AD-A243 983

DTIC
ELECTE
DEC 2 4 1991
S C D

(2)

Very Large Scale Distributed Information Processing Systems

Final Technical Report

Contract No. F29601-87-C-0072

September 27 1991

Principal Investigators:
Gerald J. Popek
Wesley W. Chu

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

Computer Science Department
School of Engineering and Applied Science
University of California
Los Angeles

91-18754

91 12 23 054

Accession For
NTIS GRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification
By Per Ltr
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Very Large Scale Distributed Information Processing Systems

Defense Advanced Research Projects Agency

Final Technical Report

DTIC COPY INSPECTED 6

1. Introduction

This final report covers the research carried out by the Very Large Distributed Information Processing Systems group at UCLA under DARPA sponsorship during the period 1987 - 1991.

This contract spanned two largely independent research efforts. During the period 1987-1989, the Tangram project under the direction of Co-PIs Dr. R. Muntz and Dr. D.S. Parker constructed an object-oriented declarative programming environment for systems performance modelling.

In 1989, the project was retargeted. The distributed operating system effort directed by Dr. G. Popek which had been ongoing during 1987-1989 was expanded. Dr. W. Chu replaced Drs. Muntz and Parker as Co-PI and initiated the fault tolerant distributed database research.

Reflecting this history of the project, this final report covers three primary areas. We will first introduce the distributed operating and file system work, followed by the database research, and finally the Tangram project. A selection of technical reports giving more detail about aspects of each of the three areas of the research follows.

1.1 Distributed Operating and Filing Systems (Dr. Gerald Popek)

1.1.1 The Ficus Replicated File System

The centerpiece of our work over the last four years is the design and implementation of Ficus, a replicated general filing environment for Unix intended to scale to very large (nationwide) networks [GHMP90]. There are three fundamental characteristics of the work which distinguish the Ficus architecture. First, it embodies an optimistic view of update in which any file or directory may be referenced or updated so long as some copy is available; conflicts are addressed when reconnection occurs [Guy91]. Second is its approach to modularity through stackable layers [HP91]. Third is its solution to the very large scale naming problem using on-disk volume grafting on demand [PGPH91]. Technology transfer efforts are underway which appear likely to result in each of these three contributions being incorporated

in commercially available versions of Unix in the next several years:

Ficus is now operational and in use at UCLA. While currently running in the context of SunOS, it has been constructed in a manner that can be added to any operating system (including many versions of Unix and Mach) that provides a VFS file system interface. The implementation consists of two stackable layers, "logical" and "physical". The logical layer provides layers above with the abstraction of a single copy, highly available file; that is, the existence of multiple replicas is made transparent by the logical layer. The physical layer implements the abstraction of an individual replica of a replicated file. It uses whatever underlying storage service it is stacked upon (such as a Unix file system or NFS) to store persistent copies of files, and manages the extended attributes about each file and directory entry. When the logical and physical layers execute on different machines, they may be separated by a transport layer which maps vnode operations across an RPC channel in a manner similar to NFS.

The stackable layers architecture provides a mechanism whereby new functionality can be added to a file system transparently to all other modules. This is in contrast to today's Unix file systems in which substantial portions must be rebuilt in order to add a new feature. Each layer supports a symmetrical interface for both: a) calls to it from above and, b) with which it performs operations on the layers below. Consequently, a new layer could be inserted anywhere in the stack, for example, to encrypt data that passes through it, without disturbing (or even having source code to) the adjacent layers. Thus, stackable layers is an architecture for extensible file systems.

The optimistic approach to replication taken here is particularly appropriate for very large or geographically dispersed filing environments. In such a domain, the network is constantly partitioned. Conventional approaches to replica management actually decrease availability for update as the number of replicas increases. Given the actual occurrence of conflicting updates is believed to be quite low, it makes more sense to detect and repair conflicts when they do occur than to take expensive measures to prevent them. Ficus employs a suite of distributed algorithms which reliably detects conflicting updates. In the case of updates to directories, where the semantics of the updates are simple and well understood by the system, Ficus is able to repair most conflicts automatically, thereby maintaining the integrity of the naming structure while providing maximal availability.

In Ficus, updates are first made to one replica of a file. Update propagation to other replicas is performed in the background on a "best effort" basis. That is, update propagation is not relied upon to maintain consistency. Replicas that are not accessible when update propagation is attempted find out about the updates upon "reconciliation" with a more informed replica. The reconciliation algorithms perform periodic sweeps of the replicated file systems, pulling in new updates, detecting conflicts, and automatically reconciling directory replicas.

Ficus has been used to connect a cluster of replicas at UCLA with remote replicas stored at Trusted Information Systems, ISI and SRI via the Internet. Elapsed time overhead for file operations is typically invisible. While system time overhead measurements show an overhead approaching 50% for worst case benchmarks, this results in noticeable delay only on heavily loaded machines when heavy name space activity occurs (e.g. recursive remove). Further, it appears that most of this overhead can be eliminated.

1.1.2 Distributed Shared Memory

Hardware and architectural characteristics strongly favor multiprocessor systems without shared memory hardware, due to switch costs and delays, as well as the ease of use of alternate LAN and optical fiber based methods. The question then becomes whether software layers can be fashioned to provide the functionality of shared memory MP systems transparently, so that standard approaches to programming and use may be employed. We demonstrated in the context of Locus that it is feasible to do so for programs with controlled sharing patterns. It is expected that many software packages exhibit the relevant behavior.

Early in the contract period, we investigated extending network transparency not only to the file system, but also to access to distributed memory in a loosely coupled distributed system. We designed and implemented distributed shared memory for the Locus disstributed operating system without any hardware support, in a manner suitable for use in a local area network [FP89].

1.1.3 Graduates

Four students in the field of distributed operating systems received their Ph.D degree under the partial support of this contract: Richard Guy [Guy91], Scott Spetka [Spet89], Brett Fleisch [Flei89], Joseph Betser [Bets88].

1.2 Fault Tolerant Distributed Database Systems (Dr. Wesley Chu)

In a distributed database system, network partitioning occurs due to site and link failures. Conventional techniques use replication to increase availability. However, network partitioning may cause blocking and only partial operability. In real-time decision support systems, availability of data is of primary importance and suspending processing is not acceptable. Further, frequently accessing remote data may be too time consuming and the user may be obliged to use local knowledge to infer the result. Since data are often correlated (for example, rank and salary; ship type and cargo), we developed a new approach that uses data inference techniques for fault tolerant and real-time query answering. Such inference techniques use accessible data and knowledge to infer inaccessible data. A knowledge base along with the database resides at each site, which can be used together to infer the inaccessible data. The knowledge base may be automatically derived from database content and application domain knowledge.

During the past two years, we have studied the acquisition of knowledge from data base and application domains [CLC90], developed open inference techniques to infer incomplete knowledge [CCH90] and also used simulation to evaluate the improvements in availability due to inference [CHL90]. We have constructed a prototype that uses SyBase as its database management system and built an inference engine to validate the proposed concept [CPC90]. The prototype confirms the feasibility of this approach and also sheds light on the following new research areas: sound inference path generation, knowledge update methodologies and correctness and completeness of the query answers, which will be investigated under the next contract.

As a generalization of the fault tolerant research, we have also extended it to cooperative distributed database systems (CoBase), in which knowledge is used to provide not only missing data but approximate, summary, and intensional answers. We have obtained some preliminary results as reported in the papers entitled "Cooperative Query Answering via Type Abstraction Hierarchy" [CCL90b], "Using Type Inference and Induced Rules to Provide Intensional Answers" [CLC91], and "A Pattern-based Approach for Deriving Intensional and Approximate Answers" [CCL91]. The cooperative concept appears to have many application areas, such as integration on heterogeneous database systems, real-time query processing, processing of image database systems, etc. These areas will also be under further investigation over the next contract period. We also plan to apply the methodology to the transportation applications.

Three Ph.D. students, A. Hwang [HWA90], R. C. Lee [Lee90], and P. Ngai [Ngai90] were graduated under the partial support of the contract. The results of their research were presented at national and international conferences.

1.3 The Tangram Modeling Environment (9/87-5/89, Dr. R. Muntz and Dr. D.S. Parker)

Today, many computers are used for the modeling of real-world systems. Demands on the extent and quality of the modeling are growing rapidly. There is an ever-increasing need for environments in which one can construct and evaluate complex models both quickly and accurately.

Successful modeling environments will require a cross-disciplinary combination of technologies:

System modeling tools
Database management
Knowledgebase management
Distributed computing

Tangram is a distributed modeling enviornment developed at UCLA. It was an innovative Prolog-based combination of DBMS and KBMS technology with access to a variety of modeling tools including stochastic, statistical, structural, equational, constraint-based, rule-based, semantic network, and object-oriented models.

A more detailed overview may be found in the attatched documents "Tangram: Project Overview," and "The Tangram Project: Publications 1987-1988." A partial list of the results of Tangram include:

1. Developed the Log(F) stream data processing language which integrates database query processing and logic programming [Nara88].

   Background/Significance: Many approaches have been proposed to integrate knowledge-based systems with database systems. However, there is an essential mismatch between AI systems and databases: they follow different processing models. The tuple-at-a-time style of AI systems is notoriously inefficient while the query-at-a-time style of databases overwhelms the workspace of an AI system with data. We can eliminate the mismatch via the *stream processing* model. We implemented a Log(F) to Prolog compiler and provided an interface to streams stored in Unix files and Ingres databases yielding a stream-based database query language.

2. Developed a distributed concurrent stream-based programming environment [LM89].

   Background/Significance: As observed above, loosely coupled multiprocessor systems currently represent the most cost effective architecture to deliver very high speed computing. The execution model of stream data processing provides an ideal paradigm for harnessing such distributed computing power. We developed the Aspen programming environment for distributed Log(F).

3. Extended Log(F) with sophisticated pattern recognition grammars [Chau89].

   Background/Significance: Parallel execution of events in a distributed system (or simulation) may be captured in an event stream for analysis. We extended Log(F) with functional grammars to produce a simple but powerful language and environment which can recognize multiple patterns in parallel in a single event stream.

4. Designed and built and demonstrated the Tangram Object-oriented Modeling Environment [PBCM89].

   Background/Significance: Tangram is a meta-modeling environment; a system for creating, storing, retrieving, updating, sharing and querying models. It features a graphical interface for constructing and querying models and allows model object behaviors to be specified using an object-oriented extension of Prolog and Log(F). Complex models may be composed of sub-models, forming a hierarchy. It features an easily extensible base set of mathematical solvers including several packages for solving Markov chains and queueing networks. The system automatically selects the most appropriate solver for a given model and query using a domain knowledge base. Tangram's base modeling domains are customizable by domain experts. Special purpose modeling environments with a look and feel tailored to individual problems may

be created quite rapidly. The system is currently in use in several commercial settings.

Four Ph.D. students participating in Tangram graduated with partial support of this contract: Hau-Ming Chau [Chau89], Thomas Page [Page89], Chung-Dak Shum [Shum89], and Sanjai Narain [Nara88].

REFERENCES

[Bets88] Joseph Betser, *Performance Evaluation and Predictions for Large Heterogeneous Distributed Systems,* Ph.D. Dissertation, University of California, Los Angeles, 1988.

[Chau89] Hau-Ming Chau, *Functional Grammar: A New Scheme for Language Analysis,* Ph.D. Dissertation, University of California, Los Angeles, 1989.

*[CCH90] Chu, W. W., Q. Chen, and A. Hwang, *Open Data Inference and its Applications,* Proceedings of CIPS Edmonton Information Technology Conference, Edmonton, Canada, October 1990.

[CCL90a] Chen, Q., W. W. Chu, and R. Lee, *Providing Cooperative Answers via Knowledge-based Type Abstraction and Refinement,* Proceedings of the 5th International Symposium on Methodologies for Intelligent Systems, Knoxville, TE, 1990.

*[CCL90b] Chu W. W., Q. Chen, and R. Lee, *Cooperative Query Answering via Type Abstraction Hierarchy,* Proceedings of the 1990 International Conference on Cooperative Knowledge Base System, Springer-Verlag, 1991.

*[CHL90] Chu, W. W., A. Hwang, R. Lee, Q. Chen, M. Merzbacher, and H. Hecht, *Fault Tolerant Distributed Database System via Data Inference,* Proceedings of the 9th Symposium on Reliable Distributed Systems, Huntsville, Alabama, October 1990.

*[CLC90] Chu, W. W., R. C. Lee, and K. Chiang, *Capture Database Semantics by Rule Induction,* UCLA Computer Science Department Technical Report, May 1990.

*[CLC91] Chu, W. W., R. C. Lee, and Q. Chen, *Using Tupe Inference and Induced Rules to Provide Intensional Answers,* Proceedings of the 7th International Conference on Data Engineering, Kobe, Japan, April 1991.

*[CPC90] Chu, W. W., T. W. Page, Q. Chen, A. Hwang, and O. T. Satyanarayanan, *Development of a Fault Tolerant Distributed Database via Inference,* IEEE Workshop on Experimental Distributed Systems, Huntsville, Alabama, October 1990.

[Flei89] Brett D. Fleisch, *Distributed Shared Memory in a Loosely Coupled Environment,*

Ph.D. Dissertation, University of California, Los Angeles, 1989.

[FP89] Brett D. Fleisch and G. Popek, *Mirage: A Coherent Distributed Shared Memory Design,* In Proceedings of the 12th ACM Symposium on Operating Systems Principles, Litchfield Park, AZ, 1989.

[GHMP90] R. Guy, J. Heidemann, W. Mak, T. Page, G. Popek, and D. Rothemeier, *Implementation of the Ficus Replicated File System,* In USENIX Proceedings, pp. 63-71, June 1990.

*[Guy91] Richard G. Guy, *Ficus: A Very Large Scale Reliable Distributed File System,* Ph.D. Dissertation (also UCLA CSD-910018), University of California, Los Angeles, 1991.

[HP91] John Heidemann and G. Popek, *A Layered Approach to file System Development,* UCLA Tech. Report CSD-910007, University of California, Los Angeles, 1991.

[Hwang90] Andy Hwang, *Fault Tolerant Distributed Database System via Data Inference,* Ph.D. Dissertation, University of California, Los Angeles, 1990.

[Lee90] Rei-Chi Lee, *Query Processing with Database Semantics,* Ph.D. Dissertation, Univeristy of California, Los Angeles, 1990. [LM89] Brian Livezey and R. Muntz, *ASPEN: A Stream Processing Environment,* In Proceedings of PARLE'89, Amsterdam, Netherlands, 1989.

*[Munt88] Richard R. Muntz and D.S. Parker, *Tangram: Project Overview,* UCLA Tech Report CSD-880032, University of California, Los Angeles, 1988.

*[Munt89] Richard R. Muntz, D.S. Parker, and Gerald J. Popek, *The Tangram Project: Publications 1987-88,* UCLA Tech Report CSD-890003, University of California, Los Angeles, January 1989.

[Nara88] Sanjai Narain, *LOG(F): An Optimal Combination of Logic Programming, Rewriting, and Lazy Evaluation,* Ph.D. Dissertation (also UCLA CSD-880040), University of California, Los Angeles, 1988.

[Ngai90] Patrick Ngai, *Constraint Propagation as an Embedded Filtering Technique for Solving Constraint Satisfaction Problems,* Ph.D. Dissertation, University of California, Los Angeles, 1990.

[Page89] Thomas W. Page Jr., *An Object-Oriented Logic Programming Environment for Modeling,* Ph.D. Dissertation (also UCLA CSD-890055, University of California, Los Angeles, 1989.

[PBCM89] Thomas W. Page Jr., S. Berson, W. Cheng, and R. Muntz, *An Object-Oriented Modeling Environment,* In Proceedings OOPSLA'89, New Orleans, LA, 1989.

[PGHP91] Thomas W. Page Jr., R. Guy, J Heidemann, G. Popek, W. Mak, and D. Rothemeier, *Management of Replicated Volume Location Data in the Ficus Replicated File System,* In Proceedings Summer USENIX Conference, Nashville, TE, 1991.

[Shum89] Chung-Dak Shum, *Alternative Representations for the Synopsis of Database Responses,* Ph.D. Dissertation, University of California, Los Angeles, 1989.

[Spet89] Scott E. Spetka, *Distributed Operating System Support for Distributed Data-intensive Applications in Network Transparent Environments,* Ph.D. Dissertation, University of California, Los Angeles, 1989.

* Attached with this report.

Ficus: A Very Large Scale Reliable Distributed File System

Richard G. Guy
June 3, 1991

Technical Report CSD-910018
Computer Science Department
University of California
Los Angeles, CA 90024-1596

UNIVERSITY OF CALIFORNIA
Los Angeles

Ficus: A Very Large Scale Reliable Distributed File System

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Richard George Guy, II

1991

© Copyright by
Richard George Guy, II
1991

Table of Contents

1 Introduction 1

1.1 Network performance 2

1.2 Scale 3

1.2.1 Partial operation 3

1.2.2 Data replication 4

1.3 Hypothesis 5

1.4 Research outline 5

1.4.1 Large scale testbed 5

1.4.2 Distributed access 8

1.4.3 Replication 10

1.4.4 Research summary 13

1.4.5 Dissertation outline 13

1.5 Related work 14

1.5.1 Distributed file systems 14

1.5.2 Replica management 16

2 Architecture 19

2.1 Stackable layers 19

2.1.1 Related work 19

2.1.2 Observations 22

2.1.3 Requirements 23

2.1.4 Ficus layer mechanism 24

2.1.5 File system decomposition 27

2.1.6 Summary 29

2.2 Volumes 29

2.2.1 Related solutions 29

2.2.2 Ficus solution overview 31

2.2.3 Graft points 32

2.2.4 Volume and file identifiers 34

2.2.5 Autografting 35
2.2.6 Creating, deleting and modifying graft points 37
2.2.7 Conflicting graft point updates 37
2.2.8 Volume summary 38
2.3 Replication 39
2.3.1 Replication overview 39
2.3.2 Logical layer 41
2.3.3 Physical layer 46
2.3.4 Vnode transport layer 49
2.4 Synchronization 54
2.4.1 Issues 54
2.4.2 Control flow examples 60
2.4.3 Summary 61
2.5 Large scale replication 62
2.5.1 Version vectors 62
2.5.2 Name space 65
2.5.3 Summary 71
2.6 Status and performance 71
2.6.1 Performance measurements 71
2.6.2 Discussion 73
2.6.3 Wide area operation 74
2.6.4 Implementation effort 74

3 Algorithms 76
3.1 Introduction 76
3.1.1 File systems 78
3.1.2 Outline 78
3.2 Algorithms 78
3.2.1 Model 79
3.2.2 Basic two-phase algorithm 80
3.2.3 Intermediate algorithm 85
3.2.4 Advanced two-phase algorithm 89

3.3 Correctness discussion 91
3.3.1 Reclamation *if* inaccessible 91
3.3.2 Reclamation *only if* inaccessible 94
3.3.3 Reclamation *exactly once* 94
3.3.4 Termination 95
3.3.5 Deadlock-free 96
3.4 Applications and observations 97
3.4.1 Directed acyclic graphs 97
3.4.2 Performance 97
3.5 Related work 98

4 Conclusions 100
4.1 Summary and conclusions 100
4.2 Future research directions 101
4.2.1 Performance tuning 102
4.2.2 Security 102
4.2.3 Databases 102
4.2.4 Typed files 102

References 104

List of Figures

2.1 Madnick's hierarchical file system design (top-down). 20
2.2 Stack with transport layer. 26
2.3 File system configuration with thick layers. 28
2.4 File system configuration with thinner layers. 28
2.5 Graft point with both parent and child volumes replicated. . . . 33
2.6 Typical Ficus layer stack. 40
2.7 File update notification and propagation. 44
2.8 Decomposed physical and storage layers. 50
2.9 Multi-layer NFS design. 53
2.10 Using NFS to interact with non-Ficus hosts. 53
2.11 Centralized synchronization service. 57
2.12 Token-based synchronization service. 57
2.13 Quorum-based synchronization service. 58
2.14 Current UFS and Ficus name space relationship. 66
2.15 Ficus global name context. 66
2.16 Volume support for global name space. (n = |replicas|) 70
2.17 Percentage overhead versus number of replicas. 72

3.1 Basic two-phase algorithm 82
3.2 One phase network example 84
3.3 Intermediate algorithm, phase one. 92
3.4 Intermediate algorithm, phase two. 93

Acknowledgments

The path to the completion of this dissertation (and beyond) has been influenced by a number of mentors. These include my parents, who encouraged my adolescent interest in computing despite the strains it placed on our relationship at the time; Steve McClain, who thought it would be interesting to teach programming to a handful of pesky high school students; Vernon Howe, who demonstrated the difference that a gifted and dedicated college teacher can make in his students' growth; Hilmer Besel, who recognized that the ignorance I felt upon graduating with a bachelor's degree in computing was evidence that a college education had taught me something after all; Jerry Popek, who welcomed me into his research group and nurtured my intellectual development throughout the past decade; the LOCUS research team, especially Bruce Walker and Evelyn Walton, who introduced me to distributed computing theory and practice; and, finally, the Ficus research team, who have been a continuing source of enthusiasm, excitement, and encouragement. I am further gratefully deeply indebted to the generous support that the Defense Advanced Research Projects Agency has invested in computer science research,[1] without which I would be (ungratefully) deeply indebted to the federal Student Loan Marketing Association.

[1]This research was sponsored under DARPA contract number F29601-87-C-0072

ABSTRACT OF THE DISSERTATION

Ficus: A Very Large Scale Reliable Distributed File System

by

Richard George Guy, II
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Gerald J. Popek, Co-Chair
Professor Walter J. Karplus, Co-Chair

The dissertation presents the issues addressed in the design of Ficus, a large scale wide area distributed file system currently operational on a modest scale at UCLA. Key aspects of providing such a service include toleration of partial operation in virtually all areas; support for large scale, optimistic data replication; and a flexible, extensible modular design.

Ficus incorporates a "stackable layers" modular architecture and full support for optimistic replication. Replication is provided by a pair of layers operating in concert above a traditional filing service. A "volume" abstraction and on-the-fly volume "grafting" mechanism are used to manage the large scale file name space.

The replication service uses a family of novel algorithms to manage the propagation of changes to the filing environment. These algorithms are fully distributed, tolerate partial operation (including nontransitive communications), and display linear storage overhead and worst case quadratic message complexity.

CHAPTER 1

Introduction

Two decades ago, the first large scale, wide area network computing experiment began at UCLA, SRI, UCSB, the University of Utah, and BBN [Cro87]. By 1980, the ARPANET connected hundreds of hosts across North America, Hawaii, and Europe. Nearly 100,000 hosts are attached to the DARPA Internet today; at this rate, the total will approach ten million by the end of the century.

Somewhat surprisingly, the phenomenal expansion of the user community has not been in response to, or even been accompanied by, changes in the style of ARPANET usage. Remote login, electronic mail and news, and bulk data transfer remain the primary services using the network. The introduction of internetworking protocols around 1980, and a domain naming system a few years later, have significantly expanded the underlying capabilities of the ARPANET, but higher level services continue to be much the same.

During this same period, local area network technology emerged from research laboratories into the commercial marketplace. Based on the same packet-switching concepts as wide area networking, local area network services rapidly evolved from remote login and simple data transfer to include distributed file systems, remote procedure call services, and even complete distributed operating systems.

The explosive growth of the ARPANET, and its transformation into the Internet, is directly attributable to the proliferation of local area networks. As service-rich local area networks have been interconnected with limited service wide area networks, interest has grown in extending the services available on wide area networks to include, among others, distributed file systems.[1]

Two prominent issues immediately encountered when extending traditional

[1] A review committee of the National Research Council endorsed the concept of a national research network in 1988 [NRC88]; in 1989, Senator Albert Gore, Jr., introduced legislation to invest $1.75 billion in a National Research and Education Network; in early 1990, the President's Council of Advisors on Science and Technology's $1.9 billion Federal High Performance Computing program (similar to Gore's proposal) has been endorsed by the Bush administration. Success of these programs hinges upon client-friendly services usable by the masses, not those endured by computer scientists.

local area network services into the wide area network arena are network performance and scale.[2]

1.1 Network performance

From its inception, the ARPANET primarily used transmission lines with much less bandwidth than an I/O channel on a typical host. The bandwidth ratio in the early 1970's was typically 1:200 (50 Kb/s ARPANET : 12 Mb/s UNIBUS); by the late 1980's, the ratio approached 1:4,000 as common workstation I/O bus rates exceeded 200 Mb/s (VME bus). The dramatic difference between I/O bus and network capacity forced network clients to be conscious of the network presence and conservative in its use.

In contrast, the 10 megabit per second local area network transmission rates of the early 1980's closely matched minicomputer I/O bus rates of the day. This similarity of bandwidth encouraged system designers to utilize network facilities in new ways, such as distributed file systems. The concept of *network transparency* became fundamental, much as virtual memory was accepted a decade earlier. Even with an order of magnitude difference in bus and network bandwidth today, local area network services continue to expand and be effective.

DARPA Internet sponsors are currently responding to the bandwidth disparity. A new Internet "backbone" with a bandwidth greater than one megabit per second is being installed in the United States. Over the next few years, the bandwidth will increase into the gigabit per second range as optical fibers replace copper wires as the primary transmission media. This realignment of network transmission capacity and host I/O bus rates lays a necessary portion of the foundation to provide wide area services that traditionally have been limited to local area networks.

Another important network performance parameter is latency induced by transmission delay. An inherent delay of 30 milliseconds is incurred by a transcontinental round trip message traveling at the speed of light. (The delay increases to $\frac{1}{2}$ second for geosynchronous satellite channels.) Services such as interprocess communication and remote procedure call for which the local latency is usually measured in microseconds are dramatically affected by large inherent latencies.

[2]In reflecting on Carnegie-Mellon University's network, Mahadev Satyanarayanan comments, "The change that would most substantially improve the usability of Andrew [CMU's campus-wide distributed system] would be a distributed file system that completely masked failures from users and application programs. It is still an open question whether this goal is achievable in conjunction with good performance and scalability." [Sat88]

In contrast, the effect of a minimum 30 millisecond delay is only moderately significant for file system services today: the average access times for typical disks are also in the low tens of milliseconds.

1.2 Scale

Local area networks are not by definition limited in scale, yet many services are implemented with assumptions of small scale. For example, the Locus distributed file system has a fixed design parameter of a maximum of thirty-two hosts. The Sun Network File System (NFS) assumes that the identity and location of every accessible filesystem is placed in a single monolithic file on every host; such a file is not manageable in practice on a large scale.

The issue of scale is more complex than simply increasing design parameters or concocting algorithms that efficiently handle large amounts of data. Large scale impacts administrative concerns such as resource management, protection, and legal issues. Scale may also affect the usability of the system: a client may be overwhelmed by complexity if large scale implies a fundamentally different mode of interaction with the computing environment.

Yet another aspect of large scale is the reliability of the system, at both a component level and overall. A distributed computing environment composed of a million hosts will have at least a million network interfaces and communications paths. The probability that all components will be operational at any single time is almost zero, and effectively zero over any useful period of time. This indicates that *partial operation* is the normal, not exceptional, mode for a system of this scale. Services intended for use in this environment must consider component failure to be a routine condition; in some cases, a failed component may *never* be repaired.

1.2.1 Partial operation

In this context, the definition of "failure" is broader than simply an unexpected denial of service which will be repaired in the near future. Failure also includes deliberate actions, such as terminating a communications link during high-tariff periods or to protect a host from external threats. An overloaded gateway may exhibit failure conditions when it is unable to respond in a timely fashion. Some components may be taken out of service permanently, either by design or disaster.

ARPANET designers anticipated the partial operation problem, and quite successfully resolved it with respect to communications paths by establishing a

multiply-linked network of communications controllers with an irregular topology. Adaptive routing algorithms are used to locate, evaluate, and select alternate communications paths. In two decades of operation, network partitioning has rarely occurred.[3].

The ARPANET solution is no panacea, however. Multiple (long-haul) links are quite expensive from an economic standpoint; in fact, the new Internet backbone is composed of just a few well-connected regional hubs that interface to area networks. Nevertheless, there continue to be single points of failure which can inhibit communication [Neu87]. A regional hub failure may isolate a significant portion of the network; a gateway or host failure may isolate just a few hosts, or prevent access to data managed by a failed or inaccessible host.

Although partial operation is unavoidable in a large scale, wide area network, its detrimental effects can often be minimized. A wide area file system is an example of a valuable service that can, but need not, be rendered impotent by partial operation. As with the ARPANET communications links example, redundancy at one or more levels is the primary tool for counteracting partial operation's negative consequences. This work proposes *data replication* as an affordable approach to cope with partial operation in a wide area file system.

1.2.2 Data replication

Data replication techniques combat the problems of partial operation by using redundancy to avoid a single point of failure (the data). Rather than accessing a specific single copy, a client accesses a varying subset of data replicas. The number of copies accessed depends on the consistency method used, but the probability of successfully accessing a subset which satisfies the consistency criteria is expected to be greater than or equal to the probability of successfully accessing a particular copy.

A number of data replication methods have been incorporated into local area network file system services: LOCUS [PW85], ROE [EF83], Isis [Bir85], PULSE [TKW85], Eden [PNP86], Guardian [Bar78], Saguaro [PSA87], Gemini [BMP87], GAFFES [GGK87], and a UNIX United variant [Bre86]. Most assume that the number of copies is small, and that they are stored among a small number of hosts. A further assumption is that failures are rare, or of suf-

[3]Three notable exceptions: the 1987 partitioning resulting from a single broken fiber despite 7-fold redundancy [Neu87], the 1980 complete network failure which occurred when a single bit was dropped from a widely propagated status word [Ros81], and the 1988 Internet worm catastrophe which prompted many sites to preemptively shutdown as a means of protection [Spa89]

ficiently short duration that clients can wait until the failure is repaired. These assumptions are invalid in a large scale system; new replication techniques must be developed to cope with large scale issues.

1.3 Hypothesis

The hypothesis of this research is that a large scale, wide area file system is feasible. To be feasible, it must compensate for partial operation; it must support a large number of hosts, clients, and data; administrative issues must be satisfactorily addressed; and the client interface must not be complex.

The true test of this hypothesis is to design, build, install, and use a wide area file system that scales to millions of hosts, compensates for partial operation, and is manageable. This research effort endeavors to accomplish the first of these phases by presenting a large scale, wide area file system design; the second is accomplished by implementing the design. Installation and extensive use of the implementation are beyond the scope of this work.

1.4 Research outline

A number of research issues are directly identifiable in the preceding discussion. Broad issues include large scale testbed design, distributed access methodology, and replica management. Each is considered in more detail in the following sections.

1.4.1 Large scale testbed

Conducting an experiment of this scale depends upon cooperation of a large number of installations, most of which must quickly perceive the potential near-term benefits (to themselves) of participating in the experiment. Maximal cooperation is dependent, in part, on minimal disruption of an organization's unrelated activity. This suggests that the experimental large scale file system services should be engineered as modules which may be easily attached to an installation's operating system environment(s).

The importance of a modular design extends beyond the initial development of a testbed. File systems depend upon many other services which can be expected to experience evolutionary, perhaps revolutionary, enhancement in the future. File system services which are intertwined with current state of the art storage

services, for example, will likely seldom benefit from novel developments. The resulting hysteresis may even condemn both intertwined services to obscurity, rather than just the one. Good modular design mitigates this problem.

1.4.1.1 Target environment

The available pool of candidate cooperating installations is the DARPA Internet community, in which UNIX is the predominant software environment. Of the various UNIX versions in common use, SunOS (Sun Microsystem's UNIX implementation) is attractive. The primary attraction is an internal design that readily supports multiple filesystem implementations; a secondary benefit is that SunOS is widely used within the Internet.

The file system portion of the SunOS UNIX kernel is built around a generic interface known as a *virtual file system*. The interface hides details of the particular file system implementation which is handling a request. AN extended version of the virtual file system interface is used to add the experimental large scale file system service to standard versions of SunOS.

1.4.1.2 Leveraging via stackable layers

Since this research focus is large scale issues, and not file system services in general, it is desirable to leverage existing (and future) services wherever possible. Primary candidate services are raw disk management and network transport of file data.

The virtual file system interface enables file system services to be leveraged in an interesting way: a novel file system service can be added, which utilizes existing services as though it (the novel service) was simply a routine client (such as a system call). This organization results in a "stack of layers" in which each layer exports the same interface.

In addition to its leveraging benefits, the stackable layers concept allows transparent services to be added. For example, a generic file system performance monitoring layer can be "slipped in" between any two stacked layers sharing the virtual file system interface.

This work uses the stackable layers model as an aid to designing and implementing large scale file system services. The existing UFS (UNIX File System) and NFS (Network File System) services within SunOS are used to provide disk management and network transport services, respectively. A new layer (or layers) is constructed that supports the virtual file system interface, and which uses UFS

and NFS services. This layer is engineered as a module which can be linked into a standard SunOS kernel.

1.4.1.3 Efficient layering

A layered implementation model often introduces additional overhead. The extra cost is especially pronounced when an upper layer builds on on lower layer services that are richer than required. To some extent, this is the case when building a large scale file system on top of UFS.

For example, the replication service provides a more powerful directory service than that currently provided by the UFS. The layered approach results in a directory service in one layer which largely replaces the service in a lower layer, but is constrained to pay for the lower layer service as well. A prototype of the Andrew File System tried this approach; it was abandoned because of significant overhead, and the layers were replaced by a single-layer, customized UFS.

In this particular case, much of the additional overhead resulting from unneeded UFS services may be avoided by exploiting various locality phenomena prevalent in UNIX system usage. Applying the results of recent studies of UNIX file access patterns to the new layers and exploiting the current finely tuned UFS implementation limits additional overhead to acceptable levels.

1.4.1.4 Naming

The primary interface between clients and a file system is the naming scheme, which includes the syntax of the name space and the operations defined over it. A well designed naming scheme is a catalyst for network file sharing, a poor design hinders and frustrates sharing.

Design goals for a large scale file system are similar in many respects to those of small scale distributed file systems. They include:

- simplicity
- ease of use
- file system boundary transparency
- syntax transparency with respect to existing file system types
- location transparency
- name transparency

- local autonomy for name management
- compatibility with embedded names in existing software
- support for large numbers of hosts, clients, and files

Existing naming schemes fail to meet one or more of these goals.

A testbed environment composed of existing installations necessarily places some limitations on the design of a naming scheme. For example, the choice of SunOS as a base system implies that the name space syntax must be compatible with the UNIX naming scheme. These limitations are not significantly constraining, however. The tree-structured UNIX name space can be supported within the context of a more general model, such as a directed acyclic graph.

The goals of simplicity and support for large scale suggest that some form of hierarchy serve as the name space model. It is not immediately clear how to integrate a few million existing, disjoint instances of the UNIX name space into a hierarchy that meets all of the design goals. Major issues of name space design and supporting mechanism must be addressed.

1.4.2 Distributed access

The file system service must be able to locate a file and provide access to it based solely on a client-provided, location transparent file name. Every host supporting the large scale name space should be able to efficiently locate *any* file named within the name space, and then support low-overhead read and update access.

1.4.2.1 Volume mapping

Ideally, the logical organization of a name space is unrelated to the physical location of the files themselves. In practice, effective resource management is achieved by physically grouping files with logically related names, yielding what is sometimes called a *volume*.[4] A name space, then, is structured as a hierarchy of volumes, each containing a rooted hierarchy of file names.

In existing distributed file systems, the macro-organization of volumes is usually represented as a table of mappings between names and volume roots. The

[4] Volumes were introduced in the Andrew File System [HKM88]; they are used here in similar ways, but with important differences in implementation and detail.

volume mapping table (sometimes known as a mount table) is external to volumes. Each host supporting the name space maintains its own volume mapping table for the portions of the name space in which it is interested.

The concept of volumes is appropriate for a large scale file system, but the mechanisms incorporated to date in smaller scale systems are generally not applicable. A large scale file system will contain millions of volumes, with new volumes being established frequently. Few installations can afford to attempt to maintain a complete table of volume mappings, yet every host must be able to locate any file named within the entire name space.

This work addresses the volume mapping table problem by distributing the table and embedding table entries within the name space itself. A new *volume graft point* file type is introduced into the large scale file system service. A graft point contains the same mapping information formerly placed within the volume mapping table. With this mechanism, a host never needs to maintain a large amount of volume mapping data. A graft point is dynamically "interpreted" in a manner similar to NFS automounting.

1.4.2.2 File transport

A file that has been located for a client is usually accessed soon thereafter. If the file is remote, its data pages must be transported to the client's host. Small scale file systems have used one of two approaches: demand paging, or whole file pre fetching and caching. Updates are handled either by asynchronous delayed write-back, or synchronous write-through to the file itself.

The whole file prefetching approach exploits the locality of usage that is typical of UNIX clients. Most UNIX file access is to small files that are read sequentially in their entirety. File updates generally are preceded by a scan of the entire file, and then the file is completely rewritten. Newly written data is often rewritten or deleted shortly thereafter. When this behavior is prevalent, whole file pre-fetching and caching has substantially less overhead than demand paging. The whole file method is not appropriate for large, random access files such as databases; demand paging is a much better scheme.

The target system's existing transport service, NFS, uses demand paging and delayed write-back. NFS has been leveraged to avoid having to design and build a new transport service at the outset. The benefits of whole-file caching can be obtained with the data replication services described in the sequel.

1.4.3 Replication

Large scale distribu ' systems inherently possess two unpleasant characteristics, partial operatio and significant communications latency. Data replication techniques have been applied to small scale local area systems to address partial operation and latency problems (although they are much less pronounced). This suggests that replication has the potential to ameliorate these problems in large scale systems.

Data replication methods maintain multiple copies of data, but strive to present the client with the illusion of a single, highly available file. Major issues include replica consistency, management algorithms, and client interface.

1.4.3.1 Consistency

Almost all existing and proposed replication mechanisms adhere to serializability[5] as a consistency definition. It is not clear, however, that enforcing strict serializability is appropriate for all, or even most, file usage in a large scale, wide area system.

Serializability protects applications from interfering with each other. Existing serializability enforcement techniques place substantial availability constraints on access, often trading off read availability requirements against update availability.

For example, the *primary copy* [AD76] strategy requires that all updates to a file be performed on a designated copy; other copies may be brought up to date in parallel with the designated copy, or at a later time. Read-only access may be serviced by any accessible copy. In this case, read availability is very high, but update availability is no greater than with a single copy.

The *majority voting* technique [Tho78] ensures that every update is applied simultaneously to (at least) a majority of replicas. Read access which is to be followed by a related update must involve at least a majority of replicas, to guarantee that the latest version of the file is read.

Quorum consensus [Gif79] is a generalization of the basic voting approach, in which replicas are *a priori* allocated fixed, but possibly differing quantities of votes, and read and update quorums can be assigned different thresholds. The sum of the thresholds must be one greater than the number of replicas, but the update threshold must always be greater than one half of the number of votes.

[5]Informally, serializability requires that all client actions be logically orderable in a serial, non-concurrent fashion. The actions themselves may be physically concurrent, but their logical order must be serial.

If update is rare relative to read, the read threshold can be made small, and the update threshold made large. Such an arrangement optimizes both performance and availability to a particular job mix, but sacrifices the availability of update in favor of read, or vice-versa.

Such approaches are inappropriate for an environment characterized b partial operation and high latency. Partial operation undermines the availability assumptions used to justify the read versus update tradeoff: the quorums required by even a modest number of replicas may never be achievable. High latency makes the cost of consulting several replicas very expensive.

The result is that the average client has lower availability and poorer service than a single client near a single replica. This is especially unfortunate since empirical studies indicate that very little concurrent file usage, and even less update, actually occurs. Concurrent updates are even more rare. Thus, the cost and service limitations implied by serializability often seem to be unwarranted.

The rarity of concurrent update access suggests that optimistic replica consistency strategies should be considered. This work adopts a novel *one-copy availability (OCA)* consistency policy which enforces *no lost updates* semantics. OCA permits read and update access when *any* data replica is accessible. A spectrum of strictness can be enforced, ranging from "use any available replica" to "any replica no older that this client has used before." Strict serializability enforcement can be constructed on top of OCA for those clients who demand it.

1.4.3.2 Replica management algorithms

The one copy availability consistency policy initially applies an update to a single file replica. The task of *propagating* the update (alternatively, the new file version) rests with a separate service. A consistency policy like OCA that allows concurrent, unsynchronized update further relies on an update *reconciliation* service to detect concurrent update activity and thereby ensure that no updates are inadvertently lost. Detection of update conflicts may be accompanied by conflict resolution in those cases in which the reconciliation service understands the semantics of the updates.

Accurate concurrent update detection can be accomplished using Parker's *version vector* technique [PPR83, PW85, Guy87, SKK90]. A version vector is a multi-dimensional version number in which each component corresponds to the number of updates applied initially to that replica. The absence of concurrent updates is indicated when a comparison of replicas' version vectors determines that one vector's components are each pair-wise greater than or equal to the cor-

responding components of another vector. Otherwise, concurrent update activity has occurred.

Concurrent updates are assumed to represent conflicting unsynchronized activity until some semantic-knowledgeable agent declares the conflict to be resolved. For many concurrently updated files, only the client can determine how the conflict is to be resolved. But concurrent updates can also be applied to the name space itself. Since the name space is closely managed by the name service, the potential exists for automatic resolution of concurrent name space updates.

A name space reconciliation service must first determine what updates should be propagated to a particular replica, and then apply those updates to it. The major issue here is how the reconciliation service learns which updates have occurred.

One approach is to maintain replica-specific update logs. The reconciliation service compares replicas' logs, identifies those updates which haven't been applied to the replica in question, and applies them to the replica and makes an appropriate log entry. Log entries should be garbage collected when they are no longer relevant, i.e., when each replica has applied the update; an algorithm such as that used in Jefferson's Global Virtual Time [Jef85] might suffice.

Name space update logs bear a striking resemblance to the name space to which they refer. It is therefore interesting to consider coalescing or embedding the log into the name space. Major benefits include a reduction in space overhead and the reduced complexity of managing only one structure.

This research investigates a log-less approach in some depth and develops a new class of low-overhead garbage collection algorithms. The algorithms place few requirements on underlying communications services, in that the topology may be arbitrary and continuously changing; no ordering of replicas is needed; global storage requirements are quadratic in the number of replicas, with a small coefficient (a few bits per replica); worst-case message complexity is quadratic, with message length as a linear function of the number of replicas.

This work incorporates these new garbage collection algorithms into the name space reconciliation service for OCA. Further research is needed to extend the basic algorithms to support dynamic growth and/or reduction of the number of replicas. Early indications suggest that growth is fairly easy to support, and reduction is more difficult.

1.4.3.3 Client interface

Replication should normally be a transparent feature of a file system. Nevertheless, there are circumstances in which it cannot be transparent, such as when conflicting file updates must be resolved. There are also times when its visibility can be helpful. For example, knowledgeable clients may wish to specify the placement and number of replicas, rather than rely on the replication service's default selections. Clients may also wish to specify stricter or weaker consistency policies than that chosen by default.

Exposing replication services to clients is a perplexing problem for a layered architecture, when the new service layers are separated from clients by other layers (which appropriately know nothing about the new services). The solution here exploits an extensible version of the virtual file system interface [HP91a], which supports a layer *bypass* mechanism that enables any layer to redirect an unknown service request to the layer immediately beneath it in a stack.

1.4.4 Research summary

The above research outline covers a broad range of issues which demand consideration in the development of large scale, wide area file system. It covers testbed development, including selection of a target operating system, identification of a suitable interface for a layered architecture, and examination of naming issues. Large scale distributed access research problems include volume management and file transport. Finally, a comprehensive approach to replication is considered.

This research does not cover all issues fundamental to a large scale file system. The most prominent, perhaps, is authentication and security. The Internet community is well aware of the importance of this aspect of large scale computing environments;[6] it must be addressed at some point, but it is outside the scope of this effort.

1.4.5 Dissertation outline

The remainder of this work consists of two main parts, the architecture and implementation of the Ficus[7] file system (Chapter 2), and a rigorous presentation

[6]The Internet worm case [Spa89] is a recent example of vulnerabilities in large scale environments.

[7]The name was inspired by the topological similarity of two trees, one from cyberspace and one from nature. The cyberspace tree is found in a large scale filing environment composed of existing standalone tree-structured name spaces connected by a shallow super-rooted sub-tree,

of the family of two-phase algorithms used in optimistic replica management (Chapter 3). Chapter 4 summarizes the research and the conclusions which can be drawn from it, and finally suggests future directions in which this research could readily proceed.

1.5 Related work

The primary collections of related work are found in distributed file systems literature and replica management research. A brief summary of significant research follows.

1.5.1 Distributed file systems

Several interesting distributed file systems have appeared in the last decade. They are reviewed here, along with relevant file access studies.

1.5.1.1 Important system implementations

The file systems mentioned here all support some form of *network transparency*. Each supports distributed access and *location transparent* naming. The extent to which *name transparency* is supported varies widely: it is guaranteed in some (e.g., LOCUS), and must be provided by convention in others (e.g., NFS).

- LOCUS
  The LOCUS [PW85] distributed operating system provides clients with the illusion of a very large, highly reliable single system. The (research) LOCUS filesystem provides extensive file replication services. Within a single partition, a very high degree of file access consistency is maintained. Concurrent, partitioned updates are tolerated, and detected later through the use of version vectors [PPR83].

  LOCUS is intended for small scale environments, such as that typified by office environments using a local area network. Each node regards the others as peers, independent of their functionality.

- NFS
  Sun Microsystem's Network File System is a distributed file system. Rep-

the 'natural' tree is the *Ficus benghalensis*, commonly known as the Banyan tree, a fig species renowned for a wide-branching visible root structure seemingly grafted high onto a main trunk.

lication is not supported; UNIX consistency semantics are approximated to some extent.

- Andrew/Coda
  The Andrew [HKM88, Sat88] filesystem provides client workstations with a highly reliable centralized file service. A client-server relationship exists between workstations and the central pool of file servers; clients do not privately share files. Client files are cached (in entirety) on the client's workstation upon the first access to the file. Andrew provides file replication only amongst the pool of file servers, and assumes that no partitioning occurs between servers storing file replicas.

  A callback mechanism is used between servers and clients for consistency control. A callback is a promise by the server to notify every client with a cached copy of a file that another client has finished updating a cached copy and the server is about to make its own copy consistent with the updated cached copy. Clients are required to privately synchronize update activity originating from distinct workstations.

  This mechanism is not resilient to communications failures, and so undetected conflicting updates can occur. This can result in a previous file version being silently overwritten.

  Andrew is intended for a campus-type environment: geographically dense, with a few thousand client workstations.

  Coda [SKK90] is a general purpose replicated filesystem service for Andrew. It uses file replication techniques inspired by the LOCUS file system to support replication between partitioned file servers. Client workstations obtain a temporary file "replica" (through Andrew's whole-file caching mechanism), which may be modified even when a server is inaccessible. Conflicting updates are reliably detected; a simplification to the version vector scheme introduces occasional false conflict notifications.

- Isis/Deceit
  The Deceit file system [SBM89] is based on Isis [BJ87] Cornell's distributed system kernel. Isis provides support for resilient objects, which Deceit uses to underly file replication. Deceit allows a broad range of file consistency semantics during network partitions, from serializability (using majority consensus protocols to obtain mutual exclusion) to optimistic, non-serializable semantics. No support for detection of partitioned conflicting updates is provided.

Deceit appears to clients to be an extension to an NFS environment. It also supports a file versioning mechanism similar to that found in the VAX/VMS operating system.

1.5.1.2 File access/placement studies

Three comprehensive file system activity traces have been performed, one in a commercial mainframe setting [Smi81] and two in university UNIX environments [OCH85, Flo86b]. These traces form the basis for a number of analyses [Smi81, OCH85, MB87, Kur88, Flo86b, Flo86a].

These studies generally showed that a few files receive a large portion of file accesses; most files accessed are small, and read sequentially in their entirety. A significant amount of working-set type locality was observed: a reference to a file was frequently followed by another reference to the same file, and with lesser frequency to a file located within the same file directory. Careful cache management strategies were shown to be extremely useful in reducing I/O and other file system overhead. Little file sharing occurred, with the exception of a few heavily accessed (for read) system files.

1.5.2 Replica management

In the past twenty years, dozens of published papers have described various replica management protocols. It is convenient to classify the protocols according to several criteria: failure mode assumptions, pessimism about concurrent updates, and mutual exclusion methodology. Specific replicated file system and directory management proposals are also discussed.

1.5.2.1 Limited failure modes

Early work on replica management assumed that inaccessible replicas had completely failed; in this model, communications failures "do not occur." Ellis' [Ell77] ring-oriented broadcast strategy utilized update history logs to recover failed replicas. Despite its limitations, various improvements have been proposed, and utilized in implementations. An improved atomic broadcast technique reduced the overhead [JB86] and was incorporated in the Isis kernel [Bir85].

1.5.2.2 Pessimistic concurrency

Most replica management protocols are *pessimistic* about concurrent update: potentially non-serializable activity is explicitly prevented from occurring. Various *mutual exclusion* techniques have been proposed to ensure that concurrent, unsynchronized updates do not happen. All of these protocols occasionally prevent update activity anywhere, even when each replica is accessible by one client or another (but no client can access all replicas).

The *primary copy* strategy was introduced by Alsberg [AD76]. One replica is designated as primary and receives all updates; secondary replicas are updated lazily. A failed or inaccessible primary replica halts all update activity. The primary copy method has been used in the INGRES database [Sto79] and the commercial LOCUS file system [PW85].

Voting techniques (also known as *majority consensus*) have higher availability than primary copy methods. Thomas' original voting method [Tho78] assigned one "vote" to each replica; an update could proceed only if a majority of replicas agreed to synchronously perform it. Unsynchronized concurrent updates are prevented by the mutual exclusion behavior of "majority."

Weighted voting [Gif79] improved upon basic voting by allowing a different number of votes at each replica, to account for varying reliability characteristics of replicas. Gifford also noted that an explicit majority of votes is not required for mutual exclusion: the actual requirement is that the *read quorum* and *update quorum* each be large enough to intersect.

Availability limitations imposed by infrequently accessible replicas have been addressed by *dynamic voting* protocols [BGS86, Her86, ES83, DB85, JM87]. These allow adjustment of quorum definitions within a "majority partition," to redefine the "size" of a majority by effectively disenfranchising currently inaccessible replicas. *Ghost* replicas have been proposed as a way to reduce the actual storage costs of voting mechanisms [Par86, RT88].

Voting has been proposed or used in a wide variety of applications: name servers [BG85], bulletin boards [Edi86], databases [VM87], reliable storage [Ber85, BY87], file directories [BDS84], and Eden kernel replicated objects [NPP86].

1.5.2.3 Optimistic concurrency

Optimistic replica management approaches exploit the observation that concurrent update is relatively rare. Some support one-copy serializability by either delaying update commit [AR85], or backing out updates at a later time [SBK85,

Dav84, Wri83]. Optimal ex post facto analysis of updates to determine which should be backed out is an NP-complete problem [Dav84, Wri83], so some updates may needlessly revoked for the sake of efficient analysis.

Optimistic non-serializable approaches have been proposed for computer conferencing [Str81], commercial information retrieval services [ABG87], bulletin board [BJS86], databases [BK85, SKS86, GAB83, All83, Fai81], and file directories [FM82, PPR83, Guy87]. Most of these base correctness criteria upon the semantics of the data and the operations performed.

1.5.2.4 Replicated directory management

The *directory replication* problem has received particular attention because of the central role it plays is designing a highly reliable distributed file system. A wide range of directory replication mechanisms have been proposed, from serializable to non-serializable.

Fischer and Michaels [FM82] presented the first detailed examination of directory replication. They recast the problem as a replicated *dictionary* problem, to focus on the basic *insert* and *delete* operations common to each. Unsynchronized, concurrent directory modifications are resolved via timestamps.

Various inefficiencies in Fischer's work were addressed by Allchin [All83]. Wuu [WB84] offered further improvements. Unfortunately, the successive improvements reduced communications complexity at the expense of storage complexity: each replica in Wuu's scheme is required to maintain a version matrix.

Bloch [BDS84] utilized weighted voting in a serializable approach to directory replication. A novel scheme was adopted in which no single directory replica need (or can be assumed to) contain a true picture of the directory's status. Several replicas must be consulted to compose a correct view of the directory. Bloch's approach is inexpensive for interrogating or adding to a directory, but expensive for deletions.

Guy [Guy87] proposed a non-serializable solution in the spirit of the LOCUS system. It is based on Parker's version vectors [PPR83], and supports a wider range of semantics than the earlier work by Fischer, et al. A novel feature of this method is that it tolerates conflicting updates after the conflict is discovered. (Earlier techniques made an immediate decision on how to resolve the conflict; the haste often resulted in unpleasant effects, such as loss of a newly created file.)

CHAPTER 2

Architecture

This chapter presents the main portions of the Ficus file system design. It begins with a discussion of the stackable layers design methodology used throughout Ficus. Section 2.2 describes Ficus volumes, which are a fundamental construct for filing in general, and replication in particular. Section 2.3 describes the Ficus replication approach and facilities, while Section 2.4 focuses specifically on file access synchronization issues in Ficus.

2.1 Stackable layers

Stackable layers is a modular structuring paradigm distinguished by the use of identical interfaces for each layer. The application of structured programming techniques to file system design is not new, nor is the concept of symmetric interfaces (which has previously been applied to terminal I/O and network protocols). The contribution of this work is the integration and extension of these concepts by the application of stackable layers to file system design.

2.1.1 Related work

The two primary classes of published work related to stackable layers are structured approaches to file system design and the use of symmetric interfaces in I/O and network protocols.

2.1.1.1 File system structuring

Previous work on structured design of file systems is typified by an early proposal for hierarchical structuring of file systems and the current state of the art in operating system support for multiple co-existing file systems.

In 1969, Madnick and Alsop [MA69] presented a modular approach to file system design which was inspired by earlier work of Dijkstra [Dij67, Dij68] and Randell [Ran68] on structured design. Their approach separates file system ser-

Symbolic file system	File directory manipulation
Basic file system	File meta-data access and management
Protection	Access control verification
Logical file system	Access methods and file structure
Physical file system	Logical to physical address mapping
Device strategies	Initiate I/O, manage buffers, allocate storage

Figure 2.1: Madnick's hierarchical file system design (top-down).

vices into discrete levels of abstraction in which each level communicates only with its immediate upper and lower levels. The resulting static organization provides a framework for implementing a file system, such as the six-layer file system design proposed in Madnick and Donovan [MD74] (see Figure 2.1). The argument presented by Madnick, *et al* for modularity is the traditional one of complexity management, both for ensuring logical completeness and for debugging and quality assurance.

Operating systems have traditionally supported exactly one file system. Sun Microsystems' SunOS[1] implementation of UNIX[2] incorporates a switch which allows multiple file system services to co-exist comfortably within a single operating system.[3]

The Virtual File System (VFS) switch mechanism [Kle86] is designed around a *vnode* data structure which implements a stylized interface between a file system and other portions of the operating system kernel. The interface is essentially an information hiding technique that exports a set of operations (analogous to methods in object-oriented terminology); all data is private to some vnode, and can only be accessed via supplied vnode operations.

The apparent initial motivation for the VFS switch was to support transparent remote file access without embedding remote access mechanisms within the existing (local access) file system implementation [Kle86]. In SunOS, local filing service is handled by the UNIX File System (UFS), while remote access is pro-

[1]SunOS is a trademark of Sun Microsystems, Inc.

[2]UNIX is a trademark of AT&T.

[3]A number of other UNIX implementations now also provide file system switches. These include AT&T's File System Switch in UNIX System V Release 4, Digital's Generic File System for ULTRIX [RKH86], and the 4.3-Reno BSD (Berkeley Standard Distribution) [KM86].

vided by the Network File System (NFS[4]) [SGK85]. NFS is "stacked" onto UFS, in the sense that it exports the VFS interface to its (local) clients and uses the VFS interface on the remote site to access file systems (such as UFS) there. NFS implements a custom data transport protocol between network sites.

Each of the dozen-plus file systems in the current (4.1) release of SunOS supports a common set of vnode operations. The natural evolution of file system services has repeatedly forced a change in the definition of the VFS interface, as new services require features not envisioned by the original designers. This lack of extensibility is a major impediment to using the VFS interface as the base for a general purpose stacking facility.

2.1.1.2 Symmetric interfaces

To date, the application of symmetric interfaces has focused on protocol design for terminal I/O in UNIX and network communications in a experimental kernel.

The STREAMS I/O system was developed by Ritchie [Rit84] as a replacement for the (then) standard line discipline mechanism used to manage input and output between processes and terminals. The existing line discipline mechanism had become extremely complex as a result of adding various *ad hoc* routines to improve performance; regularity of style and service suffered as well. A more pronounced problem was the need to support multiple line disciplines concurrently for the same process—a task never envisioned in its original design.

Ritchie's solution was to define a simple queue-oriented interface which each module uses to communicate with its neighbors in a "stream" of modules. Each module filters the items placed on its input queues (one in each direction), perhaps intercepting, substituting, augmenting or passing on the items given to it. Control messages may be queued in addition to regular data messages, which normally consist of a simple sequence of characters.

The symmetric interface of each module allows a stream to be composed of modules in whatever order is desired.[5] A further feature of the STREAMS approach is the ability to dynamically add and remove modules from the stream while it is operating.

A layered design approach has also been used in the x-kernel operating system kernel [HPA89, PHO90, HP91b]. The x-kernel is a configurable kernel designed

[4]NFS is a trademark of Sun Microsystems, Inc.

[5]Note that the modules at each "end" of the stream support the interface on only one "side"—a different interface is typically used to communicate with modules not part of the stream.

explicitly for simplifying the implementation of network protocols. Primary features of the x-kernel include a uniform interface to all protocols, late binding between protocol layers, and a light-weight layer mechanism.

The x-kernel design is deliberately intended to encourage composition of protocol stacks on-the-fly, with run-time selection of layers supplying the appropriate semantics. Layering efficiency is provided by using procedure calls, not context switches, to pass information between layers. This economy is further used to promote decomposition of protocols into multiple layers as a means of aiding flexibility.

2.1.2 Observations

Operating systems and file systems have grown significantly in size and complexity in the twenty years since Madnick's proposal was presented. Greater complexity has often manifested itself as a tendency towards monolithic system implementations which increasingly defy adequate testing and verification, and hinder improvements—especially those not conceived within the monolithic framework.

In recent years, micro-kernels in systems such as Mach [ABG86, RBF89] and Chorus [RAA90] have emerged as a response to the monolithic implementations of operating systems. However, little has been done to tackle the monolithic filing service portion of most kernels. Adding any new features to a filing environment is usually a daunting task which frequently requires reimplementation of much of the file system. This situation generally prohibits all but the major operating system vendors from providing and distributing new filing services, and even then artificially limits the services which will be offered to those which are easily added to the existing product.

The file system switch approaches are an important step in the modular direction, but have yet to be accompanied by a decomposition of monolithic file system services. A few interesting new services (e.g., RAM-based filing) have been constructed, but all are layered onto a fairly standard UFS-type file system base. The new services are therefore constrained by the services and semantics offered by a UFS: access to lower level services within the UFS is not provided, and services must be used even when their richness is unnecessary.

It is also difficult to experiment with new lower level services. For example, Rosenblum's *log-structured* file system [RO91] for the Sprite operating system [OCD88] provides UFS semantics, but uses a very different disk management algorithm from that used by Berkeley's Fast file system [MJL84]. The monolithic

nature of the various UFS implementations limits the immediate wide-spread utility of the log-structured file system (LFS) because porting LFS to other environments (which may have made semantically orthogonal but tightly-integrated changes to their UFS implementations) is a non-trivial task. A similar situation affects the RAID [PGK88] disk-array filing work.

A design technique and accompanying mechanism which provides the ability to snap together independently developed components to configure custom filing services on a site by site (or even file by file) basis would introduce unprecedented flexibility. The result would be a mechanism whereby independent researchers or vendors can deliver shrink-wrapped software modules which contribute file system functionality without rewriting, replacing or retesting large portions of the basic implementation.

2.1.3 Requirements

The above observations provide a strong motivation for stackable layers approach to file system modularity. But what specific requirements must be met by an acceptable stackable solution?

To add a bit of concreteness, suppose the goal is to design a highly available, large scale, wide area file system using a stackable layers methodology. Suppose further that remote access, replication, authentication, and a variety of as yet unknown services are to be supported in addition to traditional single-system UNIX file services.

These goals place a number of requirements on a layering mechanism; some requirements are novel individually, and the collection is unprecedented in scope:

Efficiency: Numerous thin layers embedding few abstractions are preferable to a few thick ones (to maximize the leverage of existing code), so the inherent cost of layering must be low.

Stack composition granularity: Stack contents should be customizable at a process, application, or file granularity; layer services should generally be optional, not forced upon clients. The order of layers should be determined by layer-specific semantics, not fixed *a priori*.

Extensibility: New layers with custom operations should be supported anywhere in a stack (not only the top) without disturbing existing layer implementations. New layers should be introducible at boot-time, irrespective of existing layers; binary-only distribution of new layers should be feasible.

Stacks across address space boundaries: It should be possible to construct a stack that crosses address space and machine boundaries. Graceful reaction to partial stack disintegration resulting from node or communications failures is essential.

Well-defined operations: The extensibility and address space boundary requirements imply that a layer may need to handle arguments for "unknown" operations. These arguments must be self-describing as to type, etc.

Stack fan-in, fan-out: A stack may actually be an acyclic graph, with fan-in and fan-out at any level. Fan-in occurs when several higher layers stack on a common lower layer (as might occur when a file is concurrently accessed by several nodes); fan-out occurs when a layer is stacked upon multiple lower layers (as might be the case when several files stored on distinct nodes are in use by a single client).

Scale: No artificial limits should be placed on the depth (number of layers) of a stack or on the breadth of fan-in or fan-out.

Dynamic composition: Stacks should be customizable on-the-fly, especially when the layers to be added/removed are "invisible" ones, i.e., they are semantics-free value-added layers, such as caching or monitoring layers.

Backwards compatible: Compatibility layers should be constructible which can encapsulate the new mechanism within the constraints presented by older layering services. A lower compatibility layer allows the new mechanism to leverage services implemented with other layering mechanisms; an upper compatibility layer allows older mechanisms to utilize new services, subject to limitations imposed by the intersection of the various layering mechanisms. In particular, existing system call interfaces must not be affected, so that current application software will continue to run unmodified.

2.1.4 Ficus layer mechanism

The Ficus layering mechanism meets most, but not all, of these requirements. It is derived from the SunOS VFS/vnode technique, but is much more flexible and powerful than its predecessor. The mechanism is described in detail by its designer in [HP90, HP91a]; the material that follows summarizes that work.

A Ficus layer is defined to be a set of operations which can be applied to a vnode data structure. A *vnode structure* embodies a layer's concept of a file. It normally contains a list of operations which can be applied to the vnode, and a pointer to a procedure which implements each operation. The private data held by a vnode is layer specific, of course, but usually contains one or more pointers to vnodes from the layer immediately below. Adding a new file system service is primarily a matter of implementing the desired operations for a new layer.

A vnode operation invocation contains a pointer to the vnode on which to operate, the name of the operation, a pointer to the operation's arguments, and a template which defines the type of each argument. The layer mechanism maps the operation name to the proper implementation of the operation for the indicated vnode, and invokes the operation with the supplied arguments and template.

The implementation of a vnode operation may simply forward the operation to one of its descendants (analogous to inheritance), perform some actions and then forward it to its descendants, call some other operations on its descendants, or even handle the operation entirely internally. A layer knows nothing about the type of vnode below it; it simply holds a pointer to it. At the base of the stack is a layer which has no further descendants.

2.1.4.1 Bypass operation

Sometimes the operation to be invoked is not defined, as the layer designer did not provide an implementation for that operation. (Perhaps the designer intends for the implementation to be "inherited" from some layer below; it is also possible that the operation in question had not even been conceived when the layer was built.) In this case, a *bypass* operation is executed to pass the invocation through to the next lower layer. The lowest layer in a stack returns an error if a bypass is attempted.

In general, each layer must define its own bypass operation, since the nature of the bypass may depend on a layer's semantics. Even if unusual semantics are not involved, the pointer to a lower vnode is contained in a vnode's private data area—whose structure is opaque to other layers.

The cost of a bypass operation is typically an additional procedure call which simply passes on the pointers to the arguments with which it was originally called. But when a layer "straddles" an address space boundary, the bypass operation must marshal its arguments and pass them into the other address space. The argument type template defines the arguments in sufficient detail for a bypass operation to correctly transfer arguments from one address space to another.[6]

A transport layer (one that straddles an address space boundary) is actually composed of two sets of operations: those called by the layer above, and those executing in the other address space which in turn call the layer below. For example, if a stack crosses machine boundaries via a network, the upper half of a layer might implement one end of a data transfer protocol, while the lower half implements the other end of the protocol. Figure 2.2 shows a sample stack with a transport layer.

The current Ficus implementation contains a basic layer mechanism, a generic bypass operation, and a generic transport layer pair for crossing address spaces.

[6]The External Data Representation (XDR) service in SunOS is used here; it also supports data movement between heterogeneous data formats, as often occurs when a stack crosses machine boundaries.

The implementation of a vnode operation may simply forward the operation to one of its descendants (analogous to inheritance), perform some actions and then forward it to its descendants, call some other operations on its descendants, or even handle the operation entirely internally. A layer knows nothing about the type of vnode below it; it simply holds a pointer to it. At the base of the stack is a layer which has no further descendants.

2.1.4.1 Bypass operation

Sometimes the operation to be invoked is not defined, as the layer designer did not provide an implementation for that operation. (Perhaps the designer intends for the implementation to be "inherited" from some layer below; it is also possible that the operation in question had not even been conceived when the layer was built.) In this case, a *bypass* operation is executed to pass the invocation through to the next lower layer. The lowest layer in a stack returns an error if a bypass is attempted.

In general, each layer must define its own bypass operation, since the nature of the bypass may depend on a layer's semantics. Even if unusual semantics are not involved, the pointer to a lower vnode is contained in a vnode's private data area—whose structure is opaque to other layers.

The cost of a bypass operation is typically an additional procedure call which simply passes on the pointers to the arguments with which it was originally called. But when a layer "straddles" an address space boundary, the bypass operation must marshal its arguments and pass them into the other address space. The argument type template defines the arguments in sufficient detail for a bypass operation to correctly transfer arguments from one address space to another.[6]

A transport layer (one that straddles an address space boundary) is actually composed of two sets of operations: those called by the layer above, and those executing in the other address space which in turn call the layer below. For example, if a stack crosses machine boundaries via a network, the upper half of a layer might implement one end of a data transfer protocol, while the lower half implements the other end of the protocol. Figure 2.2 shows a sample stack with a transport layer.

The current Ficus implementation contains a basic layer mechanism, a generic bypass operation, and a generic transport layer pair for crossing address spaces.

[6]The External Data Representation (XDR) service in SunOS is used here, it also supports data movement between heterogeneous data formats, as often occurs when a stack crosses machine boundaries.

should be able to incorporate or avoid such features on demand, in reaction to the dynamism inherent in general distributed computing. Rosenthal proposed a *push-down* vnode-based mechanism [Ros90] for dynamic file system stacks in a single address space; its utility is limited for distributed file systems as the mechanism does not easily extend across address spaces.

Dynamic filing stacks are not likely to be as "dynamic" as protocol streams, though, because existing file data was created in the context of a particular stack of semantically significant layers. Operational integrity requires that these layers be included, in the proper order, in later stack construction. But invisible layers, such as caching, or invisible layer pairs providing (de)compression or (de)encryption, should be dynamically stackable.

2.1.5 File system decomposition

On several occasions in the preceding discussion, it has been argued that existing file systems are monolithic (which is bad) when they should be structured as a number of layers which are relatively thin (which is good). A case study of several current file system implementations for UNIX systems supports this argument in more detail.

Consider the typical "layered" file system arrangement in SunOS that would be displayed with UFS, NFS, Rosenblum's LFS, and the Andrew File System (AFS) [HKM88]. Figure 2.3 shows a likely VFS-based arrangement.

Internally, UFS and LFS contain substantial amounts of identical code to implement standard UNIX directory semantics and so on; only the disk layout code is different. On the other hand, UFS and AFS share the same disk management code, but have different directory management code.

Figure 2.4 shows the resulting organization if the file system layers are each split into two layers, in which the basic "inode" abstraction is the dividing line. There are now two semantically equivalent implementations of disk management services, each available for use by two directory managers that provide somewhat different directory services. Note, however, that the UFS and AFS directory services are similar enough that NFS (as well as the basic OS system call services) can be layered above each.

Further decomposition of file system services is both feasible and useful, as will be seen in subsequent discussion of the Ficus replicated filing service in Section 2.3.3.4.

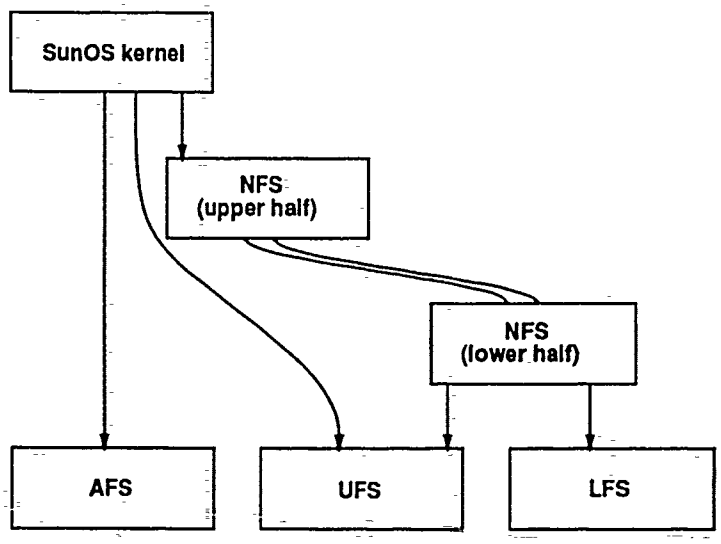


Figure 2.3: File system configuration with thick layers.

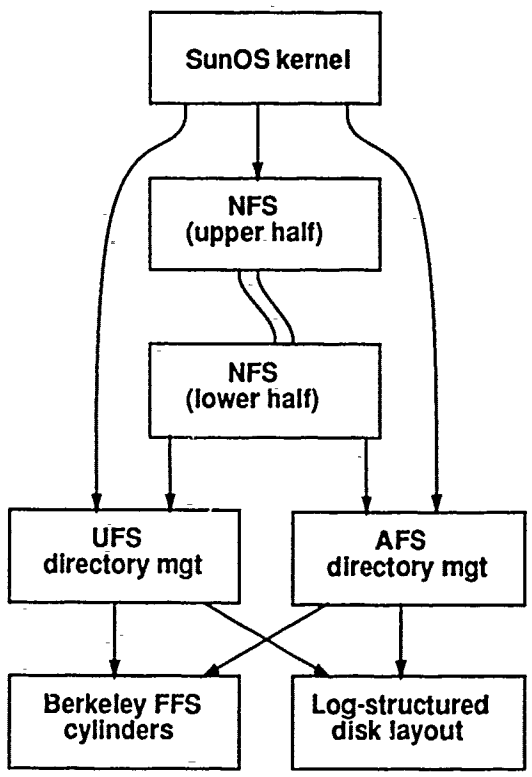


Figure 2.4: File system configuration with thinner layers.

2.1.6 Summary

The stackable layers paradigm is an promising design methodology for filing services. The unprecedented flexibility and extensibility provided by stackable layers can be expected to have a significant impact on experimentation with new file system services, improvements in existing services, and the rapid adoption of novel developments. Their use permeates the Ficus architecture.

2.2 Volumes

Ficus is intended for very large scale distributed computing environments. Such an environment might have 10^6 hosts, each with perhaps 10^5 files. Primary goals of Ficus (see Chapter 1) include name and location transparent access to all Ficus files while retaining a familiar access syntax and semantics.

Accordingly, the Ficus name space topology is a limited, directed acyclic graph of files and directories. To ease management at several levels, the name space hierarchy is divided into disjoint sub-hierarchies called *volumes.* A volume is a singly-rooted, self-contained[7], connected set of files and directories. A volume typically contains from 10^2-10^5 files, yielding an expected total of approximately 10^8 volumes.

Achieving name and location transparency at this scale implies that every host can name, locate, and access any of 10^{11} files scattered amongst 10^8 volumes on 10^6 hosts. This reduces to naming and locating volumes when (as in Ficus) volumes contain a well-defined subset of files.

Solution to this problem are very much constrained by the number of volumes in the name hierarchy, the number of replicas of volumes, the topology and failure characteristics of the communications network, the frequency or ease with which volume storage locations change, and the degree to which the hierarchy of volumes spans multiple administrative domains.

2.2.1 Related solutions

Most volume naming mechanisms are descended from the original UNIX *mounted filesystem* concept. In this model, a path name is expanded component by component within a filesystem (volume) until a specially designated directory is encountered. The special designation indicates that path name expansion should continue in the root directory of another volume, which is said to be *mounted* at

[7]Directory references do not cross volume boundaries.

that point in the hierarchy.

The traditional UNIX mounted filesystem mechanism has been widely altered or replaced to support both small and large scale distributed file systems. Examples of the former are Sun's Network File System (NFS) [SGK85] and IBM's TCF [PW85]; larger scale file systems are exemplified by AFS [Kaz88], Decorum [KLA90], Coda [SKK90], and Ficus [GHM90, PGP91]).

In a conventional single-host UNIX system, a single mount table exists which contains the mappings between the mounted-on directories and the roots of mounted volumes. However, in a distributed file system, the equivalent of the mount table must be a distributed data structure. The distributed mount table information must be replicated for reliability, and the replicas kept consistent in the face of update.

Most distributed UNIX file systems to some degree attempt to provide the same view of the name space from any site. Such *name transparency* requires mechanisms to ensure the coherence of the distributed and replicated name translation database. NFS, TCF, and AFS each employ quite different approaches to this problem.

To the degree that NFS achieves name transparency, it does so through convention and the out-of-band coordination by system administrators. Each site must explicitly mount every volume which is to be accessible from that site; NFS does not traverse mount points in remotely mounted volumes. If one administrator decides to mount a volume at a different place in the name tree, this information is not automatically propagated to other sites which also mount the volume. While allowing sites some autonomy in how they configure their name tree is viewed as a feature by some, it leads to frequent violations of name transparency which in turn significantly complicates the users' view of the distributed file system and limits the ability of users and programs to move between sites. Further, as a distributed file system scales across distinct administrative domains, the prospect of maintaining global agreement by convention becomes impossible.

IBM's TCF, like its predecessor Locus [PW85], achieves transparency by renegotiating a common view of the mount table among all sites in a partition every time the node topology (partition membership) changes. This design achieves a very high degree of network transparency in limited scale local area networks where topology change is relatively rare. However, for a network the size of the Internet, a mount table containing several volumes for each site in the network results in an unmanageably large data structure on each site. Further, in a nationwide environment, the topology is constantly in a state of flux; no algorithm which must renegotiate global agreements upon each partition membership

change may be considered. Clearly neither of the above approaches scales beyond a few tens of sites.

Cellular AFS [ZE88] (like Ficus) is designed for larger scale application. AFS employs a *Volume Location Data Base* (VLDB) for each cell (local cluster) which is replicated on the cell's backbone servers. The mount point itself contains the cell and volume identifiers. The volume identifier is used as a key to locate the volume in a copy of the VLDB within the indicated cell. Volume location information, once obtained, is cached by each site. The VLDB is managed separately from the file system using its own replication and consistency mechanism. A primary copy of the VLDB on the *system control machine* periodically polls the other replicas to pull over any updates, compute a new VLDB for the cell, and redistribute it to the replicas.

The Cellular AFS design does not permit volumes to move across cell boundaries, and does not provide location transparency across cells, as each cell's management may mount remote cell volumes anywhere in the namespace. This may be viewed as a feature or a limitation depending on where one stands on the tradeoff between cell autonomy and global transparency.

The AFS design also explicitly assumes a slowly changing VLDB. This is not likely to be the case for very large scale environments. Suppose that the number of volumes is static and that a volume is likely to be relocated to another storage device or host once a year. With 10^8 volumes, some volume can be expected to move every $\frac{1}{4}$ second. Although the rate of change for a single volume is very low, the aggregate rate of change is quite high—too rapid to allow volume location data to be distributed throughout the network in a timely fashion.

2.2.2 Ficus solution overview

Ficus uses AFS-style *on disk mounts*, and (unlike NFS) readily traverses remote mount points. The difference between the Ficus and AFS methods lies in the nature of Ficus volumes (which are replicated) and the relationship of Ficus graft points and volume location databases.

In Ficus, like AFS [HKM88], a volume is a collection of files which are managed together and which form a subtree of the name space[8]. Each logical volume

[8]Whereas a filesystem in UNIX is traditionally one-to-one with a disk partition, a volume is a logical grouping of files which says nothing about how they are mapped to disk partitions. Volumes are generally finer granularity than filesystems, it may be convenient to think of several volumes within one filesystem (say one volume for each user's home directory and sub-tree) though the actual mapping of volumes to disk partitions is a lower level issue.

in Ficus is represented by a set of volume replicas which form a maximal, but extensible, collection of containers for file replicas. Files (and directories) within a logical volume are replicated in one or more of the volume replicas.[9] Each individual volume replica is normally stored entirely within one UNIX disk partition.

Ficus and AFS differ in how volume location information is made highly available. Instead of employing large, monolithic mount tables on each site, Ficus fragments the information needed to locate volumes and places the data for an individual volume in a *graft point* (the mounted-on directory).[10]

2.2.3 Graft points

A graft point (see Figure 2.5) is a special file type used to indicate that a (specific) volume is to be transparently grafted at this point in the name space. Grafting is similar to UNIX filesystem mounting, but with a number of important differences.

A graft point maps a set of volume replicas to hosts, which in turn each maintain a private table mapping volume replicas to specific storage devices. Thus the various pieces of information required to locate and access a volume replica are stored where they will be accessible exactly where and when they will be needed.

A graft point contains a unique volume identifier and a list of volume replica and storage site address pairs. Therefore, a one-to-many mapping exists between a graft point replica and the volume replicas which can be grafted on it. Each graft point replica may have many volume replicas grafted at a time. The particular volume to be grafted onto a graft point is fixed when the graft point is created, although the number and placement of volume replicas may be dynamically changed.

A graft point may be replicated and manipulated just like any other object (file or directory) in a volume. It can be renamed or given multiple names; it can be a replicated object itself, with replication parameters independent of the referenced volume. Since a graft point resides in a "parent" volume, although referring to another volume, the graft point is subject to the replication constraints of the parent volume. There is no requirement that the replication factor (how many replicas and their location in the network) of a graft point match, or even overlap,

[9]Each volume replica must store a replica of the root node; storage of all other file and directory replicas is optional.

[10]In the sequel, the term "graft" and "graft point" is used for the Ficus notion of grafting volumes while the mount terminology is retained exclusively for the UNIX notion of mounting filesystems.

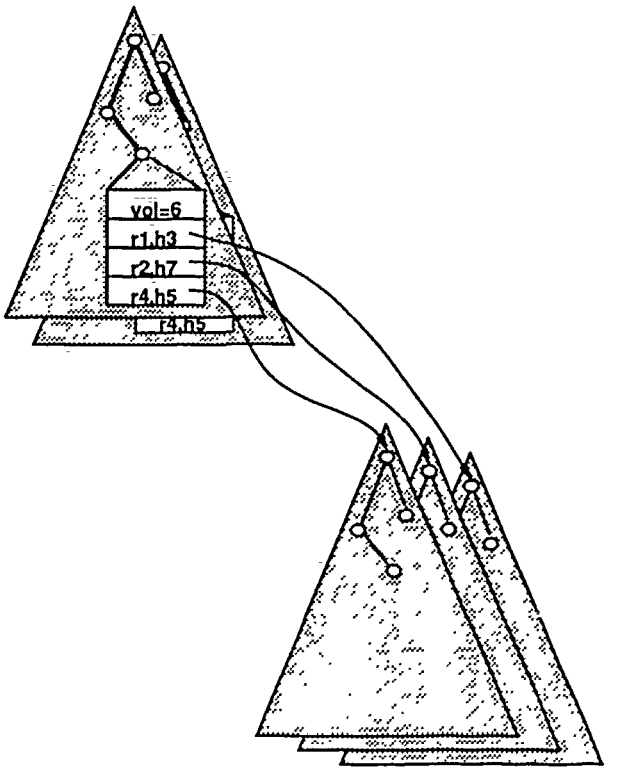


Figure 2.5: Graft point with both parent and child volumes replicated.

that of the child volume.

As it happens, the format of a graft point is compatible with that of a directory: a single bit indicates that it contains grafting information and not file name bindings. The syntactic and semantic similarity between graft points and normal file directories allows the use of the same optimistic replication and reconciliation mechanism that manages directory updates. (See Section 2.3 and Chapter 3 for details of these mechanisms.) Without building any additional mechanism, graft point updates are propagated to accessible replicas, conflicting updates are detected and automatically repaired where possible, and reported to the system administrators otherwise.

Volume replicas may be moved, created, or deleted, so long as the target volume replica and any replica of the graft point are accessible in the partition (one copy availability). This optimistic approach to replica management is critical as one of the primary motivations for adding a new volume replica may be that network partition has left only one replica still accessible, and greater reliability is desired.

This approach to managing volume location information scales to arbitrarily large networks, with no constraints on the number of volumes, volume replicas, changes in volume replication factors, or network topology and connectivity considerations.

2.2.4 Volume and file identifiers

When a file is created, it is given a globally unique, static identifier that is carried by the file (its replicas) throughout its existence. A Ficus file identifier has several components, but at its highest level of abstraction, it is a tuple ⟨*volume-id, file-id*⟩. The *volume-id* component is a globally unique identifier for the volume, while the *file-id* component is unique within that volume.

Partial network operation should not hinder a host's ability to create new volumes, so each host must be able to issue new *volume-ids* on its own. Prior to system installation, each Ficus host is issued a unique value as its *allocator-id* which the host can use in conjunction with a non-decreasing counter to issue globally unique *volume-ids.* A *volume-id* is, therefore, a tuple ⟨*allocator-id, counter-value*⟩.

Allocator-ids are issued by a central (possibly offline) service. Ficus *allocator-ids* contain space for two fields the size of an Internet host address. In most cases, one field will contain existing Internet host addresses; the other field is present to allow easy integration of existing host identifiers from other networks.

A completely detailed *volume-id* is a 3-tuple ⟨*internet-id, intranet-id, counter-value*⟩.

Individual volume replicas are further identified by a *replica-id*, so a volume replica is globally uniquely identified by the couplet ⟨*volume-id, replica-id*⟩.

A host not only has the autonomy to create new volumes at any time, it can also spawn a new volume replica, irrespective of which host established the volume originally. Each volume replica is issued a contiguous range of *replica-ids* that is disjoint from ranges issued to all other volume replicas. A new volume replica can be created from any existing volume replica possessing a non-empty range of *replica-ids*. The new volume replica's *replica-id* is taken from either the top or bottom of the existing replica's range; the new replica's range is further drawn contiguously from the older replica's range.

Within the context of a particular volume, a logical file is uniquely identified by a *file-id*. A particular file replica is then identified by appending the *replica-id* of the containing volume replica to the *file-id*, as in ⟨*file-id, replica-id*⟩. A fully specified identifier for a file replica is ⟨*volume-id, file-id, replica-id*⟩; this identifier is unique across *all* Ficus hosts in existence.

Each volume replica assigns file identifiers to new files independently. To ensure that *file-ids* are uniquely issued, a *file-id* is prefixed with the issuing volume replica's *replica-id*. A *file-id* is actually, therefore, a tuple ⟨*replica-id, unique-id*⟩.

A total of six components constitute a complete file replica identifier: ⟨*internet-it, intranet-id, counter-value, replica-id, unique-id, replica-id*⟩. The first three components constitute the *volume-id*, the fourth and fifth form the *file-id*, and the sixth identifies a particular replica. Each component is a 32-bit field, which should allow effectively unlimited growth at every level.

The values contained in any file identifier field place no constraints on the actual physical location of any file or volume replica in the network; they merely serve to uniquely identify a file or volume. There is no requirement that a host ever store a volume for which it issued a *volume-id*; nor is it necessary for a volume replica to store a replica of a file for which it provides a *file-id*.

2.2.5 Autografting

In a very large scale distributed file system, there may be millions of volumes to which one might desire transparent access. However, any one machine will only ever access a very small percentage of the available volumes. Hence it is prudent to locate and graft volumes on demand, rather than *a priori*.

If the volume in which the graft point resides is itself a replicated volume, the graft point containing the volume replica location information may also be replicated. If each parent volume replica which stores the directory in which the graft point occurs also stores a copy of the graft point, the location information is always available whenever the volume is nameable. There is very little benefit to replicating the graft point anywhere else and considerable loss if it is replicated any less.

In the course of expanding a path name, a directory is first checked to see if it is actually a graft-point. If so, and a volume replica is already grafted, pathname expansion simply continues in that volume replica's root directory. More than one replica of the grafted volume may be grafted simultaneously, but if no grafted replica is found, the system must autograft one of the volume's replicas onto the graft point.

A graft point is a table which maps volume replicas to their storage site. The sequence of records in a graft point table is in the same format as a standard directory and hence may be read with the same operations used to access directories. Each entry in a graft point is a triple of the form (*volume-id, replica-id, hostname*), identifying one replica of the volume to be grafted. The *volume-id* is a globally unique identifier for the volume. The *replica-id* identifies the specific volume replica to which the entry refers. The *hostname* identifies the host which is believed to house the volume replica.[11] The system then uses this information to select one or more of these replicas to graft. If the grafted volume replica is later found not to store a replica of a particular file, the system can return to this point and graft additional volume replicas as needed.

In order to autograft a volume replica, the system calls an application-level graft daemon on its site. Each site is responsible for mapping from volume and replica identifiers to the underlying storage device providing storage for that volume. If the *hostname* is local, the graft daemon looks in the file `/etc/voltab` for the location of the underlying volume to graft. If the *hostname* is remote, the graft daemon obtains a file handle for the remote volume by contacting the remote graft daemon (similar to an NFS mount; see [SGK85]) and completes the graft.

The system caches the pointer to the root directory of a grafted volume replica so that the graft point does not have to be fully reinterpreted each time it is traversed. (The cache supports pointers to multiple replicas for a single volume.) The graft of a volume replica which is not accessed for some time is automatically

[11] Currently *hostname* is an Internet host address, it could equally well be a Domain Naming System identifier or one from any other host naming or addressing mechanism.

pruned so it does not continue to consume resources.

2.2.6 Creating, deleting and modifying graft points

The system must support creation, deletion, moving and updating of graft points. Graft points are modified whenever the named volume replica is created, destroyed, or moved from one host to another. Moving a graft point is equivalent to creating a copy of it in a different place in the name hierarchy, and deleting the original.

While updating a graft point is a relatively rare event, when it does occur, it is generally important. Hence it is not reasonable to require that all, or even a majority of the replicas of the graft point be accessible. Further, the motivation for updating a graft point may be at its greatest precisely when the system is unstable or partitioned. Perhaps the whole reason for updating the graft point is to add an additional replica of a volume for which, due to partitions or host failures, only a single replica remains accessible; this update must be permitted, even though it cannot immediately propagate to all replicas of the graft point.

Hence, for exactly the same reason that Ficus utilizes an optimistic philosophy for maintaining the consistency of files and directories, the same philosophy must be applied to graft points. Fortunately, this is very easy to achieve in Ficus since a graft point has the same format and, as a sequence of records, very similar semantics to a directory.

2.2.7 Conflicting graft point updates

As with directories, the semantics of graft point updates are quite simple, and hence most updates which would have to be considered conflicting if viewed from a purely syntactic point of view may be automatically merged. For example, if in non-communicating partitions, two new replicas of the same volume are created, the two resulting graft point replicas will each have an entry that the other does not have. However, it is clear that the correct merged graft point should contain both entries, and this is what will occur.

Ficus uses the automatic reconciliation mechanism already in place for regular file and directory update management to manage graft point updates. A detailed description of these techniques is contained in Section 2.3 and Chapter 3; the following discussion is therefore free of most algorithmic detail.

Part of a graft point entry is mapped into the name field of a directory entry, and the remainder is placed in the file named by that entry. The couplet ⟨*volume-*

id, repl-id) is encoded as printable ASCII characters in the name field of the directory entry, while the *hostname* component constitutes the contents of the named file.

When a new child volume replica is created, an entry for the new replica (consisting of a new file and associated directory entry) is placed in one graft point replica. The Ficus file and directory automatic update propagation service sees to it that the new graft point data is propagated to all other graft point replicas.

When a volume replica is moved from one host to another, the file contents (at one graft point replica) are simply updated to reflect the new *hostname*. Should a graft point entry be concurrently updated in two graft point replicas, the normal file update conflict detection mechanism will notice that state of affairs and flag each replica in conflict. Similarly, if a graft point entry is updated in one graft point replica (to indicate that the child volume replica has moved to a new host), and concurrently deleted in another graft point replica (to indicate that access to the child volume replica is no longer possible for some reason), the regular file remove/update conflict detection mechanisms detect the conflict. Standard (manual) conflict resolution tools must then be used to resolve the conflict. In the meantime, access to the file data (i.e., the *hostname*) will be blocked as usual for conflicted files.

If a volume replica is destroyed, the graft point entry is eliminated by simply removing the directory entry for that particular volume replica in one graft point replica. Update propagation and directory reconciliation ensures that the other graft point replicas also receive notification of the change.

In general, graft point data is self-validating upon use: if it is wrong in some way (perhaps the replica has been destroyed or moved to a new host) the queried host will respond negatively to a graft request, and the autograft mechanism will try some other replica.

2.2.8 Volume summary

The Ficus volume design is the foundation for a very large scale file system name service. It supports a very large number of volumes and volume replicas, in a flexible manner. Autonomous control of volume creation, placement, and destruction is inherent, as are location transparency and name transparency.

2.3 Replication

A primary goal of the Ficus project is to provide a large scale replication service that can be readily used by a sizable client community. Such a service must, therefore, be easy to install and administer.

Easy installation implies that the service be portable to a number of operating system platforms; that existing file systems need not be converted to a new format; that existing network protocols continue to operate without interference; and that minimal changes be made to existing operating system kernels, so as not to disturb others' customizations and not unduly erode clients' confidence in their kernel's integrity.

Easy administration implies that the mechanism must be amenable to multiple bureaucratic domains; that minimal cooperation with centralized authority be required; that it be easy to integrate new hosts (including entire subnets); and that replication management be both powerful and flexible in the face of partial operation.

A second goal is that existing services be used whenever feasible. This allows one to concentrate on specific issues, rather than on peripheral areas in which contributions are not anticipated. Even when a service is not "just right," the resulting leverage often enables a prototype service to be rapidly constructed; finely tuned services can later be constructed in the context of more complete (e.g., empirical) knowledge.

The Ficus replicated file service design embodies these goals. Replication is a value-added service which can be "bolted on" to existing kernels. The design presented here also reflects a high degree of leveraged services; a number of "fine tuning" ideas drawn from this experience are described in Chapter 4. Ficus replication is a case study in providing a key piece of extended filing services using the stackable layers architecture. In particular, the replication service is largely independent of the underlying file system implementation, permitting a high degree of configuration flexibility and portability.

2.3.1 Replication overview

The initial Ficus design specified that existing services should be leveraged in their pristine form. It was clear from the outset that a suitable replication service would incorporate a persistent storage service (e.g., UFS), a network data transport service (e.g., NFS), and a stackable interface (e.g., VFS/vnode). SunOS provides all three, so it was a natural platform on which to begin.

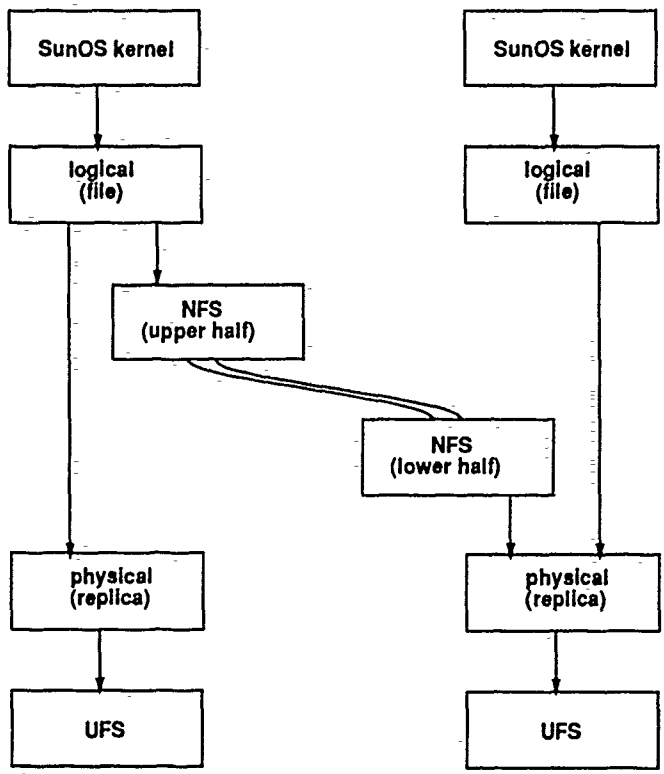


Figure 2.6: Typical Ficus layer stack.

The early prototype of implementation of the replication layers was largely successful in regards to leveraging, but demanded an ever-increasing investment in techniques to compensate for features not anticipated by the UFS, NFS, and VFS designers. In response to this problem, the design and implementation of Ficus has moved steadily towards the development of services better attuned to replication in particular, and stackable layers in general. The design described here is an enhanced version of an earlier one; it reflects the experience gained with layering and leveraging. Current implementation status is indicated as the discussion proceeds.

The Ficus file replication service is packaged as a pair of stackable layers, each building upon the abstractions provided by lower layers. Figure 2.6 shows a typical Ficus layer stack.

The *logical* layer provides to its clients (i.e., layers above it) the abstraction of a single-copy, highly available file. The *physical* layer implements the concept of a file replica. Underneath the physical layer is a *persistent storage* layer with

traditional UNIX filing service semantics. When the logical and physical layers reside on different hosts or otherwise execute in different address spaces, a *vnode transport* layer is inserted between the logical and physical layers.

The Ficus replication layers also support a file system name service intended for use in a very large scale (nationwide) distributed system. Ficus builds upon the volume mechanism described in Section 2.2, NFS-style pathname resolution, and optimistic replication techniques to provide transparent access throughout the overall name space.

2.3.2 Logical layer

The primary function of the Ficus logical layer is to provide the illusion that each file is highly available with single-copy semantics, when in reality a file may be physically represented by multiple replicas whose individual availability is not optimal. The illusion is the product of several optimistic consistency mechanisms described in the sequel; a mechanism to support serializable concurrency control is presented in Section 2.4. The optimistic mechanisms include replica selection, update propagation, and reconciliation.

Optimistic replication, like most other approaches to replication, must often choose which version of a file to use to service a file access request. Once a version decision has been made, file access performance differences may guide the final replica selection.[12]

Optimistic replication has a greater range of version choices than conventional replication mechanisms. Optimistic concurrency control and lazy update propagation yield a richer set of versions, including the possibility of conflicting versions. The volatility and scale of a large geographically distributed environment can make it infeasible even to determine the range of accessible versions. A further problem is that the appropriate version selection policy may well be client, application, instance, or data specific.

The Ficus approach is to provide a default base level policy which will often be adequate, but can also serve as the foundation for policies with different requirements. The synchronization mechanism in Section 2.4 is stacked upon the logical layer, and exploits its replica selection services. Further, a transaction layer can be constructed and stacked above the synchronization layers to provide

[12]One can imagine circumstances in which performance differences are so great as to make version issues a secondary issue. "Nearer before newer" may reasonably apply to utilities, for example, if the choice is between access to a local disk and access via voice-grade serial connection to a remote host.

full transaction semantics.

2.3.2.1 Replica selection

Three issues determine which, if any, of accessible replicas appropriately serves the client: consistency policy, cost of replica access, and cost of the selection process itself.

Replica selection primarily occurs at file open, that is, when the logical layer performs a `lookup` operation for a client. If the client specifies a particular replica, file version, or minimum file version, the logical layer will strive to locate a qualifying replica. An error is returned if no appropriate replica can be accessed.[13]

If a particular replica or version is not specified, the logical layer consults a version cache indicating the greatest version of each file opened by the layer.[14] The cache value (if present) is used as an advisory minimum value: if no compatible replica is accessible, an accessible replica will be selected.

Except when a particular replica is specified, replica selection must decide in which order to consult replicas for compatibility and eventual service to clients. The most important factor is nearness (cost of access), but transparency at several levels in Ficus makes it difficult to distinguish degrees of nearness or cost. It is relatively easy, however, to determine if a physical layer is on the same host as the logical, so a crude distinction between local and not-so-local can be made. Ficus exercises a preference for local replicas.

After nearness, Ficus currently uses a random order to consult replicas. Replica selection ceases with discovery of a compatible replica, even though some other (unconsulted) replica with a greater version may exist and be accessible.

Composing a list of file replicas to consult during replica selection is a bit complex because Ficus supports dynamic, selective replication of volumes and files. A volume may be replicated any number of times, and stored by arbitrary hosts. The number and placement of volume replicas is dynamic. Similarly, the number of file replicas is dynamic within the constraints of the replication parameters of the volume that contains the file. (More details may be found in Sections 2.2 and 2.3.3.)

The physical layer ensures that a (minimally skeletal) file replica can always be accessed for a nameable file. A skeletal replica contains a *replica list*, which replica selection uses as a starting point. (Because replica placement is dynamic,

[13]Replica selection tries to access a replica no more than once per replica, per selection.

[14]The cache is volatile, large, but finite sized.

the list followed by replica selection may change as further replicas are consulted and their replica lists and version examined.)

The logical layer ensures that a client continues to be served by the same replica unless it becomes inaccessible. In that case, Ficus will attempt to substitute a compatible replica,[15] that is, one which has a version greater than or equal to the greatest version of the original replica that is known to the logical layer.

No means exists for a logical layer to "lock" a physical replica, so a logical layer never knows with certainty the "current" version of a replica, as other logical layers might be updating the replica. A logical layer can, however, monitor the version as seen by the client: meta-data returned from the physical layer with each `read` and `write` operation contains the replica version resulting from the call.

The absence of locking implies that a logical layer cannot guarantee to provide continuous access to a particular version even in the absence of failure. It is, however, possible to provide a warning when a particular version is no longer accessible (or may not even exist anymore).

The current Ficus implementation does not contain the version cache described above, but does support access to a specified replica or version. It randomly selects a remote replica if a local is inappropriate.

2.3.2.2 Update notification and propagation

The optimistic consistency philosophy allows considerable flexibility in many aspects of replicated file management. In addition to the richer choices encountered in replica selection, more options are possible when promoting consistency among replicas. Ficus uses three asynchronous daemons in an optimistic manner to notify replicas of updates, to propagate updates, and to ensure that eventual mutual consistency is attained.

The flow of control for file update notification and propagation is displayed in Figure 2.7. In Ficus, a file update is applied immediately to only one replica. When a `write` operation is received by the logical layer, it is forwarded to the physical layer to be applied to the replica selected. After having been successfully applied to one replica, the logical layer may then notify other replicas of the update. The logical layer instance that handled the update places a summary of the update on an outgoing *update notification queue*, and then returns control to the client.

[15] If a particular replica was specified in `lookup`, substitution is not attempted.

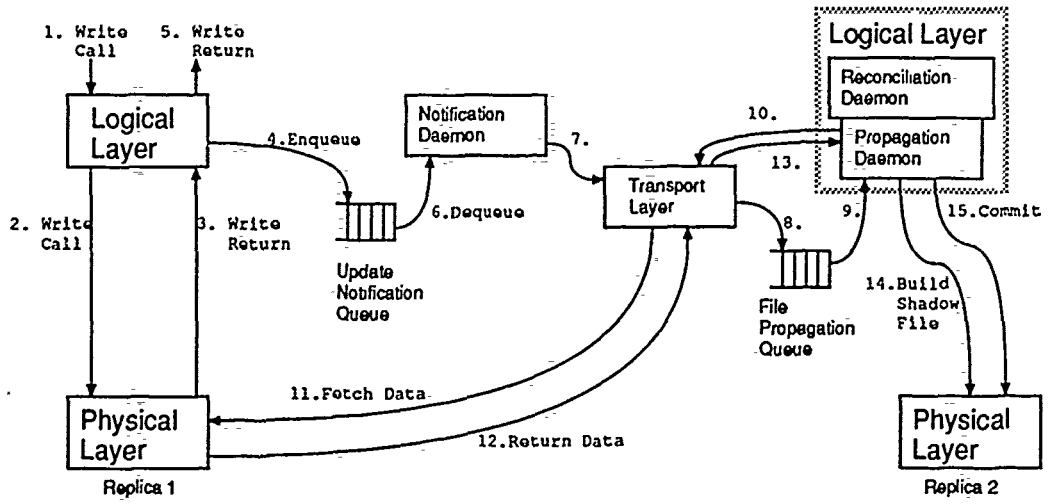


Figure 2.7: File update notification and propagation.

An *update notification daemon* periodically wakes up and services the queue, sending out notification via multicast datagram to all replica storage hosts that a new file version exists. Notification is a best-effort, one-shot attempt; inaccessible replicas are not guaranteed to receive an update notification later.

Ficus' reliance upon optimism releases it from the burden of ensuring that an update notification message is successfully delivered and processed by the receiver. If the receiver fails to update its replica for whatever reason, it is assured that it will eventually learn of the update via the reconciliation daemon running on its behalf. (See discussion below.)

An update notification is not necessarily placed on the queue as part of every update. If the logical layer has received an open operation, it may delay placing a notification in the queue until a `close` operation is received. If for some reason a close operation never arrives, an update notification might not be sent, which is analogous to a lost update notification message—perfectly acceptable by the optimistic philosophy, since reconciliation will find out about it later.[16]

The notification message contains the version vector (see Section 2.3.3) of the new file version, and a hint about the site which stores that version. It is then the responsibility of the individual replicas to pull over the update from a more

[16]A logical layer that is servicing a remote client via NFS (see Section 2.3.4.3) will never receive an open or `close` operation, so it may choose to issue an update notification for every write operation. With optimism, this is all merely an optimization.

up-to-date site.

When a replica receives an update notification, it places that notification on a queue. An *update propagation daemon* wakes up occasionally to service the queue. The notification message sometimes contains complete information about the update (as is the case for directory updates), and if so the daemon directly applies the update to its replica. If the notification only contains a summary of the update (typical for file updates), the replica which sent the notification must be queried for the new data.

Update propagation is performed atomically. Ficus contains an atomic commit mechanism used primarily by propagation to ensure that a replica's version vector properly reflects the replica's data. A shadow replica is constructed containing the new version, which is then substituted for the original replica.

No locking occurs during the construction of the shadow replica. The commit mechanism verifies that the shadow replica does not conflict with the original replica as part of the commit. It also verifies that the remote replica from which the shadow has been constructed has not changed during that time. If any changes have occurred, optimism allows update propagation to start over or abort the propagation.

2.3.2.3 Reconciliation

The reconciliation daemon shoulders the responsibility of ensuring eventual mutual consistency between replicas. For each replica housed by a node, the reconciliation daemon directly or indirectly checks every other replica to see if a new (possibly conflicting) version of the file exists. When a new version is discovered, update propagation is initiated and follows the sequence of steps outlined above.

When reconciliation discovers a remote replica in conflict with its local replica, a conflict mark is placed on the local replica. A conflict mark blocks normal access until the conflict is resolved, at which point the mark is removed. Access to marked files is permitted via a special syntax.

Ficus exploits the well-known semantics of directory files to automatically reconcile directory modifications. The details of directory reconciliation are deferred to Section 2.3.3.2, where the physical layer design provides additional context.

Imperfect communication affects how the reconciliation daemon undertakes its tasks at two levels: the order in which local replicas are inspected, and the order and timing of contact with other replicas' nodes.

For every locally housed volume, the reconciliation daemon must ensure that

every local replica is reconciled within finite time, directly or indirectly, with every other replica of that file. Since a volume has a natural tree-like structure at one level, and a linear structure internally at a lower level, the "obvious" approach for reconciliation is to reconcile files in one or another of the convenient orders. Strict adherence to an order, however, is not robust to communication failure.

If communications between two nodes is likely to be interrupted at intervals less than the length of time required by the reconciliation daemon to scan through its local replicas and query the remote node, a fixed starting point must be avoided so that files at the far end of the order are not victimized by starvation. Further, reconciliation must not be hamstrung propagating a large file that has been updated, but whose pages cannot be transferred during an operational interval. Reconciliation must either reconcile files in a random order, choose a random restarting point each time, or be able to skip over problematic files.

The overall cost of reconciliation among a set of replicas is determined in part by the inter-node pattern in which reconciliation occurs. If each node directly contacts every other node, a quadratic (in the number of nodes) message complexity results, but if indirect contact (through intermediate nodes) is used in an optimal fashion, a small-coefficient linear complexity can be achieved. The interesting problem here is to exploit indirect reconciliation when inter-node communication is in excellent condition (to avoid quadratic complexity costs), but gracefully handle degraded communications when the degradation follows no predictable pattern and may be quite volatile. The Ficus solution uses a two-node protocol that tends to structure indirect communications in a ring topology when communications links permit, and automatically adjusts to a dynamic tree topology in response to changes in the communications service.

The current Ficus reconciliation implementation is somewhat simpler than the above design. It performs a breadth-first walk of a volume replica (always beginning with the root directory), communicating with a single other replica per walk. Reconciliation also steps through the volume replica list in a round-robin fashion, without considering the currency of results from recent reconciliations with other replicas.

2.3.3 Physical layer

The physical layer performs two main functions: it supports the concept of a file replica and it implements the basic naming hierarchy adopted by Ficus. The persistent storage layer underneath it provides basic file storage services.

The structure of the current Ficus physical layer reflects an early design de-

cision to use a standard UFS as the initial persistent storage layer, as well as beginner's inexperience with respect to the power, ease, and low cost of layering. From a strictly functional perspective, the "ideal" physical and persistent storage layer design is somewhat different, as described later in Section 2.3.3.4.

2.3.3.1 Replica abstraction

Recall from Section 2.2.4 that every Ficus volume and file replica has its own unchanging globally unique identifier. Both logical and physical layers use these identifiers to unambiguously refer to a file or volume.

Every file replica also possesses two attributes, a *replica list* and *version vector*, in addition to standard UFS file attributes. The replica list is a vector of volume replica identifiers (*repl-ids*) that indicates which volume replicas maintain (or have in the past maintained) a replica of the file. Some volume replicas store a complete file replica; others store a skeletal replica that contains only extended attributes, but no client data; and yet other volume replicas may not store a file replica at all.

The replica list serves as an index into the version vector [PPR83]. A version vector is a multidimensional version number in which each component corresponds to a particular file replica. It compactly encodes the partially ordered update history of a logical file. Version vectors are used to accurately detect concurrent file update, so that *no lost updates* semantics are preserved.

2.3.3.2 Directories

The Ficus name space is quite similar to its UNIX counterpart. It uses special directory files to contain the information that represents the hierarchical name space, although the format has been extended. The main function of a directory is to map between (mutable) file names and (immutable) file identifiers. In Ficus a directory entry is a mapping from a file name to a *file-id*. The *volume-id* is implicit from the containing volume (see Section 2.2.4). Although a *file-id* is the fundamental file identifier for Ficus, it must be further mapped into a syntax and structure compatible with the interface semantics of the persistent storage layer—currently a standard UFS-style file service.

A Ficus directory entry is a tuple ⟨*name, file-id, conflict, deleted, bitvector*⟩. The *conflict*, *deleted*, and *bitvector* fields are used in the automatic reconciliation of directory files. Directory update semantics are well understood, so it is reasonable to automatically reconcile directory updates, including those that occur

concurrently at distinct replicas [Guy87].

In Ficus a directory update is either an `insert` or `delete` of a directory entry; `rename` is `insert` followed by `delete`. Concurrent directory entry insertion may result in more than one entry with identical name fields. Ficus tolerates this violation of directory semantics (all names are unambiguous within a single directory) by setting the *conflict* flag on entries with ambiguous names. This flag blocks normal name resolution pending client resolution of the conflict; it can be ignored by request.

The primary issue faced by the directory reconciliation mechanism is ascertaining which entries have been inserted into the directory replicas, and which have been deleted. To solve this problem, known as the *insert/delete ambiguity* [FM82], Ficus incorporates a logical deletion flag (the *delete* field) and a two-phase algorithm to garbage detect logically deleted entries. The algorithm, fully described in Chapter 3, uses the *bitvector* field to record its progress.

When a file is concurrently renamed, several new names result: each `rename` instance inserts a new entry, and marks the old entry logically deleted. For normal files, no conflict results even though the old entry may be "deleted" more than once. (Repeated entry deletion is not possible in UNIX) A conflict does emerge for concurrent directory rename, however.

Like UNIX, every Ficus directory contains a special entry (`..`) that references its parent directory. A directory rename, of course, must cause a `..` entry to refer to the proper parent directory. Ficus accomplishes this task by deleting the old `..` entry and inserting a new one with the correct referent. Concurrent directory rename results in a directory with multiple parents (one for each `rename` instance), and multiple ambiguous `..` entries in the directory itself, each referring to the respective parent directory. The `..` entries are marked in conflict, as usual, so backward pathname resolution will fail with a name conflict error until the conflict is resolved by removing all but one of the new directory names. Forward resolution is not affected by ambiguous `..` entries.

2.3.3.3 Extended UFS abstractions

The Ficus directory structure does not map cleanly onto a standard UNIX file system due to the additional directory entry fields used in reconciliation and the possibility of multiple directory names at one time. Ficus also requires a few additional file attributes, such as the replication list and version vector, which do not comfortably fit within the space normally reserved for attributes (the UFS *inode*). Yet, an early design goal to stack upon an unmodified UFS remains

important.

The additional richness inherent in the Ficus model led to the implementation of a full name resolution mechanism in the physical layer with the slight additional capabilities that were needed, as well as a simple file attribute service similar in spirit to the Apple Macintosh operating system *resource fork* [App85]. The UFS layer is relegated to providing a simple file service with an almost flat name space.

Ficus directories map from file names to Ficus file identifiers, which must be further mapped to UFS file names. Although Ficus file identifiers form a large flat name space which has a trivial UFS-compatible translation, the standard UFS name service is not well-suited for efficient flat name service. To overcome this efficiency problem, Ficus identifiers are mapped into a two-level UFS hierarchy which is carefully constructed to minimize linear searching and exploit expected file access locality patterns [Flo86b, Flo86a]. Extended file attributes are clustered in auxiliary files to similarly benefit from locality.

2.3.3.4 Decomposed physical and storage layers

The current physical layer is somewhat monolithic and largely duplicates (while enhancing) services provided by the UFS persistent storage layer. A cleaner design would separate the UFS and physical layers into several additional layers, each providing a narrower set of services. Replication support can then take several forms: existing UFS storage services can be used as at present, when compatibility is a premium; or, a (new) persistent storage service with a richer file model but flat name space could be used when efficiency is very important. One might also construct a UFS-style layer for placement on top of the new persistent storage layer to provide a migration path away from the old monolithic UFS. Figure 2.8 shows an exploded, layered design for the physical layer and a set of persistent storage layers.

2.3.4 Vnode transport layer

When Ficus autografts a remote volume replica, it must construct a layer stack using a vnode transport layer to cross the address space boundary between the logical and physical layers. (Refer back to Figure 2.6 on page 40.)

The transport layer is essentially a remote vnode operation service which maps calls from the layer above to operations on a vnode at a layer below, normally crossing an address space boundary (and often a machine boundary) in the process. The ideal transport layer consists solely of a bypass operation, with no

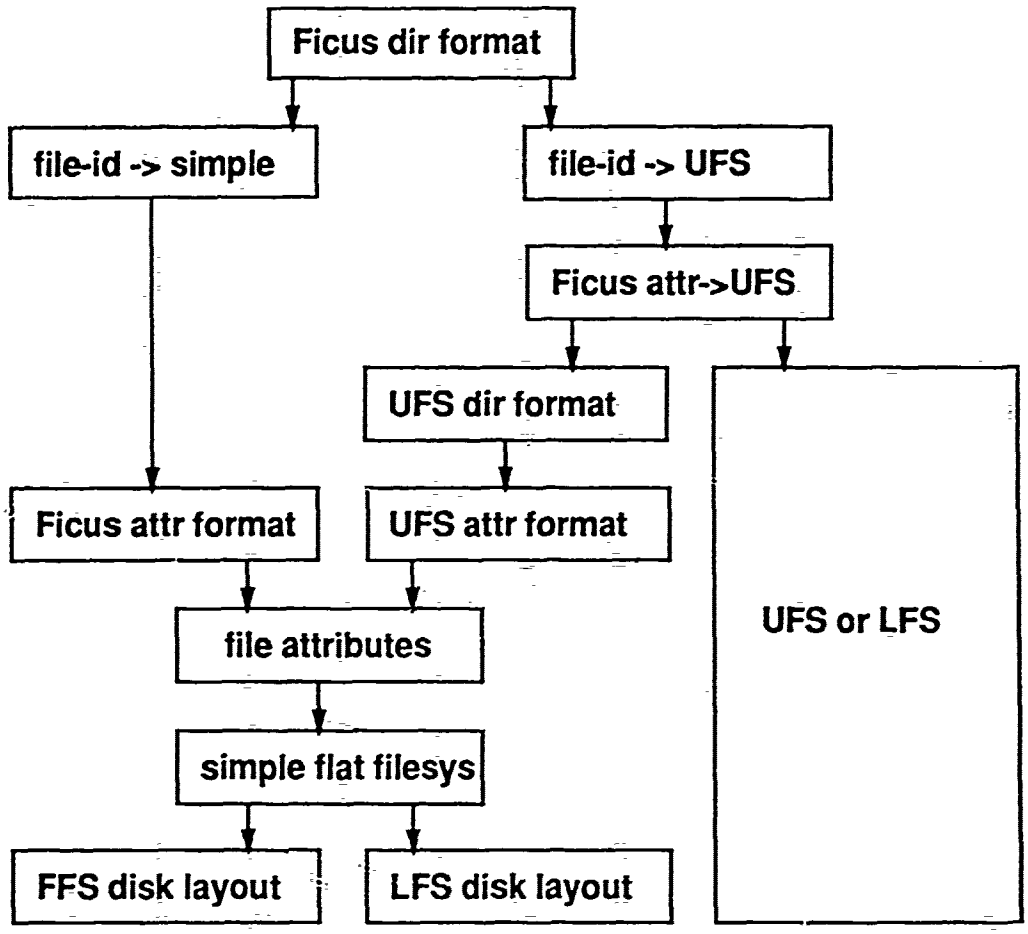


Figure 2.8: Decomposed physical and storage layers.

semantic interpretation beyond that necessary to marshal arguments. A transport layer can then be inserted between any two layers without regard for their semantics.

2.3.4.1 NFS as a transport layer

The original Ficus design and implementation used NFS as an approximation to an ideal vnode transport layer. Using NFS satisfied a design goal to leverage existing services whenever possible, and was further appealing due to the widespread adoption of NFS to provide remote file access. Introducing yet another network file access protocol had the potential to needlessly hinder acceptance of Ficus in the community at large, with little apparent benefit.

Unfortunately, NFS is not a transparent transport mechanism. It was not intended to be used between other services in a stackable layers architecture. Rather, it was meant to implement access to remote filesystems and hence, its designers' opinions about the semantics of remote file access are built into NFS. In particular, NFS was designed around the notion of stateless servers in order to simplify failure recovery.

NFS achieves a certain quality of performance by interpreting, modifying, or intercepting some vnode operation invocations. Certain interface operations which have no meaning in the context of the designer's view of remote file access (such as `open` and `close`, both stateful concepts) are not transmitted across the network to the next layer in the stack. The operations are handled (or ignored) internally, and not passed through. NFS also incorporates optimizations intended to reduce communications and improve performance. The file block caching and directory name lookup caching are not fully controllable (e.g., there is no user-level way to disable all caching), which results in unexpected behavior for layers which are not able to adopt the assumptions inherent in the NFS cache management policies.

It soon became clear that a layer attempting to leverage NFS as a generic transport service must be constructed with substantial NFS internal details in mind. The early Ficus implementation demonstrates that it is certainly possible to use NFS as a transport service, but at the expense of building extensive mechanisms to defeat numerous internal NFS mechanisms.

For example, the Ficus logical layer needs to obtain version vectors from the physical layer in order to perform replica selection. A version vector is too large to squeeze into available NFS file attributes, so some other means is required to pass version vectors across machine boundaries. The first design called for

use of a *control file*, i.e., a pseudo-file with a distinguished name known to both layers as the channel over which version vectors are to be passed. NFS caches file data blocks, so a repeated read on the control file frequently returned the cached block without actually querying the next layer—the required course of action. To defeat caching, a mechanism that cycles read request offsets through a large cycle was employed. This was not immediately successful, as NFS maintains a notion of file size so that it can optimize for read requests to non-existent data pages by simply returning a null page in response to such a read request, rather than generating network traffic to transfer a "known" data page. The control file mechanism had to be repaired so that it always reported to NFS that the file "size" was larger than the cycle size in use.

The extensible vnode interface and bypass operation were developed in large part due to unhappiness with the never ending set of counter-mechanisms being employed to outsmart NFS. The extensible interface has been installed in the current Ficus implementation, and it has been used to add a bypass routine to the NFS layer so that the counter-mechanisms could be discarded.

2.3.4.2 Multi-layer NFS

The emergence of an ideal vnode transport layer suggests the possibility of reconstructing NFS as a pair of remote file service layers surrounding a vnode transport layer. (Note that the original Ficus design attempted just the reverse: to construct an ideal vnode transport layer around an NFS base.) Various beneficial data page and attribute caching features of NFS are retained, but the core protocol is different (see Figure 2.9).

In this design, the transport layer is common to both NFS and the Ficus logical and physical layers. It can also serve as the basis for any future boundary-crossing services, or be easily replaced in its entirety by a similar transparent service.

2.3.4.3 NFS compatibility

Altering the core NFS protocol renders the new NFS layer set incompatible with the standard NFS protocol suite, yet retaining NFS compatibility with non-Ficus hosts is important. Examples of the power of layering, and the importance of standard NFS compatibility, are evident when considering the utility of an NFS layer placed above the logical layer or between the physical and UFS layers. Figure 2.10 displays several interesting possibilities. When an NFS layer is above the logical layer, an IBM PC running DOS and PC-NFS can access Ficus replicated files without being aware that replication is occurring. Similarly, configuring NFS

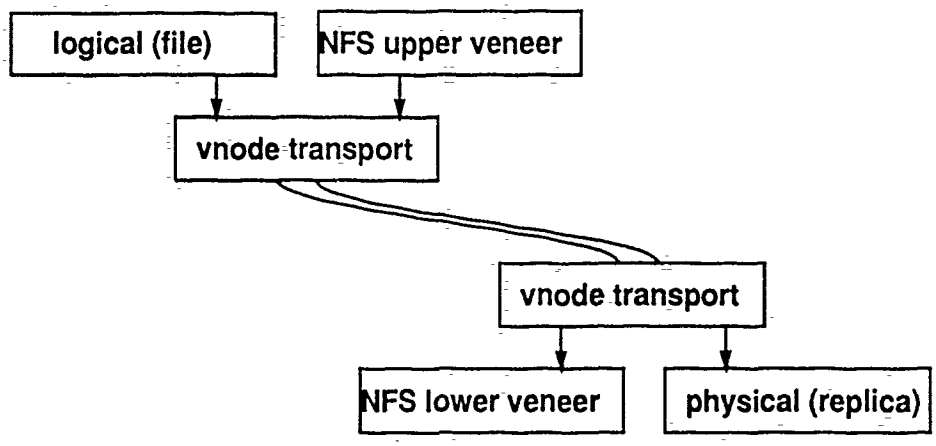


Figure 2.9: Multi-layer NFS design.

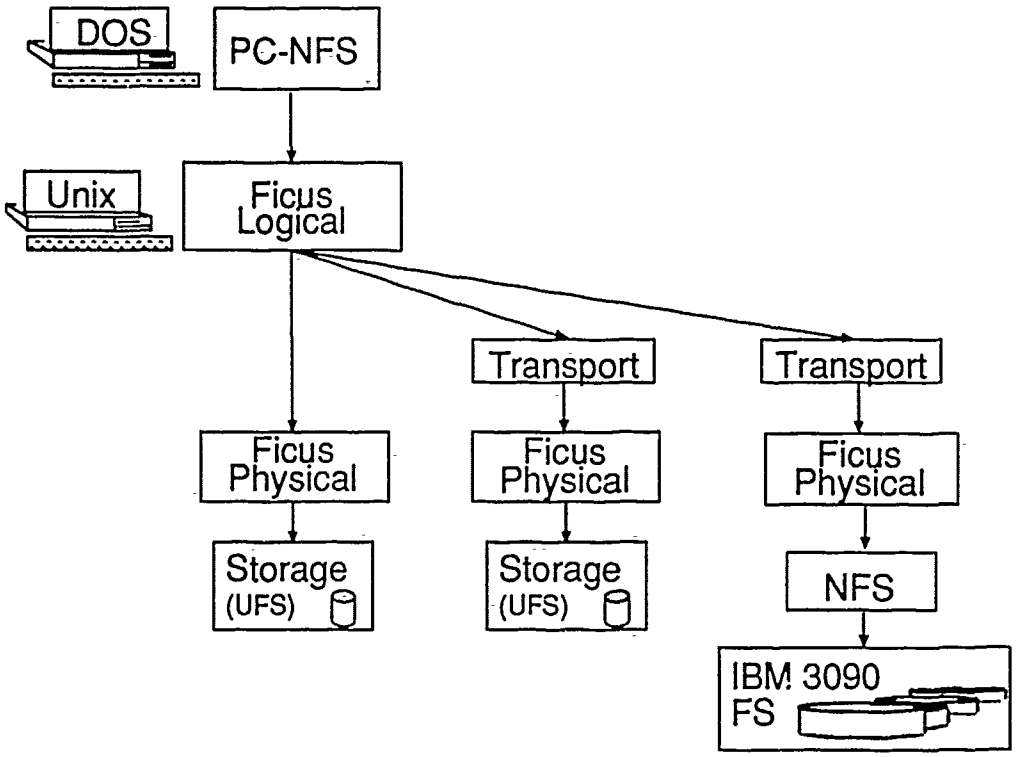


Figure 2.10: Using NFS to interact with non-Ficus hosts.

below the physical layer allows sites on which Ficus does not run to act as replica storage sites. This arrangement permits a Ficus site to store replicas on a mainframe running MVS and NFS.[17]

2.4 Synchronization

The optimistic consistency perspective is not universally appropriate: some applications require stronger consistency guarantees such as serializability. This section considers the synchronization problem of coordinating access by multiple clients (possibly on distinct hosts) to a single file; multi-file synchronization is not directly addressed.

The goal here is to identify and address issues peculiar to the Ficus environment, especially the stackable layers architecture and the use of an optimistically replicated filing service as a base. In the course of the discussion, a series of related synchronization service designs which provide successively greater flexibility and robustness is presented. These designs implement standard approaches to synchronization in the layered, optimistically replicated filing context by placing a (multi-layer) synchronization service above the Ficus logical layer.

A complete design for a particular synchronization policy is beyond the scope of this work. These designs are not currently implemented, nor are a few necessary minor enhancements to the existing layering mechanism and logical and physical layers.

2.4.1 Issues

Recall that the Ficus replicated filing service (see Figure 2.6, page 40) is constructed from three layers (logical, physical, UFS) whose execution environment(s) are intended to be set apart from the remainder of the operating system. Recall also that a logical file is typically represented by multiple physical replicas. In this context, several questions immediately arise. First, for whom is synchronization to be provided? Second, what actions are to be synchronized? Third, what is the subject of synchronization? A synchronization service must have answers to each of these in order to be well defined.

[17]This has been demonstrated using a non-Ficus SunOS host; logistics hinder an MVS demonstration of the concept.

2.4.1.1 Client identification

The issue of *client identity* is currently difficult in Ficus. The information-hiding aspect of the layering methodology obscures most identifying features of a client with the exception of a *credential* that in the existing design contains a user identifier. No notion of process family, process, or thread is routinely available to a layer. Further, the transparency provided by a transport layer hides even the identity of the host executing the original process on whose behalf the layer is working.

A synchronization service must, however, have some means of accurately identifying its clients (or vice versa: a client identifying itself to the service) lest it fail to provide an adequate degree of synchronization or fail to provide adequate access by enforcing an unnecessarily stringent synchronization policy. The designs here are based on the notion of a *synchronization capability* which supports an abstract concept of identity.

Capabilities are generated by the synchronization service on demand for knowledgeable clients via a `new-capability` operation. A capability is valid from time of issuance until the synchronization service is next initialized, which is normally at system boot time. Capabilities may be freely copied and passed among clients. All file access requests accompanied by a particular capability which are recognized by the synchronization service will be synchronized in accordance with the policy in effect.

As most clients are (at least initially) expected to be completely unaware of synchronization capabilities, a *client identification* layer normally lies between the synchronization service and its immediate regular client, the system call service of the operating system. The client identification layer ensures that all requests made of the synchronization service include a capability, even when a client does not include a capability with a file access request. In this case, the client identification layer makes a determination of the client's "identity" and appends an appropriate capability to the request.

This layer bridges the gap between the synchronization service's client model and the model supported by the host operating system. The default policy incorporated in the layer is thus quite dependent upon the native client model, be it a process, process family, thread, or some other model. To obtain the necessary information about client identity, this layer may need to violate information hiding principles and peek into some of the operating systems internal data structures and extract a process or process family identifier for use in defining a "client." This necessary interaction between the host operating system and layered ser-

vices is thus tightly confined to a single layer; the task of porting Ficus to other environments focuses primarily on this layer, and few others.[18]

Once the necessary client identification data has been obtained, the layer can determine whether it already possesses (in a volatile database) an appropriate capability to attach to the request, or whether it must first obtain a new capability from the synchronization service. The default policy used when a client has not provided a capability will vary according to environment. For example, in an academic environment a reasonable policy might be that all non-capability requests possessing the same credentials (UNIX user identifier, group identifier) be treated as emanating from a single "client" and so should be associated with a capability unique to that "client."

The identification layer supports two similar sets of file access operations: the standard set of operations (read, write, etc.) which do not include a capability parameter, and a parallel set that contains a capability. The standard (non-capability) operations consult the capability database so that a capability can be attached to the request before it is passed on to the next lower layer. The capability-included operations simply pass the request on immediately.

The important point here is that a synchronization service must assume that some appropriate notion of "client" exists, even when that concept is not well supported by either the host operating system or the layering methodology. A combination of capabilities and a layer instituting a client abstraction satisfy this requirement.

2.4.1.2 Client actions

The synchronization service acts as a funnel through which all file access requests pass. The primary actions of interest are `read` and `write`, although other requests (`open` and (`close`, for example) may also be of interest depending upon the synchronization policy to be enforced.

Suppose, for example, that the policy is to provide single-copy semantics to all callers, much as would be provided by a standalone UNIX host. The synchronization service must ensure that all `read` and `write` requests are (or appear to be) directed to a single file replica, and that any data caches maintained by lower layers are coherent with respect to synchronized access.

Two architectures to support single-copy semantics are shown in Figures 2.11 and 2.12. The first design forces all requests to pass through a single layer

[18]The transport layer is necessarily hardware architecture specific, and might be operating system specific as well.

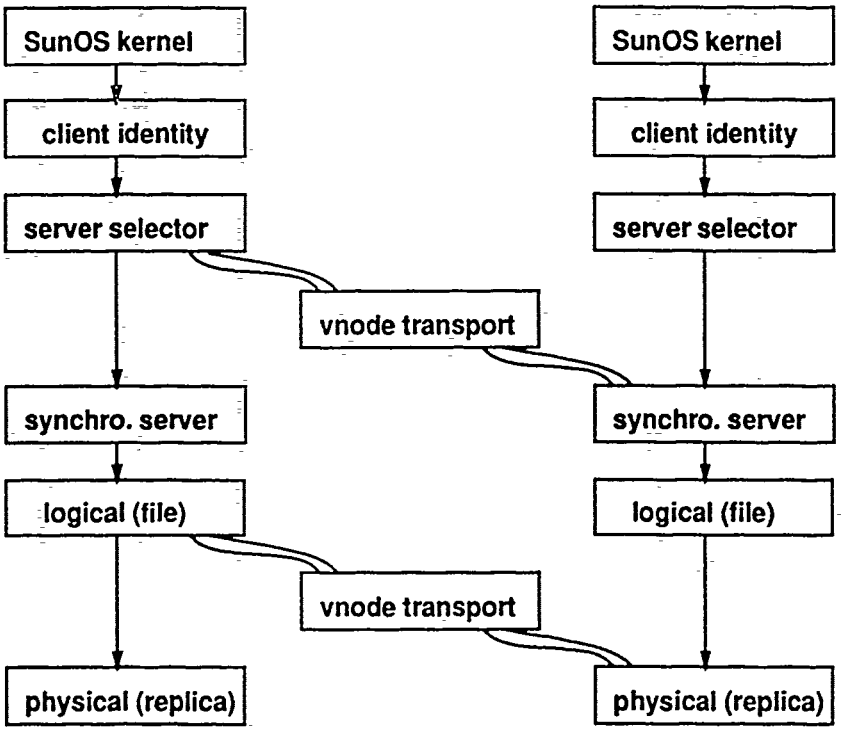


Figure 2.11: Centralized synchronization service.

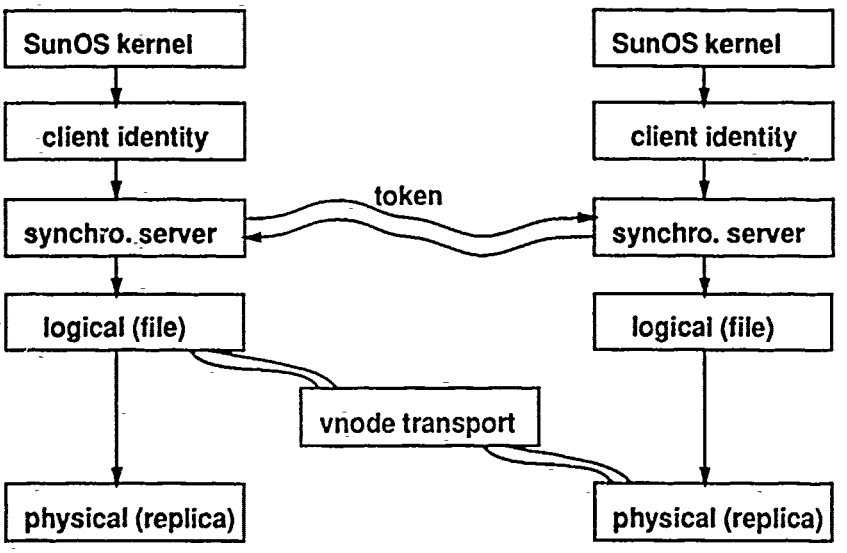


Figure 2.12: Token-based synchronization service.

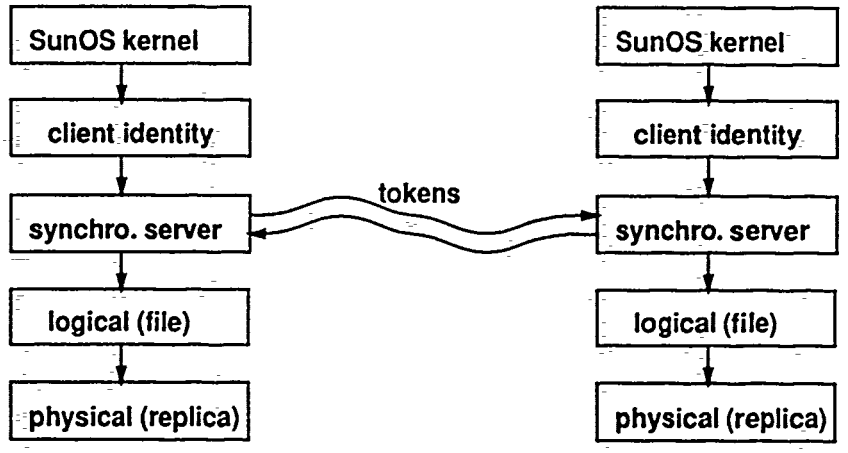


Figure 2.13: Quorum-based synchronization service.

instance, while the second uses a token mechanism to ensure that all `read` and `write` operations appear to happen to a single file, and are immediately visible to all clients.

The centralized design incorporates an additional layer that directs a request to a particular synchronization server. The "direction" information is extracted from the capability, where it was placed when generated by the server. This information consists of a host identifier and any other data required to locate the indicated server.

2.4.1.3 Objects

At its lower levels, the synchronization service must be cognizant of optimistic replication so that the appropriate quality of service may be provided. A particularly useful distinction which can be made is between synchronized access to a single file replica and synchronized access to a single logical file.

Replica-based service is adequately provided by the designs in Figures 2.11 and 2.12. Such service is constrained, however, by the robustness of the hosts maintaining the replica and executing the synchronization servers.

A synchronization service with increased robustness is readily constructed by exploiting replication to provide file-based synchronization. Figure 2.13 presents a design that uses quorum consensus replica management techniques for synchronization.

Each file replica contains a *read quorum* and *write quorum* attribute. The

quorum values are initially set at file (replica) creation. The entire "vote space" of 2^{32} votes is allocated among the replicas at all times; adding a replica requires that some replica surrender some of its voting space to the new replica.

The quorum consensus model differs from the previous one in that a set of tokens for each file (one for each replica) must be managed rather than a single token. Each token is a voting proxy for its respective replica, so a synchronization server must amass a quorum's worth of proxies before granting a file access request.

An interesting aspect of this design is that a synchronized update need only be applied to a single replica. The existing replica management mechanisms (update notification, propagation, and reconciliation) handle the details of propagating the update to other replicas. So long as the synchronization service correctly compiles the quorum-related data from the replicas when the service commences to operate, and properly manages the data after that, the optimistic mechanisms below it will satisfy its requirements.

The fact that quorum consensus can be easily implemented on top of optimistic replication is important not only to prove that serializable concurrency control can be constructed above an optimistic base, but also because optimistic services have the potential to provide a lower immediate cost for update operations. If an update can be applied to a single replica in the short term, on behalf of many replicas that will reflect the update in the longer term, the impact of the update operation on the system as a whole, and the client in particular, may be greatly reduced.

2.4.1.4 Unsynchronized actions

The layered architecture places no absolute requirements on stack composition, nor is any explicit cross-layer locking facility provided. It is, therefore, possible for clients ignorantly or deliberately to access and update file replicas without utilizing the synchronization service. If such access could disturb a synchronization service's correctness criteria, the service must monitor replica state with each access.

The necessary degree of control over replica state can be achieved by allowing a replication client to specify with each access request which version (via a version vector) of a replica is expected to satisfy the request,[19] and also by appending to the response the version vector resulting from the request (atomically, post

[19]A client can initially obtain a version vector by retrieving the extended file attributes of a replica.

read or write). If the supplied version vector does not match the actual version of the replica, an error is returned to the caller. This change to the logical and physical layer services supplies the atomicity required by a synchronization service to ensure that data is read and written as expected, without undetected interference from unsynchronized activity.

2.4.2 Control flow examples

This section outlines the typical control flow for several common synchronization scenarios. The first two examples consider groups of processes that are unaware of the replication or synchronization services. The third example assumes that the clients are cognizant of the synchronization service, but oblivious to replication.

2.4.2.1 Process family

In this example, the collection of clients requiring synchronization is a process family formed by an off-the-shelf UNIX application; it has no knowledge of the synchronization service, replication, or any other non-traditional UNIX service. This example assumes that the default synchronization policy (e.g., serializability) provided by the synchronization server layer is appropriate for the application.

Suppose that the default client identification policy is that all processes issuing requests through the client identity layer but which do not include a capability are to be treated collectively as a single "client." The identity layer could then use a single, static capability (acquired during initialization at boot time) to attach to each request it passes on down the stack.

This scenario exemplifies a basic approach to providing backwards compatibility for UNIX applications on Ficus. Treating all clients as peers, though, may place undue strictures on availability for some clients. The following scenario assumes a simple reduction in synchronization granularity based on credentials.

2.4.2.2 Independent processes with identical user-ids

Here the processes of interest are initiated by standard applications, but the default identity granularity enforced by the client identification layer is a single process. In this case, synchronization can be effected by including a user-id based synchronization registration request as part of each login instance for the particular user-id.

In response to the first registration request, the client identity layer will acquire a capability from the synchronization server, and store it in its in-core (volatile) database. Further registration requests are satisfied from the database. For each file access request, the client identity layer locates the appropriate capability in its database and attaches it to the request before it is passed on to the layer below.

The general framework for these synchronization service designs has an inherent limitation on default service: it assumes that the maximal granularity of default client identification is the set of clients of a single identification layer instance, which are usually co-resident on one host. Synchronization for clients resident on multiple hosts requires clients to be knowledgeable about the service itself. The next example considers synchronized service for clients from multiple hosts.

2.4.2.3 Clients on multiple hosts

Suppose that several processes, each on a different host, wish to have synchronized read and write access to a particular file. These processes are not aware that an optimistic replication service exists, but are knowledgeable about the synchronization capability service.

To begin, one of the processes requests and obtains a new synchronization capability from the client identity layer. (The client identity layer doesn't support such a request itself, so the bypass operation automatically passes the request on to the next lower layer, and so on, until it is received by the synchronization service layer.) The client identity layer stores the capability in its database, and then returns it to the client. Having obtained the capability, the client shares it with its colleagues via some form of interprocess communication.

Each of these clients then attaches the capability to every file access request it issues. The client identity layer simply passes the requests on to the synchronization server, where the appropriate synchronization is performed.

2.4.3 Summary

These designs are built around a general framework in which a variety of specific synchronization policies can be implemented. It addresses the primary layering and replication management problems encountered when providing strict synchronization service in Ficus. The simple base services described above should be sufficient for many applications in which optimistic consistency is inappropri-

ate.

2.5 Large scale replication

This section considers a number of issues with regards to very large numbers of replicas.

2.5.1 Version vectors

The theoretical ability of a version vector to scale up to any number of replicas does not obscure the practical reality: as the number of replicas increases, the processing, storage, and bandwidth overhead of a version vector eventually overshadows the costs of the replicated object itself. It is not readily apparent that the version vector mechanism is appropriate for large scale replication.

The basic mechanism can be adjusted in several ways to facilitate large scale replication. All of these alterations preserve the integrity of version vectors' support for "no lost updates" semantics. Some, however, may erroneously report an update/update conflict in certain circumstances; such anomalies are carefully noted in the sequel.

These adjustments all exploit various aspects of locality, with attendant implications for consistency mechanisms. Replica selection, update notification and reconciliation must each be re-examined in the context of modified version vector methods.

2.5.1.1 Compression

If a significant number of version vector components are zero-valued, conventional sparse matrix compression and manipulation techniques can be used to reduce overhead. But are version vectors sparse?

A non-zero version vector component indicates that the corresponding replica has been directly updated (as opposed to receiving propagated data) at some time in the past. Version vector sparseness, therefore, is a function of locality with respect to update access. Locality, in turn, is a function of the amount of update by distinct clients, and the concurrency displayed by shared update access.

File access locality studies have shown that shared update is rare in practice [Flo86b, Flo86a] in general purpose (university) settings. If, then, the (typi-

cally) single updating client's choice of replicas is biased towards a small subset of actual replicas, it is reasonable to expect that few of the version vector components would acquire a non-zero value.

The default replica selection policy (see Section 2.3.2.1) exhibits a bias towards a locally stored replica if one exists, and randomly selects a replica otherwise. The desired new behavior can easily be obtained by altering the selection policy to exhibit a preference for replicas known to have a non-zero vector component already.

Update notification is not especially affected by version vector compression; reconciliation might exploit the bias and choose to consult "non-zero component" replicas more frequently than others. This change does not result in false conflict reports.

2.5.1.2 Read-only replicas

Another way to reduce the cost of version vectors is to classify *a priori* some replicas to be read-only. These replicas do not have components in the version vector at all; only directly writable replicas have slots. Read-only replicas may even elect not to be mentioned in the replica list.

Declaring a replica to be read-only introduces an artificial constraint on update availability in return for reduced overhead. It forces replica selection to be sensitive to intended read-write usage; it limits update notification to writable replicas; and it alters the topology of reconciliation.

The absence of some read-only replicas in the replica list constrains the ability of replica selection (and update notification and reconciliation) to locate read-only replicas. In particular, the standard replica selection mechanism cannot locate an unmentioned replica. A small change in the autografting service, however, allows the selection mechanism to determine whether a local replica does or does not exist. If no local replica exists, replica selection is forced to choose a replica from the replica list as usual.

Read-only replicas that are not mentioned in the replica list do not receive update notifications, nor will any other replicas initiate reconciliation with them. The burden is entirely upon read-only replicas to keep up to date with writable replicas. Read-only replicas that are listed will receive update notifications, but will not be reconciled against as they are not expected to experience updates; a writable replica would normally receive little benefit from checking the status of a read-only replica.

2.5.1.3 Upgradable read-only replicas

The most onerous aspect of read-only replicas manifests itself when the only available replica is read-only, and yet the ability to apply an update is highly desired. One solution to this problem is to allow a read-only replica to be upgraded to a writable replica.

This may be accomplished in a manner similar to adding an entirely new replica: simply add the replica's *repl-id* to the (local) replica list and permit the update to proceed. The appropriate version vector component will be incremented as usual. The newly-updatable replica then needs to persuade an established writable replica to propagate the replica list modification, either by issuing an update notification or by otherwise coaxing the other replica to initiate reconciliation with it so that it can learn of the list change.

The basic reconciliation algorithms (see Chapter 3) must be slightly adjusted to cope with additional replicas. In particular, they must assume that the replica list is a monotonic, length-increasing data structure. Aside from the difficulty that received replicas lists must be routinely sorted to account for divergence in the component ordering, the algorithms must tolerate a dynamic number of participants. Further details are found in Section 3.2.4.

Upgrading a read-only replica to writable status is irreversible in this model. If upgrading is a common occurrence for a single file (which would indicate that access locality is not a property of this file), the replica list may grow to an unmanageable size. Second-class replicas address this situation.

2.5.1.4 Second-class replicas

A second-class replica begins as a read-only replica. When an update is necessary, but only a read-only replica is accessible, the update is applied to that replica. Rather than append a new slot to the version vector, a flag is set which indicates that the replica is a logical descendant of the version indicated by its version vector.[20]

As with the upgradable strategy, a flagged replica must persuade a writable replica to reconcile against it. A qualifying writable replica has a version vector that is less than or equal to the flagged version vector. Any replica with a version

[20]The idea of supporting an "updatable" replica while avoiding the expense of allocating and maintaining a version vector component was first suggested for Coda in [SKK90]. Coda version vectors are quite different in detail from the original Parker version vectors [PPR83] that are used in Ficus.

vector that is greater than the base version vector (in *any* component) is in conflict with the flagged replica—but it is the flagged replica's responsibility to resolve the conflict.

Upon successfully reconciling with the flagged replica, the writable replica's version vector component is incremented as though the update(s) were originally applied to the writable replica. The difficult part of this exchange is that the flagged replica's version vector should atomically be adjusted to reflect the new version vector of the reconciling replica. If the two vectors are not set atomically, it may later appear that the flagged replica conflicts with the reconciled replica—when, in fact, it does not. If this occurs, a false conflict will be reported.[21]

This technique requires a small modification to the atomic commit mechanism used by reconciliation when a new version is propagated into a replica. No changes to replica selection beyond those needed for upgrading are required, nor any changes to the update notification or overall reconciliation mechanisms necessary.

2.5.2 Name space

In part due to the original goal of leveraging UFS, the Ficus name space is currently a stepchild of the UFS name space. The global perspective of this relationship is shown in Figure 2.14. This name space arrangement has two major flaws: most UFS files are nameable from only one host, and UFS file names are host context dependent.

These problems can be addressed in part by altering the relationship between the Ficus name space and the (existing) UFS name space(s) so that the Ficus name space provides a global context for all file names. Figure 2.15 shows such an arrangement. In essence, this arrangement constructs an umbrella naming hierarchy above existing UFS hierarchies and in the process renders all files globally nameable in the context of a globally recognized root.[22]

The "supertree" approach to global naming forces consideration of two related issues, backward compatibility (especially with embedded file names) and shorthand naming. Both issues are aspects of the general *name context* question, outlined in Saltzer's discourse [Sal78] on the naming and binding of objects.

[21]A false conflict will also appear if a flagged replica is propagated to another (formerly) read-only replica. To maintain the integrity of the version vector, once a vector and replica are flagged, they must propagate together with the flag. Once propagated, it is impossible to distinguish this situation from one in which two read-only replicas performed independent updates, and thus each resulted with a flagged replica.

[22]The idea of a global name space incorporating Domain Naming System host names has previously been suggested in [CM89].

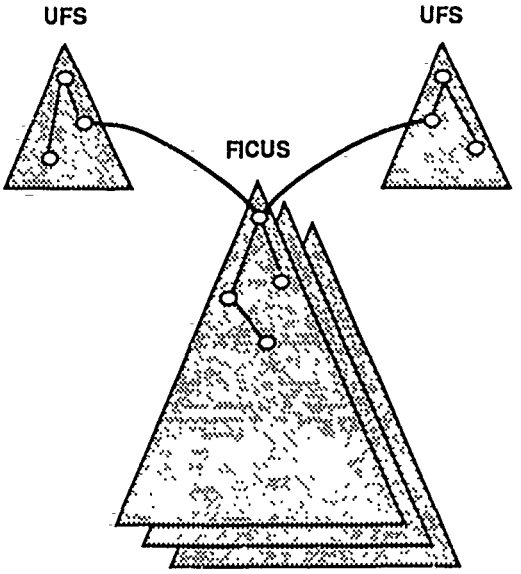


Figure 2.14: Current UFS and Ficus name space relationship.

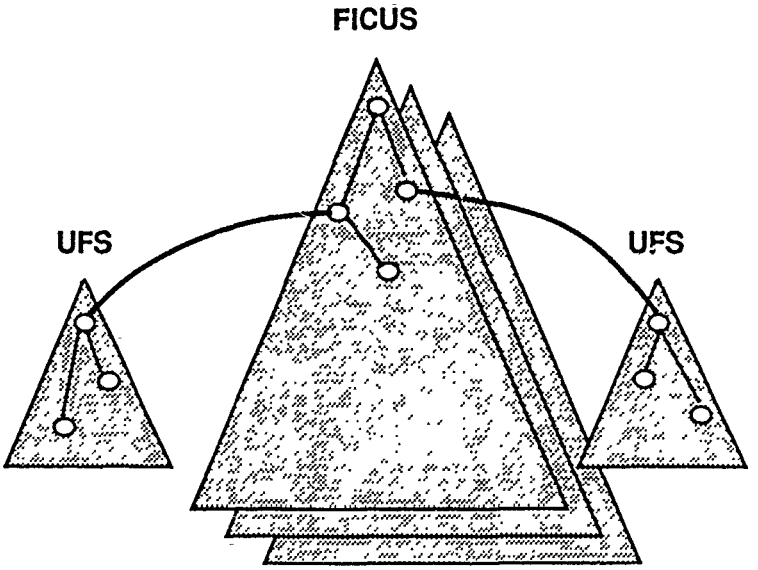


Figure 2.15: Ficus global name context.

2.5.2.1 Name contexts and closures

Applying Saltzer's terminology to Ficus, an *object* (Ficus file) is represented by a (Ficus) identifier. A *name* is used by one object to refer to another object, via a mapping from name to identifier termed a *binding*. A particular set of bindings is termed a *context*; a name is always interpreted relative to some context. In Ficus (and UNIX), a directory file (*catalog* in Saltzer's glossary) is the repository for a context.

The name context question is to determine the context relative to which a name should be resolved. For example, a UNIX file name is resolved either in the context of a *working directory* or in the context of a distinguished *root directory* if the name begins with a "/" (forward slash) character. Some shell mechanisms further support a *search list*, an ordered set of contexts in which certain (command) name resolutions will be attempted.

Saltzer introduced the *closure* concept to describe the association of an object which references another object by name with a context in which that name is (expected to be) bound. A mechanism that connects an object with a context is called a closure.

Although the global Ficus name space does not require closure support in theory, in practice a closure mechanism is essential. Existing software with embedded names such as `/bin/login` must continue to work as expected, that is, a name must map to the same object as before (pre-Ficus), despite the new supertree above an old UFS name space root. A closure mechanism is necessary to select the correct context in which to resolve the name.

Not only existing software with embedded names benefits from a closure service. The additional three or more levels of names above the previous UFS naming hierarchy significantly extends the length of all file names, which is a generally unacceptable increased burden on human clients. A flexible closure mechanism that subsumes the UNIX working directory notion and various shell context mechanisms provides an integrated solution to this problem.

2.5.2.2 Ficus closures

Neuman [Neu89] argues for a simple closure mechanism in which a context is associated with every object in a system. For example, a program image with embedded names should also contain and provide the context for name resolution. Names supplied as program input should be resolved in the user's (process) context (perhaps a working directory), and names embedded in a data file should

be interpreted in a context associated with that file. A process should always be free to redefine its own context, and to ignore contexts provided by other objects or processes.

Closure support, as with some aspects of synchronization (see Section 2.4), straddles both file system and process management. Leveraging the existing SunOS process management services necessarily constrains the completeness of closure support in Ficus, to the extent that no process service enhancements are made. A further constraint is the pre-processing that SunOS file access system calls perform on file names: names are parsed by a system call routine, and partially or completely interpreted at the system call level without ever calling the file service proper. Nevertheless, a rudimentary closure service provided as a stackable layer is feasible.

The Ficus closure service design utilizes a new distinguished character ("@") as a prefix to signify a fully qualified file name; other names are interpreted in the traditional UNIX working directory and working root contexts. An additional context attribute is provided for each file. The closure layer is normally placed somewhere above the logical layer in a Ficus layer stack.

At system initialization time, the closure layer sets the default host working root to be a fully qualified name for a volume analogous to the traditional UFS "root filesystem". An internal pointer to this volume is inherited by all processes when they are created; this is the context in which the system call pre-processing routines will interpret "/"-prefixed names.

This approach offers immediate backward file naming compatibility, except when an existing file name (component) begins with "@".[23] It also offers a context-knowledgeable process the ability to establish contexts for files, and to inquire about file contexts. A process can simply prepend a context to a file name before presenting it to the system call service for translation.

This method does not automatically provide a custom context in which an executable image will execute, nor does it provide inherited contexts for process families. Ideally, a context would automatically be prepended to file names, without the need for a process to actually do so itself, and it would be inherited by child processes as are other process environment attributes. Support for these features is dependent upon modifications to the process creation and management services.

[23]This exception appears to be an unavoidable consequence of not modifying the existing process management and file access system call mechanisms. It is easy to avoid the exception by appropriate enhancements.

2.5.2.3 Volume-based supertree

Cheriton and Manning [CM89] presented a taxonomy for classifying a global naming hierarchy into levels based upon administrative, availability, and mutability issues. Their taxonomy contains three levels: global, administrative, and managerial. Using the Domain Naming System as an example, they place the global root and its children into the global level (e.g., `/edu` and `/com`); the administrative level lists organizations and some hosts (e.g., the organizations `ucla` and `ucla/cs`, possibly the host `ucla/seas/admin`); and finally, the managerial level lists namespaces spanning a single host or less (e.g., `/bin` and `/usr/guy`).

The Ficus volume mechanism allows for an even finer granularity when circumstances warrant. In the case of the global name space, a useful criterion for determining volume granularity is based upon the desired replication factor (i.e., number of replicas) for a node in the name space: a volume boundary should be placed at any point in which the replication factors of a parent node and its children differ by at least one order of magnitude.

In the case of the DARPA Internet and its Domain Naming System, each of the top four or so levels fits the criterion. Figure 2.16 shows a particular extended branch of an example global name space, and indicates the likely replication factor of each volume shown. Note that the expected range of replication factors for each volume differs by about one order of magnitude at each level in the hierarchy. These ranges are determined primarily by the number of hosts that require a volume replica simply to operate autonomously; some hosts may wish to house replicas of volumes representing other portions of the name space as well.

The upper level volumes simply provide the overall name space structure; it is only the lower level volumes that would typically contain "real" data files, executable images, and so on. In many cases these lower volumes will be replicated only to promote higher availability, but in some cases replication will be used as a system administration tool. For example, a local area network administrator might choose to organize each node's standard library, utility, and application volumes as a set of volume replicas rather than as individual volumes.[24] Such an arrangement would ease the administrative burden when new software is to be installed, or patches applied: one could make the change to a single volume replica, and rely on the reconciliation daemons to propagate the changes.[25]

[24]Symbolic links may be useful in preserving a particular volume name space organization, and yet benefit from replicated library volumes.

[25]Installation ease should be balanced against the inherent absence of a "firewall" to prevent error propagation.

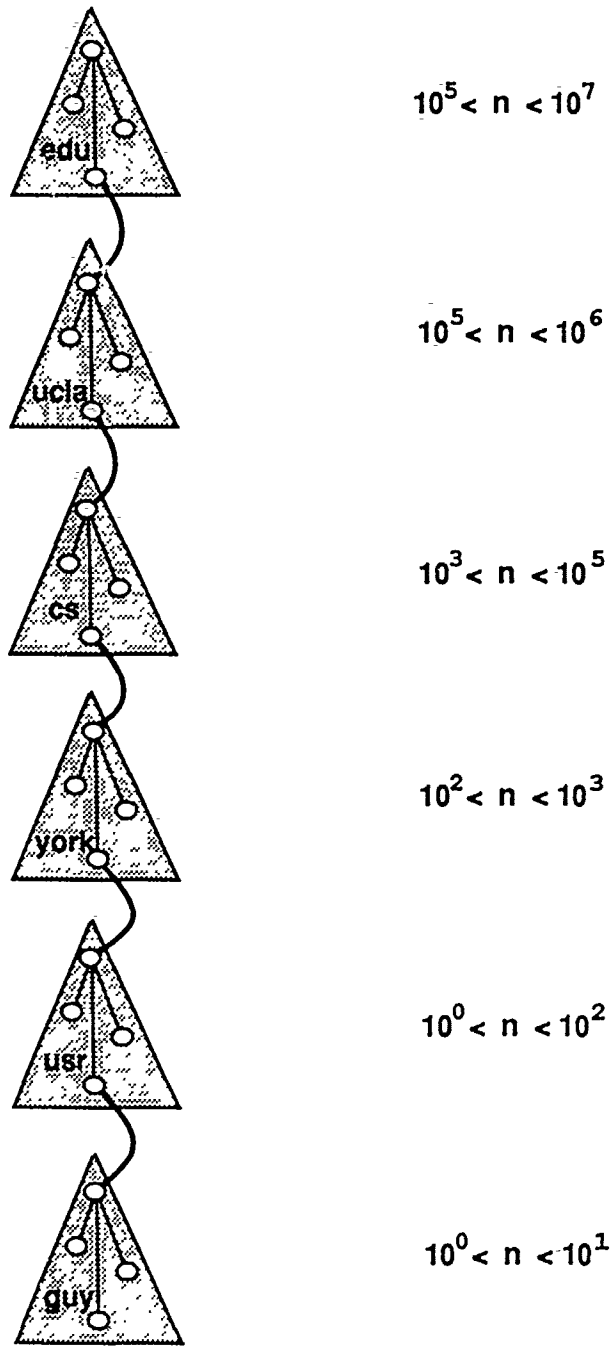


Figure 2.16: Volume support for global name space. (n = |replicas|)

2.5.3 Summary

The modest enhancements described above are important steps to be taken along the path of achieving the goal of very large scale filing. Support for large numbers of replicas is essential for the global name space to be effective. The Ficus name space transition from stepchild to umbrella is a further critical step.

2.6 Status and performance

The system as described in this chapter, including reconciliation, is operational and in daily experimental use. The system has been tested with up to eight replicas. Ficus replication has been run with geographically remote clusters; specifically, volumes have been replicated at USC/ISI, SRI, and UCLA[26].

Operational layers include the transport layer, the logical and physical replication layers, a null layer, and a measurement layer. Prototype implementations of encryption, file versions, cache consistency, and "second-class" replication layers have been constructed as class projects.

Ficus was originally based on SunOS 4.0.3 for the Sun 3 hardware family, though ports to other versions of UNIX already supporting a VFS interface should be quite straightforward. A port to SunOS 4.1 for the Sun 4 hardware family has recently been completed. The initial implementation (completed Summer 1989) used an unmodified vnode interface with extended operations supported by overloading existing operations. The current implementation uses the new extensible vnode interface described in [HP91a]. Ficus kernels are about 20% larger than a similar non-Ficus kernel.

2.6.1 Performance measurements

This section reports performance measurements for various configurations of replicated and unreplicated volumes. All measurements utilize Sun 3/60s, each with a SCSI disk and 10Mb ethernet connection. All nodes are part of the same ethernet segment, with the exception of the machines at ISI and SRI.

The first benchmark used is the *Modified Andrew Benchmark* [Ous90, HKM88], which is designed to reflect a typical mix of file operations. However, since this

[26]USC/ISI is located in Marina Del Rey and connected to UCLA (after several gateways) via Los Nettos. SRI is located in Palo Alto, California and connected to UCLA via Los Nettos to the San Diego Supercomputer Center followed by NSFnet. Thanks to Bob Balzer of ISI and Alan Downing of SRI for arranging for Ficus nodes at their respective institutes.

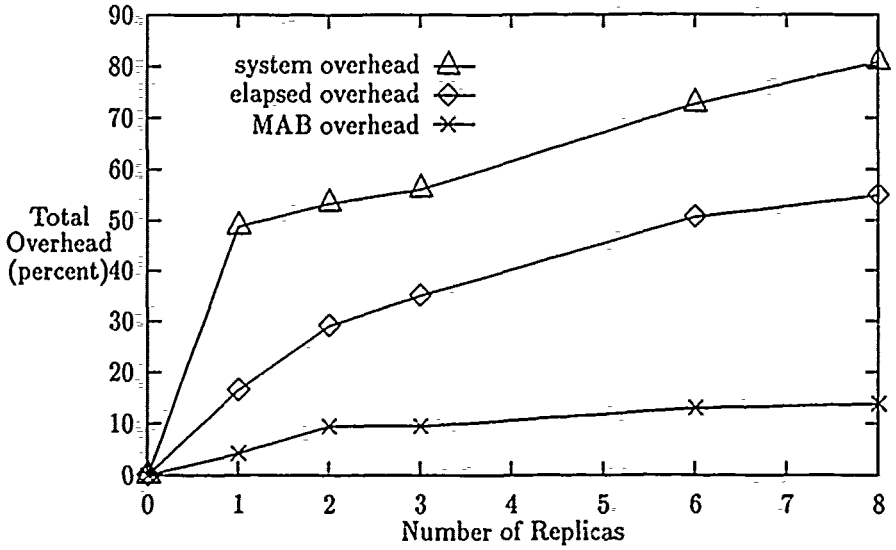


Figure 2.17: Percentage overhead versus number of replicas.

benchmark is not particularly illustrative of replicated file system performance (it is dominated by a largely cpu bound compilation phase), a second benchmark is used that is a much better worst case measure of Ficus. This second benchmark is a recursive copy ("cp -r /usr/include .") to a disk local to a Sun-3/60 from an NFS-mounted file system housed on a Sun-3/480 connected to the same ethernet segment. In the local environment, /usr/include is an unbalanced tree of depth four, with 47 directories and 1465 files totaling 4.7 Mbytes.[27] The costs studied were incurred by the site generating the activity. These measurements do not account for costs incurred by other processors.

Figure 2.17 shows the extent of Ficus costs (overhead) over normal SunOS performance. (A horizontal plot along the x-axis would indicate that Ficus was no more or less expensive than SunOS for a particular service.) Three sets of results are plotted, each measuring Ficus overhead as the target volume replication factor ranges from one to eight. One plot (*MAB overhead*) shows the overhead experienced by the Modified Andrew Benchmark (MAB) normalized against base MAB results obtained on standard SunOS. The other two plots show system time overhead (*system overhead*) and elapsed time overhead (*elapsed overhead*) displayed by the recursive copy benchmark, normalized against SunOS test results.

[27]Prior to execution of each benchmark, operating system caches were flushed to remove all references to the source and target volumes. The disk partition containing the target volume was reinitialized before each run of a benchmark. The measured node performed no other processing tasks during the benchmark.

2.6.2 Discussion

These measurements are very encouraging. For the modified Andrew benchmark, even with eight replicas, the overhead is only 14 percent. For the common case of three replicas, overhead is less than 10 percent. There is considerably more impact due to replication in the recursive copy case (between 30 and 50 percent). For each file copied, the system must place an entry in the directory (updating both the directory and its auxiliary information), create the file itself, placing its Ficus-specific attributes in the auxiliary file, notify all replicas of the directory operation to create the name for the new file, notify the replicas of the availability of the file's contents, and serve all of the asynchronous propagation requests as the replicas pull over the file contents. Copy is a worst case operation in terms of overhead for Ficus.

In interpreting these numbers, it is important to remember that Ficus applies an update synchronously to one replica and queues an "update notification" for asynchronous delivery to other "secondary" replicas. Each replica queues incoming update notifications and asynchronously processes the notifications. Directory update notifications completely describe the update, so no interrogation of the primary is needed to process a notification. File update notifications carry no data (only a version vector), so a "pull" is initiated by a secondary to bring its replica up to date. Data for updates is generally served out of the cache on the originating site.

The "system" times for all Ficus measurements are similar because the cost of asynchronous update notifications is in the background (there is some impact of increased replication factors reflected in the measurements as the interrupt handling for pull requests is included in system time).

The increased "elapsed" or "wall clock" time observed when more replicas are employed is attributed primarily to the cost of servicing requests from the the secondary in response to update notifications. It should further be noted that the recursive copy completes and the elapsed time is reported when it finishes synchronously updating the single chosen replica. It is generally the case that many of the remote replicas have not finished their asynchronous pull of the data. Thus the greater the delay in the network, or the slower the remote disks, the slower the requests arrive at the originating replica and hence the sooner the synchronous part completes. Hence, some of the numbers actually look better when the replicas are further apart.

2.6.3 Wide area operation

In the performance graphs shown, all replicas resided on machines on the same physical ethernet cable. Several of these measurements were repeated, this time locating replicas on sites connected by the Internet[28]. For the case of the recursive copy, locating one replica at SRI and one at UCLA yielded measurements for both system and elapsed time that were essentially identical to the case where both replicas were on the same local network. Measurements of a three replica configuration (UCLA, ISI, and SRI) resulted in a 36% overhead over UNIX for elapsed time (vs. 35% for the local net case) and 48% overhead for system time (vs. 56% for the local net case). For the modified Andrew benchmark the long distance three replica configuration resulted in an overhead of 19.9% compared to 9.5% when the three replicas were local.

Not surprisingly given the asynchronous update strategy, locating the background replicas at more remote sites has minimal impact on the performance of these benchmarks. Of course, access to the remote replicas is correspondingly slower, equivalent to that achieved by accessing them with NFS.

Only very preliminary efforts have yet been made to optimize the performance of the implementation as work thus far has focused on functionality. There is reason to believe that the numbers reported here can be improved substantially with careful analysis and optimization of the system's behavior (especially the effectiveness of its several caching mechanisms).

2.6.4 Implementation effort

Serious implementation work on Ficus has been underway for twenty four months. On the average, three experienced systems programmers have been engaged in full-time development. In addition, an average of two persons have worked on the design in parallel with the implementation. This author contributed approximately one man-year to the implementation effort, in addition to ongoing design research for more than three years.

The implementation has been done entirely in the C programming language. The logical and physical layers comprise about 9,500 and 12,000 lines of code, respectively.[29] The logical layer includes the update notification and propagation

[28]To avoid excessive retransmissions, the mounts across the Internet use 1K message block sizes where 8K messages are used over the ethernet.

[29]About 5,000 lines of physical layer code is devoted to Ficus directory manipulation and compatibility code to leverage UFS. Much of this code would be eliminated in a decomposed implementation, as outlined in Figure 2.8, page 50.

mechanisms; the reconciliation mechanism constitutes an additional 1,400 lines of code. The layer mechanism is composed of 1,700 lines, and miscellaneous tools contain 5,000 lines.

CHAPTER 3

Algorithms

Management of related, replicated objects is often fundamental to the design of reliable distributed systems. We are concerned both with the objects themselves: propagation of updates and reclamation of storage; as well as management of the possibly replicated directories used to keep track of and find the objects.

This chapter presents a family of algorithms for use in managing replicated objects and the accompanying graph structured directory systems. Members of this family are presented in order of increasing power and flexibility, followed by discussion of their correctness. The use of the algorithms in a replicated file system context is outlined throughout the presentation.

3.1 Introduction

Desires to improve availability and performance of information serves to motivate replicating information at locations "closer" to the data's intended use. A continuing difficulty in the operation of replicated information storage services, however, is unsatisfactory support for consistent update. Conventional methods achieve mutual consistency of data and the directories which refer to them by restricting availability for update. In the face of communications limitations, methods such as primary site, majority voting, quorum consensus, and the like reduce the performance and availability for update as the number of copies of an object or directory references is increased. This pattern is the reverse of what is desired.

There are numerous environments for which replicated storage is quite valuable. In some of these, rapid communication among sites is not suitable or even possible. Interesting examples include conventional high availability systems using redundant hardware, significant numbers of workstation users collectively engaged in a large software development project, an office workgroup composed of several widely geographically separated workgroups, large numbers of laptops operating while disconnected, and military systems subject to communications silence. These examples share several common characteristics:

- low latency communications on demand cannot be guaranteed, either due to failures or policy decisions (such as not keeping a line in operation during high tariff periods);
- updates to data and meta-data (directories) are important to allow and occur from sites whose identity could not be specified in advance;
- concurrent updates of a given data item or directory entry are quite unlikely, and in those rare cases where a conflict does occur, subsequent reconciliation is feasible. Strict transaction semantics are not required.

We argue elsewhere (see Chapter 1) that these conditions characterize a very large set of important environments, including much of today's use of distributed file systems

Our approach to providing replicated storage in these environments is called *optimistic replication.* Optimistic replication uses a *one-copy availability* concurrency control policy for both read and update: if any copy is physically accessible, read and update are permitted. Optimistic replication further guarantees *no lost updates* semantics, so it is incumbent upon the system to detect conflicting updates and manage the mutual inconsistency until it is repaired.

Conventional replica management schemes implicitly or explicitly always have the property that a set of up-to-date "authority" replicas exists. No such authority is present in optimistic replication, short of a consensus reached by all replicas—a consensus not easily obtained when a complete communications graph between all replicas is unattainable.

For example, consider the problem of creating and deleting objects under optimistic replication. Object creation can be effected by causing a single replica to exist at one node; another node may then notice that an object exists for which it lacks a replica, and it will proceed to establish one of its own. But how is an object deleted? Simply deleting a replica will not do, since in the absence of additional mechanism *that is indistinguishable from object creation*: one node has a replica, another does not, so which is it to be? Does the replica represent a newly created object, or does the "missing" replica represent a recently deleted object?

Attempting to determine whether an object is newly created or recently deleted is futile in the absence of additional information. This *create/delete ambiguity* (first noted by Fischer and Michael in [FM82]) is resolved in conventional replication schemes by appealing to an authority; in optimistic replication, some other means must be used. In this chapter, we provide solutions for this and other problems typically encountered in optimistic replication.

3.1.1 File systems

The algorithms presented in this chapter are designed to provide for management and garbage collection of distributed, selectively replicated graph structures with associated resources. In practice, they have been extensively applied to the support of an optimistically replicated hierarchical filing system and accompanying name service for UNIX(see Chapter 2).

Consider the primary components of a typical UNIX file system. Files are hierarchically organized, with designated files (directories) containing the structural details (pathname components) which indicate a file's place in the hierarchy. The hierarchy is usually a restricted form of a directed acyclic graph.

Two types of "objects" are present, files and file names. For replication management purposes, each object can be treated independently, including independent consideration of a file and its names. In the algorithm model below, a UNIX file corresponds to a multiply-named logical object, while the file's names are considered to be singly-named objects in their own right.

Although we have applied these algorithms in the context of a standard UNIX file system, they can readily be used in other applications. For example, a distributed name service such as the DARPA Internet Domain Name System could directly use these algorithms to manage its databases.

3.1.2 Outline

The next section specifies the problem to be solved in more formal terms and presents a family of algorithms to address it. Section 3.3 presents correctness arguments which aid understanding of the algorithms. Algorithm applications are discussed in Section 3.4; an outline of related research in Section 3.5 concludes the chapter.

3.2 Algorithms

The task of a management algorithm is to support the propagation of changes to names and objects, and to identify and recover all resources supporting the existence of a logical object. This section presents a simple model of object and names, followed by several reclamation algorithms which address various combinations of object properties.

3.2.1 Model

Our model provides clients with a persistent storage service for a collection of entities called *objects*. A *logical object* is represented by a finite set of *physical object replicas*. Each object has a *replication factor* which defines the intended quantity and placement of replicas.[1] Clients access an object via a *logical name*, which is represented by a finite set of *physical name replicas*. Each name has a replication factor separate from the object it names and from other names for the object.

The system creates a new logical object by establishing a single physical object replica and a single physical name replica. Additional physical name and object replicas for this object are established asynchronously as indicated by the relevant replication factor. The first physical name replica to be established for a logical name is tagged with a unique value that distinguishes this particular usage of the name from all others; all physical name replicas for this logical name carry this same tag.

An object is initially created with one (logical) name. Some objects may be restricted to only the original name; other objects allow names to be added or removed at will. Each object replica maintains a reference count indicating the number of (local) name replicas which refer to it.[2] Name removal will leave an object inaccessible when no physical name replicas for the object exist. New names can only be added to an accessible object, so an inaccessible object is permanently inaccessible. Resources held by an inaccessible object are subject to reclamation.

Name removal is effected by marking a name replica 'deleted', which prevents its use in accessing an object. This indelible mark eventually propagates to other name replicas, but until then, the object may be accessible via unmarked name replicas. An object replica's reference count is decremented atomically with marking a name replica.

An additional name for an object may be established provided that the physical object replica referenced by the to-be-established initial name replica currently has a non-zero reference count.

The difficulty of replica management is determined (in part) by several issues:

[1] In this chapter, we use the term replica to include all of the resources at a node which are devoted to the (logical) object. Typically, this includes a copy of the object's "client data," as well as any replication or other bookkeeping meta-data associated with the object. Resources consumed by meta-data must be reclaimed as well as resources used by client data.

[2] A name replica is reflected in the reference count of exactly one replica.

- static versus dynamic naming
- object mutability
- equivalence of name and object replication factors
- static versus dynamic replication factor

The algorithms presented in the next several subsections vary in their ability to handle these issues, ranging from the simplest combination (fixed name, immutable object with equivalent static replication factors for both name and object) to the most difficult (dynamically named mutable object with non-equivalent dynamic name and object replication factors).

To aid clarity of discussion, we assume that no more than one replica is stored by any given node. The algorithms generalize directly to multiple replicas per node.

We make minimum assumptions about the available communications environment to assure successful operation of the algorithms in practice. All we require is that information be able to flow from any node to any other in the network over time if relayed through intervening nodes. More formally:

> nodes N_1 and N_m are *time flow connected* if there is a finite sequence of nodes $N_1, N_2, \ldots, N_m$ such that for $1 \leq i < m$, a message can successfully be sent from N_i to N_{i+1} at time t_i, and $t_i < t_{i+1}$.

We require that every pair of nodes is time flow connected starting at any time.

This property does not require, for example, that any pair of nodes be operational simultaneously, but it does mean that no relevant node can be down indefinitely.

We also assume that nodes are truthful: Byzantine behavior does not occur. Finally, history only moves forward: a node must never "roll back" from a reported state, so stable storage of any reported state must precede that report.

3.2.2 Basic two-phase algorithm

The basic two-phase algorithm is appropriate for the simplest kind of replicated object: static single name, immutable object, and equivalent fixed name and object replication factors. The task at hand is simply to garbage collect. Subsequent more difficult types of management tasks adapt this algorithm to accomplish their goals.

The basic reclamation algorithm proceeds in two phases at each node. The first phase begins executing at a node when the node learns the object is to be reclaimed, that is, when its (single) name replica is marked 'deleted.' This mark is then also placed on the object replica. Actual physical reclamation of the object replica (and name replica) will not occur until after the node completes its second phase of the algorithm. Figure 3.1 lists the basic two-phase algorithm in pseudo-code.

Each node concurrently executes the algorithm, and shares its progress with other nodes. Sharing improves the algorithm's efficiency, but more importantly, it enables the algorithm to cope with pathological communications failures.

3.2.2.1 Phase one

The first phase proceeds by composing a list of nodes that have their object replica marked deleted. In effect, each node is engaged in the same activity: collecting information about the deletion status of each object replica. A node completes its first phase when every replica is listed as marked deleted.

When two nodes cannot directly communicate, information propagates by way of intermediate nodes. Indirect communication is, in fact, an integral part of the algorithm: when nodes inquire about each other's status, algorithm status as well as deletion status is shared. The list of replicas marked deleted which is maintained by a node is shared with other nodes, who in turn incorporate the information into their own lists.

A node that has completed phase one has limited knowledge about the status of other nodes. It knows that all have marked their name replicas and object replicas deleted, and thus have themselves begun executing phase one of the algorithm. However, a node at this stage makes no assumptions about the knowledge other nodes have of *it*. It is quite possible that no other node is aware that the node in question has marked its replica deleted, as the flow of information is not guaranteed to be a two-way exchange at any step.

3.2.2.2 Phase two

Immediately upon completing phase one, a node begins executing phase two. In this phase, a node compiles a list of nodes that it learns have finished phase one. The first node placed on this second list is, of course, itself: phase two began at this node precisely because it had finished phase one. As with the earlier phase, phase two at this node is complete when all nodes are listed. The same style of

```
/* variables and data structures:
   Let set R := replication factor,
        r,s   element drawn from R,
        self  is element of R,

        P1[]  binary vector of size |R|,
        P2[]  binary vector of size |R|,
        R[]   binary vector of size |R|.
*/
begin:  while (my name replica is not
                        marked deleted)
                { donothing; }
        mark my object replica deleted;

        P1[r] := 0, for all replicas r;
        P2[r] := 0, for all replicas r;

phase1: P1[self] := 1;
        while (P1[r] == 0 for any r) {
                R[r] := 0, for all r;
                choose some r to query;
                ask r for its P1 vector;
                if r responds {
                    R[] := r's response;
                    foreach (R[s] == 1)
                       { P1[s] := 1; }
                }
        }

phase2: P2[self] := 1;
        while (P2[r] == 0 for any r) {
                R[r] := 0, for all r;
                choose some r to query;
                ask r for its P2 vector;
                if r responds {
                    R[] := r's response;
                    foreach (R[s] == 1)
                       { P2[s] := 1; }
                }
        }

postlude:
        reclaim object replica resources;
        reclaim name replica resources;
```

Figure 3.1: Basic two-phase algorithm

list sharing utilized in phase one also occurs in phase two.

Nodes placed on a phase two list are those that know that all replicas are marked deleted. A node with a complete phase two list therefore knows that all nodes know all replicas are marked deleted. This global state is vital to providing "once reclaimed, never re-established" behavior: it allows a node (finished with phase two) to reclaim all local resources devoted to the replicated object and to forget about it entirely, secure in the knowledge that the replica will never be re-established in response to a query from another node about the object.[3]

A node that is striving to finish phase two might query a node which has already reclaimed its resources and forgotten about the object. The query response will indicate that no such object is known, which the inquirer will (correctly) interpret to mean that the queried node has completed phase two. The inquiring node uses this inferred status to complete its own second phase, and proceed with reclaiming its object and name replicas' physical resources.

3.2.2.3 One phase is not enough

The algorithm's first phase ensures that all nodes with replicas are aware that the object's resources are to be reclaimed. This property guarantees that no replica will survive the reclamation effort without having been aware that reclamation was in progress. The second phase guarantees that all portions of the distributed algorithm will terminate despite barriers to information flow that are formed as nodes reclaim their replicas' resources.

In order to appreciate why one phase is insufficient, consider a hypothetical one phase algorithm and its execution in a particular class of network configuration behaviors. In the imagined algorithm, a node reclaims a replica's resources upon learning that all extant replicas are aware that reclamation is in progress. The network configuration of interest (see Figure 3.2) is composed of a group of well connected nodes and two nodes which are weakly connected to the group and very weakly connected to each other.

Suppose that an object is initially created at Node Z, with replicas to be established at all nodes A, B, ... Y, Z. Soon after establishing a replica, Node A determines that the object should be reclaimed. According to the algorithm,

[3] "Once reclaimed, never re-established" behavior is important for both practical and theoretical reasons: resource allocation and deallocation is costly and should be done only when necessary; and, removing the possibility of re-allocation greatly simplifies algorithm termination arguments. Note that the "logical deletion" design hides re-establishment issues from clients, so it is strictly an internal systems issue.

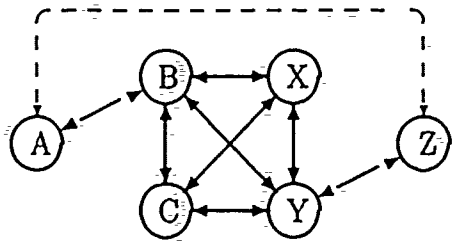


Figure 3.2: One phase network example

Node A notes that it is self-aware of reclamation, and begins the process of acquiring knowledge about other replicas' reclamation status. Suppose that the link between nodes A and B is the only remaining (albeit weak) link from Node A to the others.

Now consider Node B's possible perspectives upon receiving an inquiry from Node A (which contains the information that reclamation is in progress): Node B is either aware of the object already (because a replica exists at Node B), or it is not aware (no replica exists at Node B).

In the first case (Node B is already aware of the object), Node B notes that reclamation is in progress and Nodes A and B are cognizant of it. Node B, in turn, attempts to contact other nodes. Suppose the well connected group of nodes (B - Y) rapidly succeeds in learning that reclamation is in progress, and even manages to get a response back from Node Z acknowledging that reclamation is in progress. Further suppose that Node B is the first node to learn that every node is aware of the intent to reclaim. Node B therefore reclaims its resources and forgets entirely about the object (including the fact that its replica was reclaimed).

Continuing the scenario, when Node B receives the initial inquiry from Node A, it replies quickly. Unfortunately, congestion on the link causes the reply to be lost. Node A eventually sends another message to Node B inquiring about Node B's reclamation status, since it failed to get a response to the first message. By the time Node B receives the second message, its replica is already reclaimed. This scenario forces consideration of case two: Node B is unaware of the object.

Another way in which Node B might be unaware of the object is that Node B has never learned of the object before receiving the inquiry from Node A. (Perhaps Node A learned of the object directly from Node Z via the very weak link between them.) From Node B's perspective, these two situations are indistinguishable, yet its response must differ for the two scenarios: in one, Node B must establish a replica (whose body may be empty) to support indirect communication to other nodes about the reclamation initiated by Node A; in the other, all nodes are (or

were) aware of reclamation, and do not need (or want) to re-establish replicas.

Failure to support indirect communication that may be critical to algorithm termination is unacceptable. It is also unacceptable to simply re-establish replicas just in case indirect communication support is needed: re-establishment in the above scenario is a side-effect of an event (successful transmission of a message) whose frequency is both unbounded (by the algorithm) and may not contribute to progress towards termination.

The two-phase nature of our algorithm provides an ignorant node with the ability always to distinguish "never knew" from "forgotten". An ignorant node which receives a phase one message correctly concludes "never knew", and establishes a replica to provide support for indirect communication. An ignorant node concludes "forgotten" upon receiving a phase two message, and does not establish a replica. (An ignorant node's reply to a phase two inquiry–"I know nothing"–implicitly means "I finished phase two, and so can you," which is all the inquirer needs to know to reclaim its replica's resources and terminate.)

3.2.3 Intermediate algorithm

The basic algorithm in the previous section applies to fixed-name, immutable objects with identical static name and object replication factors. In this section we relax the first two constraints to allow dynamic naming and object updates, while continuing to require name and object replication factors to be both identical and static.

Dynamic naming and object mutability each complicate the reclamation problem, and the combination of the two is especially difficult. Dynamic naming introduces a global stable state detection problem, while object mutability requires special mechanisms to prevent inadvertent data loss.

3.2.3.1 Dynamic naming

A necessary, but not sufficient, condition for object reclamation is that the object have no names. In our model, 'no names' means that every name replica referring to an object replica has been marked deleted.

In the basic two-phase algorithm, object reclamation inevitably follows name removal; the two phases ensure that all replicas will be reclaimed, exactly once. In the dynamic naming case, it is much harder to determine whether or not reclamation is to occur: names may be added or removed at any node at any time, since optimism allows unsynchronized concurrent updates across non-communicating

nodes. When a name is added at one node concurrently with a name removal at another node, a transient situation may arise in which a node has no names for an object for a time, until the new name propagates to that node. During this time, reclamation of the 'nameless' object replica would be premature, even though it has a zero-valued reference count.

Premature reclamation must be avoided because of the potential for data loss. Concurrent update, name creation, and name removal together with the non-atomicity of name and object propagation leave open the possibility that the only object replica reflecting an update could temporarily have a zero-valued reference count. Such a replica must not be prematurely reclaimed.

Although a single replica's zero-valued reference count may be a transient condition, when all replicas have zero references a global stable state [CL85] exists. The problem, then, is to detect that all object replicas simultaneously have zero-valued reference counts in an environment when simultaneous or pseudo-simultaneous queries of all nodes is not feasible.

We provide an adaptation of our basic two-phase algorithm which exploits the rules governing name additions to achieve a relatively inexpensive, fully distributed mechanism for determining the global zero-valued reference count stable state. The adaptation requires that each object replica maintain a monotonic counter in parallel with the reference counter, and that the algorithm compile and distribute a vector of these new counters.

The new counter is incremented atomically with the reference counter, but it is never decremented. It functions as a 'total name counter' to reflect the number of name replicas at this node which have referred to the object replica. The total name counter from each replica is used to determine that a zero-valued reference counter has been quiescent between interrogations.

3.2.3.2 Algorithm for dynamic naming

This intermediate two-phase algorithm is triggered at a node when the object replica's reference count is zero. In the first phase, two parallel vectors are maintained. One vector indicates which replicas have reported a zero-valued reference count, and the other contains the total name counter value reported by those replicas.

A node has completed its first phase when all replicas are listed with total name count values recorded in the parallel vectors. This also implies that each replica reported a zero-valued reference counter. These parallel vectors are shared with other nodes executing the algorithm.

The second phase proceeds similarly, with two parallel vectors of total name count values and report indicators. In this phase, the total name count values recorded reflect a replica's total name count value at some point after the queried replica has finished phase one.

As a node is collecting values in the second phase, it compares the newly reported values with those recorded in its phase one vector. If any discrepancy is discovered (i.e., the corresponding values are not identical), the algorithm aborts, initializes its vectors, and restarts phase one. This abort occurs when the transient behavior described above occurs.

A node finishes its second phase when all replicas have reported values to it, and the values are identical to those collected in the first phase. At this point, all object replicas are guaranteed to be permanently inaccessible.

3.2.3.3 Mutability

As presented, the intermediate algorithm will determine that an object is globally inaccessible. A further condition is necessary (and sufficient) to allow physical reclamation to proceed: data must not be lost inadvertently as an unavoidable consequence of optimism. We are not concerned here with the kind of 'inadvertent loss' that results when a client mistakenly removes a name, but with the consequences of concurrent update and name removal.

Consider a scenario with one object, two names, three replicas, and three clients. (Imagine a journal paper draft, with three collaborating colleagues.) Suppose that each of the nodes is isolated, but optimism allows each author to continue working. One author makes revisions to his object replica. Each of the other two authors decides (differently) that one of the two names is superfluous, and removes it. Each of the clients will be understandably disappointed if the object is reclaimed (since it eventually will be declared globally inaccessible), especially the one who updated it.

Our approach to the general problem of *remove/update conflicts* is to assume that name removal is undertaken in the context of an object replica. We invest the name removal operation with the additional semantics that a client wishes object reclamation (when no names exist) if no other object replicas are newer than (or in conflict with) the object replica which is initially affected by the name removal.

To accommodate the additional semantics, the reclamation algorithm must determine which of the object versions represented by the replicas is the latest, and which is the latest version to provide a context for name removal. (Opti-

mism also introduces the possibility that no 'latest' version exists, such as when unsynchronized concurrent updates occur to distinct replicas, thus generating an update/update conflict.) After identifying the latest object version and removal context version, it is trivial to decide whether a remove/update conflict exists.

Version identification and context recollection can be readily accomplished with *version vectors*, which provide a multi-dimensional version numbering technique for replicas [PPR83]. We augment the object replica model with two data items: a 'current' version vector, and a 'removal context' version vector. The current version vector always identifies the current value of the object replica. The removal context vector is replaced by a copy of the current version vector when a name removal operation is issued with this object replica providing context. Each replica's removal context vector will be checked to see that no remove/update conflict exists.

3.2.3.4 Remove/update conflict algorithm

It is easy to modify the intermediate two-phase algorithm to collect and compare the various vectors and determine if a remove/update conflict exists: each instance of the algorithm can collect (and share) sets of vectors, and perform the appropriate comparisons when the sets are complete. This approach, however, imposes quadratic storage and message size complexity upon each instance of the algorithm.[4]

Linear storage complexity can be achieved by exploiting the (partial) ordering of version vector values. Instead of collecting each replica's version vector values, an algorithm instance can retain only the greatest (latest) vector value encountered, along with a vector indicating which replica's vectors have been consulted and whether the vectors conflict with the greatest values seen to this point in the algorithm's execution.

The linear optimization is not free, however. A two-phase consultation scheme is required to collect the vectors and correctly assert that a particular vector value is greatest, or that no value is greatest due to conflicting versions. As it happens, these two phases can be executed in parallel with the two-phase algorithm that determines global inaccessibility, so the cost is effectively eliminated.

Once global inaccessibility and remove/update conflict status are determined, a decision can be made whether to reclaim an object replica's resources. If a remove/update conflict is discovered, reclamation will not occur; proper action at this point is application dependent. (An example is described in a later section.)

[4]Each version vector is of length n, of which n must be collected in each set ($n = |\text{replicas}|$).

Figures 3.3 and 3.4 show the intermediate algorithm.

3.2.4 Advanced two-phase algorithm

The previous algorithms each assume that object and name replication factors are fixed at creation time, and are identical. In practice, these constraints are not attractive. Changing circumstances of network behavior or object usage may necessitate adding, deleting, or moving replicas, which can not be usefully predicted when an object is created. It should also be possible to change an object's replication factor without directly affecting object names.

Note that an object (or name) replication factor is itself a replicated data structure which is used to manage other replicated data structures. The version vector technique used to manage updates to replicated data can not easily be applied to managing updates to version vectors themselves.

Our system supports an approximation to an ideal flexible replication factor mechanism: a replication factor can grow to be very large (2^{32} replicas), with masks used to 'shrink' a replication factor. One mask is used to indicate that particular replicas should be (irrevocably) ignored during algorithm execution. The second mask permits an object replica to avoid the expense of storing the object itself any further, but the 'skeletal' replica must continue to participate in algorithm execution. In short, a replication factor monotonically increases in physical size, with adjustments available to reduce the actual number of physical copies of a client's data which are maintained.

Increasing a replication factor is straightforward. Any replica's replication factor can be augmented simply by adding a (globally unique) replica identifier to its list of replicas. A replica can form a new replication factor while executing the one of the two-phase algorithms by taking the union of its replication factor and that reported by another replica.

A replication factor's 'ignore' mask provides a way for a replica to be forever ignored. This is especially useful when recovery of a destroyed replica is impossible or too expensive. Indicating that a replica is to be ignored is an irrevocable action. Like an increase in replication factor, a new ignore mask is formed by taking the union of the local mask and one reported by another replica.

The 'skeletal' mask indicates which object replicas don't actually store any client data. This mask is maintained in an optimistic fashion, but without conflict detection: mask updates cause a new timestamp to be generated for the mask; the mask with the latest timestamp is deemed to be correct.

3.2.4.1 Algorithm

Very few changes need to be made to the intermediate two-phase algorithm to support dynamic name and object replication factors. Each replication factor must support two additional parallel data structures (the masks), and the algorithm must check reported replication factors for changes. Care must be exercised, though, when increasing a replication factor not to violate the semantics of an in-progress reclamation algorithm.

Our two-phase algorithms have two critical points: when a node finishes phase one, it believes that all replicas have been consulted; and when a node finishes phase two, it believes that all replicas have finished phase one.

While a node is currently in phase one, its replication factor can be augmented safely because every other node must consult it at least once more, during phase two. When this node is consulted, other nodes will learn about the additional replica(s). But a replication factor must not be augmented to create a new replica when the 'source' replica's node is in phase two.

For brevity, we do not show these minor algorithm modifications in a separate figure.

3.3 Correctness discussion

The basic two-phase reclamation algorithm is *correct* if and only if these conditions are satisfied:

- object reclamation occurs if, and only if, the object is globally inaccessible
- for each replica of an inaccessible object, reclamation occurs exactly once, in finite time
- all algorithm executions terminate in finite time
- all algorithm executions are free from deadlock

We first show that reclamation occurs *if* an object is inaccessible, followed by the *only if* direction. We then show that reclamation occurs exactly once in finite time by proving that it occurs at least once, and at most once.

3.3.1 Reclamation *if* inaccessible

The "information flow" requirement governing network behavior ensures that it is possible for each node to learn of status changes at every other node. Since each node periodically uses the propagation protocol to incorporate other replicas' status changes into its own replica, and since all replicas are guaranteed to be available at the same time, each node will, in fact, learn in finite time of the status of every other replica. Therefore, every logical name deletion will eventually be reflected at every node, as each name replica will be indelibly marked deleted.

Following the deletion of every name for an object, in finite time all name replicas will be marked deleted. Each object replica will, in turn, have a zero-valued reference count, and be inaccessible.

The first phase of the algorithm simply collects the information that, when consulted, each replica was inaccessible. The second phase similarly collects information from each node. By the previous argument, each node is guaranteed to learn the desired information. At the conclusion of executing its second phase, a node reclaims its resources. Since each node is guaranteed to finish its phases if the object is inaccessible, each node will reclaim the resources consumed by the object.

```
/* major changes to basic algorithm
   show asterisks in first column.  */

/* new vectors:
        NCR  total name count response
        NC1  total name count, phase1
        NC2  total name count, phase2
        NV   total name count validation
        VV   replica's version vector
        VVR  version vector response
        SVV  saved version vector response
        RC   removal context vector
        RCR  removal context response
   new scalars:
        C    reference count response
        RU   remove/update conflict flag
*/

begin:  while (my ref-counter non-zero)
            { donothing; }
        reset all elements of vectors:
        P1, P2, NCR, NC1, NC2, NV
        RU := 0;

phase1: P1[self] := 1;
        while (P1[r] == 0 for any r) {
                NCR[r] := 0, for all r;
                choose some r to query;
*               ask r for its C, NC1, P1;
*                   VV, RC
*               if r responds with C==0 {
*                   NCR[] := r's NC1;
*                   NV[] := r's P1;

*                   foreach (NV[s] == 1) {
*                       NC1[s] := NCR[s];
                        P1[s] := 1;
                    }

*                   VVR := r's VV;
*                   RCR := r's RC;
*                   if (VVR >= VV)
*                       { SVV := VVR; }
*                   if (RCR >= RC)
*                       { RC := RCR; }
                }
        }
```

Figure 3.3: Intermediate algorithm, phase one.

```
phase2: P2[self] := 1;
        while (P2[r] == 0 for any r) {
             NCR[r] := 0, for all r;
             choose some r to query;
*            ask r for its C, NC2, P2;
*                VV, RC
*            if r responds with C==0 {
*                NCR[] := r's NC2;
*                NV[] := r's P2;
*                foreach (NV[s] == 1) {
*                  NC2[s] := NCR[s];
*                  P2[s] := 1;
*                  if (NC1[s] != NC2[s])
*                      goto begin;
                 }
*                VVR := r's VV;
*                RCR := r's RC;
*                if (VVR conflicts SVV or
*                    RCR conflicts RC)
*                    { RU := 1}
*                if (VVR >= VV)
*                    { SVV := VVR; }
*                if (RCR >= RC)
*                    { RC := RCR; }
             } else if (C > 0)
                 { goto begin; }
        }
postlude:
        if (RU == 0) {
          reclaim object replica resources
          reclaim name replica resources
        } else {put object into orphanage}
```

Figure 3.4: Intermediate algorithm, phase two.

3.3.2 Reclamation *only if* inaccessible

We argue by contradiction. Suppose reclamation of an object replica occurred without the object being inaccessible. Therefore, some object replica must have a non-zero reference count at the end of a node's second phase.

But, the algorithm's first phase demonstrated that each replica had a zero-valued reference count (though not necessarily simultaneously), and the second phase ensured that each replica's reference count had not changed between the first and second reference count queries. Since the second set of queries strictly followed the first set, a point in time must exist at which all replicas were simultaneously inaccessible. Global inaccessibility is a global stable state, by the restrictions placed on additional name creation. Therefore, a non-zero object replica reference can not exist, which contradicts the hypothesis.

3.3.3 Reclamation *exactly once*

If the object is inaccessible, each replica will be reclaimed at the end of its node's execution of the algorithm, as per the above arguments. Therefore, each replica is reclaimed at least once.

Multiple reclamation requires multiple establishment of a replica. Replica establishment occurs when a node without a replica receives a message that indicates that the receiver is intended to have a replica and there is no indication in the message of the replica's prior existence. Therefore, to re-establish a replica, a node which has already reclaimed its replica must receive a message about the object which does not indicate that the replica is known to have existed.

It suffices to hypothesize that such a message is received, and then prove that such a message cannot arrive. We do so by classifying all messages and showing that none of the types which could cause replica establishment will be received after reclamation.

Every message about an object replica implicitly indicates a "phase" of algorithm execution. In addition to phase one and phase two messages, nodes routinely send status query and response messages to learn of object updates when the algorithm is not executing. For convenience, consider these routine messages to be "phase zero" messages.

When a node without a replica receives a message, its decision whether to create a replica is based on the phase of the sender:

zero A phase zero message contains no indication whether the receiving node ever had a replica. Therefore, a replica must be established.

one A phase one message may or may not indicate that the replica has ever existed. If it does not indicate that the replica existed, a replica must be established. If it indicates that a replica once existed, an anomalous condition has been encountered. (See discussion below.)

two A prerequisite for entering phase two is that all replicas have been consulted, which implies that all replicas exist. Therefore, the replica has previously existed, been reclaimed, and must not be re-established.

A node which has already reclaimed its replica normally expects to receive only phase two messages, because a condition of phase two completion is to determine that all other nodes have finished phase one. Since phase two messages can not cause a replica to be re-established, only the receipt of phase zero or phase one messages after reclamation might cause a replica to be established again.

Phase zero or phase one messages received by a node which has completed phase two and reclaimed its replica could only have been sent *before* the sender began phase two. Such messages have been delayed in transit, long enough for the sender to finish phase one and the receiver to finish phase two.

The algorithm is resilient to delayed messages which are received within the next phase: phase one messages received by a node in the midst of phase two are quite normal, as are phase zero messages received during phase one. It is only when message delay exceeds one phase that replica re-establishment might occur.

We assume that message delays does not exceed the time required for one complete phase. If this bound is invalid, algorithm execution can be artificially slowed to increase the length of a phase until a valid bound is achieved. It is, therefore, feasible to prevent phase zero or phase one messages from arriving after reclamation occurs.

The hypothesized message received after a replica has been reclaimed must be from one of the three phases, but since delayed phase zero and phase one messages can be prevented and phase two messages do not cause replica establishment, no message which could cause replica establishment will be received. This contradicts the hypothesis that such a message might be received, and so replica re-establishment (and subsequent reclamation) after an initial reclamation is not possible.

3.3.4 Termination

We show that the algorithm terminates by defining a partial order on the possible states of a node during the algorithm's execution, and showing that all state tran-

sitions are monotonic with respect to this order. (We showed above that sufficient transitions will occur, based on the finite time information flow assumption.)

A node's algorithm execution status is primarily determined by the list compiled in each phase of replicas consulted. The set of valid list values comprises all subsets of the (finite) set of replicas indicated in the object's replication factor. A partial order based on cardinality can then be defined over these subsets.

A state transition (list change) is defined in the algorithm to be a set union operation, which is monotonic over the partial order. A partial order is acyclic, so all algorithm state transitions are acyclic. Progress towards termination is guaranteed, unless deadlock occurs.

The intermediate algorithm occasionally aborts and restarts. The only circumstance in which abort occurs (a mismatch of total name count vectors) is bounded in occurrence by the product of the number of names and the cardinality of the object's replication factor. Since the number of aborts at a node is bounded, some algorithm execution will not abort, and so the above termination argument holds.

3.3.5 Deadlock-free

We show that the protocol is free from deadlock by developing a *waits for* graph model and proving that it is acyclic for all algorithm executions.

Recall that the propagation protocol underlying state transitions is non-blocking, so a node is never blocked waiting for a particular response from another node. It therefore suffices to consider the algorithm's behavior at the higher level of phase transitions, where 'waiting' does occur.

Define a total order over the states 'accessible', 'phase one', 'phase two', and 'reclaimed' such that accessible < phase one < phase two < reclaimed.

A node transitions from accessible to phase one when its replica becomes inaccessible, and from phase one to phase two when it learns that all nodes have transitioned to phase one. It transitions from phase two to reclaimed upon learning that all nodes have transitioned to phase two.

With the exception of the initial transition from accessible to phase one, a node waits for all other nodes to reach the same state as itself, before transitioning to a later (fully ordered) state. Therefore, a node only waits for "lesser" nodes; since "lesser" is acyclic, no cycles can occur in the waits-for graph and so the protocol is deadlock-free.

3.4 Applications and observations

Which two-phase algorithms are appropriate for managing a UNIX file system? UNIX files are mutable, dynamically named objects, so at least the intermediate algorithm should be used for them. File names (directory entries) are simple objects which can be managed with the basic two-phase algorithm.

While the intermediate algorithm is a sufficient base upon which to construct a usable file system, the additional cost of implementing and using the advanced algorithm (with flexible replication factors) is negligible. Ficus incorporates the advanced algorithm to manage its files.

The advanced algorithm is also used to support the name service that connects subtrees together to form a large connected hierarchical filing environment. This name service plays a role similar to NIS (Yellow Pages) in NFS, or volume support in AFS. The implementations of these two applications (file hierarchy and volume hierarchy) are common, so multiple facilities were not required.

3.4.1 Directed acyclic graphs

The UNIX filing environment is a simple directed acyclic graph (dag) structure. These algorithms may be applied to an arbitrary graph structure as well, so long as there are no disconnected self-referential subgraphs. Additional mechanism is needed to handle that case.

In fact, modest mechanism beyond that discussed in this chapter is required even to handle dags. That is because the discussion was cast in terms of a single logical object. The additional facilities are simple, and discussed in [Guy87].

3.4.2 Performance

Performance of these algorithms is, of course, important. A suitable measure is the number of messages that must be exchanged in order to cause a set of n nodes with replicas to reach agreement. One would expect that the worst case could be expensive, since the underlying minimum communications assumptions do not allow a stylized pattern of interaction always to be employed. The worst case indeed requires $O(n^2)$ messages, as most nodes talk to most of the other nodes to complete each phase.[5]

[5]In each phase, in the worst case, a first node pulls from the $n-1$ other nodes to become knowledgeable. A second node then pulls from the remaining $n-2$ unknowledgeable nodes, and then the first, knowledgeable one. The third node pulls from the remaining $n-3$ unknowl-

However, in practice the situation is far better, since we can communicate in a stylized manner most of the time. As a simple example, if the nodes order their communications in a ring, then a total of $3n - 1$ messages are used.[6]

3.5 Related work

Our work is related to several areas of research: the "gossip" problem, which has received substantial formal treatment; optimistic file systems, including LOCUS, Coda, and Deceit; optimistic "dictionaries" (directories) in file systems; and, distributed garbage collection.

In the gossip problem, each node in a graph must communicate a unique item to every other node in the graph. A variety of papers have appeared in the twenty years of its study [HHL88], yielding complexity results under varying communications assumptions.

Heddaya, Hsu, and Weihl [HHW89] used a two-phase gossip protocol to manage distributed event histories of updates to object replicas. A timestamp vector is used to determine when history elements may be safely discarded. Their solution does not address the problem of completely forgetting that a history exists, but only forgetting items in the history.

LOCUS [WPE83, PW85] is an intellectual ancestor of the Ficus file system which incorporates these algorithms. LOCUS system prototypes incorporated more limited replica management algorithms, from which the algorithms presented here are descended.

The Coda project [SKK90] has similar goals to our own Ficus work and bases its replica management on the LOCUS version vector [PPR83] mechanism and an earlier draft of this work [Guy87].

Fischer and Michael [FM82] proposed recasting the replicated directory maintenance problem as a replicated "dictionary" problem, with slightly (but significantly) different semantics. A timestamp vector was used to infer from a comparison of two dictionary replicas which entries had been inserted and which had been deleted.

---

edgeable nodes, and then one of the knowledgeable ones. Thus each phase requires $(n-1)+(n-1)+(n-2)+(n-3)+\ldots+1 = \frac{(n+1)n}{2} - 1$ pulls, and there are two phases. Thus, $n^2 + n - 2$ pull messages are required.

[6] Assume that a single message is active in the ring at any time. This ever-changing message flows around the ring three times. Phase one of the algorithm begins for all nodes in the first round trip. Phase one completes and phase two begins for all nodes during the second round trip. Phase two completes for all nodes during the third round trip.

Allchin [All83] and Wuu and Bernstein [WB84] expanded upon Fischer and Michael's approach to use two-dimensional timestamp matrices to reduce the number of messages exchanged, with small variations in semantics.

None of these works addressed the general problem of reclaiming resources of named replicated objects; they were concerned with "dictionary entries" as isolated entities.

Wiseman's survey [Wis88] of distributed garbage collection methods includes several techniques based on reference counting, but none are designed for use on replicated objects, and none are directly applicable to imperfectly connected networks.

CHAPTER 4

Conclusions

This final chapter summarizes the research and its results. It also spotlights several areas in which future research is expected to be fruitful.

4.1 Summary and conclusions

This dissertation presents a new architectural paradigm, *stackable layers*, as a methodology for designing and implementing a wide range of filing services for operating systems. Coupled with a new volume management strategy and a novel two-phase algorithm family, the stackable layers technique has been used to construct a large scale replicated file service. The Ficus replicated file system is in experimental use; neither layering nor replication demonstrate unacceptable performance degradation.

This work reinforces the fact that today's file management systems are exceedingly complex pieces of software. The challenges presented by the globalization of computer networks will make them all the more so. Many features will have to be added to traditional filing services: selective replication, data security, user authentication, and type conversion among heterogeneous storage conventions are but a few examples. The stackable layers architecture provides a methodology for extensibility which is crucial for the advancement of distributed file system technology. As a case study, Ficus demonstrates that the stackable architecture is logically feasible and can, with care, be made to perform satisfactorily.

All of the experience gained in the course of this research supports the view that optimistic replication is very attractive, whether it is merely between one's home computer and the office network, or in a very large corporate information system. High performance, high availability, scalable distributed computing service is feasible; it is hoped that the facilities described in this work will make that high quality service commonplace, as they require no special hardware and can easily be added to many existing systems. Many applications should benefit from the ease with which the basic optimistic replication reconciliation service can be retargeted beyond its initial use for directory management, as is shown

by its successful use to manage Ficus' replicated volume location tables.

The use of stackable layers as the framework for the Ficus architecture has been an unqualified boon. The ability to leverage a common filing service directly permitted one to focus on development of new functionality inherent in the replication service, and avoid much of the traditional cost of building an ideal substrate at the outset. The modularity afforded by the architecture, along with the ability of the transport layer to map operations across address space boundaries, allowed new layers to be developed and debugged in user space, and then moved into the kernel only after they were working. This substantially simplified the testing and debugging enterprise. Layers can indeed be transparently inserted between other layers, and even surround other layers. A replication service can be added to a layer stack without modifying existing layers, and yet perform well.

The availability of a general reconciliation service is also very useful. Usually, one must deal with the many boundary and error conditions that occur in a distributed program with a considerable variety of cleanup and management code throughout the system software. Instead, in Ficus failures may occur more freely without as much special handling to ensure the integrity and consistency of the data structure environment. The reconciliation service cleans up later. For example, volume grafting was made considerably easier by the (easy) transformation of its necessarily replicated data structures into Ficus directory entries. No special code was needed to maintain their consistency. There is thus reason to believe that services such as those provided by Ficus will be of substantial utility in general, and easy to include as a third-party contribution to a user's system.

The two-phase algorithm family addresses the heretofore unsolved garbage collection problem for replicated data structures. It may also be useful in similar contexts, such as cleaning up storage used in reliable broadcast protocols.

The final and perhaps most significant conclusion is that this work opens up a number of relevant research directions where one can expect to make rapid progress, and provides the tools to investigate them. The following section provides several suggestions for future work.

4.2 Future research directions

The most prominent area of immediate future research is to demonstrate that large scale is workable in practice. Further, a variety of additional layers are of interest. General service areas include performance tuning, security, and databases.

4.2.1 Performance tuning

Two immediately useful layers would be ones that take measurements and cache data.

A measurement layer could easily and transparently collect the kinds of performance data normally sought after when measuring the cost of a particular service. A simple measurement layer that counts vnode operations can usefully be placed between any other layers on a stack, without regard for what operations and semantics those layers support. A prototype measurement layer constructed for an early Ficus implementation demonstrated the feasibility of collecting data in this way, but more complex measurement layers need to be constructed, for example, to gather trace data and operation duration.

File system services are often initially designed with functionality and good performance as primary goals. It is not always clear from the outset, however, which portions of a service design will benefit from caching or other enhancements. A general purpose caching layer would be of great utility in experimenting with performance improvements. Service-specific caching layers can also be expected to appear.

4.2.2 Security

A large scale file system will be vulnerable to security problems to an unprecedented degree. The use of encryption and authentication services is clearly desirable, but the appropriate means is as yet unknown. A related issue is that of accounting for resource usage; the current state of affairs (no charges) soon may not be acceptable.

4.2.3 Databases

It remains to be demonstrated (in practice) that database-oriented services such as serializable concurrency control and transactions can be constructed on an optimistic replicated file service base, and can further provide the high-performance data transfers that are required by this class of clients.

4.2.4 Typed files

If the stackable layers approach were to be applied at a single file granularity, not simply a volume granularity, it seems feasible to provide support for typed files. Each file might maintain as an attribute the particular services used to produce

the file; the attribute would then later be used at file open time to construct a custom, "typed" file service for that one file.

References

[ABG86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for UNIX Development." In *USENIX Conference Proceedings*, pp. 93–113. USENIX, June 1986.

[ABG87] Rafael Alonso, Daniel Barbará, Hector Garcia-Molina, and Soraya Abad. "Quasi-copies: Efficient Data Sharing for Information Retrieval Systems." Technical Report CS-TR-101-87, Princeton University, September 1987.

[AD76] P. A. Alsberg and J. D. Day. "A Principle for Resilient Sharing of Distributed Resources." In *Proceedings of the International Conference on Software Engineering*, pp. 562–570, October 1976.

[All83] James E. Allchin. "A Suite of Robust Algorithms for Maintaining Replicated Data Using Weak Consistency Conditions." In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

[App85] Apple Computer, Inc. *Inside Macintosh.* Apple Computer, 1985.

[AR85] J. A. Anyanwu and Brian Randell. "Update and Merge of Partitioned Distributed Systems." Technical Report 204, University of Newcastle Upon Tyne, 1985.

[Bar78] Joel F. Bartlett. "A NonStop operating system." In *Proceedings of the Eleventh International Conference on Systems Sciences*, pp. 103–117. Society for Computer Simulation, January 1978.

[BDS84] Joshua J. Bloch, Dean Daniels, and Alfred Z. Spector. "Weighted Voting for Directories: A Comprehensive Study." Technical Report CMU-CS-84-114, Carnegie-Mellon University, Pittsburgh, PA, 1984.

[Ber85] Arthur J. Bernstein. "A Loosely Coupled Distributed System for Reliably Storing Data." *IEEE Transactions on Software Engineering*, 11(5):446–454, May 1985.

[BG85] Daniel Barbará and Hector Garcia-Molina. "Mutual Exclusion in Partitioned Distributed Systems." Technical Report CS-001-346, Princeton University, July 1985.

[BGS86] Daniel Barbará, Hector Garcia-Molina, and Annemarie Spauster. "Protocols for Dynamic Vote Reassignment." In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pp. 195–205, August 1986.

[Bir85] Kenneth P. Birman. "Replication and Fault Tolerance in the Isis System." Technical Report TR 85-668, Cornell University, March 1985.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. "A Logic of Authentication." *ACM Transactions on Computer Systems*, **5**(1):47–76, February 1987.

[BJS86] Kenneth P. Birman, Thomas A. Joseph, Frank Schmuck, and Pat Stephenson. "Programming with Shared Bulletin Boards in Asynchronous Distributed Systems." Technical Report TR 86-772, Cornell University, August 1986.

[BK85] Barbara T. Blaustein and Charles W. Kaufman. "Updating Replicated Data during Communications Failures." In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pp. 49–58, August 1985.

[BMP87] Walter A. Burkhard, Bruce E. Martin, and Jehan-François Pâris. "The Gemini Replicated File System Test-bed." In *Proceedings of the Third International Conference on Data Engineering*, pp. 441–488. IEEE, February 1987.

[Bre86] O. P. Brereton. "Management of Replicated Files in a UNIX Environment." *Software–Pratice and Experience*, **16**(8):771–780, August 1986.

[BY87] F. B. Bastani and I-Ling Yen. "A Fault Tolerant Replicated Storage System." In *Proceedings of the Third International Conference on Data Engineering*, pp. 449–454. IEEE, February 1987.

[CL85] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems*, **3**(1):63–75, February 1985.

[CM89] David R. Cheriton and Timothy P. Mann. "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance." *ACM Transactions on Computer Systems*, **7**:147–183, May 1989.

[Cro87] Stephen D. Crocker. "The Origins of RFCs." Network Working Group Request for Comm ts: 1000, August 1987.

[Dav84] Susan B. Davidson. "Optimism and Consistency in Partitioned Distributed Database Systems." *ACM Transactions on Database Systems*, 9(3):456–481, September 1984.

[DB85] Dančo Davčev and Walter A. Burkhard. "Consistency and Recovery Control for Replicated Files." In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 87–96. ACM, December 1985.

[Dij67] Edsgar W. Dijkstra. "The structure of the THE multiprogramming system." In *Proceedings of the Symposium on Operating Systems Principles*. ACM, October 1967.

[Dij68] Edsgar W. Dijkstra. "Complexity controlled by hierarchical ordering of function and variability." Working paper for the NATO conference on computer software engineering at Garmisch, Germany, October 1968.

[Edi86] Judy Lynn Edighoffer. "Distributed, Replicated Computer Bulletin Board Service." Technical Report STAN-CS-86-1133, Stanford University, June 1986.

[EF83] Carla Schlatter Ellis and R. A. Floyd. "The ROE File System." In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 175–181, October 1983.

[Ell77] Clarence A. Ellis. "Consistency and Correctness of Duplicate Database Systems." In *Proceedings of the Sixth Symposium on Operating Systems Principles*, pp. 67–84, November 1977.

[ES83] Derek L. Eager and Kenneth C. Sevcik. "Achieving Robustness in Distributed Database Systems." *ACM Transactions on Database Systems*, 8(3):354–381, September 1983.

[Fai81] Sergio Zarur Faissol. *Operation of Distributed Database Systems Under Network Partition*. Ph.D. dissertation, University of California, Los Angeles, 1981.

[Flo86a] Rick Floyd. "Directory Reference Patterns in a UNIX Environment." Technical Report TR-179, University of Rochester, August 1986.

[Flo86b] Rick Floyd. "Short-Term File Reference Patterns in a UNIX Environment." Technical Report TR-177, University of Rochester, March 1986.

[FM82] Michael J. Fischer and Alan Michael. "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network." In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[GAB83] Hector Garcia-Molina, Tim Allen, Barbara Blaustein, R. Mark Chilenskas, and Daniel R. Ries. "Data-patch: Integrating Inconsistent Copies of a Database after a Partition." In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 38–44, October 1983.

[GGK87] Shai Gozani, Mary Gray, Srinivasan Keshav, Vijay Madisetti, Ethan Munson, Mendel Rosenblum, Steve Schoettler, Mark Sullivan, and Douglas Terry. "GAFFES: The design of a globally distributed file system." Technical Report UCB/CSD/87/361, Unviversity of California, Berkeley, June 1987.

[GHM90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. "Implementation of the Ficus Replicated File System." In *USENIX Conference Proceedings*, pp. 63–71. USENIX, June 1990.

[Gif79] D. K. Gifford. "Weighted Voting for Replicated Data." In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pp. 150–162. ACM, December 1979.

[Guy87] Richard G. Guy. *"A Replicated Filesystem Design for a Distributed UNIX System."*. Master's thesis, University of California, Los Angeles, 1987.

[Her86] Maurice Herlihy. "Dynamic Quorum Adjustment for Partitioned Data." Technical Report CMU-CS-86-147, Carnegie-Mellon University, September 1986.

[HHL88] Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Arthur L. Liestman. "A Survey of Gossiping and Broadcasting in Communication Networks." *NETWORKS*, 18:319–349, 1988.

[HHW89] Abdelsalam Heddaya, Meichun Hsu, and William Weihl. "Two Phase Gossip: Managing Distributed Event Histories." *Information Sciences*, **49**:35–57, October 1989.

[HKM88] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[HP90] John S. Heidemann and Gerald J. Popek. "An Extensible, Stackable Method of File System Development." Technical Report CSD-900044, University of California, Los Angeles, December 1990.

[HP91a] John S. Heidemann and Gerald J. Popek. "A Layered Approach to File System Development." Technical Report CSD-910007, University of California, Los Angeles, March 1991. Submitted for publication.

[HP91b] Norman C. Hutchinson and Larry L. Peterson. "The x-Kernel: An Architecture for Implementing Network Protocols." *IEEE Transactions on Software Engineering*, **17**(1):64–76, January 1991.

[HPA89] Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques." In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pp. 91–101. ACM, December 1989.

[JB86] Thomas A. Joseph and Kenneth P. Birman. "Low Cost Management of Replicated Data in Fault Tolerant Distributed Systems." *ACM Transactions on Computer Systems*, 4(1):54–68, February 1986.

[Jef85] David R. Jefferson. "Virtual Time." *ACM Transactions on Programming Languages and Systems*, **7**(3):404–425, July 1985.

[JM87] Sushil Jajodia and David Mutchler. "Dynamic Voting." In *Proceedings of the 1987 Annual Conference of theACM Special Interest Group on Management of Data*, pp. 227–238, May 1987.

[Kaz88] Michael Leon Kazar. "Synchronization and Caching Issues in the Andrew File System." In *USENIX Conference Proceedings*, pp. 31–43. USENIX, February 1988.

[KLA90] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. "DEcorum File

System Architectural Overview." In *USENIX Conference Proceedings*, pp. 151–163. USENIX, June 1990.

[Kle86] S. R. Kleiman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." In *USENIX Conference Proceedings*, pp. 238–247. USENIX, June 1986.

[KM86] Michael J. Karels and Marshall Kirk McKusick. "Toward a Compatible Filesystem Interface." In *Proceedings of the European Unix User's Group*, p. 15. EUUG, September 1986.

[Kur88] Øivind Kure. "Optimization of File Migration in Distributed Systems." Technical Report UCB/CSD 88/413, Unviversity of California, Berkeley, April 1988.

[MA69] Stuart E. Madnick and Joseph W. Alsop, II. "A modular approach to file system design." In *AFIPS Conference Proceedings Spring Joint Computer Conference*, pp. 1–13. AFIPS Press, May 1969.

[MB87] John H. Maloney and Andrew P. Black. "File Sessions: A Technique and its Application to the UNIX File System." In *Proceedings of the Third International Conference on Data Engineering*, pp. 54–61. IEEE, February 1987.

[MD74] Stuart E. Madnick and John J. Donovan. *Operating Systems.* McGraw-Hill Book Company, 1974.

[MJL84] Michael McKusick, William Joy, Samuel Leffler, and R. Fabry. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[Neu87] Peter G. Neumann. "ARPANET Partial Outage Despite "Redundancy"." *ACM Software Engineering Notes*, **12**(1):17, January 1987.

[Neu89] B. Clifford Neuman. "The Need for Closure in Large Distributed Systems." *Operating System Review*, **23**(4):28–30, October 1989.

[NPP86] Jerre D. Noe, Andrew B. Proudfoot, and Calton Pu. "Replication in Distributed Systems: the Eden Experience." In *Proceedings of the Fall Joint Computer Conference*, pp. 1197–1209. IEEE, November 1986.

[NRC88] National Research Network Review Committee of the National Research Council. "Toward a national research network." National Academy Press, 1988.

[OCD88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. "The Sprite Network Operating System." *IEEE Computer*, pp. 23–36, February 1988.

[OCH85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System." Technical Report UCB/CSD 85/230, UCB, 1985.

[Ous90] John K. Ousterhout. "Why Aren't Operating Systems Geting Faster As Fast as Hardware?" In *USENIX Conference Proceedings*, pp. 247–256. USENIX, June 1990.

[Par86] Jehan-François Pâris. "Voting with Witnesses: A Consistency scheme for Replicated Files." In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pp. 606–612, May 1986.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *ACM SIGMOD 88*, pp. 109–116, June 1988.

[PGP91] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. "Architecture of the Ficus Scalable Replicated File System." Technical Report CSD-910005, University of California, Los Angeles, March 1991. Submitted for publication.

[PHO90] Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao. "The x-Kernel: A Platform for Accessing Internet Resources." *IEEE Computer*, **23**(5):23–33, May 1990.

[PNP86] Calton Pu, Jerre D. Noe, and Andrew B. Proudfoot. "Regeneration of replicated objects: a technique for increased availability." In *Proceedings of the Second International Conference on Data Engineering*, February 1986.

[PPR83] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. "Detection of Mutual Inconsistency in Distributed Systems." *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[PSA87] Titus Purdin, Richard Schlichting, and Gregory Andrews. "A File Replication Facility for Berkeley UNIX." *Software—Practice and Experience*, **17**(12):923–940, December 1987.

[PW85] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture.* The MIT Press, 1985.

[RAA90] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. "Overview of the CHORUS Distributed Operating System." Technical Report CS/TR-90-25, Chorus systèmes, April 1990.

[Ran68] Brian Randell. "Towards a methodology of computer system design." Working paper for the NATO conference on computer software engineering at Garmisch, Germany, October 1968.

[RBF89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. "Mach: A Foundation for Open Systems." In *Proceedings of the Second Workshop on Workstation Operating Systems*, pp. 109–113. IEEE, September 1989.

[Rit84] Dennis M. Ritchie. "A Stream Input-Output System." *AT&T Bell Laboratories Technical Journal*, **63**(8):1897–1910, October 1984.

[RKH86] R. Rodriguez, M. Koehler, and R. Hyde. "The Generic File System." In *USENIX Conference Proceedings*, pp. 260–269. USENIX, June 1986.

[RO91] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System." Technical report, University of California, Berkeley, March 1991.

[Ros81] Eric Rosen. "Vulnerabilities of network control protocols." *ACM Software Engineering Notes*, **6**(1):6–8, January 1981.

[Ros90] David S. H. Rosenthal. "Evolving the Vnode Interface." In *USENIX Conference Proceedings*, pp. 107–118. USENIX, June 1990.

[RT88] Robbert van Renesse and Andrew S. Tanenbaum. "Voting with Ghosts." In *Proceedings of the Eigth International Conference on Distributed Computing Systems*, pp. 456–461. ACM, June 1988.

[Sal78] J. H. Saltzer. "Naming and Binding of Objects." In R. Bayer, editor, *Operating Systems*, volume 60 of *Lecture notes in Computer Science*, chapter 3, pp. 99–208. Springer Verlag, 1978.

[Sat88] Mahadev Satyanarayanan. "On the Influence of Scale in a Distributed System." In *Proceedings of the Tenth International Conference on Software Engineering*, pp. 10–18, April 1988.

[SBK85] Sunil K. Sarin, Barbara T. Blaustein, and Charles W. Kaufman. "System Architecture for Partition-Tolerant Distributed Databases." *IEEE Transactions on Computers*, **34**(12):1158–1163, December 1985.

[SBM89] Alex Siegel, Kenneth Birman, and Keith Marzullo. "Deceit: A Flexible Distributed File System." Technical Report TR 89-1042, Cornell University, November 1989.

[SGK85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and Implementation of the Sun Network File System." In *USENIX Conference Proceedings*, pp. 119–130. USENIX, June 1985.

[SKK90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. "Coda: A Highly Available File System for a Distributed Workstation Environment." *IEEE Transactions on Computers*, **39**(4):447–459, April 1990.

[SKS86] Sunil K. Sarin, Charles W. Kaufman, and Janet E. Somers. "Using History Information to Process Delayed Database Updates." In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pp. 71–78, August 1986.

[Smi81] Alan J. Smith. "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms." *IEEE Transactions on Software Engineering*, **7**(4), July 1981.

[Spa89] Eugene H. Spafford. "The Internet Worm: Crisis and Aftermath." *Communications of the ACM*, **32**(6):678–687, June 1989.

[Sto79] Michael Stonebraker. "Concurrency Control and Consistency of Multiple Copies of Data in Distrubted INGRES." *IEEE Transactions on Software Engineering*, **5**(3), May 1979.

[Str81] B. Ivan Strom. "Consistency of Redundant Databases in a Weakly Coupled Distributed Computer Conferencing System." In *Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 143–153, February 1981.

[Tho78] R. H. Thomas. "A Solution to the Concurrency Control Problem for Multiple Copy Databases." In *Proceedings of the 16th IEEE Computer Society International Conference*. IEEE, Spring 1978.

[TKW85] G. M. Tomlinson, D. Keeffe, I. C. Wand, and A. J. Wellings. "The PULSE Distributed File System." *Software–Pratice and Experience*, **15**(11):1087–1101, November 1985.

[VM87] K. Vidyasankar and Toshimi Minoura. "An Optimistic Resiliency Control Scheme for Distributed Database Systems." *Lecture Notes in Computer Science*, **312**:297–309, July 1987.

[WB84] Gene T. J. Wuu and Arthur J. Bernstein. "Efficient Solutions to the Replicated Log and Dictionary Problems." In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, August 1984.

[Wis88] Simon R. Wiseman. *Garbage Collection in Distributed Systems*. Ph.D. dissertation, University of Newcastle Upon Tyne, November 1988.

[WPE83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. "The LOCUS Distributed Operating System." In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pp. 49–70. ACM, October 1983.

[Wri83] David. D. Wright. "On Merging Partitioned Databases." In *Proceedings of the 1983 Annual Meeting of theACM Special Interest Group on Management of Data*, pp. 6–14, May 1983.

[ZE88] Edward R. Zayas and Craig F. Everhart. "Design and Specification of the Cellular Andrew Environment." Technical Report CMU-ITC-070, Carnegie-Mellon University, August 1988.

Fault Tolerant Distributed Database System via Data Inference

by

Wesley W. Chu, Andy Y. Hwang, Rei-Chi Lee,
Qiming Chen, Matthew Merzbacher,
and Herbert Hecht

Fault Tolerant Distributed Database System via Data Inference*

Wesley W. Chu, Andy Y. Hwang, Rei-Chi Lee, Qiming Chen and Matthew Merzbacher
Computer Science Department
University of California, Los Angeles
and
Herbert Hecht
Sohar Inc.
Los Angeles, CA

Abstract

A knowledge-based approach is proposed for query processing during network partitions. The approach uses available domain and summary knowledge to infer inaccessible data to answer the given query. A rule induction technique is used to extract correlated knowledge between attributes from the database contents. This knowledge is represented as rules for data inference. Based on a set of queries, simulation is used to evaluate the effectiveness of the proposed data inference technique for improving data availability under network partitions. Object allocation has a significant impact on data availability. Allocating objects that increase remote redundancy and reduce local redundancy increases data availability during network partitions. A prototype distributed database system that uses the proposed inference technique with correlated knowledge from a ship database has been implemented at UCLA. Our experience reveals that the proposed inference technique can significantly improve the availability of distributed database during network partitions.

1. INTRODUCTION

To improve the reliability and response time of distributed database systems, databases are often partitioned into fragment objects which are replicated and stored at several sites. Such fragment replication requires additional communication and processing overhead for maintaining consistency among the replicated copies. Further, due to channel and node failures, a network may be partitioned into two or more isolated parts. Since fragments may not be fully replicated at all sites, certain fragments may be inaccessible during network partitions. Most prior work used syntactic information to handle operations during network partitioning which lead to blocking or a partially operable system [GARC87]. However, in many real time applications, the availability of data is of primary importance. It is often not acceptable for a site to suspend processing when it cannot communicate with other sites. Because database attributes are often correlated and contain redundant information (e.g., salary and rank, ship type and cargo), we propose to use a data inference technique to *infer inaccessible data from accessible data.* Such a knowledge-based approach can greatly increase the availability of distributed database systems [CHU90a] [CHU90b].

This research is supported by DARPA contract F29601-87-C-0072, ONR contract N00014-88-K0434 and RADC contract F3060288C008

Correlated knowledge can be categorized into two levels. At the schema level, correlated knowledge between objects is represented as *inference paths.* Inference paths suggest proper objects and directions that the system should select for data inference. At the instance level, *correlated rules* are used to represent their detailed correlations. In our approach, a rule induction technique is used to induce correlated knowledge from the database contents. The induced knowledge is then represented in the inference graph where each node represents an object or an attribute in the domain and each edge represents the inferential relationships among objects. Each edge has an inferential confidence which indicates the degree the edge can be used to infer other objects. Depending on target objects, object availability status, and inferential relationships among objects, proper inference paths are selected for inference. In this paper, we shall first present our approach for knowledge acquisition and data inference. We then present the architecture of a DDBMS with data inference. Simulation is used to evaluate the effectiveness of the inference technique for improving data availability during network partitions and the data availability under different database fragment allocations. Finally, to validate the feasibility of our proposed approach, we present the implementation of a prototype inference system for a commercially distributed database system with correlated rules derived from a navy ship database.

2. KNOWLEDGE ACQUISITION AND DATA INFERENCE

2.1 Rule Induction

Because database attributes are often correlated and contain redundant information, data inference can be used to infer inaccessible data objects from other *accessible and correlated* data objects. In our approach, a rule induction technique is used to induce correlated rules from the database contents. Although different forms of rules may exist in a database, we shall only acquire the pairwise relationship among attributes. To induce rules between attributes X and Y, we use relational operations to retrieve instances of (X,Y) pairs from the database, and then select those pairs in which X has an unique corresponding Y value. Assuming relation RR contains attribute pair (X,Y), the algorithm to induce correlated rules for inference path X --> Y can be described as follows :

Rule Induction Algorithm:

1. *Retrieving (X,Y) value pairs*

CH2912-4/90/0000/0086$01.00 © 1990 IEEE

Retrieve the instances of the (X,Y) pair from the database. Let S be the resultant relation. The corresponding QUEL statement is:

```
Range of r is RR
retrieve into S unique (r.X, r.Y)
sorted by r.Y
```

2. *Removing (X,Y) pairs in which X has different Y value*

Retrieve all the (X,Y) pairs in which X has multiple values of Y. Let T be the resultant relation. The corresponding QUEL statement is:

```
Range of r is RR
Range of s is S
retrieve into T unique (s.X, s.Y)
where (r.X = s.X and r.Y != s.Y)
```

we then remove all the (X,Y) pairs from S in which X has different Y values.

```
Range of s is S
Range of t is T
delete s
where (s.X = t.X and s.Y = t.Y)
```

3. *Constructing Rules*

The acquired rules are summarized in the range form :

Rule: if $x_1 \leq X \leq x_2$ then $Y = y$

or in the set form :

Rule : if X in $\{ x_1, x_2, ... x_n \}$ then $Y = y$

The induced rules can be classified into *intra-relation* and *inter-relation* rules. Intra-relation rules describe correlated knowledge between attributes within the same relation while inter-relation rules describe knowledge between attributes of different relations. Since attributes within one relation are allocated at the same site, for fault tolerance applications, the inter-relation rules are more useful than intra-relation rules.

The total space for the rule base should be much smaller than its original data since rules are represented as summarized information. The induced rules represent the current states of database instances which may contain both static and dynamic parts of database characteristics. Static rules such as integrity constraints do not change while dynamic rules may change as data values are updated. However, the induced rules are less volatile than the original data since the acquired rules are summarized as range or set form.

To reduce the size of the rule base, we may discard the rules which cover too few pairs of instances. For example, in the ship database, ship name can uniquely determine its ship type. However, the volume of correlated rules between ship name and ship type is too large since each rule covers only one pair instance of ship name and ship type. Since rules which cover more instance pairs are usually less volatile, this also reduces the overhead of maintaining the rules. We shall use a naval ship database as a testbed to illustrate the above knowledge acquisition approach. The ship database was created by the System Development Corporation (SDC, now UNISYS) to provide a fairly generic database based on [JANE81]. For illustration purposes, we use a portion of the ship database which only contains the following relations

SHIP = (ShipId, ShipName, Class)
SONAR = (Class, Type, Sonar)
TYPE = (Type, Surface, TypeName)

The result of applying the rule induction algorithm to an instance of the sample database is given in Appendix A. Both *intra-relation* and *inter-relation* rules are acquired from rule induction. For instance, inference path *TYPE.type* --> *TYPE.surface* represents intra-relation knowledge since it describes correlated knowledge between *type* attribute and *surface* attribute within the TYPE relation. Inference path *SHIP.class* --> *TYPE.type* represents inter-relation knowledge since it describes correlated knowledge between *class* attribute in the SHIP relation and *type* attribute in the TYPE relation.

The rules between ship classes and ship types are fairly stable since the classification of ships into different ship types does not change often. Further, the acquired rules also indicate that shiptype information can be used to determine whether the ship is a surface or subsurface ship. We also note that certain rules are less static. For instance, the correlated rules between ship classes and sonars are dynamic since the sonars that the ships installed may change with time.

2.2 Inference Graph and Path

Inference Graph

An *inference graph* is a directed graph $G_I = (V_I, E_I)$ where V_I is a set of vertices each of which represents an object or attribute of the domain, and E_I is a set of edges which represent the inferential relationships among objects/attributes. Each edge has an inferential confidence which indicates the degree the edge can be used to infer other objects. An edge from X to Y is *universal* if the information about Y can be precisely inferred from attribute X. The *inferential confidence* for U edge is 1 since the requested information can be inferred exactly. An edge is *existential* from X to Y if the information of X can only infer partial information of Y. Therefore, an inference graph can be constructed based on the induced correlated knowledge.

The induced rules can be deterministic or probabilistic. The rule is deterministic if by knowing the instance information of one object, the other object's instance information can be exactly inferred. For example, the rule " if 'S101' ≤ Shipid ≤ 'S120' then Battlegroup = 'B02' " is deterministic since by knowing the ship's id is between S101 and S120, its battle group can be uniquely determined. However, some rules are probabilistic since by knowing the instance information of one object, another object may have several alternate instances. For example, a rule may indicate that weapon "SAM02" has 0.8 probability of being installed at DD shiptype and 0.2 probability of being installed at another shiptype DDG.

Inferential confidence of edges, I_e , can be determined by the total number of instances for the target object and the number of instances that are covered by the rules. For example, in the ship database, an inferential edge exists

from "weapon" to "shiptype" with the following correlated rules :

Rule 1 : if weapon = "AAM01" then shiptype = "CV"
Rule 2 : if weapon = "SAM01" then shiptype = "DD"
Rule 3 : if weapon = "SAM02" then shiptype = "DD" with p = 0.8
if weapon = "SAM02" then shiptype = "DDG" with p = 0.2
Rule 4 : if weapon = "ASW03" then shiptype = "SS" with p = 0.6
if weapon = "ASW03" then shiptype = "SSG" with p = 0.4
Rule 5 : if weapon in "ASW05,ASW07" then shiptype = "SSBN"

The inferential confidence from weapon to shiptype, I_c(weapon -> shiptype), can be determined by the total number of shiptype instances and the number of instances that are covered by the rules. Let us consider the deterministic rules, assuming that access frequency for each shiptype is uniformly distributed and that we have 80 tuples out of the total of 100 tuples covered by rules 1,2 and 5, then I_c(weapon -> shiptype) equals 0.8 since 80% of the time we can uniquely infer each ship's type from the weapon it carries.

Inference Path

An inference path can be constructed from inference edges. An inference path from X to Y exists if X can be connected to Y through one or more inferential edge(s). A path from X to Y is *universal* if and only if all the edges on the path are U edges. For a universal path, its *inferential confidence* is 1 and the requested information can be completely inferred. A path from X to Y is *existential* if information of X can infer only partial information of Y. Under such a condition, several paths may be required to infer an object. Each path may provide partial information and these intermediate results will be extracted or merged to derive the answer.

The confidence measurement should provide both completeness and correctness of the inference results. For an inference system only consisting of deterministic rules, all the inference results obtained via correlated rules are exact. The universal paths can infer the target object's information completely while the existential paths infer only partial information about the target objects. The correctness issue is related to the probabilistic rules which depends on whether the inference edges are independent. The evaluation of the correctness of an inference path is complicated particularly when the inference edges are dependent. Further research is needed to provide the correctness of an inference path.

When a network partition renders the required data objects inaccessible, depending on object availability status and inferential relationships between objects, the inference system is invoked to infer the missing data. For each inaccessible data object i, the inference system will traverse the constructed inference graph and form a search tree to find those accessible objects that can be used to infer i. Based on a predefined selection criterion, certain inference paths are selected and correlated rules are applied to infer the missing objects.

Our inference approach is different from the conventional deductive database approach. Conventional deductive database systems developed so far are in general underlain by the relational view mechanism or the first-order logic programming [GALL84] [REIT84]. In deductive database, an attempt to answer a query is referred to as satisfying a goal based on the prespecified facts and rules. From the inference path point of view, the execution order of the inference processes is prespecified in the conventional logic programming framework. In our inference environment, due to the incompleteness of the correlated knowledge, the inferred results may not be complete. To infer the missing information, the intermediate results from separate processes have to be merged. Since we cannot predict the outcome of intermediate results, the merging process has to be planned dynamically according to the inferred results. Therefore, totally prespecified inference paths, as in the conventional logic programming environment, are not adequate for our inference requirements. For detail discussions of the above issue, interesting readers should refer to [CHU90d]

3. THE ARCHITECTURE OF A DDBMS WITH DATA INFERENCE

A distributed database system with inference capability consists of a query parser and analyzer, an information module, and an inference system as shown in Fig. 1.

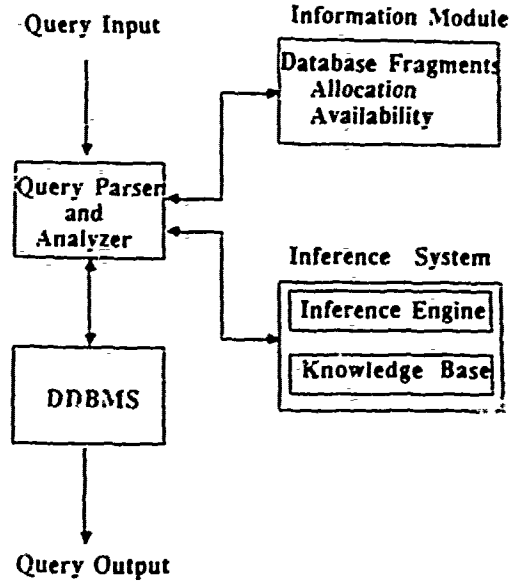


Figure 1 DDBMS with Data Inference

Information module provides allocation and availability information of all the attributes in the system. An inference system consists of a knowledge base and an inference engine. The correlated knowledge between attributes is represented as rules and is stored in the knowledge base. During normal operations, queries can be processed by the query processor since all the database fragments are accessible. When a network partition occurs, the information module and inference system will be invoked if any of the required attributes is inaccessible. Based on the availability status and the correlated knowledge between attributes, the inference engine modifies the original query to a new one

so that all the required data for the query is accessible from the requested site. Depending on the physical allocation of the database fragments and the domain semantics, the modified query may provide the exact, approximate or summarized answer of the original query.

4. IMPROVING AVAILABILITY USING DATA INFERENCE AND ALLOCATION

We shall use simulation to evaluate the availability improvement with the proposed inference technique under different fragment allocations. The input parameters for the simulation include : attributes and their allocation, querying site and its querying attributes, inferential capabilities between attributes, network partition topology and cost function. In the simulation, a given query is first referred to the object availability table to check if all the target attributes are available. If not, the inference module will be invoked to iteratively apply each inferential path and check to see if it will improve data availability or reduce its cost without sacrificing availability. The object availability table is updated according to the inference results. A given query can then be answered by referring to the new availability table.

Consider the following distributed ship database that consists of five database fragments :

SHIP(shipname,class,base)
TYPE(type,surface,typename)
SONAR(type,class,sonar)
INSTALL(weapon,shipid)
WARFARE(weapon,warfare)

where SHIP and SONAR are allocated to site SF, INSTALL to site Norfolk and TYPE, WARFARE to site LA. Further, a set of inference paths are also induced for the above database as shown in figure 2. For instance, our correlated knowledge indicates that shiptype information can be inferred from shipclass with confidence 1 and sonar information can be inferred from shipclass with confidence 0.7.

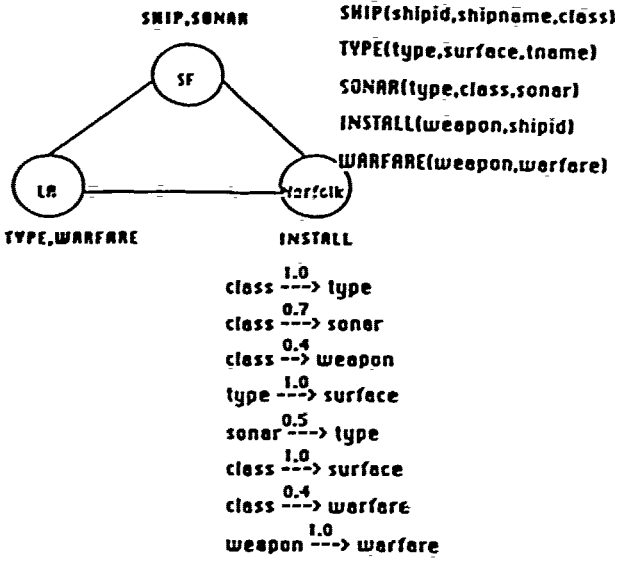


Figure 2 A Distributed Database System and its Inference paths

4.1 Availability Improvement with Data Inference

We can classify queries into two types. Let a query be called a type I query if all of its requested attributes are accessible. When any of the requested attributes is inaccessible, the query is classified as a type II query. Based on the above definition, for a given set of queries and attribute allocation, we define *query mix factor*, α, as the percentage of type I queries in the system. We shall now use simulation to evaluate the effect of data inference on the improvement in availability for selected α's. Assuming that after several link failures, the network is partitioned into two parts where LA and Norfolk are at one part and SF is at another as shown in figure 3a. Our simulation first randomly generates a set of queries. Each query in this set requires accessing different attributes at different sites. For a given attribute allocation, depending on the accessibility status, we can classify each query into a type I or a type II query. We can evaluate the probability of answering queries correctly for a specific partition, e.g. partition SF, $P_{SF}(\alpha)$ as a function of α. When the system consists of all type I queries; i.e., $\alpha = 1$, all the queries can be answered by accessing the database directly. Therefore, $P_{SF}(\alpha) = 1$. Since α decreases as type II queries increase, thus $P_{SF}(\alpha)$ decreases as α decreases. When all the queries are type II queries, then $\alpha = 0$. Since the requested data is inaccessible, no queries can be answered. Thus, $P_{SF}(\alpha) = 0$ without data inference. Therefore, without data inference, the probability of answering queries correctly is 0. However, if the data inference technique is used, certain missing data may be inferred. As a result, some queries can be answered using the correlated knowledge. With the inference paths given in figure 2, our simulation results reveal that 32% of the queries can still be answered which shows the effectiveness of our inference approach. The percentage of queries that can be answered depends on many factors such as object allocation, network partition topology, correlated knowledge between attributes, etc., which will be discussed later.

For a given query, based on the correlated knowledge between objects, there may exist several inference paths to infer the answer. Some may provide more accurate answers but require a longer execution time, while the others may take a shorter time but yields less accuracy. The overhead for inference consists of communication delay, database accessing and knowledge base processing delay. Therefore, there is a cost/performance trade-off in selecting an inference path. Based on the predefined cost function, one approach is to select the path that provides better confidence for answering the query or provides less cost with the same level of confidence. Further, using multiple inference paths may achieve better availability. Let us consider the following example. Assuming a query is initiated at SF that requires accessing *shiptype* and *warfare* information. With the database fragment allocation and resultant network partitions as shown in figure 4a, the given query cannot be answered since *warfare* information cannot be accessed from site SF. However, there exists several inference paths to infer the *warfare* information. It can be remotely inferred from the *radar* information at site LA with confidence 0.6 or locally inferred from the *class* information with confidence 0.4. In our simulation, the path from *radar* is selected to infer *warfare* since it provides higher confidence even though its cost is higher than the other path. On the other hand, the path from *class* to *warfare* will be selected if cost is the criterion for selection. If we combine the above two paths for inferring *warfare*, then the total confidence will be greater than using a single path. Assuming the information inferred from each path is in

dependent, then the total inferential confidence will be 1 - (1 - 0.4) * (1 - 0.6) = 0.76, which is greater than 0.6.

Note that combining knowledge from different paths requires special algebraic tools. The usual relational join operation is inadequate for merging intermediate knowledge from different paths since some information may be dropped during the join operation. As a result, a special union operation is used for merging results from different paths in [CHU90c].

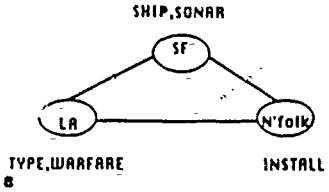


Figure 3a

A network is partitioned into 2 sub-networks: site SF and site LA, N'folk

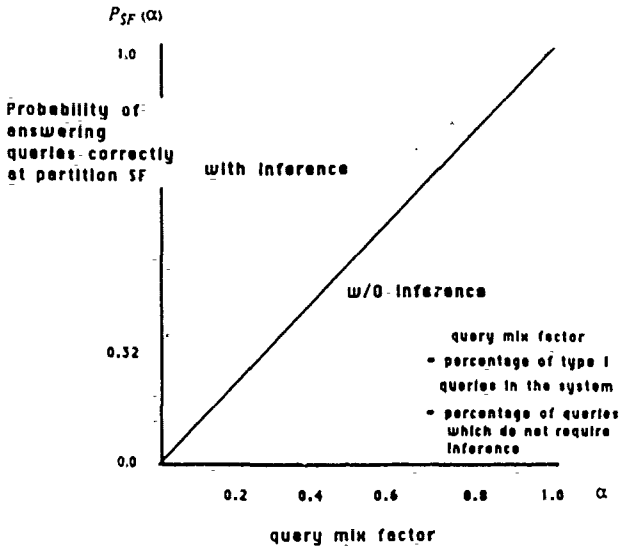


Figure 3b Probability of answering queries correctly with and w/o inference

4.2 Allocation of Database Fragments to Improve Availability

Since data inference introduces a certain degree of *information redundancy*, inference can be viewed as *virtual replication*. In general, there are two types of information redundancies: remote redundancy and local redundancy. Local redundancy exists if the same information is duplicated at a site. For instance, if object i can infer object j, allocating them at the same site yields local redundancy for object j. Thus, allocating objects i and j at different sites provides remote redundancy which increases the availability of object j. Note that local redundancy reduces data availability, while remote redundancy increases data availability.

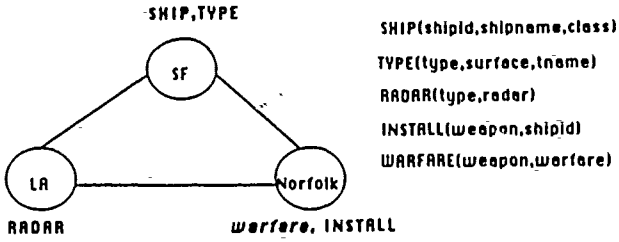


Figure 4a

A network is partitioned into 2 sub-networks

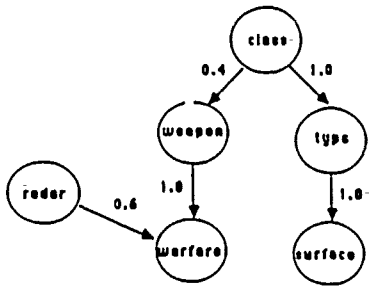


Figure 4b

Inference graph for the database in figure 4a

We shall use simulation to illustrate the above concept. For a three-node network with three database fragment allocations as shown in figure 5, since SHIP and SONAR both contain *shipclass* information, allocating them at the SF site (allocation A) yields redundant *shipclass* information at that site. Allocation B also introduces a certain level of local redundancy for *shiptype* information since that information already exists in the TYPE relation and it can also be inferred from *shipclass* in the SHIP relation. Thus, allocation A yields redundant *shipclass* information and allocation B yields redundant *shiptype* information for SF site. Since *shipclass* can infer more information than *shiptype* can infer, allocation B yields better availability than allocation A. Allocation C allocates SHIP and WARFARE at the same site which does not introduce local redundancy for site SF. Therefore, allocation C provides better availability than allocations A and B.

Optimal allocation for normal operations may be different from that during network partitioning. From the locality point of view, allocating two frequently co-referenced database fragments at the same site reduces communication cost and thus response time. However, from the data inference point of view, allocating two uncorrelated database fragments to the same site and two strongly correlated database fragments at different sites provide higher inferential capability and thus increases the virtual replication of those two database fragments. Since locality and correlation may be dependent, we need to consider both factors in allocating database fragments to different sites in a distributed database design.

5. AN IMPLEMENTATION

In the following, we discuss the implementation of a data inference engine. An example based on a ship database is also included to illustrate the inference process.

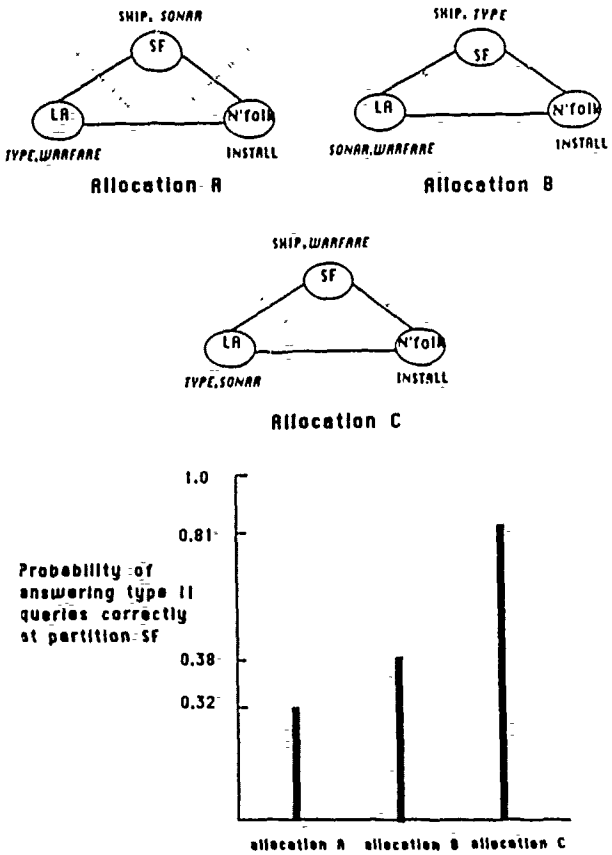


Figure 5 Probability of answering type II queries correctly for different data allocations

5.1 Inference Engine

An experimental inference system has been implemented for a distributed database running on a set of Sun 3/60 workstations interconnected by an Ethernet at UCLA. The system is based on the relational model where all the source and target data objects are relations. The inference actions are extensions of the relational operations which allow us to build the inference engine on top of Sybase, a relational database system. The inference system operates in the following way :

a) When a network partition renders required data objects inaccessible, the inference system develops an inference plan based on the given query, object availability status, database schema and correlated knowledge stored in the rule base.

b) Data inference is then carried out via the inference plan which consists of a set of derivations and the execution sequence of those derivations. Each derivation process represents a derivation from certain available data objects to an intermediate or final data inference result. Three general types of derivations are implemented in the system :

   1) Derive new relation based on certain source relations. It is specified as relational views and implemented through the view generation mechanism.

   2) Instantiate relations based on summary information and correlated knowledge. The instantiation process is implemented through the relation alteration mechanism.

   3) Combining intermediate results in terms of appropriate system operations (viewed as meta-rules). Due to the incompleteness of the inference results, the combination of two relations is implemented through a special union operation developed in [CHU90c]. For more discussion and formalism of the union operation, interested users should refer to [CHU90c] [CHEN89].

c) Select the required data objects from the final result. In the current implementation, since the target objects to be inferred are relations, the inference process is designed to infer as much of the missing relations as possible. The required attribute information is then selected from the final result.

5.2 A Data Inference Example

Consider a distributed database that consists of three database fragments : SHIP(shipid,sname,class), INSTALL(shipid,weapon), and CLASS(class,type,tname) which are stored at sites LA, SF and Norfolk respectively. When the site SF is partitioned, the following query cannot be answered since relation INSTALL is not accessible :

Q1 : " Find the ship names that carry weapon 'AAM01' "

Since the target objects are relations, our inference engine needs to make an inference plan to select relevant inference paths for inferring the missing relation INSTALL. The inference engine currently exhaustively searches all the derivations in the knowledge base and selects the relevant derivations. In this example, the following two derivations are used to infer the missing INSTALL relation :

DERIVATION 1 : select shipid, type from SHIP, CLASS

DERIVATION 2 : CLASS(type) --> INSTALL(weapon)

Derivation 1 represents the first type of derivation where the deductive rule is expressed by a view definition. This derivation creates a temporary relation which contains shipid and type information. Derivation 2 illustrates the second type of derivation, where derivation is performed from [type] to [type,weapon]. This derivation also creates a temporary relation with shiptype and weapon information. While information of shiptype is filled by accessing the CLASS relation, weapon information is filled based on the provided correlated rules between shiptype and weapon. The above two intermediate results are then combined by the special union operation developed in [CHU90c]. The resultant relation, referred to as INSTALL_INF, is used to replace the inaccessible INSTALL relation. Query Q1 can be answered by joining SHIP relation with the INSTALL_INF relation.

6. DISCUSSIONS

In a distributed database system, data fragments are often replicated to increase data availability in case of failures [CERI84] [ELAB85]. However, performance can be adversely affected by replication because of the communication required to ensure consistency between copies of the data. While full replication provides better availability than partial replication, it introduces more communica-

tion and processing overhead [GARC82]. However, in any partially replicated scheme, some fragments may be inaccessible during network partition.

Data inference provides an alternative to data replication for increasing availability. As a result, the number of physical replicas may be reduced without reducing the number of logical replicas. To improve availability during network partition, objects which cannot be inferred should be replicated and stored at different sites. Further, to reduce access time, objects with high transaction rates at several sites should be replicated at these sites. To increase inference capability and further improve availability, we may want to replicate objects with high inferential capability. Therefore, with selective database fragment replication and the use of data inference, the availability of the database system can be significantly improved.

The degree of improvement in availability is affected by the database fragment allocation. Since inference provides virtual replication of database fragments, to maximize the benefit, data fragments should be allocated to maximize the virtual replication of all the fragments (weighted by the access frequency). For a given allocation, the availability of the database fragments also depends on how the network is partitioned. Further research is needed to determine the best allocation for database fragments under network partition.

While we have constructed a system which is able to infer missing data, several important problems remain. Our approach uses the rule induction mechanism to induce a set of rules which describe the correlation between attributes. However, only correlations between individual attributes are used. Although a set of attributes (two or more) may also infer the value of another set of attributes, the efficient selection of correlated sets is difficult due to the combinatorial explosion of such correlations. Further research in selecting for correlated sets is needed.

So far, we have considered knowledge acquisition in a slowly changing database environment. If data values are changing dynamically, timely updating of the knowledge base becomes an issue. One approach would be to trigger a knowledge base update every time the database changes state, thus keeping the knowledge base up-to-date at all times. To reduce communication overhead, we may select only a subset of the knowledge base to be consistent at all times, with the rest having weaker consistency. We need to determine what type of knowledge is critical with respect to the mix of expected queries and how to best keep that knowledge consistent. For less critical knowledge, where weaker consistency will suffice, we must determine how much and what types of inconsistency are allowable. Further research is needed to investigate the cost of maintaining a knowledge base in a dynamically varying environment, and also the effect of using weakly consistent knowledge for data inference.

7. CONCLUSION

A knowledge-based approach is proposed to improve data availability for query processing during network partitions. The approach uses available domain and summary knowledge to infer inaccessible data to answer the query. A rule induction algorithm is used to acquire correlated knowledge for data inference application. Simulation is used to evaluate the effectiveness of the proposed data inference technique to improve data availability under network partitions. Object allocation has a significant impact on data availability. Allocating objects that increase remote redundancy and reduce local redundancy increases data availability during network partitions. A prototype inference system has been implemented on a distributed database system that runs on a network of Sun 3/60 workstations at UCLA. Using ship database as a testbed, our experience reveals that the proposed rule induction technique is capable of obtaining useful correlated knowledge for data inference. As a result, data inference can significantly improve the availability of the distributed database during network partitions.

Acknowledgements

The authors would like to thank Brian Boesch of DARPA ISTO and Joseph Giordano of RADC for their encouragement and support for carrying the study and implementing the prototype system. We also thank G. Popek and T. Page of UCLA for their stimulating discussions.

REFERENCE

[CERI84] Ceri, S. and Pelagatti, G. "Distributed Databases Principles and Systems". New York, NY: McGraw-Hill, 1984

[CHEN89] Chen, Qiming "A High Order Logic Programming Framework for Complex Objects Reasoning", International Computer Software & Applications Conference (COMPSAC 89), 1989, USA.

[CHU90a] Chu, Wesley, Hwang, Andy, Hecht, Herbert and Tai, Ann "Design Considerations of a Fault Tolerant Distributed Database System by Inference Technique", *Extended Abstract*, Proceedings of PARBASE-90, March 6 - 8, 1990, Miami

[CHU90b] Chu, Wesley, Hwang, Andy "Inference Techniques for a Fault Tolerant Distributed Database System", *Extended Abstract*, Proceedings of PARBASE-90, March 6 - 8, 1990, Miami

[CHU90c] Chu, Wesley, Hwang, Andy, Chen, Qiming and Lee, Rei-Chi "An Inference Technique for Distributed Query Processing in a Partitioned Network", Technical Report, CSD-900005, February, 1990, UCLA

[CHU90d] Chu, Wesley, Chen, Qiming, and Hwang, Y. Andy "On Open Data Inference and its Applications", submitted for publication

[ELAB85] El Abbadi, Skeen, D. and Cristian, F., "An efficient fault-tolerant protocol for replicated data management", in Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, pp. 215 - 229, 1985

[GALL84] Gallaire, H. Minker, J. Nicolas, J. "Logic and Databases : A Deductive Approach", ACM Computing Surveys, Vol. 16, No 2, June 1984.

[GARC82] Garcia-Molina, Hector "Reliability issues for fully replicated distributed databases," IEEE Computer, Vol. 15, pp. 34 - 42, Sept. 1982

[GARC87] Garcia-Molina, H. and Abbott, R. K. "Reliable Distributed Database Management", *Proc. of the IEEE*, May 1987, pp. 601-620.

[JANE81] Jane's Fighting Ships, Jane's Publishing Co., 1981.

[REIT84] Reiter, R. "Towards a Logical Reconstruction of Relational Database Theory", in On Conceptual Modeling, pp 191-234, Springer- Verlag Ed ., 1984.

Appendix A

Sample Ship Database and Its Induced Rules:

Relation SHIP

sid	name	class
S101	Wisconsin	C02
S102	Dale	C03
S103	America	D12
S104	Barry	C11
S105	Texas	C12
S106	John_Hancock	S03
S107	Peterson	D02
S108	Nicholson	D04
S109	John_Rodgers	D12
S110	Paul	D14
S111	Donald_Berry	SN04
S112	Clark	D15
S113	Thomas_Hart	SN02
S114	Iowa	D12
S115	John_Hall	S02
S116	Seahorse	S03
S117	Rams	SN02
S118	Dallas	SN03
S119	Delta_I	SN04
S120	Delta_II	SN11
S121	Bathsh	SN12
S122	Blue6sh	S15
S123	Atlanta	S15
S124	LaJolla	S16
S125	Skate	S18

Relation SONAR

class	type	sonar
C02	CG	SQS-26
C03	CG	SQS-23
C03	CG	SQS-53A
C11	CV	SQS-53
C12	CV	SQS-53
D02	DD	SQS-26
D02	DD	SQS-53A
D04	DD	SQS-23
D12	DDG	TACTAS
D14	DDG	SQS-23
D15	DDG	SQS-23
S02	SS	BQS-4
S02	SS	BQQ-2
S03	SS	BQQ-2
SN02	SSN	BQQ-2
SN03	SSN	BQS-8
SN04	SSN	BQS-8
SN11	SSBN	LF-bow
SN12	SSBN	LF-bow
S15	SSG	hercules
S16	SSG	hercules
S18	SSG	hercules

Relation TYPE

type	ss	name
CG	surface	guided missile carrier
CV	surface	aircraft carrier
DD	surface	destroyer
DDG	surface	guided missile destroyer
SS	subsurface	patrol submarine
SSBN	subsurface	ballistic nuclear missile submarine
SSG	subsurface	guided missile submarine
SSN	subsurface	nuclear submarine

The Rules:

Intra Relation Rules

Entity	IF					THEN
TYPE(Type --> ss)	CG	S	Type	S	DDG	ss = surface
TYPE(Type --> ss)	SS	S	Type	S	SSG	ss = subsurface
SONAR(Class --> Type)	C02	S	Class	S	C03	Type = CG
SONAR(Class --> Type)	C11	S	Class	S	C12	Type = CV
SONAR(Class --> Type)	D02	S	Class	S	D04	Type = DD
SONAR(Class --> Type)	D12	S	Class	S	D15	Type = DDG
SONAR(Class --> Type)	S02	S	Class	S	S03	Type = SS
SONAR(Class --> Type)	SN02	S	Class	S	SN04	Type = SSN
SONAR(Class --> Type)	SN11	S	Class	S	SN12	Type = SSBN
SONAR(Class --> Type)	S15	S	Class	S	S18	Type = SSG
SONAR(Class --> Sonar)	C02	S	Class	S	C02	Sonar = SQS-26
SONAR(Class --> Sonar)	C11	S	Class	S	C12	Sonar = SQS-53
SONAR(Class --> Sonar)	D04	S	Class	S	D04	Sonar = SQS-23
SONAR(Class --> Sonar)	D12	S	Class	S	D12	Sonar = TACTUS
SONAR(Class --> Sonar)	D14	S	Class	S	D15	Sonar = SQS-23
SONAR(Class --> Sonar)	S03	S	Class	S	SN02	Sonar = BQQ-2
SONAR(Class --> Sonar)	SN03	S	Class	S	SN04	Sonar = BQS-8
SONAR(Class --> Sonar)	SN11	S	Class	S	SN12	Sonar = LF bow
SONAR(Class --> Sonar)	S16	S	Class	S	S18	Sonar = hercules
SONAR(Sonar --> Type)	SQS-53	S	Sonar	S	SQS-53	Type = CV
SONAR(Sonar --> Type)	TACTUS	S	Sonar	S	TACTUS	Type = DDG
SONAR(Sonar --> Type)	BQS-8	S	Sonar	S	BQS-8	Type = SSN
SONAR(Sonar --> Type)	LF bow	S	Sonar	S	LF-bow	Type = SSBN
SONAR(Sonar --> Type)	hercules	S	Sonar	S	Hercules	Type = SSG

The Rules (continued):

Inter-Relation Rules

Relationship	IF					THEN
SHIP(Class) --> SONAR(Type)	C02	S	Class	S	C03	Type = CG
SHIP(Class) --> SONAR(Type)	C11	S	Class	S	C12	Type = CV
SHIP(Class) --> SONAR(Type)	D02	S	Class	S	D04	Type = DD
SHIP(Class) --> SONAR(Type)	D12	S	Class	S	D15	Type = DDG
SHIP(Class) --> SONAR(Type)	S02	S	Class	S	S03	Type = SS
SHIP(Class) --> SONAR(Type)	SN02	S	Class	S	SN04	Type = SSN
SHIP(Class) --> SONAR(Type)	SN11	S	Class	S	SN12	Type = SSBN
SHIP(Class) --> SONAR(Type)	S15	S	Class	S	S18	Type = SSG
SHIP(Class) --> TYPE(Type)	C02	S	Class	S	C03	Type = CG
SHIP(Class) --> TYPE(Type)	C11	S	Class	S	C12	Type = CV
SHIP(Class) --> TYPE(Type)	D02	S	Class	S	D04	Type = DD
SHIP(Class) --> TYPE(Type)	D12	S	Class	S	D15	Type = DDG
SHIP(Class) --> TYPE(Type)	S02	S	Class	S	S03	Type = SS
SHIP(Class) --> TYPE(Type)	SN02	S	Class	S	SN04	Type = SSN
SHIP(Class) --> TYPE(Type)	SN11	S	Class	S	SN12	Type = SSBN
SHIP(Class) --> TYPE(Type)	S15	S	Class	S	S18	Type = SSG
SHIP(Class) --> SONAR(Sonar)	C02	S	Class	S	C02	Sonar = SQS-26
SHIP(Class) --> SONAR(Sonar)	C11	S	Class	S	C12	Sonar = SQS-53
SHIP(Class) --> SONAR(Sonar)	D04	S	Class	S	D04	Sonar = SQS-23
SHIP(Class) --> SONAR(Sonar)	D12	S	Class	S	D12	Sonar = TACTUS
SHIP(Class) --> SONAR(Sonar)	D14	S	Class	S	D15	Sonar = SQS-23
SHIP(Class) --> SONAR(Sonar)	S03	S	Class	S	SN02	Sonar = BQQ-2
SHIP(Class) --> SONAR(Sonar)	SN03	S	Class	S	SN04	Sonar = BQS-8
SHIP(Class) --> SONAR(Sonar)	SN11	S	Class	S	SN12	Sonar = LF-bow
SHIP(Class) --> SONAR(Sonar)	S16	S	Class	S	S18	Sonar = hercules
SHIP(Class) --> TYPE(ss)	C02	S	Class	S	D15	ss = surface
SHIP(Class) --> TYPE(ss)	S02	S	Class	S	SN12	ss = subsurface
SONAR(Class) --> TYPE(Type)	C02	S	Class	S	C03	Type = CG
SONAR(Class) --> TYPE(Type)	C11	S	Class	S	C12	Type = CV
SONAR(Class) --> TYPE(Type)	D02	S	Class	S	D04	Type = DD
SONAR(Class) --> TYPE(Type)	D12	S	Class	S	D15	Type = DDG
SONAR(Class) --> TYPE(Type)	S02	S	Class	S	S03	Type = SS
SONAR(Class) --> TYPE(Type)	SN02	S	Class	S	SN04	Type = SSN
SONAR(Class) --> TYPE(Type)	SN11	S	Class	S	SN12	Type = SSBN
SONAR(Class) --> TYPE(Type)	S15	S	Class	S	S18	Type = SSG
SONAR(type) --> TYPE(ss)	CG	S	type	S	DDG	ss = surface
SONAR(type) --> TYPE(ss)	SS	S	type	S	SSN	ss = subsurface
SONAR(Sonar) --> TYPE(Type)	SQS-53	S	Sonar	S	SQS-53	Type = CV
SONAR(Sonar) --> TYPE(Type)	TACTUS	S	Sonar	S	TACTUS	Type = DDG
SONAR(Sonar) --> TYPE(Type)	BQS-8	S	Sonar	S	BQS-8	Type = SSN
SONAR(Sonar) --> TYPE(Type)	LF-bow	S	Sonar	S	LF bow	Type = SSBN
SONAR(Sonar) --> TYPE(Type)	hercules	S	Sonar	S	Hercules	Type = SSG
SONAR(Class) --> TYPE(ss)	C02	S	Class	S	D15	ss = surface
SONAR(Class) --> TYPE(ss)	S02	S	Class	S	SN12	ss = subsurface

Proceedings of the IEEE Workshop on Experimental Distributed Systems
Huntsville, Alabama Oct. 1990

Development of a Fault Tolerant Distributed Database via Inference*

Wesley W. Chu Thomas W. Page Jr. Qiming Chen Andy Y. Hwang
O. T. Satyanarayanan

Department of Computer Science
University of California Los Angeles

1 Introduction

This work sets out to test the philosophy that not only can syntactic redundancy (replication) be exploited to improve fault tolerance, but that most data are correlated, containing redundant information at the semantic level as well. In order to be of use, this redundancy must be recognized, automatically extracted, and encoded as rules which can be used as input to an inference engine [5, 4]. The database itself must be engineered to make use of the inference engine to infer the inaccessible data from accessible data. The inferred data may be exact or approximate. However, in cases where time critical decisions must be made even though portions of the database are unavailable due to network partitions or site failures, having such inferred data (with completeness and correctness measures) is often preferable to no data at all.

While inference enhances availability for query answering access, we can also employ semantic information about the data and transactions to improve availability for *update*. Given the semantics of an update transaction on replicated data, it is often preferable to permit transactions to commit during network partition even though purely syntactic definitions of correctness (serializability) may be violated[1]. The semantic knowledge can then be used to restore the database when the partition heals. In this workshop, we report on the experience of building a knowledge based distributed database testbed on top of a commercial relational database in which to experiment with semantics for fault tolerance.

2 Data Inference

Our data inference system is based on the relational model where all the source and target data objects are relations. The inference actions are extensions of relational operations which enabled us to build the inference engine on top of a commercial relational database system. Currently, two types of rules are used by the inference engine - deductive rules specified in terms of relational operations and correlated rules which are specified as summarized knowledge.

A rule induction technique is used to extract correlated knowledge between attributes from the database relations. In our implementation, only correlations between pairs of attributes are used. Although two or more attributes may also infer the value of a set of attributes, the efficient selection of correlated sets is difficult due to the combinatorial explosion of such correlations. To induce rules between attributes X and Y, we use relational operations to retrieve instances of (X,Y) pairs from the database, and then select those pairs in which X has a corresponding unique Y value. The detailed algorithm is presented in [6]. The acquired rules are

*This work was sponsored by DARPA under contract number F29601-87-C-0072.

[1] This is based on the optimistic belief that that conflicting updates are sufficiently rare that it is better to detect and repair conflicts than to prevent them using existing algorithms which reduce update availability.

in the range form *if* $x_1 \leq X \leq x_2$ *then* $Y = y$ or in the set form *if* $X \in \{x_1, x_2, \ldots, x_n\}$ *then* $Y = y$.

When a network partition occurs, the inference system develops a plan which consists of a set of derivations and the execution sequence of those derivations. The inference plan is based on the given query, object availability status, database schema, and correlated knowledge stored in the rule base. Each derivation process represents a derivation from certain available data objects to an intermediate and final data inference result.

Three types of derivations are implemented in the system. First, new relations can be derived based on other relations. The derived relations are specified as relational views and implemented through the view generation mechanism. The second method consists of valuations of incomplete relations based on summary information and correlated knowledge. The valuation process is implemented through the relation altering mechanism. Finally, intermediate results can be combined via appropriate operations. The combination of two relations can be implemented through relational outer-join, which keeps all the necessary incomplete tuples appearing in the intermediate results. These tuples may be valuated through other derivations or combined with the data obtained from other derivations.

The required data objects are selected from the results of the inference process. The tuples in the missing relation are inferred, as completely as possible, and the required attributes selected from the final result.

3 Overview of the Architecture

In order to avoid rebuilding large amounts of software that are not central to our research, we are using an off-the-shelf, commercial database server for local relational data management. We have chosen Sybase because it supports a client/server model, supports a relatively standard SQL interface, has elements of an extensible architecture, and competitive performance. However, our modular architecture renders our data inference and semantic based concurrency control highly independent of our choice of database engines.

The system architecture consists of a front-end process per user session and a pool of back-end database servers, at least one per site. The front-end process consists of an SQL parser, an object availability module, an inference engine, and a distribution layer. Users submit queries to the front-end where they are parsed to form a query tree. The tree is then passed to an object availability evaluator which checks the status of the storage sites for each data object named in the query. The object evaluator uses the distribution module to determine data object locations and node/link status information. If any data objects are found to be unavailable due to network partition, the list of missing objects and the parse tree are submitted to the inference engine.

For each missing object, the inference engine attempts to infer an approximation. The knowledge base for the inference engine is stored (fully replicated) in the underlying local database server. The inference engine can infer a replacement data object for each missing object and modify the query tree to reference the inferred object. Alternatively, the inference engine may simply modify the query to an equivalent one which accesses only available data. In either case, a modified query tree is returned to the parser. The modified parse tree is converted back to SQL and submitted to the distribution module for execution[2]. Figure 1 shows a schematic view of the architecture.

4 Experiences

We have a working prototype of a distributed database, knowledge induction mechanism, and inference engine. The system automatically induces a

[2]The distribution module simulates a distributed database on top of the collection of single site servers. It provides access to remote data and simulates distributed joins by forming temporary local copies, performs update propagation to replicated relations, and coordinates two phase commit.

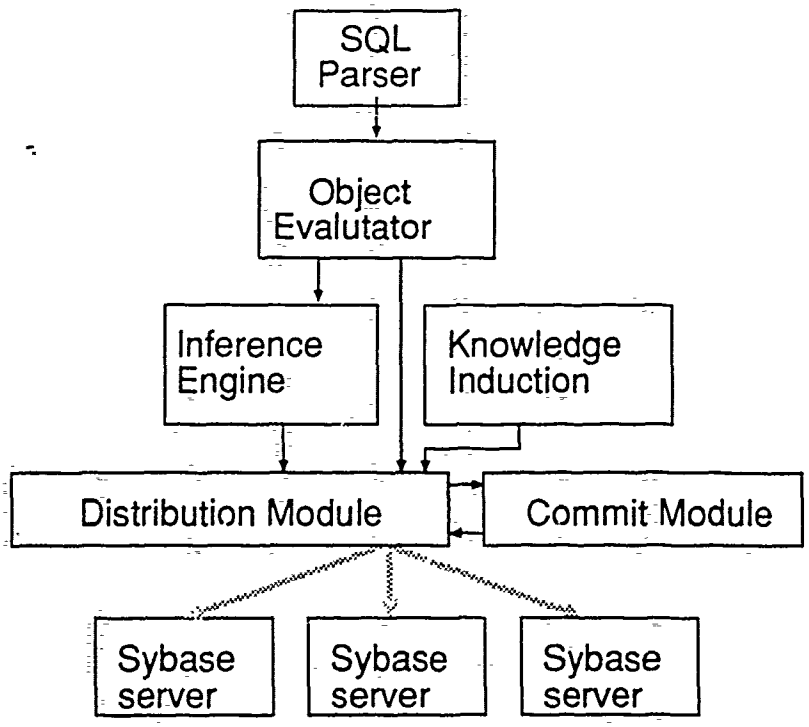


Figure 1: Inferential Database Testbed Architecture

set of summary rules from the data instance and domain model. Sets of sites can be disconnected from the network and the system automatically infers the data rendered inaccessible, answering queries which would otherwise be impossible in conventional distributed databases. The prototype effectively demonstrates the potential of inference as a technique to improve fault tolerance during network partition.

Knowledge Schema

The inference engine makes use of the rules induced by the knowledge induction mechanism to construct temporary relations from summarized knowledge and accessible relations. The rules were also stored in the database in a relational form for uniformity of access and storage. This necessitated the schema of the rules and relations to be known to the inference engine in order to be able to access them. To avoid the overhead of communicating with the Sybase server to determine the schema for each inference cycle, a copy of the schema of all the relations in the database was maintained at the client's site. Consistency between the client and server copy was maintained by adopting a write-through policy, *i.e.*, any changes made by the client would also result in the server's copy being updated. This was justified based on the observation that updates to the schema were much lower compared to the reads required to access base relations and rules.

Cacheing

To construct an inaccessible relation, the inference engine makes use of several rules, each of which serve the purpose of inferring some tuples of the missing relation. In order to avoid the overhead of communicating with the Sybase server each time a new rule is required, all the rules that are instrumental in inferring a particular relation were batched together and cached at the client's site for use by the inference engine. One more technique adopted to improve performance was the cacheing

of base relations at the client's site. Some of the rules required accessing other available relations to infer the tuples of the missing relation. Rather than requesting an available base relation from the server each time a rule is applied, the base relation is cached at the client's site and then all rules which refer to the base relation are successively applied to construct the missing tuples. The set of rules required to infer a missing relation and the base relations used by these rules are predetermined by the knowledge induction mechanism. This is facilitated to a large extent by the static nature of the application, which had a predominance of queries and all updates were made to existing base relations.

As mentioned above, rules refer to other base relations to infer tuples of a missing relation. If it turns out that some of these base relations are also inaccessible, they in turn have to be inferred, which is possible only if there are no cyclic dependencies between relations. In our experience, this is not true even for moderately large applications. This leads to the rather complex problem of optimal data assignment in order to maximize availability during network partitions.

Commercial Database

The commercial database server Sybase, on which the knowledge based distributed database testbed was implemented, both hindered and facilitated our implementation. Sybase logs every update operation on its log device to aid recovery, including updates on relations in the temporary database, which can be used as scratch space in the server by all database users. We avoided the overhead of a disk logging operation for each update by specifying a UNIX file as a log device, so that the actual disk accesses were controlled by the file system's buffering mechanism. However, we discovered during the course of the implementation that frequent updates very quickly filled up the log device which impeded the server from accepting all further updates. This necessitated database dumps to be made very frequently during peak operational periods in order to clear the log device.

Our initial design required our front-end to do an actual login to a Sybase server for each query submitted by a user. This did not have any noticeable performance overhead for normal operations, where the rate at which queries were submitted to the system was governed by the user. However, during network partitions, the inference engine generated a large number of queries in rapid succession in order to access rules and base relations, which lead to a very rapid performance degradation. We solved the problem by having the client manage a pool of connections to several servers. Each client on initiation would open at least one connection to each one of the available servers in the system and all requests are routed through this connection as much as possible. Further connections are opened depending on load requirements. This was, of course, the obvious approach, we were surprised, however, that the system was unusably slow until we made this optimization. It is advisable to minimize the number of open connections to each server from a particular client since the server supports a limited number of concurrent client connections. A large number of open connections also results in an increase in the response time of the server.

Database Error Handling

We are able to create dynamic network partitions and reconfigure sites back into the system during our demonstrations. This is facilitated to a large extent by Sybase's user supplied error handling mechanism, which prevents the client from catastrophically aborting its execution if a severe error is detected. This enables the client to abort the query in progress gracefully and clean up its connections to the inaccessible server. A new connection is opened up as soon as the partition is repaired and the server's presence detected by the client. The aborted query is processed by the inference mechanism as explained above.

5 Ideal Architecture

What the prototype does not do, in hindsight, is demonstrate how distributed databases should be

architected to take advantage of inference techniques. In order to recognize that inference is required to materialize an inaccessible object, the query must first be parsed and the relevant objects identified. As commercial databases do not typically provide an interface below the level of the parser, we are forced to parse the query ourselves, manipulate the parse tree, turn it back into SQL, and submit it to the database through the high level interface. Further, the inference techniques require the invention of new relational operators (eg. open S-union [3, 5]). As we do not have source code to the database engine, we implement the new operators in the address space of the front-end. Consequently, intermediate results frequently have to cross out of the database's address space using a "tuple-at-a-time" interface across the boundary. These factors combine to produce unacceptable performance.

What is needed is a truly open architecture for distributed databases. Intelligent application programs need to be able to interact with a database service other than simply through the high-level language interface. The program should be able to access and modify the parsed query. It should be able to call relational operators directly. Most importantly, the set of relational operators should be *extensible* so that specialized, application-supplied operations like open S-union can execute in the database's address space, avoiding expensive copying. Of course, protection of the integrity of the data must be guaranteed, perhaps through providing extensibility via an interpreted language such as Push [1] (proposed for operating system kernel extensibility for the Raid database [2]).

References

[1] Bharat Bhargava, Enrique Mafla, and John Riedl. Experimental facility for implementing distributed database services in operating systems. Department of Computer Sciences, Purdue University, Submitted for publication, 1990.

[2] Bharat Bhargava and John Riedl. The Raid distributed database system. *IEEE Transactions on Software Engineering*, 15(6), June 1989.

[3] Qiming Chen. A high order logic programming framework for complex objects. In *International Computer Software & Applications Conference (COMPSAC 89)*, 1989.

[4] Wesley W. Chu, Andy Hwang, Qiming Chen, and Rei-Chi Lee. An inference technique for distributed query processing in a partitioned network. Technical Report CSD-900005, Department of Computer Science, University of California Los Angeles, February 1990.

[5] Wesley W. Chu, Andy Hwang, Rei-Chi Lee, and Qiming Chen. Fault tolerant distributed database system via data inference. Proceedings of the Ninth Symposium on Reliable Distributed Systems, October 1990.

[6] Wesley W. Chu, Rei-Chi Lee, and Kuorong Chiang. Capture database semantics by rule induction. Technical Report CSD-900013, Department of Computer Science, University of California Los Angeles, May 1990.

Proceedings of the CIPS Edmonton '90
Canada, Oct. 1990

ON OPEN DATA INFERENCE AND ITS APPLICATIONS*

Wesley W. Chu, Qiming Chen, and Andy Y. Hwang

Computer Science Department
University of California
Los Angeles

ABSTRACT

An open data inference technique is proposed which uses domain and summary knowledge to infer inaccessible data for query processing during network partitions. The open nature of data inference is due to the incomplete knowledge available about data and the need to combine partial inference results from separate processes to derive cooperative answers. To underlie such inference, new algebraic tools are developed for handling incomplete information. Further, a weaker correctness criterion, called *toleration*, is introduced to evaluate inference results. The above concepts have been implemented on a prototype Cooperative Distributed Database system, CDB, at UCLA. Our preliminary experimental results reveal that open inference can significantly improve the availability of distributed databases during network partitions.

1. INTRODUCTION

To improve the reliability and response time in distributed systems, databases are often partitioned into fragments which are replicated and stored at several sites. Such fragment replication requires additional communication and processing overhead to maintain consistency among the replicated copies. Further, due to channel and node failures, a network may be partitioned into two or more isolated parts. Since fragments may not be fully replicated at all sites, certain fragments may be inaccessible during network partitions. Most prior work using syntactic information to handle operations during network partitioning leads to blocking or a partially operable system [GARC87]. However, in many real time applications, the availability of data is of primary importance. It is often not acceptable for a site to suspend processing when it cannot communicate with other sites. Because database attributes are often correlated and contain redundant information (e.g., salary and rank, ship type and ship class), data inference techniques can be used to *infer inaccessible data from correlated and accessible data* during network partitions [CHU90]. Such a knowledge-based approach can greatly increase the availability of distributed database systems. However, in general, such inferences must be made from incomplete information. This is because

1. the incomplete correlated knowledge and source objects.

2. the need to infer data based on partial results from separate reasoning processes as follows :

   (a) inferring the missing data from multiple data sources and combining the partial results;

   (b) inferring data based on partial results from previous phases, goals or subgoals, which may also vary with time and depend on the network status and other events therefore cannot be totally pre-planned;

* This research is supported by DARPA contract F29601-87-C-0072 and ONR contract N00014-88-K0434

(c) cooperative multi-agent systems require combining results from individual agents.

We shall refer to this kind of inference as **open data inference**. It aims at deriving and filtering data from different knowledge sources, and measuring the inference results in situations where the system state is unknown and unknowable. Since other events may occur during the inference which are not predictable, open inference is a particular issue which is part of the intersection of *logic programming* and *null values* in an *open environment*.

Null value is a special case of the incomplete information problem which has been studied extensively in relational database and inference theory. The efforts made from the relational database point of view concentrate on how to understand the meaning of nulls occurring in the query answers. Different interpretations on nulls are proposed. The formal treatment of nulls under the *unknown* interpretation was given [CODD86] [BIS81] [BIS82]. The treatment of nulls under the *does not exist* interpretation was proposed in [ZANI84]. In [VASS79] the problem of managing nulls with both the *unknown* and *does not exist* is taken into account. In the context of the *open world* database [REIT78], Zaniolo [ZANI84] introduced the *no information* interpretation of nulls. [ROTH85] extended this approach by considering *unknown, does not exist* and *no information* interpretations of nulls together. [GOTT88] defined operations which may return *true, false, unknown, does not exist*, and *open* to support a locally-controlled open world database. This allows the definition of portions of a traditional closed world database as open world. A taxonomy on the meanings of nulls is given in [OLA89]. However, these studies are in the context of relational database and without concern about logic reasoning.

From the logic point of view, the absent terms are related to *Skolem constants*. The absence of negative facts was the first concern. This problem was solved by the introduction of a *closed world assumption* [REIT78] by assuming that negative facts may simply be inferred from the absence of their positive counterparts, which has become one of the foundations of deductive databases. The issue of query evaluation for databases containing *existing but unknown* marked nulls was studied in [REIT86]. Incompleteness is also frequently discussed together with indefiniteness, where data are said to be indefinite if they are of the disjunctive form, e.g. *a or b*. This problem was tackled by Reiter with a precise solution on the basis of the proof-theoretic point of view, and also discussed in [LIP79] [WILL88]. The model developed by Lipski allows attribute values to be of either set or range types, a query applied to a database containing incomplete information must specify the answer as the set of values which might possibly or definitely satisfy the query. Additional approaches to incomplete information reasoning use meta-language techniques [LEV81] [KON81] based on information known about the domain of discourse. It is this fact that many studies on this issue are based on *nonmonotonic logics* [ETH88].

However, we cannot directly adopt the above approaches to our open data inference applications for the following reasons. First, the interpretations of nulls studied from the relational database point of view are not integrated to the present logic programming framework underlying our implementation. Second, many of the above approaches are based on rather strong assumptions. For instance, even though the data is unknown, they require that certain definite information (e.g. data existing) must be known. However, the above assumptions are not suitable for open data inference since the system state is unknown and unknowable. The existence of unknown data is still unknown. Therefore, in our approach, we classify the status of data into two states : closed (including nonexisting) and open (unknown, whether existing or nonexisting). Finally, many of the above studies from the logic point of view cover fields of incomplete reasoning concerned with negation, set, disjunction, etc, and in general, nonmonotonic reasoning, but not exactly in the context of open data inference. Our interests are in such issues as the impact of open data inference environment to the present logic programming model theory and the evaluation of the results of open data inference.

Our effort consists of extending the present algebraic notions of logic programming for dealing with open data inference. Thus, extended algebraic tools are developed to handle incomplete objects. Further, a semantic framework is proposed to underlie the open data inference. Our solution is characterized by the following :

(a) Extending the logic programming framework to accommodate incomplete objects.
(b) Supporting dynamic inference planning.

(c) Introducing toleration as a weaker correctness criterion.

In this paper we shall first discuss the characteristics of open data inference. Next, the algebra on incomplete objects is developed. Then the notions of satisfaction and toleration are discussed. Finally, we present the implementation issues which include the inference engine and the combination of inference results from different derivations.

2. THE CHARACTERISTICS OF OPEN DATA INFERENCE

Let us discuss the differences between the open data inference and the conventional data inference based on logic programming. Throughout this paper, we shall use --> and <-- to represent logical implications, and use → to represent mappings.

The deductive database systems developed so far are based on the relational view mechanism or first-order logic programming [GAL84] [RET84]. In a deductive database, relation names are treated as predicates and tuples as predicated atomic formulas. An attempt to answer a query is referred to as satisfying a goal based on prespecified facts and rules. For example, given the following logic program P (variables are represented by capital symbols),

```
{
    p(X,Y,Z) <-- r(X,Y), q(X,Z).
    q(X,Y) <-- s(X,Y).
    r(a,b).
    s(a,c).
}
```

there is a Herbrand base associated with P, denoted as B_P. It is the set of ground atoms that can be formed by using predicate symbols and ground terms from P, such as

{ p(a,b,c), p(a,c,b), ... r(a,b), r(a,c), r(b,c), ... q(a,b), ...s(a,b), s(b,c), ... }

A (Herbrand) model of a given program P is identified with a **subset** of B_P containing at least the facts that can be derived from P [LLOY83]. For example, the following is an interpretation and a model of P.

M = {p(a,b,c), q(a,c), r(a,b), s(a,c)}.

The derived fact p(a,b,c) is satisfied by the above interpretation, i.e., p(a,b,c) ∈ M.

Therefore, in logic programming, both the original and the derived facts appearing in the Herbrand base are represented as ground formulas containing no null value. The notion of logic satisfaction and model are based on Herbrand interpretations formed out of ground formulas. Further, a goal can succeed only if the derivation path from the given facts to the goal is complete.

However, the *open data inference* environment is characterized by the incompleteness of domain knowledge and the need for combining partial inference results from separate reasoning processes for the purpose of utilizing multiple knowledge sources, carrying out dynamically scheduled inferences, or cooperating multiple intelligent agents. The above requirements are not supported by the conventional deductive database approach. To illustrate it, let us consider the following program P':

{

staff1(EMP#, NAME, SEX, OCCUPATION) <--
person(EMP#, NAME, SEX), job(EMP#, OCCUPATION).
person(05, John, δ).
job(05, δ).

}

based on the following relational schema

staff1(emp#, name, sex, occupation).
person(emp#, name, sex).
job(emp#, occupation).

where job(05, δ) and person(05, John, δ) contains an "δ" representing the unknown information. The conventional logic programming framework does not provide notions to derive any new fact based on such incomplete facts. However, by using some reasoning processes, the following new fact may be derived:

staff1(05, John, δ, δ).

Even though the derived information still contains unknown information, it may be used for assisting a decision making or for merging with other intermediate results to make further deductions.

Under the conventional notions of logic programming, it is not possible to include the given incomplete fact job(05, δ) or the inferred incomplete result staff1(05, John, δ, δ) in the Herbrand base B_P'. Further, since a model of P' is a subset of B_P', we do not have a base to determine whether staff1(05, John, δ, δ) holds or not. Thus

1. From the model semantics point of view, a fact may not be included in the models of a program P if it cannot be inferred from the rules and facts given in P. Since there is no notion to accommodate incomplete facts and results in the Herbrand base, a model of a program may never contain any incomplete fact. Further, no weaker correctness criterion is provided to evaluate the incomplete results.

2. From the inference path point of view, the execution order of the inference processes is pre-specified in the conventional logic programming framework. In our inference environment, due to the incompleteness of the available knowledge, the inferred results may not be complete. To infer the missing information, the intermediate results from separate processes have to be merged. Since we cannot predict the outcome of the intermediate results, the merging process must be planned dynamically according to the inferred results. Therefore, totally pre-specified inference paths, as in the conventional logic programming environment, are not adequate for our inference requirements.

To solve these problems, we shall first extend the base of a program to allow both original and derived facts to be incomplete. For instance, the incomplete fact job(05, δ) is accepted to the extended base, as is the derived incomplete fact staff1(05, John, δ, δ). Further, we shall extend the interpretation of a program to allow the involvement of incomplete information, and define such an interpretation as a subset of the above extended base. Thus for example we can have the following interpretation that does not satisfy but yet does not "violate" the program P':

{staff1(05, John, δ, δ), job(05, δ), person(05, John, δ)}.

In other words, we allow the derivation from job(05, δ) and person(05, John, δ) to staff1(05, John, δ, δ) according to the given rule in P'. To evaluate incomplete inference results, we shall introduce an extended correctness criterion which is weaker than the notion of satisfaction.

Next, let us consider the combining of intermediate incomplete results from multiple knowledge sources for obtaining cooperative answers. Assume we need to infer the relation

staff(emp#, name, sex, occupation, status)

by combining the results of two separate programs P_1 and P_2 where P_1 can be used to infer the relation

staff1(emp#, name, sex, occupation)

and P_2 can be used to infer the relation

staff2(emp#, name, sex, status).

The program P_1 is specified as

```
{
    staff1(EMP#, NAME, SEX, OCCUPATION) <--
            person(EMP#, NAME, SEX), job(EMP#, OCCUPATION).
    person(05, John, δ).
    person(06, Smith, male).
    job(05, engineer).
    job(06, δ).
}
```

From program P_1, an open interpretation M_1 containing the derived facts

```
staff1(05, John, δ, engineer).
staff1(06, Smith, male, δ).
```

can be formed. Given the following additional relation schema

wk_status(emp#, status).

the program P_2 is specified as

```
{
    staff2(EMP#, NAME, SEX, STATUS) <--
            person(EMP#, NAME, SEX), wk_status(EMP#, STATUS).
    person(05, δ, male).
    person(06, δ, δ).
    wk_status(05, δ).
    wk_status(06, fulltime).
}
```

From program P_2, an open interpretation M_2 containing the derived facts

```
staff2(05, δ, male, δ).
staff2(06, δ, δ, fulltime).
```

can be formed. Assuming EMP# is the key of the relations "staff1" and "staff2", the above facts can be combined to obtain a (more) complete interpretation containing

```
staff(05, John, male, engineer, δ).
staff(06, Smith, male, δ, fulltime).
```

This combination generally involves the union of different tables with common attributes, the reduction of resulting

relations, and the merging of certain tuples in terms of appropriate algebraic tools as will be discussed in the next section. Therefore, in open data inference, it is necessary to keep as many incomplete partial results as possible so that they can be combined to yield more complete results. Such a combining process is not explicitly specified by the domain-specific rules, but rather is handled by the system as a meta facility and performed according to the goal and the data semantics.

We refer to a program which contains facts (source data) and rules as a *Data Inference Program (DIP)* and refer to the execution of a DIP as a derivation. An open data inference consists of one or more statically or even dynamically planned derivations. Each derivation is the execution of a DIP. In order to develop its semantics, we shall discuss issues such as the algebra of incomplete object reasoning, two different levels of correctness criteria, and the implementation of an inference engine based on the above concepts. In this paper, we do not address the inference planning problem and assume all the rules are consistent.

3. ALGEBRA FOR OPEN OBJECTS

In order to handle incomplete and dynamically reconstructed database objects, we shall discuss the notion of *variable null*, *open object* and *valuation*, and develop extended algebraic operations that are applicable to both *closed objects* (not containing unknown components), and *open objects* (containing unknown components).

Variable Null

The treatment of "incomplete" information in the relational model has been addressed based on the Closed World Assumption (CWA) and Open World Assumption (OWA) [REIT78]. Under CWA, only the facts expressed by the database are true. Thus, a null may be interpreted either as an existing but "unknown" fact [CODD79][BISK81], or as a "non-existing" one [VASS79][ZANI84][CODD86]. Under OWA, besides the facts specified in the database, no further information is available, thus things are left open [RKS85][GOTT88][OLA89]. Since our goal is to develop a model theory for open inference, we concentrate on the impact of incomplete information on the inference process and classify a null as :

- a *variable-null* denoted as " δ " which may be substituted by different actual values, or
- an actual "non-exist" value called *undefined* and denoted as " ϕ ".

Since ϕ is an actual value, it may be used to substitute δ .

Open-Range Object

To model database objects formally, the existence of some finite sets of values referred to as domains is assumed, and the special value " ϕ " is introduced. The product over domains $D_1, D_2, ..., D_n$, denoted as $D_1 \times D_2 ... \times D_n$, is the set of all tuples $[x_1, x_2, ..., x_n]$ such that $\forall i \in \{1,...,n\}$ $x_i \in D_i$. A relation schema, called a *range*, consists of a list of attributes $A_1, A_2, ..., A_n$, where each A_i is a subset of a domain. Unique Name Assumption (UNA) on attributes is assumed. Tuples and relations are generally called range-objects where an attribute value is allowed to be "ϕ" or "δ". A *closed range-object* is free of *variable-nulls*. An *open range-object* contains *variable-nulls*. Thus, a tuple is open if at least one of its attribute values is *variable-null*. A relation is open if it contains at least one open tuple. An open range-object cannot be compared with other range-objects. For example, assuming A,B,C are attribute names, we cannot determine whether [A:1, B:2, C:δ] and [A:1, B:2, C:δ] are equal since both range-objects are left open, and the *variable-nulls* in each tuple may stand for different actual values. To define relationships and operations on open range-objects, it is necessary to extend the notion of equality to represent syntactically identical objects. Therefore, we adopt the notion of *symbolic equality* , denoted as == , as described in [CODD86][GZC87][GOTT88]. Under this notion, all the *variable-nulls* represented by the same notation " δ " are symbolically equal. Further, two tuples are symbolically equal if the values of each attribute are symbolically equal. Two relations are symbolically equal if their tuples are pairwise symbolically equal.

The notion of closed and open objects is related to the notion of closed formula in logic programming. A non-closed formula may contain one or more specific variables such as X, Y, etc. However, in the framework described here, all the unknown components in the open objects are syntactically specified by the same *variable-null* notation.

Valuation

In order to define the notion of satisfaction for open data inference, the concept of *valuation* is introduced. Valuation plays the role of instantiation of variables and *variable-nulls* in terms of actual values under attribute type constraints.

[Valuation]

The set of valuation mappings, Ω, from the set of objects (open or closed) to the set of closed objects are defined as follows :

(a) For a constant value $a \in A$ on attribute A, $a \rightarrow a \in \Omega$.
(b) For a variable z on attribute A, $(\forall a \in A)\ z \rightarrow a \in \Omega$.
(c) For a null-variable δ on attribute A, $(\forall a \in A \cup \{\phi\})\ \delta \rightarrow a \in \Omega$.
(d) For a tuple $t = [A_1{:}t_1, ..., A_n{:}t_n]$, $t' = [A_1{:}t_1', ..., A_n{:}t_n']$,
$\forall i \in \{1,...,n\}\ t_i \rightarrow t_i' \in \Omega$ --> $t \rightarrow t' \in \Omega$.
(e) For a relation $R = \{t_1, ..., t_n\}$, $\forall i \in \{1,...,n\}\ \exists t \in R'\ (t_i \rightarrow t \in \Omega)$ --> $R \rightarrow R' \in \Omega$.

An open object is *partially valuated* if not all the *variable-nulls* are instantiated by actual values. For instance, as shown below, the relation "SHIP" is a partial valuation of another relation "ship" according to the following rules :

IF 'B01' ≤ battle_group ≤ 'B02' THEN radar = 'SPS'.
IF 'S120'≤ ship_id ≤ 'S150' THEN battle_group = 'B03'.

ship			SHIP		
ship_id	battle_group	radar	ship_id	battle_group	radar
S100	B01	δ	S100	B01	SPS
S122	δ	δ	S122	B03	δ

Note that valuations of variable-nulls should not be referred to as substitutions. Since all variable-nulls are represented by the same notation "δ", there is no way to separately represent each variable-null and the value used to substitute for it. The usual concept of substitution in logic is only applicable to distinct variables. However, valuation does obey object typing constraints, thus we state that the valuation of a null-variable δ on attribute A is defined as

$$(\forall a \in A \cup \{\phi\})\ \delta \rightarrow a \in \Omega.$$

Sub-tuple, Sub-membership and Sub-containment on Closed and Open Objects

Data inference consists of mappings between range-objects. Open data inference involves open range-objects. In order to study its interpretation semantics, appropriate algebraic tools are required. This includes special set membership and set containment between range objects, and the extension of these notions for dealing with open range-objects. We first define the *sub-tuple* relationship between tuples. Let t, t' be tuples with attribute list W and

W', and t.X be the attribute value of t on the attribute X; t is the sub-tuple of t', denoted as $t \leq^* t'$, and is defined as:

$$t \leq^* t' \text{ iff } W \subseteq W' \wedge \forall X \in W\,(t.X = \phi \vee t.X = t'.X)$$

For example, we have the following sub-tuple relationships:

$$[a,b] \leq^* [a,b,c], \qquad [a,b] \leq^* [a,b,\delta], \qquad [a,\phi] \leq^* [a,b,\delta].$$

The third sub-tuple relationship holds since the first tuple contains the special non-exist value (i.e., ϕ). We then define a special set membership and set containment [CHEN89a][CHEN89b] called *s-Membership* and *s-Containment*, denoted as $\in^*$ and $\subseteq^*$ respectively. Let t be a tuple and R be a relation, we say $t \in^* R$ if t is a sub-tuple of any tuple in R. Further, let R and S be two relations, we say $R \subseteq^* S$ if every tuple in R is an s-Member of S. That is,

$$t \in^* R \text{ iff } (\exists t' \in R)\, t \leq^* t'.$$

$$R \subseteq^* S \text{ iff } (\forall t \in R)\, t \in^* S.$$

In this case, R is also called the *sub-relation* of S. For example, given the following relations "ship1" and "ship2", we have ship1 $\subseteq^*$ ship2

ship1		ship2		
ship_id	battle_group	ship_id	battle_group	radar
S100	B01	S100	B01	SPS
S122	B03	S122	B03	ϕ
		S130	B01	δ

Now let us extend the above notions to open range-objects. By using symbolic equality $==$, we can develop the notions of *open sub-tuple* $\leq\hat{}$, *open s-Membership* $\in\hat{}$ and *open s-Containment* $\subseteq\hat{}$. Let t, t' be tuples with attribute sets W and W' respectively; t is the sub-tuple of t', denoted as $t \leq\hat{} t'$, and is defined as:

$$t \leq\hat{} t' \text{ iff } W \subseteq W' \wedge \forall X \in W\,(t.X == \delta \wedge t'.X \neq \phi \vee t.X == t'.X \vee t.X = \phi).$$

In short, the sub-tuple relationship $t \leq^* t'$ is extended to $t \leq\hat{} t'$ by allowing t to contain *variable-nulls* (i.e. δ). For example, we have the following open sub-tuple relationships: $[a,\delta] \leq\hat{} [a,b]$ and $[a,\delta] \leq\hat{} [a,\delta]$. Consequently, we can also introduce open s-Membership $\in\hat{}$ and open s-Containment $\subseteq\hat{}$ similar to the s-Membership and s-Containment shown above. Let t be a tuple and R, S be relations, then

$$t \in\hat{} R \text{ iff } (\exists t' \in R)\, t \leq\hat{} t'.$$

$$R \subseteq\hat{} S \text{ iff } (\forall t \in R)\, t \in\hat{} S.$$

We call a relation *s-reduced* if none of the tuples in the relation is the sub-tuple of another tuple. For the set of s-reduced relations, it can be proved that the $\subseteq\hat{}$ relationship is reflexive, transitive, and antisymmetric. We can further show that the set of s-reduced relations form a partial order lattice under the $\subseteq\hat{}$ relationship.

It is easy to see that the $\leq^*$, $\in^*$, $\subseteq^*$ relationships are special cases of the corresponding open relationships. In fact, when we say that relation R is openly contained in relation S under $\leq\hat{}$ or $\subseteq\hat{}$ relationship, we mean that S contains R, or S contains a valuation of R, or S contains a partial valuation of R.

The reason $\subseteq^{\wedge}$ is weaker than $\subseteq^{*}$ can be explained as follows : let R and S be two relations; $R \subseteq^{\wedge} S$ implies that there exists valuations from R to R' and from S to S' such that $R' \subseteq^{*} S'$. Conversely, if $R \subseteq^{\wedge} S$ does not hold, no such valuations exist. More generally, we have

Theorem 3.1

Let R and S be two relations. $R \subseteq^{\wedge} S$ implies that for each $S \rightarrow S' \in \Omega$, there exists a closed relation R', such that

$$R \rightarrow R' \in \Omega \wedge R' \subseteq^{*} S'.$$

Proof outline : see appendix.

The notions introduced in this section provide us with the mathematical tools for handling the interpretation semantics of data inference involving open objects in the next section.

4. SATISFACTION AND TOLERATI

In logic programming, the model of a program is the interpretation which satisfies all the rules and facts specified in that program. In open data inference, since rules may not be sufficient for inducing all the necessary data, both intermediate and final results may be left open. Since intermediate results can be used as base objects for further inferencing, the base data objects in a DIP may be open. This requires us to study the interpretation semantics in which both source and target data may be incomplete.

4.1 F-base

In logic programming, the static Herbrand base B_P of a program P contains only the predicate symbols appearing in P, and any possible (Herbrand) interpretation of P is identified with a subset of B_P [LLOY83]. Therefore, only predefined predicates (relations) can be derived during the inference. To accommodate dynamically generated predicates during open inference, under the UNA on attributes, we need to extend the base of a program to the following. Let P be a DIP and $A = \{A_1,...,A_n\}$ be the set of attributes that appear in the relational schema in P. The F-base Φ_P of P is the set of all products over the power set elements of A, except $\emptyset$, that is,

$$\Phi_P = \{ A_1,..., A_n, A_1 \times A_2, A_1 \times A_3, ..., A_1 \times A_2 \times A_3, ... \}$$

At the relation level, each element of Φ_P is associated with a virtual relation which can be syntactically (may not be semantically) constructed by using attributes in P, referred to as the base of that virtual relation. The instance of an actual relation R in P is a subset of its base. For a DIP in the F-base, there are no restrictions to forming relation schemas. Relations and views are not fixed to the ones predefined in the DIP. Therefore, predicates, which do not occur in the original facts and rules can be generated during the inference. This is the major difference of the F-base defined here from the Herbrand base that is defined in the logic programming theory.

This notion allows us to discuss the interpretations of intermediate relations, viewed as sets of intermediate predicates or atomic formulas, which are not predefined in the rules but generated by the inference process at runtime.

4.2 Closed Interpretation and Open Interpretation

In our system, a DIP consists of relations and rules (tuple or relation oriented). An interpretation I of a data inference program P is drawn from Φ_P such that

$$(\forall r \in I)((\exists R \in \Phi_P)\ r \subseteq^* R)).$$

That is, every relation in the interpretation is the sub-relation of a relation in Φ_P. The general form of an interpretation is

$$I = \{R_1, ..., R_n\}$$

An interpretation is closed if it contains only closed relations; an interpretation is open if it contains at least one open relation. There exist valuation mappings from open interpretations to closed interpretations. In the following discussions, for simplicity, interpretations are handled at the relation level rather than at the tuple level.

4.3 The Notions of Satisfaction and Toleration

The inference results can be evaluated in terms of two levels of correctness criteria: satisfaction and toleration. The notion of satisfaction is usually for closed objects. We shall now extend the meaning of satisfaction for the open data inference which involves open range-objects. The satisfaction of a possibly open range-object in a DIP by an interpretation I means I contains an appropriate valuation of that range-object. Further none of the rules specified in that DIP are not violated.

[Satisfaction]

Let I be an interpretation. The notion of satisfaction, denoted as $\models$, is defined as

a) For a tuple t, $I \models t$ iff $(\exists R \in I)\ t \rightarrow t' \in \Omega \wedge t' \in^* R$.
For a relation r, $I \models r$ iff $(\exists R \in I)\ r \rightarrow r' \in \Omega \wedge r' \subseteq^* R$.
b) For a rule $h \leftarrow b_1,...,b_n$, for a substitution θ_I based on I,
$I \models (h \leftarrow b_1,...,b_n)$ iff $I \models b_1\theta_I, ..., I \models b_n\theta_I$ implies $I \models h\theta_I$.
c) For a data inference program P, $I \models P$ iff $\forall p \in P\ (I \models p)$.

For example, as shown below, range-object "SHIP" is correctly derived from another range-object "ship" based on the rule r. In this case, the DIP containing range-object "ship" and rule "r" is satisfied by the interpretation {SHIP}.

rule r : IF 'S120' ≤ ship_id ≤ 'S150' THEN battle_group = 'B03'.

ship

ship_id	battle_group
S100	B01
S122	δ

SHIP

ship_id	battle_group
S100	B01
S122	B03

For a Data Inference Program, P, and an interpretation I of that program, we say I is the model of P iff $I \models P$. In the above example, {SHIP} is a model of the given DIP.

The execution of an open data inference may not yield a model containing complete and exact information but rather a tolerant interpretation containing partial information. To accommodate this, we introduce a weaker correctness notion, called toleration, denoted as $\vdash$. In general, a derivation program is *tolerated by an interpretation* if the known facts and rules of the program are not violated and there exist valuations of the open objects involved in the interpretation that makes the interpretation satisfy the program.

[Toleration]

Let I be an interpretation. The notion of toleration, denoted as $\vdash$, is defined as

a) For a tuple t, $I \vdash t$ iff $(\exists R \in I)\ t \in^{\wedge} R$.
For a relation r, $I \vdash r$ iff $(\exists R \in I)\ r \subseteq^{\wedge} R$.
b) For a rule $h \leftarrow b_1,\ldots,b_n$,
$I \vdash (h \leftarrow b_1,\ldots,b_n)$ iff $I \models (h \leftarrow b_1,\ldots,b_n)$.
c) For a data inference program P, $I \vdash P$ iff $\forall p \in P\ (I \vdash p)$.

Now let us observe the following example, where range-object "$SHIP_1$" is partially valuated from another range object "$ship_1$" based on the rule r :

IF 'S120' ≤ ship_id ≤ 'S150' THEN battle_group = 'B03'.

$ship_1$

ship_id	battle_group
S100	B01
S122	δ
S300	δ

$SHIP_1$

ship_id	battle_group
S100	B01
S122	B03
S300	δ

Let $I_1 = \{SHIP_1\}$ be an interpretation. For the given DIP containing rule r and range object $ship_1$, we cannot say that I_1 satisfies the DIP since I_1 is still open. However, $SHIP_1$ is indeed a reasonable derivation of $ship_1$ although it is still open. In general, when a possibly open range-object in a DIP is tolerated by an interpretation I, then I contains an appropriate *partial* valuation of that range-object and the rules in the DIP are not violated. For instance, in the above example, the execution of the given DIP containing rule r and range object $ship_1$ generates a tolerant interpretation $I_1 = \{SHIP_1\}$. We can say range object $ship_1$ is tolerant by the interpretation I_1 since I_1 contains an appropriate *partial* valuation of $ship_1$ and I_1 does not violate rule r.

From theorem 3.1 and the definition of toleration, we can have the following :

Theorem 4.1

Let P be a data inference program and I be an interpretation of P. Then $I \models P \rightarrow I \vdash P$

Proof outline : see appendix.

5. IMPLEMENTATION

An experimental data inference system has been implemented on a prototype cooperative distributed database, CDB, running on a set of Sun 3/60 workstations interconnected by an Ethernet at UCLA. The data inference system is based on the relational model where all the source and target data objects are relations. The inference actions are extensions of the relational operations which allow us to build the inference engine on top of Sybase, a re-

lational database system. Currently, two types of rules are available :

1. Deductive rules specified in terms of relational operations.
2. Correlated rules which are specified as summarized knowledge. This consists of condition and action parts such as "if 'S120' ≤ ship_id ≤ 'S150' then battle_group = 'B03'".

A rule induction technique is used to extract correlated knowledge between attributes from the database contents. In our implementation, only correlations between individual attributes are used. Although a set of attributes (two or more) may also infer the value of another set of attributes, the efficient selection of correlated sets is difficult due to the combinatorial explosion of such correlations. To induce rules between attributes X and Y, we use relational operations to retrieve instances of (X,Y) pairs from the database, and then select those pairs in which X has a unique corresponding Y value. For a detailed algorithm, interested readers should refer to [CHU90]. The acquired rules are summarized in the range form, as

$$\text{IF } x_1 \le X \le x_2 \text{ THEN } Y = y.$$

or in the set form, as

$$\text{IF } X \in \{x_1, x_2, .., x_n\} \text{ THEN } Y = y.$$

When a network partition occurs, the inference system operates in the following way :

a) Based on the given query, object availability status, database schema and correlated knowledge stored in the rule base, the inference system develops a plan which consists of a set of derivations and the execution sequence of those derivations. Each derivation process represents a derivation from certain available data objects to an intermediate or final data inference result.

b) Carry out the inference plan.

Three general types of derivations are implemented in the system :

1. Deriving new relations based on certain source relations. It is specified as relational views and implemented through the view generation mechanism.
2. Valuations of incomplete relations based on summary information and correlated knowledge. The valuation process is implemented through the relation alteration mechanism.
3. Combining intermediate results via appropriate system operations (viewed as meta-rules) to keep all the necessary open tuples appearing in the intermediate results. These tuples may be valuated through other derivations or combined with the data obtained from other derivations.

In our approach, type 1 and type 2 derivations are treated as basic units for type 3 derivation. To infer the missing information, the data inference system first selects certain type 1 and type 2 derivations. Since our inference approach is designed to combine results from different derivations, the improper combination of results may generate redundant information. Further, although each derivation may provide valid information, an arbitrary combination of different results may generate invalid information. Therefore, proper operations and control rules are required such that neither redundant nor invalid information is generated.

The meta-operations for combining tables consists of *natural outer-join* [DATE83], *reduction* and *merge* operations. *natural outer-join* is is used to combine relations involving incomplete data by

generating their least upper bound. *Reduction* is used to remove redundant information in the same table. Such redundancy exists when a tuple appears more than once in the same table. Further, a tuple may not be able to provide extra information if it is the sub-tuple of other tuple(s) in the same table. Removing such a tuple does not lose any information since all the information it can provide can also be provided by the other tuple(s). Furthermore, in our inference environment, *natural outer-join* is combined with *merge* operation to integrate inference results from different derivations. We say the merge result is safe with respect to the original database if no invalid information is generated. This requires the provision of data schema information. For ex-, two tuples can be merged when they have the same key value. Therefore, data inference has to be integrated with knowledge acquisition and schema design stage so that the required knowledge for inferencing is provided.

c) Select the required data objects from the inference result. In the current implementation, the target objects to be inferred are relations. The inference process infers as much of the missing relations as possible. In some case an iterative inference is required up to the saturation of the derived result.

As a data inference example, consider a distributed database that consists of three database fragments: SHIP(ship_id,sname,class), INSTALL(ship_id,weapon), CLASS(class,type,tname) which are stored at sites LA, SF and NYC respectively. When the site SF is partitioned, the following query cannot be answered since relation INSTALL is not accessible :

Q1 : " Find the ship names that carry weapon 'AAM01' "

Since the target objects are relations, our inference engine needs to make an inference plan, selecting relevant inference paths for inferring the missing relation INSTALL. We have not yet implemented a dynamic inference plan. Currently, to infer the missing relation the inference engine exhaustively searches all the derivations in the knowledge base and selects the relevant derivations. In this example, the following two derivations are used to infer the missing INSTALL relation :

DERIVATION 1 : select ship_id, type from SHIP, CLASS
DERIVATION 2 : CLASS(type) --> INSTALL(weapon)

Derivation 1 represents the first type of derivation where the deductive rule is expressed by a view definition. This derivation creates a temporary relation which contains ship_id and type information. Derivation 2 illustrates the second type of derivation, where derivation is performed from [type] to [type,weapon]. This derivation also creates a temporary relation with shiptype and weapon information. While information of shiptype is filled by accessing CLASS object, weapon information is filled based on the provided correlated rules between shiptype and weapon. The above two intermediate results are then combined by the third type of derivation. The resulting relation is used to replace the inaccessible information.

6. CONCLUSIONS

We have proposed the use of open data inference for distributed query processing to improve database availability. This open nature of data inference is due to the incomplete knowledge about data and the need of combining partial inference results from separate reasoning processes. New algebraic tools are developed to support such inference. To evaluate inference results under incomplete knowledge, a weaker correctness criterion, called *toleration*, is introduced. This open inference technique has been implemented at UCLA on a prototype cooperative distributed database system (CDB). Our experience reveals that the proposed open inference approach can significantly improve the availability of the distributed database during network partitions.

REFERENCES

[BIS81] Biskup, J. "A Formal Approach to Null Values in Database Relations", In Advance in Database theory, Vol.1, Plenum, New York, 1981.

[BIS82] Biskup, J. "A Foundation of Codd's Relational Maybe-operations", Tech. Rep. Computer Science Department, Univ. of Dortmund, West Germany, 1982.

[BROD84] Brodie, M., J. Mylopoulos, and J. W. Schmidt (eds.) *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer, New York, 1984.

[CHEN89a] Chen, Qiming "A High Order Logic Programming Framework for Complex Objects reasoning", International Computer Software and Applications Conference (COMPSAC89), 1989, USA.

[CHEN89b] Chen, Qiming and Wesley Chu, "A High Order Logic Programming Language (HILOG) for NON-1NF Deductive Databases", Proc. of 1st International Conference on Deductive and Object-Oriented Databases, 1989, Japan.

[CHU90] Chu, Wesley, Andy Hwang, Rei-Chi Lee, Qiming Chen, Matthew Merzbacher, and Herbert Hecht, "Fault Tolerant Distributed Database via Data Inference", 9th Symposium on Reliable Distributed System, Huntsville, Alabama, October, 1990.

[CODD86] Codd, E. F. "Missing Information (Applicable and Inapplicable) in Relational Databases, SIGMOD RECORD, Vol. 15, no. 4 December 1986.

[DATE83] Date, C. J. "The Outer Join", Proc. of the 2nd International Conference on Databases, UK, 1983.

[ETH88] Etherington, D. "Reasoning with Incomplete Information", Morgan Kaufmann Publishers Inc. 1988.

[GALL84] Gallaire, H., J. Minker, J. Nicolas, "Logic and Databases : A Deductive Approach", ACM Computing Surveys, Vol. 16, No 2, June 1984.

[GARC87] Garcia-Molina, H. and R. K. Abbott, "Reliable Distributed Database Management", *Proc. of the IEEE*, May 1987, pp. 601-620.

[GOTT88] Gottlob, Georg and Roberto Zicari, "Closed World Databases Opened Through Null Values", Proc. of 14th VLDB Conference 1988, pp. 50 - 61

[HAMM81] Hammer, M., and D. McLeod, "Database Description with SDM: A Semantic Database Model," *ACM Transactions on Database Systems*, Vol. 6, No. 3, September 1981.

[KON81] Konolige, K. "A Metalanguage Representation of Databases for Deductive Question Answering System", Proc. of 7th IJCAI, 1981.

[LEV81] Levesque, H. "The Interaction with Incomplete Knowledge Bases : A Formal Treatment", Proc. of 7th IJCAI, 1981.

[LIP79] Lipski, W. "On semantic issues connected with incomplete information databases", ACM Trans. on Database Systems 4 (1979).

[LLOY83] Lloyd, J. "Foundation of Logic Programming", Springer-Verlag, 1983.

[OLA89] Ola, Adegbemiga and Gultekin Ozsoyoglu, "A Family of Incomplete Relational Database Models", Proc. of 15th VLDB pp. 23 - 31

[REIT78] Reiter, R. "On Closed World Databases Logic and Databases", PLenum, New York, 1978.

[REIT84] Reiter, R. "Towards a Logical Reconstruction of Relational Database Theory", in On Conceptual Modeling, pp 191-234, Springer- Verlag Ed ., 1984.

[REIT86] Reiter, R. "A Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values", JACM Vol.33, No.2 April 1986.

[ROTH85] Roth, M., H. Korth, and A. Silberschats, "Null values in Non-1NF Relational Databases, Rep. TR-85-32, Univ. of Texas at Austin, 1985.

[VASS79] Vassiliou, Y. "Null Values in Database Management : A Denotational Semantics Approach", ACM-SIGMOD 1979, pp. 162 - 169

[WIL88] Williams, M. and Q. Kong, "Incomplete Information in a Deductive Database", Data and Knowledge Engineering 3 (1988).

[ZANI84] Zaniolo, C. "Database Relations with Null Values", Journal of Computer and System Science, Vol 28, No. 1, 1984 pp. 142 - 166

APPENDICES

Theorem 3.1

Let R and S be two relations. $R \subseteq^{\wedge} S$ implies that for each $S \rightarrow S' \in \Omega$, there exists a closed relation R', such that

$$R \rightarrow R' \in \Omega \wedge R' \subseteq^{*} S'.$$

Prove Outline :

We denote the set of closed relations valuated from S as Γ_S. Based on the definitions of valuation, closed s-containment and open s-containment, the theorem can be proven in the following cases :

case 1 : R and S are both closed, then simply $\Gamma_S = \{S\}$ and $R \rightarrow R \in \Omega \wedge R \subseteq^{*} S$.

case 2 : R is open and S is closed. Clearly $\Gamma_S = \{S\}$. There exists a valuation from R to R' defined as follows : for any attribute X and any such tuple $r \in R$ that $r.X = \delta$, valuate r.X to r'.X ($r' \in R'$) such that $r'.X = s.X$ ($s \in S$). By the definition of $\subseteq^{*}$, it can be shown that $R' \subseteq^{*} S$.

case 3 : R is closed but S is open. By the definition of $\subseteq^{\wedge}$ and $\subseteq^{*}$ $R \subseteq^{*} S$. Therefore for any S' where $S \rightarrow S' \in \Omega$, $R \subseteq^{*} S'$. Thus $R \subseteq^{*} S'$ for any S' in $\Gamma_S = \{S' \mid S \rightarrow S' \in \Omega\}$.

case 4 : R and S are both open. By the definition of $\subseteq^{\wedge}$ and valuation, for any S' where $S \rightarrow S' \in \Omega$, $R \subseteq^{\wedge} S'$. As described in case 2, for any S' valuated from S, there exists a relation R' valuated from R such that $R' \subseteq^{*} S'$.

Theorem 4.1

Let P be a data inference program and I be an interpretation of P, $I \models P \dashrightarrow I \vdash P$

Prove Outline :

For satisfaction, given a tuple t, $I \models t$ iff $(\exists R \in I)\ t \rightarrow t' \in \Omega \wedge t' \in^{*} R$; given a relation r, $I \models r$ iff $(\exists R \in I)\ r \rightarrow r' \in \Omega \wedge r' \subseteq^{*} R$. However, for toleration, given a tuple t, $I \vdash t$ iff $(\exists R \in I)\ t \in^{\wedge} R$; given a relation r, $I \vdash r$ iff $(\exists R \in I)\ r \subseteq^{\wedge} R$. From the definitions of closed and open relationships between range objects, if $t \rightarrow t' \in \Omega \wedge t' \in^{*} R$ holds, then $t \in^{\wedge} R$ holds; if $r \rightarrow r' \in \Omega \wedge r' \subseteq^{*} R$ holds, then $r \subseteq^{\wedge} R$ holds.

Using Type Inference and Induced Rules to Provide Intensional Answers

Wesley W. Chu, Rei-Chi Lee and Qiming Chen

Computer Science Department
University of California, Los Angeles
Los Angeles, California 90024

ABSTRACT

An intensional answer provides characteristics rather than a listing of all the instances that satisfy a query. This paper presents a new approach that uses knowledge induction and type inference to provide intensional answers. Machine learning techniques are used to analyze database contents and induce a set of If-then rules. Type inference which is based on forward inference and backward inference is developed that uses database type hierarchies to derive the intensional answers for a query. Further, more precise intensional answers can be derived by properly merging the type inference results from multiple type hierarchies. A prototype intensional query processing system which uses the proposed approach has been implemented. Using a ship database as a test bed, we demonstrate the effectiveness of the use of type inference and induced rules to derive specific intensional answers.

1. Introduction

Conventional database systems provide answers in the form of an enumeration of database instances retrieved from the database. Although such an answer conveys information to the users, a general description of the answer or summarized or approximate answers are often more useful. Meta-data of the database such as integrity constraints and semantic rules can be used to infer hidden information within the database. For example, integrity constraints were used to improve query processing performance [KING81, HAMM80] and to derive intensional answers [MOTR89].

Type hierarchies specify the subtype and supertype relationships in a database application domain. This can be used to improve query processing [CHU90] and to provide an aggregate response to queries [SHUM88]. Such an aggregate response can be provided to the users as the intensional answer. However, only very limited forms of intensional answers can be generated by using the type hierarchy alone without database intensional knowledge. To remedy this problem we propose to use database intensional knowledge to derive more specific intensional answers. As in many other systems, one of the major problems of knowledge base design is the acquisition of knowledge. In this paper, we propose an approach which first uses rule induction to derive intensional knowledge by analyzing database contents, and then uses the generated rules to derive more specific intensional answers based on the type hierarchy. The approach consists of two phases: the knowledge acquisition and the intensional query answering. In the knowledge acquisiton phase, a model-based knowledge acquisition methodology is proposed to facilitate the knowledge acquisition. A Knowledge-based Entity-Relationship (KER) model is developed for specification of the induced knowledge. Based on the type hierarchy, rules can then be generated and maintained in the knowledge base. In the intensional query answering phase, intensional answers are derived by *type inference* via traversing the type hierarchies. Query condition is used to direct the traversal direction. The process stops when the desired type specifications are obtained then provided as the intensional answers.

In this paper, we will first present the methodology that uses database schema and knowledge induction techniques to extract useful meta-data from the database. Next, we present the use of induced rules and type hierarchies to derive intensional answers. Finally, we use a ship database as a testbed and present examples that use the proposed type inference to derive intensional answers.

2. Knowledge Induction

2.1 Database Semantics

To construct the database schema, objects with similar characteristics or properties are grouped into object types and subtypes. These semantics, referred to as *classification semantics* or *classification characteristics*, are useful in knowledge-based data processing. Table 1 presents an example of the navy battleship characteristics that classify ships into ship types with different displacement ranges. These characteristics are database semantics describing the ship database domain and can be used to derive intensional answers. Since the database instances follow these characteristics, these characteristics can be induced by the following model-based learning methodology.

Category	Type	Type Name	Displacement (in tons)		
Subsurface	SSBN	Ballistic Nuclear Missile Submarine	7250	-	16600
	SSN	Nuclear Submarine	1720	-	6000
Surface	CVN	Attack Aircraft Carrier	75700	-	81600
	CV	Aircraft Carrier	41900	-	61000
	BB	Battleship	45000	-	45000
	CGN	Guided Nuclear Missile Cruiser	7600		14200
	CG	Guided Missile Cruiser	5670	-	13700
	CA	Gun Cruiser	17000		17000
	DDG	Guided Missile Destroyer	3370		8300
	DD	Destroyer	2425	-	7810
	FFG	Guided Missile Frigate	3605		3605
	FF	Frigate	2360	-	3011

Table 1. Classification Characteristics of Navy Battleships

* This research is supported in part by the DARPA Contract F29601-87-C-0072, the ONR Contract N00014-88-K-0434, and the Hughes Micro Contract 90-032.

CH2968-6/0000/0396$01.00 © 1991 IEEE

2.2 Model-based Learning Methodology

The acquisition of knowledge is one of the most difficult problems in the development of a knowledge-based system. Currently, knowledge acquisition is still largely a manual process which is very time-consuming. Further, it is often not possible for domain experts to describe their expertise to others. To remedy this problem, machine learning techniques can be used to construct the knowledge base. Inductive learning [QUIN79, MICH83] is a machine learning technique that has been used in AI research. For a given concept and a set of training examples representing the concept, it finds a description for the concept such that all positive examples satisfy and all negative examples contradict the description. Thus, using the database contents as the set of training examples, object classification characteristics embedded within the database can be induced. Since a database schema is created by the designer based on the semantic of the application, such semantic can be used as the candidates for rule induction. Therefore, we propose to use machine learning to acquire database characteristics and to use the database schema to guide the rule induction process.

The semantic knowledge associated with each database are: *intra-object knowledge* and *inter-object knowledge*. Intra-object knowledge defines specific properties of each entity set such as the attribute domains, value ranges, relationships between attributes, etc., and restricts the allowable instances of an entity set. For example, the displacement of an Attack Aircraft Carrier is in the range of 75,700 tons - 81,600 tons. The inter-object knowledge specifies the constraints that the instances of a relationship set must satisfy. For example, the relationship VISIT involves entities of SHIP and PORT and satisfies the constraint that the draft of the ship must be less than the depth of the port. The inter-object knowledge can be induced from the interrelationship between SHIP and PORT linked by the VISIT relationship.

2.3 The Knowledge-based (KER) Data Model

To enhance the modeling of such capabilities as type hierarchy and knowledge specification, we introduce a Knowledge-based E-R (KER) model, an extension of the Entity-Relationship Model [CHEN76] and provides the following three generic constructs of data modeling [BROD84, HAMM81, MCLE82]:

1. **has/with** (*aggregation*) which links an object with another object and specifies a certain property of the object (e.g., a CLASS has an instructor);

2. **isa/with** (or **contains/with**) (*generalization/specialization*) which links an object type with another object type and specifies an object as a subtype of another object (e.g., PROFESSOR is-a subtype of PERSON or PERSON contains PROFESSOR, STUDENT, and STAFF);

3. **has-instance** (*classification*) which links a type to an object that is an instance of that type (e.g., "John Smith" is an instance of PROFESSOR).

Note that in addition to the semantic constructs provided by most semantic data models, KER also provides knowledge specification which is represented by the *with-constraint* information. Such knowledge specification associated with each database definition is useful for knowledge-based data processing.

In KER, an entity is a distinctly identified object, for example, a specific person, a department, or a course. An entity set is a collection of entities. Each of these entities is distinguished by a unique identifier. The set of unique identifiers is called the primary key of the entity set. A relationship specifies the connections between different entities. Conceptually, both entity type and relationship type can be considered as object type and can be modeled using the has/with construct. For example, Figure 1 shows an object type SUBMARINE represented in KER.

```
object type SUBMARINE

  has
  key:     ShipId        domain:   char[10]
  has:     ShipName      domain:   char[20]
  has:     ShipType      domain:   char[4]
  has:     ShipClass     domain:   char[4]
  has:     Displacement  domain:   integer
  has:     Fleet         domain:   integer

  with     Displacement  in [2000..30000]
```

Figure 1. The KER representation of an object type SUBMARINE.

A type hierarchy uses specialization/generalization constructs (isa or contains relationships) to define the subtype and supertype relationships. For example, SSBN (Ballistic Nuclear Missile Submarine) is a subtype of SUBMARINE, and CLASS-0101 is a subtype of SSBN, and therefore, a type hierarchy consisting of SUBMARINE, SSBN, and CLASS-0101 is formed (see Figure 2).

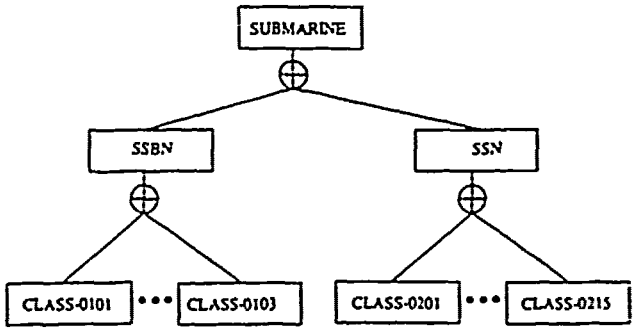


Figure 2. A Type Hierarchy SUBMARINE

A subtype inherits all the properties of its supertypes, unless the properties already have been redefined in the subtype. For example, type SUBMARINE has attributes ShipID ,and ShipName, and type SSBN has attribute TypeID and TypeName; subtype CLASS-0101 will automatically inherit properties ShipID and ShipName from supertype SUBMARINE, and inherit properties TypeID and TypeName from another supertype SSBN.

A subtype can also be derived from another type by providing a *derivation specification*. For example, one can define a subtype SSBN (all the ships with ship type SSBN) of type SUBMARINE by specifying:

SSBN isa SUBMARINE with ShipType = "SSBN"

The with-clause defines the derivation specification of the subtype SSBN. It can also be considered as associating a constraint with this subtype.

The type hierarchy is represented in KER as:

E_1 isa E with Ψ_1
E_2 isa E with Ψ_2
...
E_n isa E with Ψ_n

or alternatively, it can also be represented as:

E contains $E_1, E_2, ..., E_n$ with Ψ.

This definition states that the instances of E can be divided into n disjoint subsets $E_1, E_2, ..., E_n$, with the constraint Ψ. Each E_i is a subtype of E.

To provide a graphical representation of the interrelationships among the entity types/subtypes, relationship types, and derivation specification, we can extend the ER diagram by adding the type hierarchy with constraint representation as shown in Figure 3. A representation of a ship database schema by the KER Diagram is shown in Figure 4.

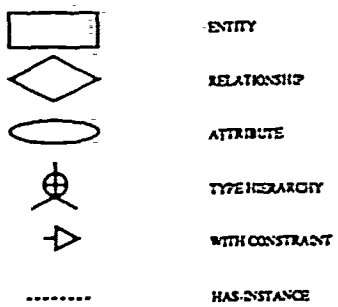


Figure 3. Components of the KER Diagram

2.4 A Rule Induction Example

Rule induction is a process to generate a set of rules to classify objects into classes called classification. In databases, objects with similar properties are defined as the same object type. Thus, object hierarchies may be used to form such a classification. The inputs for rule induction are object instances, schema describing object types hierarchies, and criteria to evaluate the classification quality. In our implementation, the set of object instances is represented as relations and the database schema is represented in the KER model. The induction system generates the classification characteristics for each class based on the object hierarchy.

To illustrate the proposd rule induction and learning methodology, we use the nuclear submarine portion of a ship database* as a test bed which consists of the following relations (For a sample database instances, see Appendix):

SUBMARINE = (*Id, Name, Class, Fleet*)
CLASS = (*Class, ClassName, Type, Displacement*)
TYPE = (*Type, TypeName*)
SONAR = (*Sonar, SonarType*)
INSTALL = (*Ship, Sonar*)

The database consists of five entity types: SUBMARINE, CLASS, TYPE, SONAR, SONAR_TYPE and one relationship type: INSTALL. The three entity types SUBMARINE, TYPE, and CLASS form a type hierarchy for SUBMARINE, submarines with different fleet number also form a type hierarchy for SUBMARINE, and the entities SONAR and SONAR TYPE form another type hierarch as shown in Figure 4. Each submarine type contains a set of submarine classes and each submarine class contains a set of submarine instances. For example, Submarines are divided into two types: SSBN (Ballistic Nuclear Missile Submarine) and SSN (Nuclear Submarine). The SSBN ships contain three classes of ships: 0101 (Ohio), 0102 (Benjamin Franklin), and 0103 (Lafayette), and there are three ships that belong to the ship class 0103 (Lafayette). Each ship class has its specific characteristics such as displace-

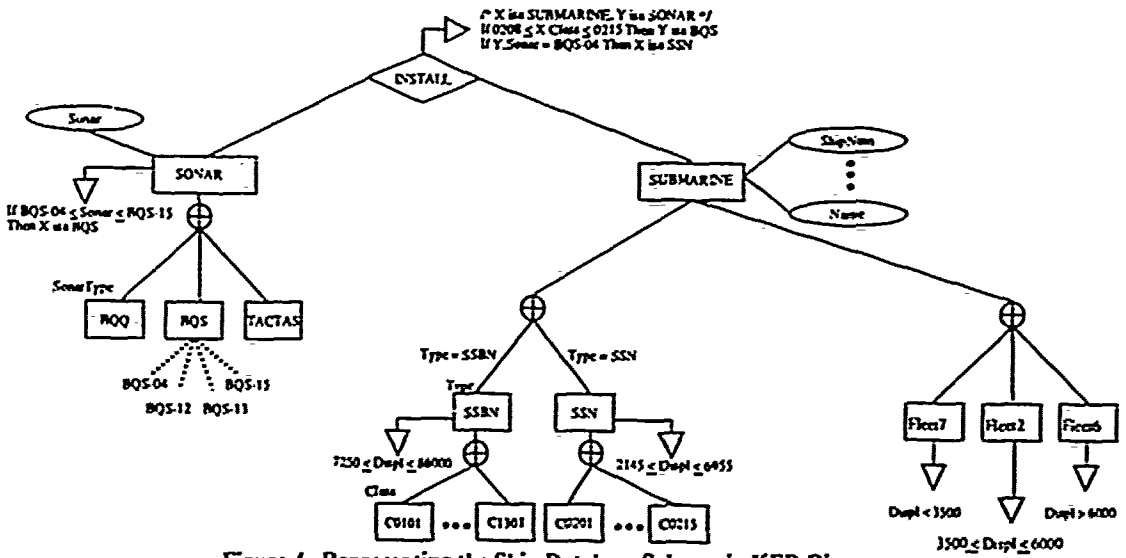


Figure 4. Representing the Ship Database Schema in KER Diagram

* The ship database was created by the System Development Corporation (now UNISYS) to provide a generic naval database based on [JANE81].

ment, length, beam, etc. For tactical or strategic reasons, different sonars are installed on different ships. The relationship INSTALL indicates the sonars installed on the different ships.

Applying the proposed knowledge acquisition technique to the ship database, rules are generated and grouped by the object types as follows:

(1) SUBMARINE

R_1: if SSN623 ≤ *Id* ≤ SSN637 then x isa C0103
R_2: if SSN648 ≤ *Id* ≤ SSN665 then x isa C0204
R_3: if SSN673 ≤ *Id* ≤ SSN686 then x isa C0204
R_4: if SSN692 ≤ *Id* ≤ SSN704 then x isa C0201
R_5: if 0101 ≤ *Class* ≤ 0103 then x isa SSBN
R_6: if 0201 ≤ *Class* ≤ 0215 then x isa SSN
R_7: if Skate ≤ *ClassName* ≤ Thresher then x isa SSN
R_8: if 2145 ≤ *Displacement* ≤ 6955 then x isa SSN
R_9: if 7250 ≤ *Displacement* ≤ 30000 then x isa SSBN
R_{10}: if *Displacement* < 3500 then x isa Fleet_2
R_{11}: if 3500 ≤ *Displacement* ≤ 6000 then x isa Fleet_6
R_{12}: if *Displacement* > 6000 then x isa Fleet_7

(2) SONAR

R_{13}: if BQQ-2 ≤ *Sonar* ≤ BQQ-8 then x isa BQQ
R_{14}: if BQS-04 ≤ *Sonar* ≤ BQS-15 then x isa BQS

(3) INSTALL (x isa SUBMARINE and y isa SONAR)

R_{15}: if SSN582 ≤ x.*Id* = SSN601 then y isa BQS
R_{16}: if SSN604 ≤ x.*Id* = SSN671 then y isa BQQ
R_{17}: if x.*Class* = 0203 then y isa BQQ
R_{18}: if 0205 ≤ x.*Class* ≤ 0207 then y isa BQQ
R_{19}: if 0208 ≤ x.*Class* ≤ 0215 then y isa BQS
R_{20}: if y.*Sonar* = BQS-04 then x isa SSN

3. Intensional Query Answering

A relational database is made up of the extension database (EDB) and the intension database (IDB) [GALL78, NICO78]. The EDB is the set of tuples contained in the relations. It is expressed in *relations* over domain values. The IDB is the set of general rules (i.e., meta-data) about data stored in the EDB. It is expressed in closed well-formed formulas in first-order predicate calculus.

An answer to a query is the set of data values that satisfy a qualification specified in the query. Generally, query answers are retrieved from the EDB. An *intensional answer* to a query provides the characterizations of the set of data values that satisfies the query [MOTR89]. In many applications, users are satisfied with or prefer to obtain summarized answers rather than the answers from the EDB. Such summarized or abstract answers can be represented as intensional answers.

The intra- and inter-object knowledge specifying the inter-relationships between the database objects are the essential components of the intensional database. This knowledge can be induced by our model-based knowledge acquisition methodology. Using these induced rules and based on the database schema, the condition, and object types specified in the query, the intensional answers may be derived by traversing the type hierarchies of the object types as specified in the query. We call this technique *type inference*. For example, the entities SUBMARINE, SSN (Nuclear Missile Submarines), and SSBN (Ballistic Nuclear Missile Submarines) forms a type hierarchy where the set of SUBMARINES can be divided into two disjoint subsets: SSBN and SSN. Representing this type hierarchy togther with the induced rules in KER, we have the following intensional knowledge which can be used to provide intensional answers to queries that involve SUBMARINE.

```
SSBN isa SUBMARINE with ShipType = "SSBN"
SSN isa SUBMARINE with ShipType = "SSN"

object type SUBMARINE

    has key: ShipId        domain: char[20]
    ...
    has:     Displacement  domain: integer

    with /* x isa SUBMARINE */

        if x.Displacement ≥ 7250 then x isa SSBN
        if x.Displacement ≤ 6955 then x isa SSN
        ...
```

Figure 5. A Type Hierarchy of Submarine with Induced Rules.

3.1 Type and Type Hierarchy

Type hierarchy allows objects to be represented and processed at different knowledge levels. An answer to a query is the set of database instances satisfying the query qualification. An intensional answer to a query is a set of descriptions that characterizes the set of database instances. To derive the set of descriptions, it is necessary to consider object representations at different levels of type hierarchy and to derive the set of descriptions as the intensional answers. Before we describe how intensional answers can be derived via type inference, let us first define *type*.

Type: A *type* can be defined recursively as the following:

a) A primitive type (e.g., integer, real, etc).

b) If $t_1, ..., t_n$ are types and $a_1, ..., a_n$ are attributes,

then $t:\{a_1:t_1, ..., a_n: t_n\}$ is a type called a tuple-type which can be abbreviated as t.

c) If $t_1, ..., t_n$ are types, then $t:\{t_1, ..., t_n\}$ is a type called a set-type which can be abbreviated as t.

This type definition actually covers the definition of the entity type and the relationship type of the KER model. Definition b defines the relationship type in the KER model which states that a type t can be constructed from other types, $t_1, ..., t_n$, and each t_i is a component of (or a part of) t and can be denoted as:

t_i IS_PART_OF t.

Definition c states that a type t can be constructed from or decomposed into other types, $t_1, ..., t_n$, and each t_i is a called a *subtype* of t and denoted as:

t_i IS_A t.

Definition c actually defines a type hierarchy with t as the root in the KER model. There are two ways to interpret such a type definition: *partition-based* and *construction-based*. Based on the partition view, a type hierarchy is built from top (root) to bottom, that is, a super type is partitioned into many subtypes and each subtype is in turn partitioned into sub-subtypes, and so on. Based on the construction view, a type hierarchy is built from bottom to top; that is, a super type is constructed from a set of subtypes. Both interpretations imply that a type at a higher level is a superset of all the types below it. Any type described at a higher level has wider coverage than a type defined at a lower level. For example, a type SSBN covers the ballistic nuclear missile submarines while its supertype SUBMARINE covers all the submarines.

3.2 Deriving Intensional Answers via Type Inference

Traditional relational databases support only flat relations and each entity or relationship is mapped into a relational table. Query is answered by accessing data from the database. Type inference is a process of tree traversal on the type hierarchies. Thus, type inference is a process to move up and down along the type hierarchy to expand or reduce the variable scope of the query. Therefore, using type inference together with the induced knowledge allows us to derive intensional answers without accessing the physical data from the database.

Intensional answers can be derived by *forward inference* and *backward inference* with the induced rules. Forward inference uses the known facts to derive more facts, i.e., given a rule "if X then Y", and a fact "X is true", we can conclude "Y" is true. Backward inference uses the known facts to infer what must be true according to the induced rules, i.e., given a rule "if X then Y", and a fact "Y is true", we can conclude "X" must be true. Forward inference and backward inference can be combined to derive more specific intensional answers.

3.2.1 Forward Type Inference

The process of using forward inference to derive intensional answers is to derive *the most specific type description* in the type hierarchy that satisfies the query condition. For example, consider a query asking for submarines with displacement greater than 8,000. Using the intensional knowledge as stated in Figure 5, we can traverse down from the submarine hierarchy (Figure 4) to derive an intensional answer "SSBN" since the condition "Displacement > 8000" is subsumed by "Displacement ≥ 7250".

Using the rule induction technique, a set of classification rules that are associated with each type definition can be induced and maintained with each type hierarchy using the *with* constraint specification. The process of forward inference is then a tree traversal, starting with the root type, on the type hierarchy constrained by the classificaiton rules and the query condition. If the query condition satisfies a classification rule for the subtypes, then the traversal continues on that subtype for obtaining more specific subtype descriptions. The traversal stops when the query condition satisfies all classification rules of the subtypes and the intensional answer is the immediate super-type of these subtypes.

The process of forward inference can be described as follows: Each given query Q can be represented as a conjunction of a set of selection conditions on the entity types as:

$$Q = \{T_1 \wedge \Psi_1\} \wedge ... \wedge \{T_n \wedge \Psi_n\}$$

where each T_i is an entity type reference and each Ψ_i is the query condition on each T_i. During forward inferencing, each Ψ_i is used to determine the subsumption relationship between Ψ_i and the *with* constraints associated with T_i. The most specific type description of the type hierarchy rooted at T_i that satisfies the query condition Ψ_i is the intensional answer.

3.2.2 Backward Type Inference

Backward inference uses the known facts to infer what must be true according to the induced rules. For example, given a rule "if x *isa* SUBMARINE and x.Displacement ≥ 7250, then x *isa* SSBN". If "x *isa* SSBN", we can conclude that some submarines must have displacement ≥ 7250 otherwise we will not have such an induced rule in the knowledge base. The backward inference described here is different from the backward chaining in logic programming such as PROLOG which uses backward reasoning to prove goals. Using backward inference, we can only derive descriptions of a *subset* of the extensional answers. For example, there might have some SSBN ships with displacements less than 7250.

The process of the backward inference is also a tree traversal but starts from the bottom to the top. Unlike forward inference, backward inference derives a set of classification rules that characterize the query results. Due to the inheritance property of type hierarchy, each subtype inherits the properties from its super-types which can be used as the intensional answers. For each subtype in a query, the traversal of backward inference is performed from the subtype to its ancestors in the type hierarchy. The classification rules for attributes that are specified in the query are collected along the traversal path. These rules state the conditions that the subtype must hold in the database. The traversal stops when it reaches the root of the type hierarchy and the set of classification rules collected is merged and becomes the intensional answer.

3.2.3 Properties of Intensional Answers

Using forward inference, the intensional answer gives a description of instances that includes the answers. The intensional answers derived from forward inference characterize a set of instances *containing* the extensional answer. Using backward inference, the intensional answer gives only a description of partial answers. As a result, there may be other extensional answers that satisfy the query condition but are not included in the intensional answer. Therefore, the intensional answer derived from backward inference characterizes a set of answers *contained in* the extensional answer. Forward inference and backward inference can be also combined to derive more specific intensional answers.

For forward inference, an attribute type can be reduced to a more specific type and the rules that are associated with the specific types can be used for inference. For backward inference, more knowledge can be derived from the rules which in turn can also be used to further reduce the attribute type. Therefore, the intensional answers that are generated by the proposed type inference are *relevant**.

3.3 Type Inference with Multiple Type Hierarchies

For a given object type, multiple type hierarchies may be formed. For example, given the SUBMARINE relation example, we can construct one type hierarchy based on the value of attribute *Type* and another type hierarchy based on the value of attribute *Fleet* (See Figure 4). Each type hierarchy is associated with certain knowledge represented in the rules.

When a query is specified, these hierarchies can be traversed at the same time to derive the intensional answers. The results from the multiple tree traversal can then be *merged* to obtain the final results. As a result, more precise and relevant intensional answers can be derived via type inference using multiple type hierarchies.

4. Ship Database Examples

We shall now use type inference to derive intensional answers. Given the rules induced in Section 2.4, let us consider the following examples:

Example 1:

Find the Ids, Names, Classes, and Types of the SUBMARINE with Displacement greater than 8000.

```
SELECT   SUBMARINE.ID, SUBMARINE.NAME
         SUBMARINE.CLASS, CLASS.TYPE
FROM     SUBMARINE, CLASS
WHERE    SUBMARINE.CLASS = CLASS.CLASS
AND      CLASS.DISPLACEMENT > 8000
```

The extensional answer of the above query is:

id	name	class	type
SSBN730	Rhode Island	0101	SSBN
SSBN130	Typhoon	1301	SSBN

Using forward inference with the induced rule R_9 and the definition of SSBN in the database schema, we derive the following intensional answer which provides a summarized answer for the query:

A_I = *"Ship type SSBN has displacement greater than 8000"*

Example 2:

Find the names and classes of the SSBN ships.

```
SELECT SUBMARINE.NAME, SUBMARINE.CLASS
FROM   SUBMARINE, CLASS
WHERE  SUBMARINE.CLASS = CLASS.CLASS
```

The following is the extensional answer to the above query:

name	class
Nathaniel Hale	0103
Daniel Boone	0103
Sam Rayburn	0103
Lewis and Clark	0102
Mariano G. Vallejo	0102
Rhode Island	0101
Typhoon	1301

Using backward inference with the induced rule R_5, the following intensional answer can be derived for this query:

A_I = *"Ship Classes in the range of 0101 to 0103 are SSBN."*

Note that ship class 1301 is also a SSBN (see Appendix), but is not included in the answer. This is because backward inference is used to derive the intensional answer which yields only a partial answer. As a result, the answer is incomplete. Note the following rule

R_{new}: if x.*Class* = 1301 then x isa SSBN.

is satisfied only by a single instance. For efficiency reasons, R_{new} is not maintained in the knowledge base. However, if this rule is maintained by the system, then the derived intensional answer will be complete.

Example 3:

List the names, classes and types of SUBMARINEs equipped with sonar BQS-04.

```
SELECT   SUBMARINE.NAME,
         SUBMARINE.CLASS, CLASS.TYPE
FROM     SUBMARINE, CLASS, INSTALL
WHERE    SUBMARINE.CLASS = CLASS.CLASS
AND      SUBMARINE.ID = INSTALL.SHIP
AND      INSTALL.SONAR = "BQS-04"
```

The extensional answer of the above query is:

name	class	type
Bonefish	0215	SSN
Seadragon	0212	SSN
Snook	0209	SSN
Robert E. Lee	0208	SSN

Using forward inference, from rule R_{20}, we know the ship type must be SSN; and from rule R_{14}, we know the sonar type is BQS. Next, using backward inference with the rule R_{19}, we conclude that the answers must contain ships with class from 0208 to 0215 (See Figure 4). We therefore have the following intensional answer:

A_I = *"Ship type SSN with class 0208 to 0215 is equipped with sonar BQS-04."*

In this example, we combine both forward and backward inferences to derive the specific intensional answer from two object types (SUBMARINE and SONAR) that are related by the INSTALL relation.

* *Relevance* concerns with avoiding intensional answers that have little or no value to the user [MOTR90].

Example 4:

List the Id, Name, Fleet, and Type of the SUBMARINE with displacement less than 3000.

```
SELECT   SUBMARINE.ID, SUBMARINE.NAME
         SUBMARINE.FLEET, CLASS.TYPE
FROM     SUBMARINE, CLASS
WHERE    SUBMARINE.CLASS = CLASS.CLASS
AND      CLASS.DISPLACEMENT < 3000
```

The extensional answer of the above query is:

id	name	type	fleet
SSBN582	Bonefish	SSN	2
SSBN584	Seadragon	SSN	2

Using the type hierarchy based on *Type*, we will get the following intensional answer:

A_{I_1}: "Ships are *SSN*."

Using the type hierarchy based on *Fleet*, we will get another intensional answer:

A_{I_2}: "Ships belong to *Fleet 2*."

These two intensional answers can be *merged* to obtain a more precise answer:

A_I: "Ships are *SSN* and belong to *Fleet 2*."

Note that by merging type inference results from multiple type hierarchies, a more precise intensional answer are derived.

5. Conclusions

In this paper, we present an approach using type inference and induced rules to provide intensional answers to queries. An inductive learning technique is developed to induce knowledge from the database contents. Using the induced knowledge, inference can be performed on the type hierarchies to derive intensional answers.

A machine learning technique is used to acquire the rules from database contents. These rules are stored in rule relations. Forward and backward type inferences can be used individually or combined to derive intensional answers. Further, inference with multiple type hierarchies may provide more precise intensional answers than inference with single type hierarchy.

Our experiments reveal that induced rules can play an important role in type inference in providing intensional answers. Further, type inference with induced rules is a more effective technique to derive intensional answers than using integrity constraints when the database schema have strong type hierarchy and semantic knowledge.

References

[BROD84] Brodie, M., Mylopoulos, J., and Schmidt, J. W., (eds.) *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, 1984.

[CHEN76] Chen, P.P.S., "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Trans. on Database Systems*, Vo. 1, No. 1, Mar 1976.

[CHU90] Chu, W. W., Lee, R., "Semantic Query Optimization via Database Restructuring," Proc. of the 8th Intl. Congress of Cybernetics and Systems, 1990.

[HAMM80] Hammer, M. and Zdonik, S. B., Jr., "Knowledge-based query processing," In *Proc. of the 6th Intl. Conf. on Very Large Data Base.*, pp. 137-147, 1980.

[HAMM81] Hammer, M., and McLeod, D., "Database Description with SDM: A Semantic Database Model," *ACM Transactions on Database Systems*, Vol. 6, No. 3, Sept 1981.

[JANE81] "*Jane's Fighting Ships*", Jane's Publishing Co., 1981.

[KING81] King, J. J., "QUIST: A system for semantic query optimization in relational databases," In *proc. of the 7th Intl. Conf. on Very Large Data Base.* pp. 510-517, 1981.

[MCLE82] McLeod, D., and Smith, J. M., "Abstraction in Database," *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, SIGMOD Rec.*, Vol. 11, No. 2, 1981.

[MICH83] Michalski, R. S., et al, (eds.) *Machine Learning: An Artificial Intelligence Approach*, Tioga Press, Palo Alto, 1983.

[MOTR89] Motro, A., "Using Integrity Constraints to Provide Intensional Answers to Relational Queries," In *Proc. 15th Intl. Conf. on Very Large Data Bases*, 1989.

[MOTR90] Motro, A., "Intensional Answers to Database Queries," working draft, 1990.

[QUIN79] Quinlan, J. R., "Induction Over Large Data Bases", STAN-CS-79-739, Stanford University, 1979.

[SHUM88] Shum, C. D., and R. Muntz, "An Information-Theoretic Study on Aggregate Responses," in *Proc. 14th Intl. Conf. on Very Large Data Bases*, 1988.

Appendix. An Example Ship Database

A Ship Database:

SUBMARINE			
Id	Name	Class	Fleet
SSBN130	Typhoon	1301	7
SSBN623	Nathaniel Hale	0103	7
SSBN629	Daniel Boone	0103	7
SSBN635	Sam Rayburn	0103	7
SSBN644	Lewis and Clark	0102	7
SSBN658	Mariano G. Vallejo	0102	7
SSBN730	Rhode Island	0101	7
SSN582	Bonefish	0215	2
SSN584	Seadragon	0212	2
SSN592	Snook	0209	2
SSN601	Robert E. Lee	0208	7
SSN604	Haddo	0205	7
SSN610	Thomas A. Edison	0207	7
SSN614	Greenling	0205	6
SSN648	Aspro	0204	6
SSN660	Sand Lance	0204	6
SSN666	Hawkbill	0204	6
SSN671	Narwhal	0203	6
SSN673	Flying Fish	0204	6
SSN679	Silversides	0204	6
SSN686	L. Mendel Rivers	0204	6
SSN692	Omaha	0201	6
SSN698	Bremerton	0201	6
SSN704	Baltimore	0201	6

TYPE	
Type	TypeName
SSBN	ballistic nuclear missile sub
SSN	nuclear submarine

CLASS			
Class	ClassName	Type	Displacement
0101	Ohio	SSBN	16600
0102	Benjamin Franklin	SSBN	7250
0103	Lafayette	SSBN	7250
0201	LosAngeles	SSN	6000
0203	Narwhal	SSN	4450
0204	Sturgeon	SSN	3640
0205	Thresher	SSN	3750
0207	Ethan Allen	SSN	6955
0208	George Washington	SSN	6019
0209	Skipjack	SSN	3075
0212	Skate	SSN	2360
0215	Barbel	SSN	2145
1301	Typhoon	SSBN	30000

INSTALL	
Ship	Sonar
SSBN130	BQQ-2
SSBN623	BQQ-5
SSBN629	BQQ-5
SSBN635	BQS-12
SSBN644	BQQ-5
SSBN658	BQS-12
SSBN730	BQQ-5
SSN582	BQS-04
SSN584	BQS-04
SSN592	BQS-04
SSN601	BQS-04
SSN604	BQQ-2
SSN610	BQQ-5
SSN614	BQQ-2
SSN648	BQQ-2
SSN660	BQQ-5
SSN666	BQQ-8
SSN671	BQQ-2
SSN673	BQS-12
SSN679	BQS-13
SSN686	BQQ-2
SSN692	BQS-15
SSN698	TACTAS
SSN704	BQQ-5

SONAR	
Sonar	SonarType
BQQ-2	BQQ
BQQ-5	BQQ
BQQ-8	BQQ
BQS-04	BQS
BQS-12	BQS
BQS-13	BQS
BQS-15	BQS
TACTAS	TACTAS

Reprinted from the Proceedings of the IMS-91, Kyoto Japan, April 1991

A Pattern-based Approach for Deriving Approximate and Intensional Answers*

Wesley W. Chu, Qiming Chen and Rei-Chi Lee

Department of Computer Science
University of California, Los Angeles
Los Angeles, California

Abstract

A pattern-based approach is proposed to derive the approximate and intensional query answers when the exact answer is unavailable or too time consuming to generate. The approximate and intensional query answers may be refined if more time is available. Since the pattern-based query processing performs mainly main-memory based manipulations without database access until the last step of generating the final results, it should provide faster response to queries than the conventional query processing in real-time applications.

Introduction

Conventional query processing technique has many short comings in supporting real-time applications. It cannot provide approximate answers or derive relevant intensional information when the exact answer is not available within a time limit. Due to the computation complexity of obtaining precise data, a query processing may not be completed in a restricted time limit and the users get no answer in return. Although the research on processing time-constrained specific queries, such as on the incremental evaluation of aggregate queries [LIU87, OZS90] has been conducted, presently there is no methodology available for processing time-constrained general queries. To remedy this problem, we propose to use a pattern-based approach to derive approximate and intensional answers.

In this paper we shall first introduce the notions of pattern and abstract pattern class, then present the proposed time-constrained query processing techniques using abstract pattern classes and pattern-based intensional knowledge.

1. Pattern and Abstract Pattern Class

Database objects may be divided into classes based on their commonly shared properties. For instance, a relation can be viewed as a class of objects. A pattern is defined on a class of objects by specifying *a query condition*. For example, based on the class of objects of the following relation schema

EMPLOYEE : (ID, name, address, department, position, salary)

such a condition as

salary ≥ 20K,

or

department = 'research'

can be used to define a pattern on the class EMPLOYEE. The objects of that class satisfying a pattern condition *match the pattern*. The set of Object identifiers (Oid's) of those objects who match the pattern with condition C form an **Abstract Pattern Class (APC)**. These Oid's are called the instances of the APC. The number of distinguished Oid's in an APC is called its **population**. An APC denoted as P(C), where P is its name and C is its condition, can be abbreviated as *P*.

An APC can subsume another in the following two situations :

1. When the pattern with a weaker condition C_1 covers the pattern with a stronger condition C_2, the APC defined by C_1 subsumes the APC defined by C_2. For example, the APC defined by condition "salary>20K" subsumes the APC defined by condition "salary>50K" on the same class EMPLOYEE.

2. When a super-class *A* subsumes a sub-class *B*, then the APC defined on *A* subsumes the APC defined on *B* by the same pattern condition. For example, the APC defined by "sex='M'" on a super-class PERSON subsumes the APC defined by "sex='M'" on its sub-class EMPLOYEE.

A Subsumption relationship introduces the partial ordering and underlies the inheritance among APC's.

The Oid's of a class of objects can be divided into a large number of APC's defined by various conditions. To reduce the amount of predefined APC's we can maintain the APC's defined on a single attribute, and use them combinationally. For example, for the class EMPLOYEE, a set of APC's can be defined on the attribute "salary" as

P_1("salary>20K"),
P_2("salary>50K"),
P_3("salary>70K"),

and a set of APC's can be defined on the attribute "depart-

*This research is supported by DARPA contract F29601-87-C0072.

ment" as

P_A("department='research'"),
P_B("department='marketing'"),
P_C("department='administration'").

Since we do not restrict the set of APC's defined on a single attribute to be mutually disjoint, they do not necessarily form a partition [SPY87]. Instead, they may form a sumsumption based partial order or even a total order such as the set of APC's defined on the attribute "salary".

Figure 1 shows an example of pattern definitions based on the attributes of "address", "salary" and "department" for the class EMPLOYEE.

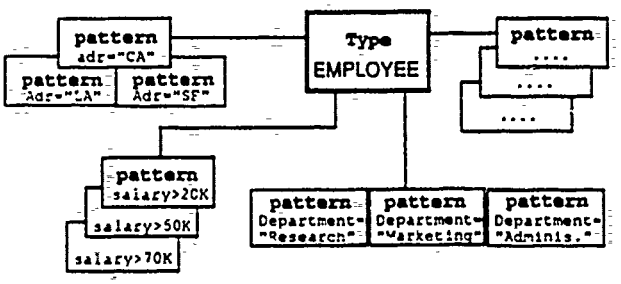


Figure 1. Patterns Defined for EMPLOYEE

In the following sections we shall present two techniques for using the notion of pattern to speed up query response in real-time applications. First we shall use the Oid in the APC to provide indexing for speeding up data access, where inexact matching between a query and a pattern yields approximate answering. Second we shall use the pattern-related rules to provide intensional answers.

2. Speed up Query Processing by Abstract Pattern Classes

A query can be viewed as a pattern called a **query pattern**. The set of Oid's of those objects who match a query pattern is called the **Query Pattern Class (QPC)** for the query pattern. A query pattern matches a predefined pattern means that the QPC of the query pattern is identical to the APC of the predefined pattern, then the Oid's contained in the APC can be used as pointers for accessing the desired data. Based on a known application domain and a set of frequently used queries, it is possible to provide a set of patterns with relatively large APC populations for speeding up query processing. Therefore creating APC's is similar to replicating data, except that APC's only contain Oid's rather than complete data and thus require less storage.

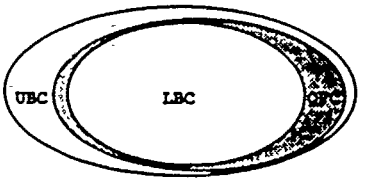
Let us first consider simple queries whose conditions involve only one attribute. An exact match often occurs between a simple query and a pattern defined on such an attribute whose values do not form an order. For example, a query pattern with condition "department='marketing'" matches a pattern defined by the same condition. In this case the QPC of the query pattern is identical to the APC of the predefined pattern, then the Oid's contained in the APC can be used to speed up database access by index techniques.

However, an exact match is only a special case for the patterns based on such an attribute whose values can be ordered. In this case we are more interested in the situation where the QPC of a simple query is contained in a predefined APC, P_1, and contains another predefined APC, P_2. In such a situation, P_1 contains the complete set of Oid's for answering the query but also contains some extra Oid's; and all the Oid's of P_2 can be used to answer the query but they are incomplete. Although neither P_1 nor P_2 matches the QPC exactly, they indicate the range of the answer.

Note that in the above situation P_1 subsumes P_2. In fact, in this approach we intentionally organize the APC's defined on a single attribute in a partial order with subsumption relationship among them. For example, based on the APC's defined on attribute "salary", for a simple query with condition "Salary ≥ 30K", its QPC is within the APC with condition "salary>20K" and subsumes the APC with condition "salary>50K".

For pattern-based query processing, A QPC may fall between a more general APC, referred to as the **Upper Bound Class** (UBC), and a more specific APC, referred to as the **Lower Bound Class** (LBC), where the UBC is the *least APC among the available APC's subsuming the QPC*, and the LBC is the *greatest APC among the available APC's subsumed by the QPC*. If the exact matching cannot be achieved, a neighborhood is selected where the QPC *contains* the LBC but is *contained* by the UBC (see Figure 2). With respect to the QPC, the UBC is *complete but not all correct* while LBC is *totally correct but not complete*.

The upper bound and the lower bound computed for a query may be returned as an approximate answer as the real-time deadline approaches. The approximation can be iteratively refined by testing each tuple (in the case of the UBC) or adding additional tuples (for LBC's). Such an incremental query processing can improve the accuracy of the result as time permits.



The Neighborhood of A Query Pattern Class (QPC)
with a Upper Bound Class (UBC) and a Lower Bound Class (LBC)

Figure 2. A Query Pattern Class Bounded by LBC and UBC

Now let us consider complex queries whose conditions involve multiple attributes. Such a query condition can be stepwise decomposed into disjunction and conjunction of query conditions.

Patterns defined on a single attribute may form a pattern hierarchy. Since patterns can be defined on multiple attributes, multiple pattern hierarchies for a single relation can be formed based on database schema. For example, the EMPLOYEE relation has several pattern hierarchies: by "department", "position", "salary" range, or "age" range. When a

query is related to a combination of these attributes, these hierarchies can be traversed and the results *merged* via operations on APC's to obtain the desired query answer.

The QPC of a query with **disjunctive** conditions is the *union of the QPC's for each participating condition.* For example, the QPC of a query with condition

"salary>50K ∨ department='research'"

is the union of the QPC for

"salary>50K"

and the QPC for

"department='research'".

In general, let the query condition C be expressed as the conjunction of $C_1 \vee ..., \vee C_n$, the corresponding QPC's, UBC's and LBC's be $QPC_{C_1}, ..., QPC_{C_n}$, $UBC_{C_1}, ..., UBC_{C_n}$, and $LBC_{C_1}, ..., LBC_{C_n}$ respectively, then the QPC, UBC and LBC for this query can be generated as

$$QPC_C = \bigcup_{i=1}^{n} QPC_{C_i},$$

$$UBC_C = \bigcup_{i=1}^{n} UBC_{C_i}, \text{ and}$$

$$LBC_C = \bigcup_{i=1}^{n} LBC_{C_i}.$$

The QPC for a query with **conjunctive** conditions is the *intersection of the QPC's for each participating condition.* For example, the QPC of a query with condition

"salary>50K ∧ department='research'"

is the intersection of the QPC for

"salary>50K"

and the QPC for

"department='research'".

In general, let the query condition C be expressed as the conjunction of $C_1 \wedge ..., \wedge C_n$, the corresponding QPC's, UBC's and LBC's be $QPC_{C_1}, ..., QPC_{C_n}$, $UBC_{C_1}, ..., UBC_{C_n}$, and $LBC_{C_1}, ..., LBC_{C_n}$ respectively, then the QPC for this query can be generated as

$$QPC_C = \bigcap_{i=1}^{n} QPC_{C_i},$$

$$UBC_C = \bigcap_{i=1}^{n} UBC_{C_i}, \text{ and}$$

$$LBC_C = \bigcap_{i=1}^{n} LBC_{C_i}.$$

Using the above set union and intersection operations, we can find the QPC, UBC or LBC for the query. However, for a query with conjunctive conditions, the difference between its QPC and UBC, and between its QPC and LBC, may lie beyond the approximation tolerance. To remedy this problem, for the frequently used queries with complex conditions, defining specific patterns with complex conditions is recommended. Further, maintaining and organizing APC's are application-dependent.

3. Speed up Query Processing by Pattern-based Intensional Knowledge

Intensional knowledge induced from the database domain and expressed as rules can be specified on each pattern for facilitating intensional query answering. For example, all employees in Los Angeles require security clearance and can be reached by the phone number (213) 208-2222. Further, the knowledge of one pattern may be used to imply the knowledge about another pattern. For example, all employees in the research department make more than 50K (see Figure 3).

An intensional answer provides a description rather than detailed data to a query. For example, instead of listing all the persons that satisfy the query condition "salary > 50K", an intensional but possibly partial answer to that query might be

"The salary of employees in the research department > 50K".

In a real-time system, when the deadline is approaching, the system will provide intensional and partial answers, if a complete answer is not available. When more time is available, the system can refine the answer by continuing the inference process on other patterns for further facts, or terminate the inference process and start retrieving instances from the database.

The subsumption relationship between patterns introduces the partial ordering and allows the **inheritance of intensional knowledge among patterns.** Further, given a set of APC's the APC resulted from their intersection *inherits* intensional knowledge from the set of APC's.

Since our approach performs mainly memory-based inference without physical database access until the last step of generating the final result, it should take less time than conventional query processing. Since patterns are used to match a set of instances that satisfy a given condition, they provide a finer granularity schema and therefore more specific intensional knowledge than that of types [CHU91].

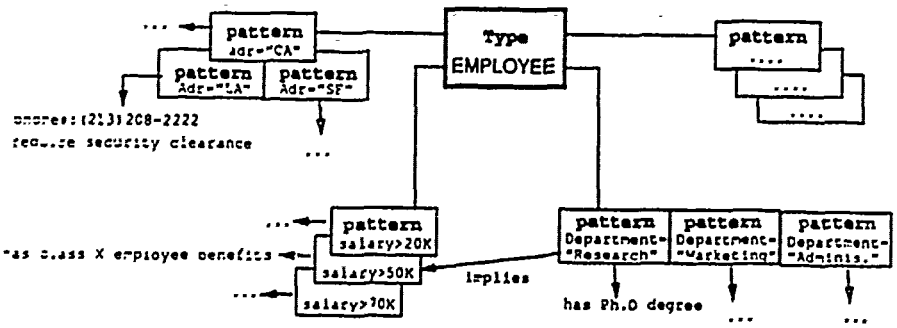


Figure 3. Intensional Knowledge of Patterns

4. Summary

The pattern-based approach presented in this paper supports approximate and intensional query processing. We have implemented a prototype system at UCLA using a naval ship database as a testbed on Sybase, a commercial relational database management system. Pattern hierarchies and other knowledge of the naval database are represented in LOOM [MACG89]. Our preliminary experience indicates that the proposed approach is an efficient method for providing approximate and intensional answers for real-time applications.

References

[CHU91] W. Chu, R. Lee and Q. Chen, "Using Type Inference and Induced Rules to Provide Intensional Query Answering," Proc. 7th International Conference on Data Engineering, Japan, 1991.

[LIU87] J. W. S. Liu, K. J. Lin, and S. Natarajan, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," Proc. IEEE Real-Time Syst. Symp., 1987.

[MAC89] MacGregor, R. and Yen, T. "LOOM: Integrating Multiple AI Paradigms," USC/Information Sciences Institute, Technical Report, 1989.

[OZS90] G. Ozsoyoglu, Z. Ozsoyoglu and W. Hou, "Research in Time- and Error-Constrained Database Query Processing," 7th IEEE Workshop on Real-time Operating Systems and Software, 1990.

[SPY87] N. Spyratos, "The Partition Model : A deductive Database Model," ACM TODS, Vol.12, No.1, 1987.

Computer Science Department Technical Report

University of California

Los Angeles, CA 90024-1596

CAPTURE DATABASE SEMANTICS BY RULE INDUCTION

Wesley W. Chu

Rei-Chi Lee

Kuorong Chiang

May 1990

CSD-900013

Capture Database Semantics by Rule Induction *

Wesley W. Chu, Rei-Chi Lee and Kuorong Chiang

Computer Science Department
University of California, Los Angeles
Los Angeles, California 90024

ABSTRACT

Database semantics can be classified into *database structure* and *database characteristics*. Database structure specifies the inter-relationships between database objects, while database characteristics defines the unique characteristics and properties of each object. To provide knowledge-based data processing, we need to gather and maintain knowledge about database characteristics. To capture the database characteristics, a Knowledge-based Entity Relationship (KER) Model that provides knowledge specification is proposed. A knowledge acquisition methodology is developed that uses machine learnine to induce the database characteristics knowledge from the database instances. Using a ship database as a test bed, a knowledge base has been generated by the proposed methodology. The induced knowledge has been used for semantic query optimization and data inference applications.

* This research is supported by DARPA Contract F29601-87-C-0072.

1. Introduction

The use of knowledge to support intelligent data processing has gained increasing attention in many database areas. For example, integrity constraints have been used as semantic knowledge to improve query processing [KING81, HAMM80]. Most of these works rely on human specified constraints and very few, if any, tools exist in gathering this knowledge.

In recent years, much effort has been devoted to the development of semantic data models [HAMM81, MCLE82, BORD84]. Most of these works emphasize structure modeling and allow for describing the database in a more natural way than traditional data models. These models are used mainly to capture the semantics of the database structure as described in the database schema. However, when a database is designed, most database designers use some *semantic rules* to distinguish among similar objects and specify the database schema according to these rules. In many cases, objects are often classified into different categories according to certain *characteristics* or *properties*. To distinguish them from the semantics of *database structure*, we shall refer to these semantics as *database characteristics* which are useful in knowledge-based data processing.

The type of database characteristics that are specified as integrity constraints [HAMM75] by human experts is not only time consuming to acquire but also is not too useful for knowledge-based data processing. To remedy this problem, we propose to use the database schema to guide the learning process and use the machine learning technique to induce database characteristics from the database. The database schema is specified in a knowledge-based Entity-Relationship (KER) Model which provides knowledge specification capability.

In this paper, we shall first discuss the problem of knowledge acquisition in databases. Next, we propose a model-based knowledge acquisition methodology that is based on the knowledge-based Entity-Relationship (KER) Model. We then present a rule induction algorithm and an example. Finally, we present the use of the induced rules for semantic query optimization, intensional query answering, and data inference applications.

2. Knowledge Acquisition in Databases

2.1 Knowledge Acquisition

The acquisition of knowledge is one of the most difficult problems in the development of a knowledge-based system. Currently, the acquisition of knowledge is still largely a manual process. The process usually involves a knowledge engineer who uses some expert system tools to transform the available knowledge into some internal form (knowledge representation) that is understandable by the expert system. It usually involves [MICH83]:

1. studying application literature to obtain fundamental background information,
2. interacting with the domain expert to obtain the expert level knowledge,
3. translating and encoding the expert knowledge for the system,
4. refining the knowledge base through testing and further interaction with the domain experts.

Such a manual process is very time-consuming. Further, even if the domain experts have the expertise, they may often not be able to describe their own expertise to others. As a result, useful knowledge may not be easy to collect. To remedy this problem, we propose to use the machine learning technique to construct the knowledge base. Rather than using knowledge engineers learning the application, or the domain experts learning the expert system tools and using their understanding of the application to construct the knowledge base, we propose to use machine learning technique to understand the database application and to create the knowledge base automatically.

2.2 Knowledge Acquisition by Rule Induction

Rule Induction [QUIN79, MICH83] is a technique of machine learning that has been used in AI research to induce rules from a set of training examples. For a given concept and a set of training examples representing the concept, find a description for the concept such that all

the positive examples satisfy the description and all the negative examples contradict the description. One approach is to examine the training examples simultaneously to determine which descriptors are most significant in identifying the concept from other related concepts. This approach recursively determines a set of descriptors that classify each example and selects the best descriptor from a set of examples based on a statistical estimation or a theoretical information content. The set of examples is then partitioned into subsets $S_1, S_2, ..., S_n$ according to the values of the descriptor for each example. This technique is recursively applied to each S_i until each subset contains only positive examples so that the set of descriptors describes the example set. Although the automated approach speeds up the knowledge acquisition process, it has been used mainly in applications when the size of training examples is small. To apply this technique directly to a database would be too costly because the database usually consists of a very large volume of data. However, since a database schema is created by the designer based on the semantic characteristics of the application, and since semantic characteristics are the candidates for rule induction, we can use the database schema to guide the knowledge acquisition by machine learning and generate the rules automatically.

3. Model-based Knowledge Acquisition Methodology

3.1 The Knowledge-based Entity-Relationship (KER) Model

To enhance the modeling of such capabilities as type hierarchy and knowledge specification, we introduce a Knowledge-based E-R (KER) model, an extension of the Entity-Relationship Model [Chen76]. KER provides the following three generic constructs of data modeling:

1. **has/with** (*aggregation*) which links an object with another object and specifies a certain property of the object (e.g., a CLASS has an instructor);

2. **isa/with** or **contains/with** (*generalization/specialization*) which links an object type with another object type and specifies an object as a subtype of another object (e.g., PROFESSOR is-a subtype of PERSON or PERSON contains PROFES-

SOR, STUDENT, and STAFF);

3. **has-instance** (*classification*) which links a type to an object that is an instance of that type (e.g., "John Smith" is an instance of PROFESSOR).

Note that in addition to the semantic constructs provided by most semantic data models, KER also provides knowledge specification which is represented by the *with-constraint* information. Such knowledge specification associated with each database definition is useful for knowledge-based data processing.

In KER, an entity is a distinctly identified object, for example, a specific person, a department, or a course. An entity set is a collection of entities. Each of these entities is distinguished by a unique identifier. The set of unique identifiers is called the primary key of the entity set. A relationship specifies the connections between different entities. Conceptually, both entity type and relationship type can be considered as an object type and can be modeled using the **has/with** construct. For example, Figure 1 shows an object type SUBMARINE represented in KER.

```
object type SUBMARINE

    has key:   ShipId        domain:   char[10]
    has:       ShipName      domain:   char[20]
    has:       ShipType      domain:   char[4]
    has:       ShipClass     domain:   char[4]
    has:       Displacement  domain:   integer

    with  Displacement in [2000..30000]
```

Figure 1. The KER representation of an object type SUBMARINE.

The object type can also be represented mathematically as:

$$\{ [a_1, a_2, ..., a_n] \mid a_1 \in D_1, a_2 \in D_2, a_n \in D_n \ with\ \Psi \}$$

where each tuple $[a_1, a_2, ..., a_n]$ is an instance of such a type. Note that each a_i defines an attribute of the object type, and D_i specifies its attribute domain while Ψ states constraints on the allowable values the tuple can have. An attribute domain can also be an entity type. The system

provides a set of basic domains such as **integer**, **real**, **string**, and **date**. A more complex domain can be constructed from these basic domains. For example, we can define a domain AGE on the basic domain INTEGER with the range [0..200]. A BNF description of the KER model is given in the Appendix A.

A type hierarchy uses specialization/generalization constructs (**isa** or **contains** relationships) to define the subtype and supertype relationships. For example, SSBN (Ballistic Nuclear Missile Submarine) is a subtype of SUBMARINE, and CLASS-0101 is a subtype of SSBN, and therefore, a type hierarchy consisting of SUBMARINE, SSBN, and CLASS-0101 is formed (see Figure 2).

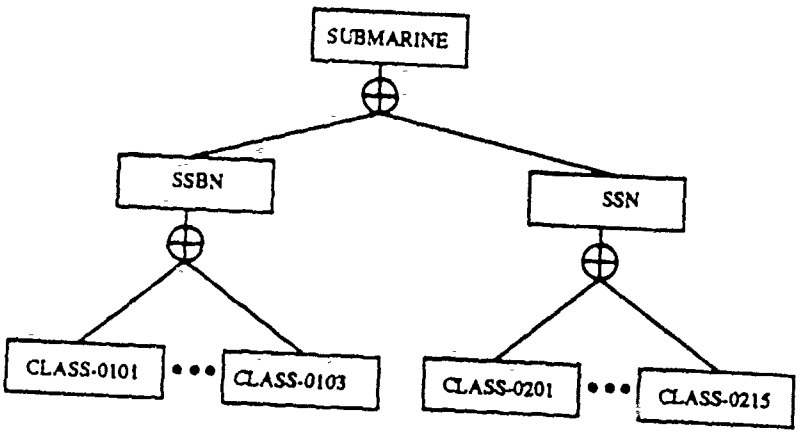


Figure 2. A Type Hierarchy SUBMARINE

A subtype inherits all the properties of its supertypes, unless some of the properties have been redefined in the subtype. For example, type SUBMARINE has attributes ShipID ,and ShipName, and type SSBN has attribute TypeID and TypeName; subtype CLASS-0101 will automatically inherit properties ShipID and ShipName from supertype SUBMARINE, and inherit properties TypeID and TypeName from another supertype SSBN.

A subtype can also be derived from another type by providing a *derivation specification*. For example, one can define a subtype SSBN (all the ships with ship type SSBN) of type SUBMARINE by specifying:

SSBN **isa** SUBMARINE **with** ShipType = "SSBN"

The with-clause defines the derivation specification of the subtype SSBN. It can also be considered as associating a constraint with this subtype.

The type hierarchy is represented in KER as:

E_1 **isa** E **with** Ψ_1

E_2 **isa** E **with** Ψ_2

...

E_n **isa** E **with** Ψ_n

or alternatively, it can also be represented as:

E **contains** $E_1, E_2, ..., E_n$ **with** Ψ.

This definition states that the instances of E can be divided into n disjoint subsets $E_1, E_2, ..., E_n$, with the constraint Ψ. Each E_i is a subtype of E.

To provide a graphical representation of the inter-relationships among the entity types/subtypes, relationship types, and derivation specification, we can extend the ER diagram by adding the type hierarchy with constraint representation as shown in Figure 3. A representation of a ship database schema by the KER Diagram is shown in Figure 4.

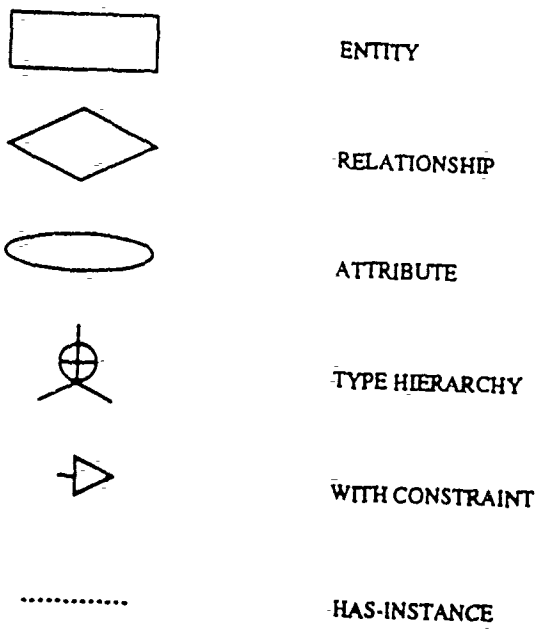


Figure 3. Components of the KER Diagram

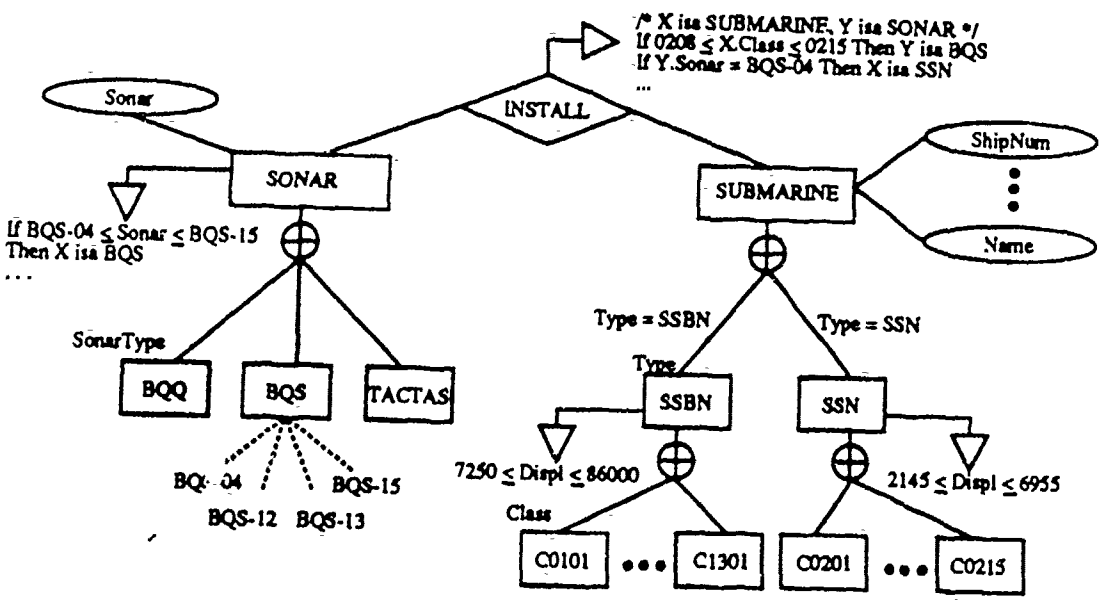


Figure 4. Representing the Ship Database Schema in KER Diagram

3.2 Classification of Semantic Knowledge

Semantic knowledge of a database can be divided into two categories: *enterprise knowledge* and *database knowledge*. Enterprise knowledge is the semantics of the database application. For example, the integrity constraint is the enterprise knowledge. Database knowledge is an instance of the enterprise knowledge which describes the current database contents. For example, enterprise knowledge specifies that the displacement of a ship must be greater than 2,000 tons, while database knowledge specifies that the displacements of the ships are between 2,360 tons to 81,600 tons. Thus, database knowledge is more effective for query optimization.

The semantic knowledge associated with each database are *domain knowledge*, *intra-object knowledge*, and *inter-object knowledge* as follows:

Domain Knowledge:

Domain knowledge defines specific properties of each entity set such as the attribute domains, value ranges, etc., and restricts the allowable instances of an entity set. For example, the displacement of a ship is in the range of 2,000 tons - 100,000 tons, and the displacement of an Attack Aircraft Carrier is in the range of 75,700 tons - 81,600 tons.

Intra-Object Knowledge:

Intra-Object knowledge specifies the relationships between attributes within an object type. For example, the object type submarine has the intra-object knowledge that if the displacement is less than 7,000 then it is a nuclear submarine (SSBN).

Inter-Object Knowledge:

The inter-object knowledge specifies the constraints that the instances of a relationship set must satisfy. For example, the relationship VISIT involves entities of

SHIP and PORT and satisfies the constraint that the draft of the ship must be less than the depth of the port. The inter-object knowledge can be induced from the interrelationship between SHIP and PORT linked by the VISIT relationship.

3.3 The Knowledge Acquisition Methodology

After classifying different types of knowledge and defining the target attributes for knowledge acquisition, let us now describe our Knowledge Acquisition Methodology (KAM), which consists of three stages: *schema generation*, *automated knowledge acquisition*, and *knowledge base refinement* as follows:

1. *Schema Generation*:

   The DBA uses the KER model to define a database schema. This step includes:

   a. Identifying entities and associated attributes.

   b. Identifying generalization hierarchies. If the database already exists, use the clustering indexes to define subtype entities. The indexes are the target attributes.

   c. Identifying aggregation hierarchy. Designate each of the referential keys as the target attributes. A referential key is the attribute of a relationship which is a key to some other entity.

2. *Automated Knowledge Acquisition*:

   a. Determine the *domain constraint* for each attribute of each entity type.

   b. Use the *rule induction algorithm* (described in the next section) to induce *inter-structure* and *intra-structure* knowledge related to the target attributes from the database.

3. *Knowledge Base Refinement*:

Based on the expert's own knowledge, domain experts can refine the rules in the knowledge base to improve system performance. Unlike the manual approach to knowledge acquisition, our methodology uses the database schema to guide the learning process to induce knowledge from the database contents. Such an automated process reduces the time for knowledge acquisition.

3.4 The Rule Induction Algorithm

We have implemented a rule induction algorithm in EQUEL (Embedded QUEL) and C on top of the INGRES system. Rule induction is performed using the relational operations to generate semantic rules for pairwise attributes. We shall present the algorithm that induces correlated relationships of the rule scheme X --> Y for attribute pair (X, Y).

Rule Induction Algorithm.

1. *Retrieving (X, Y) value pairs*

   Retrieve into S the instances of the (X, Y) pair from the database. The corresponding QUEL statement is:

```
range of r is relation
retrieve into S unique (r.Y, r.X)
sort by r.Y
```

2. *Removing inconsistent (X, Y) value pairs*

   Retrieve all the (X, Y) pairs that have multiple Y values for the same value of X. Let T be the result of this relation.

```
range of r is relation
range of s is S
retrieve into T unique (s.Y, s.X)
where (r.X = s.X and r.Y != s.Y)
```

   Remove all the (X, Y) pairs that have different Y values for the same X value from S.

```
range of s is S
range of t is T
delete s
where (s.X = t.X and s.Y = t.Y)
```

3. *Constructing Rules*

For each distinct value of Y in S, say y, determine the *value range* x of X and create a rule in the form of

$$\textbf{if } x_1 \leq X \leq x_2 \textbf{ then } Y = y.$$

A value range is defined as a consecutive sequence of X values that occur in the database. The rules generated for the same attribute pair (X, Y) consist of the *rule set* designated by the rule scheme X --> Y. Note that when $x_1 = x_2$, then the rule reduces to

$$\textbf{if } X = x \textbf{ then } Y = y.$$

4. *Pruning the Rule Set*

Although storing more rules in the knowledge base provides more opportunities for inference, it also increases the overhead for storing and searching these rules. Therefore, when the number of rules generated becomes too large, the system must reduce the size of the rule set. In general, there are two criteria for rule pruning:

1. *Coverage Measure.*

   Each rule is satisfied by a certain number of database instances. Coverage specifies the number of database instances where the rule is satisfied.

   For applications such as semantic query processing, the higher the coverage measure of a rule, the higher the probability that this rule will be used. Thus, we keep these rules that have a coverage measure higher than a prespecified number N_c, which is a cut-off point for pruning the rules.

Based on this criteria, we remove rules from the knowledge base when the coverage measure is less than N_c. Selecting the N_c depends on the tradeoff between the applicability of the rules and the overhead for storing and searching these rules.

2. *Completeness Measure.*

Each rule scheme (X --> Y) contains a set of rules specifying the relationship between the attributes X and Y. The completeness measure is the sum of the coverage of the rules of the same scheme divided by the total number of (X, Y) value pairs in the database. If the completeness measure is equal to 100%, that means we can always find a rule (and Y's value) for each X's value.

For data inference applications, higher completeness measure enable us to infer more accurate answers. Therefore, for such applications, we want to keep the completeness measure of rule schema higher than a prespecified number C_c, which is a cut-off point for pruning the rules. Thus, we remove all the rule schema where completeness measure are less than C_c.

4. A Rule Induction Example

To experiment the proposed knowledge acquisition methodology, we shall use a ship database which was created by the System Development Corporation (now UNISYS) to provide a generic naval database based on [JANE81]. The database is currently running on INGRES on a Sun 3/60 machine. We shall use the nuclear submarine portion of the database which consists the following relations (a sample database instance is given in the Appendix C):

SUBMARINE = (*Id*, *Name*, *Class*)

CLASS = (*Class*, *ClassName*, *Type*, *Displacement*)

TYPE = (*Type*, *TypeName*)

SONAR = (*Sonar*, *SonarType*)

INSTALL = (*Ship*, *Sonar*)

The database consists of five entity types: SUBMARINE, CLASS, TYPE, SONAR, SONAR_TYPE and one relationship type: INSTALL. The three entity types SUBMARINE, TYPE, and CLASS form a ship hierarchy and the entities SONAR and SONAR TYPE form another hierarchy as shown in Figure 4. Each submarine type contains a set of submarine classes and each submarine class contains a set of submarine instances. For example, submarines are divided into two types: SSBN (Ballistic Nuclear Missile Submarine) and SSN (Nuclear Submarine). The SSBN ships contain three classes of ships: 0101 (Ohio), 0102 (Benjamin Franklin), and 0103 (Lafayette), and there are three ships that belong to the ship class 0103 (Lafayette).

Each ship class has its specific characteristics such as displacement, length, beam, etc.. For tactical or strategic reasons, different sonars are installed on different ships. The relationship INSTALL indicates the sonars installed on the different ships. A textual representation of the database schema is given in Appendix B.

Applying our knowledge acquisition technique to the ship database generates 17 rules as shown below (rules are grouped by object types):

(1) SUBMARINE

R_1: **if** SSN623 $\leq$ *Id* $\leq$ SSN635 **then** x **isa** C0103
R_2: **if** SSN648 $\leq$ *Id* $\leq$ SSN666 **then** x **isa** C0204
R_3: **if** SSN673 $\leq$ *Id* $\leq$ SSN686 **then** x **isa** C0204
R_4: **if** SSN692 $\leq$ *Id* $\leq$ SSN704 **then** x **isa** C0201

(2) CLASS

R_5: **if** 0101 $\leq$ *Class* $\leq$ 0103 **then** x **isa** SSBN
R_6: **if** 0201 $\leq$ *Class* $\leq$ 0215 **then** x **isa** SSN
R_7: **if** Skate $\leq$ *ClassName* $\leq$ Thresher **then** x **isa** SSN
R_8: **if** 2145 $\leq$ *Displacement* $\leq$ 6955 **then** x **isa** SSN
R_9: **if** 7250 $\leq$ *Displacement* $\leq$ 30000 **then** x **isa** SSBN

(3) SONAR

R_{10}: **if** BQQ-2 $\leq$ *Sonar* $\leq$ BQQ-8 **then** x **isa** BQQ

R_{11}: if BQS-04 $\leq$ *Sonar* $\leq$ BQS-15 then x isa BQS

(4) INSTALL (x isa SUBMARINE and y isa SONAR)

R_{12}: if SSN582 $\leq$ x.*Id* = SSN601 then y isa BQS
R_{13}: if SSN604 $\leq$ x.*Id* = SSN671 then y isa BQQ
R_{14}: if x.*Class* = 0203 then y isa BQQ
R_{15}: if 0205 $\leq$ x.*Class* $\leq$ 0207 then y isa BQQ
R_{16}: if 0208 $\leq$ x.*Class* $\leq$ 0215 then y isa BQS
R_{17}: if y.*Sonar* = BQS-04 then x isa SSN

For the intra-object relationships, we have found rules about the relationships between *Ship Id* and *Ship Class*; *Ship Class* and *Ship Type*; *Class Name* and *Ship Type*, and *Displacement and Ship Type*. The classification of the submarines into different classes and types is fairly stable, thus, these rules are stable as well. For the inter-object relationships, the following typical rules have been found: the submarines of the classes from 0205 to 0207 carry only the "BQQ" type of sonars (R_{15}); and the sonar "BQS-4" is only carried by the SSN (Nuclear Submarine) type of submarines (R_{17}); etc.

5. Applications

In this section, we shall illustrate the use of induced rules in such areas as semantic query optimization, intensional query processing, and data inference applications.

5.1 Semantic Query Optimization

Semantic query optimization uses the semantics to transform a given query to a new query. The transformed new query produces the same results as the original query but is more efficient to process [KING81, HAMM80]. Integrity constraints are commonly used as semantic knowledge for query transformation. However, due to the generality of the integrity constraints that describe all the valid states of the database instances, they are not effective for semantic query optimization.

Using the proposed rule induction, the relationships between the attributes as well as the inter-relationships between relations can be captured as semantic rules. These semantic rules provide more efficient semantic query transformation than the integrity constraints.

It is well-known that using a clustering index provides faster access to relations than that of a segment scan. Therefore, a clustering index is often used to answer the queries. Due to similar data characteristics, objects are often clustered together. If we select the clustering index as the *target attribute* during rule induction, useful classification rules representing the characteristics of each cluster can be induced. For example, using ShipType as a clustering index, we obtained the induced rules in Section 4. Based on R_9, the following query

Q: *"List the names of the submarines with displacement greater than 8,000."*

can be transformed to

Q′: *"List the names of the SSBN submarines with displacement greater than 8,000."*

Processing Q′ is much faster than Q, since processing Q requires a scan of the entire relation, while Q′ can use ShipType as a clustering index.

5.2 Intensional Query Processing

Conventional query processing provides answers in the form of an enumeration of database instances retrieved from the database. Intensional query processing provides answers that characterize the instances satisfying the query rather than listing all the instances [SHUM88, MOTR89]. Traditionally, the knowledge about the database structure such as type hierarchy is used to derive intensional answers.

However, due to the limited semantics in database structure, using the type hierarchy alone can only generate very limited forms of intensional answers. The induced rules can be used to derive much more specific intensional answers. Based on the database schema, intensional answers can be derived from the induced rules by traversing down the type hierarchies of the object types as specified in the query. Such a technique is called *type inference* [CHU 90a]. For example, considering the query

"List the submarines with displacement greater than 8,000."

Since the condition "*Displacement* > *8000*" is subsumed by "*Displacement* ≥ 7250", based on the induced rule R_9, we can traverse down from the submarine hierarchy to derive an intensional answer for the query which is "SSBN".

5.3 Data Inference Applications

In a distributed database system, databases are often partitioned into fragments and replicated and stored at several sites. However, during network partition, the data fragments may be inaccessible which reduces data availability. Since database attributes are often related to each other, the values of certain attributes often can be *inferred* from other attributes. To improve data availability, we can use *data inference* to infer *inaccessible* data from *accessible data* [CHU 90b].

Using the proposed knowledge acquisition approach, correlated knowledge between attributes can be extracted from the database contents. Since these rules represent summarized information, the storage size of these rules is much less than that of the replicated copies of the data. Such induced rules can then be replicated at each site or certain critical sites to improve data availability during network partitions. For example, given the query,

"*Which submarines carry the BQS type of sonar?*"

To process the above query, we need the INSTALL relation. However, due to network partitioning, the INSTALL relation (See Section 4) is no longer available. As a result, we are unable to answer the query by conventional query processing. However, using the induced rule R_{12}, we can derive that the Id of the ships are in the range from "SSN582" to "SSN601". Then, from the SUBMARINE relation we can derive the following answer for the query:

{SSN582, SSN584, SSN592, SSN601}.

For more discussion on this, the readers should refer to [CHU 90b].

6. Conclusions

Database semnantics can be classified into database structure and database characteristics. Semantic data models emphasize the modeling structural aspects of the database, while the database characteristics define the unique characteristics and properties of objects which are important to knowledge-based data processing. A Knowledge-based Entity-Relationship (KER) Model is proposed to provide knowledge specification capability and to maintain the database semantic knowledge. A knowledge acquisition methodology is developed that is based upon the KER Model and machine learning techniques to acquire the database characteristics from the database. These database characteristics are useful for semantic query optimization and data inference applications.

REFERENCES

[BROD84] Brodie, M., Mylopoulos, J., and Schmidt, J. W., (eds.) *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer, New York, 1984.

[CHEN76] Chen, P.P.S., "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transaction on Database Systems*, Vo. 1, No. 1, March 1976.

[CHU 90a] Chu, W. W, and Lee, R., "Using Type Inference and Induced Rules to Provide Intensional Answers," Technical Report CSD-9000006, Computer Science Department, UCLA, Los Angeles, 1990.

[CHU 90b] Chu, W. W, and et al, "An Inference Technique for Distributed Query Processing in a Partitioned Network," Technical Report CSD-9000005, Computer Science Department, UCLA, Los Angeles, 1990.

[HAMM75] Hammer, M., and McLeod, D., "Semantic integrity in a relational data base system," In *Proceedings of the First International Conference on Very Large Data Bases*, IEEE, New York, pp. 25-47, 1975.

[HAMM80] Hammer, M. and Zdonik, S. B., Jr., "Knowledge-based query processing," In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct. 1-3). IEEE, New York, pp. 137-147, 1980.

[HAMM81] Hammer, M., and McLeod, D., "Database Description with SDM: A Semantic Database Model," *ACM Transactions on Database Systems*, Vol. 6, No. 3, September 1981.

[JANE81] *Jane's Fighting Ships*, Jane's Publishing Co., 1981.

[KING81] King, J. J., "QUIST: A system for semantic query optimization in relational databases," In *proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Sept. 9-11). IEEE, New York, pp. 510-517.

[MCLE82] McLeod, D., and Smith, J. M., "Abstraction in Database," *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, SIGMOD Record*, Vol. 11, No. 2, February 1981.

[MICH83] Michalski, R. S., et al, (eds.) *Machine Learning: An Artificial Intelligence Approach*, Tioga Press, Palo Alto, 1983.

[QUIN79] Quinlan, J. R., "Induction Over Large Data Bases", STAN-CS-79-739, Stanford University, 1979.

Appendix A. A BNF definition of the KER Model

We will use the following BNF conventions:

<...> non-terminal symbol

{ x } x appears 0 or more times

[x] x appears 0 or 1 time

x1 | x2 | ... | xn x1 or x2 or ... or xn

'l' literal symbol

A.1 Data Definition Statements

<KER definition> ::=

<domain definitions> |

<object type definitions> |

<type hierarchy definitions>

A.2 Domain Definition Statements

<domain definitions> ::=

<domain definition> {, <domain definition> }

<domain definition> ::=

domain <domain name> **is** <domain description>

[<domain sepcification>]

<domain name> ::= **identifier**

<domain description> ::= <standard domain> | <object domain>

<standard domain> ::= **string** | **integer** | **real** | **date**

<domain specification> ::=

<range specification> |

<set specification>

<range specification> ::=

range <lower boundary> <value> ',' <value> <upper boundary>

<lower boundary> ::= '[' | '('

<upper boundary> ::= ']' | ')'

<set specification> ::=

set of '{' <value> {, <value> } '}'

<value> ::= **identifier** | <integer> | <real>

<object domain> ::= <object type name>

<object type name> ::= **identifier**

A.3 Object Type Definition Statements

<object type definition> ::=

object type <object type name>

<attribute list>

<with constraints>

<attribute list> ::=

<attribute> {, <attribute> }

<attribute> ::=

has [**key**] ':' <attribute name> **domain** <domain name>

<with constraints> ::=

with <constraints>

A.4 Type Hierarchy Definition Statements

<type hierarchy definition> ::=

<object type name> **contains** <sub-type list>

[<attribute list>]

[<with constraints>]

<sub-type list> ::=

<object type name> {, <object type name> }

A.5 Constraint Definition Statements

<constraints> ::=

<constraint> {, <constraint> }

<constraint> ::=

<domain range constraint> |

<semantic rule>

<domain range constraint> ::=

<attribute name> **in** <domain sepcification>

<semantic rule> ::=

<constraint rule> |

<structure rule>

<constraint rule> ::=

if <premise> **then** <consequence>

<premise> ::=

<conjuntives>

<conjunctives> ::=

<clause> { **and** <clause> }

<clause> ::=

<attribute> <operator> **constant**

<consequence> ::=

<attribute> '=' **constant**

<structure rule> ::=

if <role definitions>

and <conjunctives>

then <variable> **isa** <object type name>

<role definitions> :: =

<role> { **and** <role> }

<role> ::= <variable> **isa** <object type name>

<variable> ::= **identifier**

Appendix B. A KER Representation of a Naval Ship Database Schema.

B.1 Domain Definitions

```
domain: NAME                isa CHAR[20]

        domain: CLASS_NAME      isa NAME
        domain: SHIP_NAME       isa NAME

domain: TYPE_NAME           isa CHAR[30]

domain: SONAR_NAME          isa CHAR[8]
```

B.2 Object Type Definitions

```
object type CLASS

        has key:    Class          domain: CHAR[4]
        has:        Type           domain: type
        has:        ClassName      domain: CLASS_NAME
        has:        Displacement   domain: INTEGER

        with /* constraint rules */

        if "0101" ≤ Class ≤ "0103" then Type = "SSBN"
        if "0201" ≤ Class ≤ "0216" then Type = "SSN"

CLASS contains SSBN, SSNs

        Bwith /* x isa CLASS */

        if 2145 ≤ x.Displacement ≤ 6955 then x isa SSN
        if 7250 ≤ x.Displacement ≤ 30000 then x isa SSBN

object type SUBMARINE

        has key:    Id             domain: CHAR[7]
        has:        Name           domain: SHIP_NAME
        has:        Class          domain: class
SUBMARINE contains C0101, ..., C1301

object type TYPE

        has key:    Type           domain: CHAR[4]
        has:        TypeName       domain: TYPE_NAME

object type SONAR

        has key:    Sonar          domain: CHAR[8]
        has:        SonarType      domain: SONAR-NAME

SONAR contains BQQ, BQS, TACTAS
```

```
with /* x isa SONAR */

if BQQ-2 ≤ x.Sonar ≤ BQQ-8 then x isa BQQ
if BQS-04 ≤ x.Sonar ≤ BQS-15 then x isa BQS
if x.Sonar = "TACTAS" then x isa TACTAS
```

object type INSTALL

```
has key:    Ship      domain: SUBMARINE
has:        Sonar     domain: SONAR

with /* x isa SUBMARINE and y isa SONAR */

if x.Class = 0203 then y isa BQQ
if 0205 ≤ x.Class ≤ 0207 then y isa BQQ
if 0208 ≤ x.Class ≤ 0215 then y isa BQS
if y.Sonar = BQS-04 then x isa SSN
```

Appendix C. A Ship Database and Its Induced Rules

A Ship Database:

Relation SUBMARINE		
Id	Name	Class
SSBN130	Typhoon	1301
SSBN623	Nathaniel Hale	0103
SSBN629	Daniel Boone	0103
SSBN635	Sam Rayburn	0103
SSBN644	Lewis and Clark	0102
SSBN658	Mariano G. Vallejo	0102
SSBN730	Rhode Island	0101
SSN582	Bonefish	0215
SSN584	Seadragon	0212
SSN592	Snook	0209
SSN601	Robert E. Lee	0208
SSN604	Haddo	0205
SSN610	Thomas A. Edison	0207
SSN614	Greenling	0205
SSN648	Aspro	0204
SSN660	Sand Lance	0204
SSN666	Hawkbill	0204
SSN671	Narwhal	0203
SSN673	Flying Fish	0204
SSN679	Silversides	0204
SSN686	L. Mendel Rivers	0204
SSN692	Omaha	0201
SSN698	Bremerton	0201
SSN704	Baltimore	0201

Relation INSTALL	
Ship	Sonar
SSBN130	BQQ-2
SSBN623	BQQ-5
SSBN629	BQQ-5
SSBN635	BQS-12
SSBN644	BQQ-5
SSBN658	BQS-12
SSBN730	BQQ-5
SSN582	BQS-04
SSN584	BQS-04
SSN592	BQS-04
SSN601	BQS-04
SSN604	BQQ-2
SSN610	BQQ-5
SSN614	BQQ-2
SSN648	BQQ-2
SSN660	BQQ-5
SSN666	BQQ-8
SSN671	BQQ-2
SSN673	BQS-12
SSN679	BQS-13
SSN686	BQQ-2
SSN692	BQS-15
SSN698	TACTAS
SSN704	BQQ-5

Relation TYPE	
Type	TypeName
SSBN	ballistic nuclear missile sub
SSN	nuclear submarine

Relation SONAR	
Sonar	SonarType
BQQ-2	BQQ
BQQ-5	BQQ
BQQ-8	BQQ
BQS-04	BQS
BQS-12	BQS
BQS-13	BQS
BQS-15	BQS
TACTAS	TACTAS

Relation CLASS			
Class	ClassName	Type	Displacement
0101	Ohio	SSBN	16600
0102	Benjamin Franklin	SSBN	7250
0103	Lafayette	SSBN	7250
0201	LosAngeles	SSN	6000
0203	Narwhal	SSN	4450
0204	Sturgeon	SSN	3640
0205	Thresher	SSN	3750
0207	Ethan Allen	SSN	6955
0208	George Washington	SSN	6019
0209	Skipjack	SSN	3075
0212	Skate	SSN	2360
0215	Barbel	SSN	2145
1301	Typhoon	SSBN	30000

Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

TANGRAM: PROJECT OVERVIEW

R. R. Muntz
D. Stott Parker

April 1988
CSD-880032

Tangram: Project Overview †

R.R. Muntz

D. Stott Parker

Computer Science Dept.
University of California
Los Angeles, CA 90024-1596

ABSTRACT

Today, most computers are used for the modeling of real-world systems. Demands on the extent and quality of the modeling are growing rapidly. There is an ever-increasing need for environments in which one can construct and evaluate complex models both quickly and accurately.

Successful modeling environments will require a cross-disciplinary combination of different technologies:

System modeling tools
Database management
Knowledge base management
Distributed computing

None of these technologies by itself provides all that is needed. A modeling environment must offer high-speed retrieval and exploration of knowledge about systems, as well as integration of diverse information sources with existing modeling tools.

Tangram is a distributed modeling environment being developed at UCLA. It is an innovative Prolog-based combination of DBMS and KBMS technology with access to a variety of modeling tools.

April 28, 1988

†Supported by DARPA contract F29601-87-C-0072.

For more information contact:

Richard R. Muntz (muntz@cs.ucla.edu)
(213) 825-3546 ofc., 825-7879 scy.

D. Stott Parker (stott@cs.ucla.edu)
(213) 825-6871 ofc., 825-1322 scy.

Tangram: Project Overview †

R.R. Muntz

D. Stott Parker

Computer Science Dept.
University of California
Los Angeles, CA 90024-1596

1. Introduction: Modeling

> Harrison Brown warns that modern economies are complex and vulnerable because dangerous dynamic forces are at work (growing population, decreasing resource base, growing gap between rich and poor nations, political troublemaking on a world-wide scale, etc.). Despite these dynamic forces and the complexity of world economies, policy makers continue to make decisions without the benefit of the powerful analytical tools that are available to them. That the decisions they make are bad, is self-evident.
>
> – George Dantzig [31]

Today, numerous environments containing specialized tools are under development. Often these tools are for creating and managing models of some kind [35, 42, 74]. There are many kinds of analytic models alone:

- Stochastic processes/queueing models
- Statistical models
- Structural models
- Equational models (numeric or symbolic)
- General constraint-based models
- Rule-based models
- Semantic network/entity-relationship models
- Object-based models, with methods or behaviors.

In diverse science and engineering disciplines, ranging from medicine to the social sciences, models are applied in computer-aided specification, design, and analysis, including simulative and statistical analysis. The diversity of the models used stems partly from their evolutionary development in different fields.

Modeling will be one of the main enterprises of the information age. Today's emphasis on modeling will only increase in the future. Darwinian pressures force information systems to grow in sophistication, to better answer questions such as 'What happened?', 'What is going on?', and 'What would happen if...?' Future modeling environments must deal with three significant needs:

- Structurally complex data
- Deep interpretation of data
- Integrated management of data.

1.1. Structurally Complex Data

Data used in models is becoming more detailed in terms of structural complexity. Present modeling systems have trouble in representing this detail. For example, while structurally simple data is handled well by relational database technology, recently a great deal of attention has been given to more structurally complex data that has interesting, non-tabular structure. This data is commonly called 'complex objects' or 'non-first normal form' (NFNF) records in the data management field, and semantic networks or knowledge bases in other contexts.

With increasing detail of description, the list of differences between any two objects will grow. *Deep models* – models involving great detail, or multiple levels of detail – are always more structured than simple or generic models, which are sometimes called *shallow models* by comparison. As we model objects in greater detail, we must deal with the fact that some objects have individualized attributes that may be unknown or irrelevant for other objects. Relational structures model objects homogeneously, with every object being represented by the same sequence of parameters or attributes. Relational storage therefore becomes impractical when we seek greater detail.

The ability to store complex structures is powerful: with it, one can store not only facts about objects, but also rules governing the behavior of objects. This has immediate consequences in modeling, since it permits distinctions between data and interpretation of data to disappear.

1.2. Deep Interpretation of Data

In addition to managing more detailed data, the operations one wishes to do on this data become more sophisticated, or deep. These operations have been impractical in the past for a number of reasons.

First, analytical or abstracted models are often used instead of doing direct analysis of real-world data. There are inherent problems with such models:

(1) Abstracted models are often shallow. In some cases an abstracted model permits the researcher to get at essential aspects of the real-world system being studied, but then only at those aspects. Detailed information about a system requires direct analysis of larger quantities of data.

(2) Abstracted models require *estimation* of suitable parameter values. Typically, parameter estimates are defined as either the result of statistical queries against observed data, or the solutions of a system of constraints. It is not always straightforward to obtain these estimates.

(3) To be of real use, an abstracted model must be *validated* against the real-world data it supposedly describes. Again, while this can be thought of as a query (Does the model match the data?), validation is an unpleasant responsibility that is often avoided by model-builders.

Second, models may be too large for conventional machines, even mainframes. Simulation of microprocessors with significant numbers of gates takes weeks of CRAY-2 time. Interpretation of more detailed data will require significantly more processing power than is available today. Some argue that the current push into supercomputers is justifiable only because it will permit us to model systems that have been intractable in the past.

Third, researchers developing models have lacked appropriate data management tools. Laborious data extraction from tapes or files is a traditional problem in large-scale modeling.

Unfortunately, using modern DBMS will not be adequate for modeling needs.* The queries permissible in today's DBMS are very restricted: They map tables into tables using a handfull of well-defined operators. More generally, queries should be *any* kind of computable feature extraction operation.

It is important to consider what one really wants from a modeling environment. Three general operations are performed against a collection of data in the process of modeling: *abstraction* of models, *prediction* of the future using models, and *validation* of models against the past.

Abstraction is the least understood of the three kinds of operations, since it involves some sort of creative or associative recall on the part of the modeler. The process of abstraction requires compiling information from different contexts, or perspectives. In other words, abstraction requires feature extraction, filtering, data reduction, etc. Important operations today include:

standard relational query
aggregate computation (min, max, average, count, sum)
statistical analysis (regression, analysis of variance, etc.)
pattern recognition (trends, transients, etc.)
estimation of model parameters
estimation of certainty factors
induction of rules

No existing tools, even the most advanced database systems, provide what is needed here.

Prediction is a process of evaluating various possible scenarios consistent with a current world model. It typically involves some sort of simulation. Mathematical models involve structurally simple data, and either the solution of systems of equalities and inequalities or continuous-time simulation. What we might call structured models (with structurally complex data) involve either discrete event simulation, or some variety of rule- or object-based simulation.

Validation, finally, matches predictions obtained from a model of a system against the actual behavior of the system. This involves the definition of metrics or figures of merit on model performance, and verifying somehow (statistically or formally) that a model achieves a level of accuracy.

These three operations are used in an abstraction-prediction-validation cycle. The results of each cycle are used in refining the model for better accuracy. A reasonable modeling environment must support these three operations and their iterative application in the refinement of a model.

1.3. Integrated Management of Data

To see the need for integrated data management, consider the problems encountered in a simple case study. A 4-site model of the LOCUS distributed operating system was simulated using the

*Interestingly, work in the database field has approached modeling from another direction. A database is, after all, a model of some enterprise, or more generally of some collection of objects and relationships among them. The DBMS of today permit only static models. That is, they capture the *state* of some system. The behavior of these systems must be captured with external programs, not by the DBMS itself. Thus the DBMS accomodates simple models on vast quantities of data, instead of intricate models involving a few parameters.

PAWS simulation package by Steven Berson of UCLA.†

Altogether, nine different load-balancing algorithms were considered. In each experiment, labelled by single letters in the table below, all sites used the same criteria for deciding when to migrate process load to other sites. The criteria and offered load varied among the experiments: load balancing was done by inspecting cpu and/or disk queue lengths, and offered load differed by distribution and by number of customers (between 2 and 20):

Experiment	*Load balancing on*	*Work done*
r	disk queue lengths	Exponential
s	cpu queue length	Exponential
u	sum of cpu and disk queue lengths	Exponential
v	sum of cpu and disk queue lengths	Hyperexponential
w	sum of cpu and disk queue lengths	Erlangian
x	NO BALANCING, NO REMOTE ACCESS	Exponential
U	NO BALANCING	Exponential
V	NO BALANCING	Hyperexponential
W	NO BALANCING	Erlangian

The resulting average response times, in milliseconds, were obtained:

Experiment	*Number of Customers*									
	2	4	6	8	10	12	14	16	18	20
r	2490	2390	2440	3200	3520	4610	5830	6630	8900	10500
s	2320	2450	2820	3150	4080	4510	5160	7240	9510	11700
u	2470	2450	2620	2560	3600	3670	5010	6130	7920	10400
v	2430	2610	2260	2720	3210	3860	4660	5280	7680	7750
w	2330	2570	2720	2930	3260	4050	4690	6350	8060	9690
x	2750	2990	3140	3490	3910	6230	6190	8570	9060	10500
U	2700	2880	3680	4410	5130	5310	7190	8650	11100	13600
V	2220	3520	3100	3710	4780	6340	7480	9660	10900	12400
W	2580	3150	3200	3950	4640	5950	7780	8830	10600	13800

These results show that the load balancing strategies taking both cpu and disk queue lengths into account performed better than the others. It is difficult to 'get a feel' for what the differences are here, however.

Modest as it is, this experience underscores the importance of an environment for modeling:

(1) Data management of experiments requires enormous effort. The data above were extracted manually from the printed output of many PAWS runs (possibly with errors), required creation of several different versions of the data for different tools (*S*, *grap*, *tbl*, etc.), and so forth. The data above cannot be queried automatically now.

(2) Various types of measurement data that must be captured beyond the standard statistics (utilization, throughput, queue length, queueing time, chain population, point-to-point

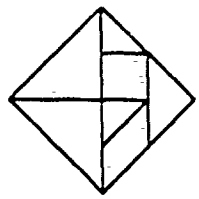
† Each site had 1 cpu and 1 disk. The access patterns were selected to be 80% local, 20% remote disk access, with an average disk seek time of 30ms. Load on the system was generated by between 2 and 20 customers. The PAWS simulation system was used to simulate several tens of thousands of events, in order to obtain mean response time.

timing, etc.) typified above. Event traces, for example, are needed for detailed understanding of what goes on in distributed systems: cyclic behavior, correlated events, race conditions, catastrophe-theoretic behavior, etc.

(3) There are many different kinds of queries that may be made against measurement data: statistical summary and analysis, pattern detection on time series, browsing through subsets of event trace, animation with start/stop/replay, and of course graphical display (bar plots, contour plots, histograms, scatter plots, time-series plots, etc. [10])

2. Tangram: A Modeling Environment

Tangram is an environment for developing models of the scope suggested above. It is implemented as an extension of Prolog that includes integration with the Unix environment and database managers, and provides distributed processing constructs. This section gives an informal overview of the design of the environment. In later sections we describe more thoroughly the individual research projects which together comprise Tangram.



2.1. Functions of Tangram

The main goal of Tangram is to provide an interactive modeling environment. The experiences of the previous case study highlight some functions that such an environment should provide. A functional diagram of Tangram in use is sketched in Figure 1. With this system a user should be able to:

1. Select models from the Model Base
2. Select an experiment from the Model Base
3. Run the experiment
   - 3.1 Find available tools from the environment Knowledge Base
   - 3.2 Execute the experiment with a specific tool (or testbed)
   - 3.3 Store results of the experiment in the Measurement Data Base
4. Query the results of the experiment interactively

We describe these capabilities further below.

2.1.1. Model Management

Just as DBMS are managers for data, Tangram is a manager of models. Model management includes the storage and retrieval of 'data dictionary' knowledge about available models, workloads (load generators, benchmarks), experiments, experiment output, and tools. Where multiple models are used to describe a single system from different viewpoints or levels of abstraction, model management also provides information on how these models relate.

The Tangram environment is to support various modeling packages, and possess knowledge on how these packages are applied. This knowledge comprises an expert system on easy, effective access to modeling tools. The SACON system developed by Bennett et al. [13] illustrates the kind of functionality needed: SACON inspects a structure and recommends a particular subprogram from the (large, complex) MARC environment for structural analysis. For example, given the description of an airplane wing, it applies knowledge about the domain to decide to use the MARC inelastic-fatigue program to analyze stress and deflection of the wing.

2.1.2. Measurement Data Management

Modeling experiments generate massive quantities of data. Tangram is concerned with issues in capturing this data from different tools, translating it to a common format, storing it, and supporting arbitrary queries with parallel processing. This presents challenges in developing data management technology. Not only is the data structured, but it also contains important temporal information. Also, current DBMS do not support 'exploration' of the data in the way that exploratory data analysis systems such as *S* [10] do. It is important to be able to support exploration of a model, encouraging a modeler to get an intuitive understanding of its behavior. The modeler should be able to view his model actually 'running' with various kinds of graphical displays, for example.

Parallelism can make interactive real-time modeling possible, where it would not be possible on this scale otherwise. We see stream processing as the most natural parallel data management paradigm. A great portion of the Tangram project is, then, concerned with stream processing. The *Tangram Stream Processor (TSP)* is a stream-based system founded on the abstraction of transducers. A transducer maps input streams to output streams. We discuss TSP in greater detail in a later section.

2.1.3. Support for Advanced Modeling Tools

Current modeling tools typically force the modeler into expressing his model in a limited framework, and investigating the model's behavior with a limited set of query facilities. These tools are rarely extensible, i.e., they do not permit addition of new features. Tangram provides tools that permit 'declarative' specification of models supporting complex structuring of knowledge and deep interpretation as discussed earlier. We currently envision an object-oriented environment for developing these tools, supporting knowledge base management and arbitrary query processing. The environment will be extensible, permitting the addition of new kinds of models.

We have developed a methodology for building modeling tools based on Markov processes. A prototype system has been built based on this methodology [14]. In the system, users specify system components in an object-oriented framework. This level of specification is significantly higher than that provided by most modeling tools, which require input in the form of Markov chains, Petri nets, etc. Lower-level derivates (such as Markov chains) can be obtained from this specification when this is desired, or the specification can be simulated directly.

In the longer range we are developing methodologies for designing and implementing modeling tools that are applicable to computer systems research. These will include analytic, statistical, simulation as well as expert system-like 'conceptual' modeling. We are currently extending the work already done with Partial Order Programming [69].

2.2. Prerequisites on Implementation

Tangram is implemented primarily in Prolog. Prolog is an excellent starting point for developing a modeling environment for at least two reasons:

---

*Currently Tangram is being implemented on top of SICStus Prolog. SICStus Prolog is a portable Prolog environment developed in *C* at the Swedish Institute of Computer Science.

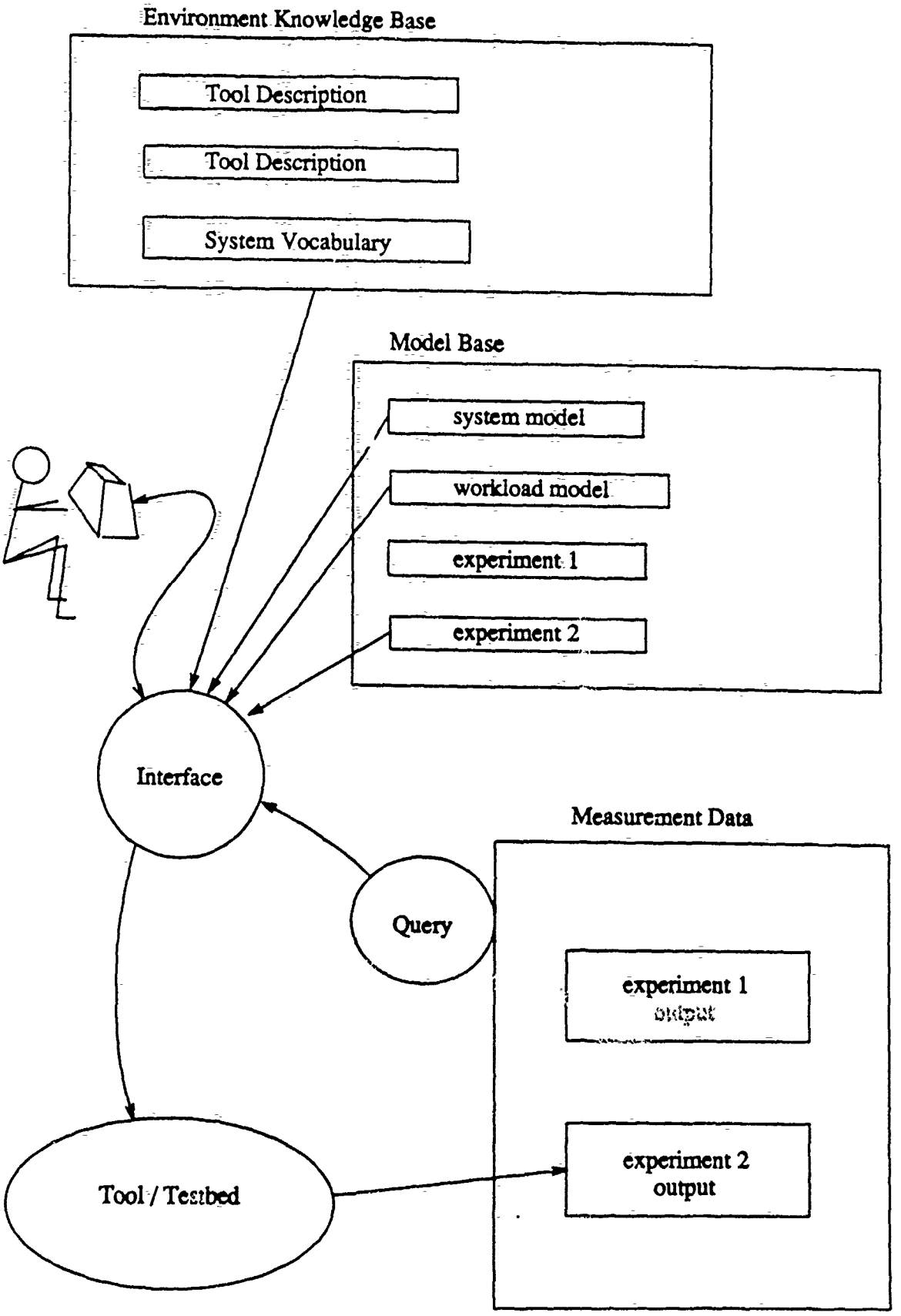


Figure 1. Tangram Modeling Environment Functions

(1) Prolog is unarguably the best candidate as a database/knowledge base language. It subsumes relational databases, supports complex structure in data through its terms and rules, and its first-order logic foundations are appropriate in many situations: unification and pattern matching, logical derivation and intensional query processing, backtracking and search, and most generally declarativeness. It easily supports increases in structure of models and 'expert system' techniques for interpretation of these models.

(2) Prolog is flexible. It is an outstanding vehicle for rapid prototyping, and permits easy access to existing systems that perform computationally intensive tasks efficiently.

To be effective, however, an environment based on Prolog must offer the features cited below.

2.2.1. Industrial Strength

As we pointed out earlier, increases in the quantity of data and in the complexity of interpretation require distributed computing/supercomputer technologies. Massive parallelism is needed to deal with the increased volume of information. Interactive display of model behavior is essential for effective modeling. An environment like Tangram incorporating these techniques will be successful only if it provides 'industrial strength' performance. Keys to success here are:

- Optimization
- Advanced data management technology
- High-performance interactive graphics
- Support at the operating systems level

2.2.2. Integration

A modeling environment must be able to combine many different systems of different types elegantly and efficiently. The many tools and testbeds for developing models that have taken man-centuries to develop should be accessible directly and conveniently from a single workstation. Prolog is excellent for representing and making inferences how these tools and testbeds should be accessed, but efficient access prohibits the use of *'glue job'* connections between them and Prolog. Database systems, for example, require stream access rather than the tuple-at-a-time access encouraged by Prolog. Keys to success here are:

- Modularization
- General ability to connect with diverse programs
- General knowledge representation of program functions
- Support for translation tools

2.2.3. Support for Evolution and Multiple Models

There are many ways to represent the same information. As models are refined over time they become more detailed, and focus on specific aspects of systems. Also, different models using different abstractions are necessary to represent complex systems accurately and efficiently.

Both evolution of models and different views of complex systems require different kinds of models, and hence different languages or *paradigms* for capturing different aspects of the real world.

Bobrow [16] criticizes Prolog on the grounds that there are many programming paradigms other than logic programming, and existing Prolog environments should, but do not yet, support them.

For modeling in particular, this criticism is of the essence. Paradigms can be low-level in nature, such as with parallel/distributed processing, object-oriented programming, or constraint satisfaction. They can also be more high-level or 'semantic' in nature, such as with extended queueing networks. Keys to success here are:

- Support for multiple modeling paradigms
- Representation of connections among paradigms

2.3. Partial Evaluation as an Implementation Strategy

The list of prerequisites above may seem somewhat imposing. It is not immediately apparent how they may all be achieved, or achieved with Prolog as a foundation.

Tangram uses partial evaluation as a performance-oriented mechanism for extending what paradigms are available in the Prolog environment. Software in the Tangram environment is written in one of two ways:

(1) As ordinary *Tangram Prolog* code. This code looks like 'standard' Edinburgh-style Prolog code, with the exception that a module system developed at UCLA is used, and many UCLA-specific predicates have been added.

(2) As code from an appropriate *paradigm*. A paradigm consists of both a language, and an interpreter for that language. In Tangram, there are many specialized paradigms, including:

- Parallel/Distributed processing
- Stream processing
- Object-oriented and functional programming
- Constraint satisfaction

Extension through addition of new paradigms is encouraged.

Currently paradigms in Tangram are implemented using a general partial evaluation scheme. With partial evaluation, the paradigm interpreter is used to translate paradigm language statements into Tangram Prolog code which can be optimized and subsequently executed. Loosely speaking, partial evaluation uses an interpreter as a 'macro' for expanding input statements into Prolog statements. As much evaluation of the 'macro' as possible is performed at expansion time, so partial evaluation can be thought of as a general optimization technique.

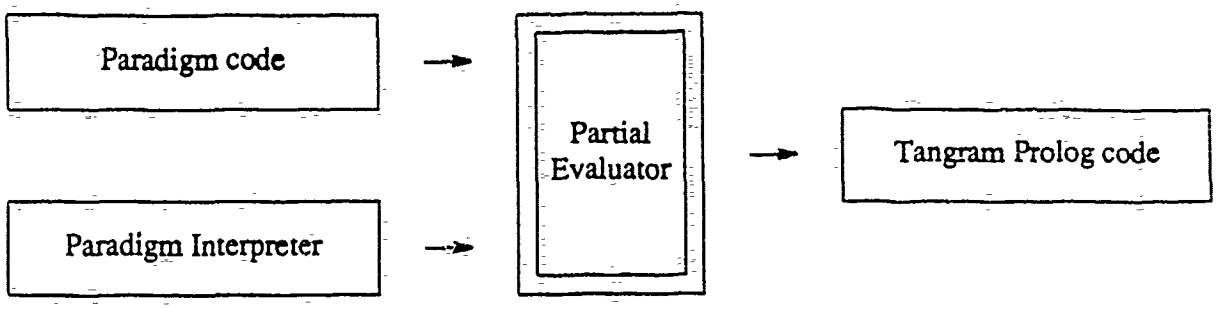


Figure 2. Partial Evaluation

Thus the interpreter for a paradigm can be used either for execution or compilation: paradigm code is either interpreted directly in Prolog, or partially evaluated into Prolog for subsequent execution. Recent work on partial evaluation is summarized in [87], along with issues in using it for interpreting parallel programming languages. Speedup factors of 40 have been reported from the use of partial evaluation instead of direct interpretation.

Partial evaluation can be performed multiple times, of course. That is, paradigms can be used hierarchically or accumulatively. For example, we can write the interpreter for a constraint satisfaction paradigm in the language of the object-oriented paradigm In this case multiple levels of partial evaluation are needed.

2.4. Project Overview

The individual tasks of the Tangram project are summarized in the following sections:

- Computer Systems Modeling
- Constraint Processing
- Stream Data Processing
- Industrial Strength Prolog

We have selected computer systems modeling as an initial modeling domain, as this is a domain in which we are expert. Constraint processing is necessary for the specification of models, and stream processing is necessary for the evaluation of model output. The industrial strength Prolog extends Prolog to be an effective language for 'real' applications, as opposed to the 'toy' applications for which it has been used in the past.

3. Computer Systems Modeling Applications

The only good environments are those that are used by their developers. To help guide direction of the Tangram environment, we have selected the general application area of knowledge-based modeling of complex computer systems. Within this area, Tangram is focussed on two projects:

- Complex Computer System Modeling
- Distributed DBMS Performance Modeling

3.1. Complex Computer System Modeling (Steven Berson, Bill Cheng, Dick Muntz)

This project focuses on developing methodologies for computer system modeling. The main objectives of this project are the following:

(1) To construct an environment that integrates various analysis tools so that computer system modeling experts can access them through a common interface. The environment should also be capable of giving expert advice on how a model of a computer system can be analyzed, and why. Such an environment would also allow integration of new tools.

(2) To facilitate the construction of modeling packages that are tailored to particular application domains for non-experts.

Computer system modeling falls into general domains such as performance analysis, availability analysis, and reliability analysis. In order to analyze a computer system, a model in a certain 'domain' has to be constructed. For each kind of domain, there exist many tools that can analyze models in the domain. However, tools are usually applicable only to a limited range of problem areas, and they will perform with different efficiencies in different problem areas. One of the goals of our environment is to manage the complex relationships between various domains and tools. Users of our environment should be able to describe their models in the form that is natural for their application domain, and the system should be capable of translating that description to the form required by the appropriate analysis tool. A prototype for modeling computer systems based on Markov processes [14] is now running, and is being extended.

Sometimes, the exact analysis of a complex computer system model is infeasible due to the size of the model; it is then necessary to perform approximate analysis of the model. There are different techniques for the approximate analysis of computer systems, and each of them works well under different conditions. With knowledge based techniques, our environment can assist users in using these approximate analysis techniques.

Currently, we are focusing on Markov processes and queueing networks. Continuing work on the current prototype includes:

(1) Graphical interfaces for entering, editing, and displaying modeling descriptions.

(2) Query facilities based on the high-level model.

(3) Model debugging/consistency checking.

(4) Extensions to allow the modeler to specify approximation analysis techniques.

(5) Heuristic interpretation of the analysis results.

(6) Model 'optimization', e.g. providing automatic state space reduction where possible.

We also wish to explore the problems of dealing with real computer systems. This requires developing methods for:

(1) Describing real computer systems: both their structure and the measurements that are available.

(2) Query and display facilities based on the system description.

(3) Correlation of analytic and/or simulation models with a conceptual model of the real computer system. This would provide the basis for model validation.

3.2. Distributed DBMS Performance Modeling (Ron Hu, Dick Muntz)

A performance measurement environment is being developed for the distributed data management system manufactured by Teradata, Inc. This system provides a rich environment in which to study the application of our modeling methodology. The system itself has a complex structure both in terms of the hardware and software architecture. The performance issues include: configuration planning, evaluation of hardware/software design alternatives, evaluation of query optimization strategies, logical and physical database design, and regression testing of the performance of new software releases. We plan to investigate the application of our modeling environment to this set of problems. The study will concentrate on the database aspects as well as the unique features of the architecture which we expect to provide new insights. The system described in [47] is a simple approximation to what we have in mind, and resembles the Tangram environment illustrated in Figure 1.

4. Constraint Processing

Until recently, models often enforced a strong distinction between the data of the model (parameters, initial values, etc.) and the model itself. Modeling was done as a very rigid kind of programming, with a strong distinction between programs and data.

This distinction has begun to blur. Largely a result of the influence of AI tools in software, new modeling tools have developed in which one can treat parts of the model program as parameters, or define parameters with additional models. Basically the idea is that one can specify behavior information (rules) in the same way one specifies descriptive information (facts).

In its essence, this idea amounts to what is commonly called *declarativeness*. That is, one should be able to describe real-world systems by declaring the *constraints* on their behavior, rather than by giving programs or other procedural descriptions that somehow fulfill these constraints. Not only is it easier to specify constraints declaratively, it is also easier to validate the correctness of a model that is specified declaratively. It is notoriously difficult to verify that a program satisfies the constraints that are expected of a model.

An instructive example of the importance of declarativeness comes from today's database systems. During the 1970s database systems moved away from 'procedural' query languages to declarative languages in which users could specify the results they wished, without having to specify how the results should be obtained. Today, fourth-generation languages (4GLs) are accepted as important means for specifying queries and data processing requirements. Constraint processing can be viewed as the continuation of the evolutionary development from DBMS to more powerful information processing systems.

Constraint processing is a modeling paradigm in which models may be developed declaratively. It is the modeling paradigm of choice within the Tangram project. Constraint processing is also used in other ways in Tangram, both in program analysis ('abstract interpretation') for optimization of programs, and in graphical display of information.

4.1. Partial Order Programming

A good deal of the Tangram work in constraint processing is based on partial order programming [69]. Partial order programming is a new paradigm developed by Parker at UCLA in which statements are constraints over partial orders. In this paradigm a problem has the form

minimize	u
subject to	$u_1 \sqsupseteq v_1$
	$u_2 \sqsupseteq v_2$
	$\cdots$

where u is the *goal*, and $u_1 \sqsupseteq v_1, u_2 \sqsupseteq v_2, \cdots$ is a collection of constraints called the *program*. A solution of the problem is a minimal value for u determined by values for u_1, v_1, etc. satisfying the constraints. The domain of values here is a *partial order*, a domain D with ordering relation $\sqsupseteq$.

The partial order programming paradigm has interesting properties:

(1) It generalizes mathematical programming, dynamic programming, and computer programming paradigms (logic, functional, and others) cleanly, and offers a foundation both for studying and combining paradigms.

(2) It takes thorough advantage of known results for continuous functions on complete partial orders, when the constraints involve expressions using only continuous and monotone operators. These programs have an elegant semantics coinciding with recent results on the relaxation solution method for constraint problems.

(3) It provides a framework that may be effective in modeling complex systems, and in knowledge representation for cognitive computation problems.

Recently we have applied partial order programming in the formalization of *directed logic programs* [70], logic programs which have specified input and output arguments. Such a formalization is important in defining the semantics of stream processing, and it unifies a number of diverse knowledge representation primitives as well.

4.2. Knowledge Representation (Stott Parker)

Modern knowledge representation systems tend to be frame-based systems or production systems with inheritance. They are largely ad hoc and support only certain kinds of shallow models [67]. The knowledge representation system planned for Tangram is formally founded on partial order programming, and will provide the following useful features:

Common systems of inference (syllogisms, parts, roles) [5]
Constraint solution mechanisms, including propagation and relaxation
Naming and Events
Semantic Unification
Contexts and Modalities
Meta-Level Capability & Planning
Schemata/Episodes/Definitions/Scripts/Prototypes/Defaults/Situations

These features should prove useful in the representation of both computer systems and of software packages used to model these systems. See [68].

We are developing a system for automated simple linear regression modeling. Certainly linear regression is the statistical test that is most useful in analyzing data. Our system is similar to the REX system described in [37], but iteratively refines a Box and Cox model, taking into consideration bias, normal distribution of residuals, and so forth. Issues in developing the system include how to represent functionality knowledge of the regression analysis tools available, and control knowledge for obtaining the desired transforms on the data from the regression tools. Currently the *S* statistical analysis package is being used.

A major issue in the design of the system is how much emphasis to put on interaction with the user about solution strategy. While statisticians prefer to explore their data and make decisions about transformations themselves, less knowledgeable users may prefer to have the system operate without any interaction in analysis. The issue is the degree to which statistical strategy in data exploration and analysis can be automated. Effective use of statistical analysis requires three kinds of expertise:

- Data expertise (general knowledge of what is in the data)
- System expertise (knowledge of what generated the data)
- Statistical expertise (knowledge of what the analysis means)

A modeler will have all three kinds of expertise only rarely. Statistical expertise helps avoid over- and under-interpreting data, and appears to be the most inviting to automation.

4.3. Constraint Satisfaction (Richard Huntsinger, Stott Parker)

The goal of the constraint satisfaction component of the project is the development of a powerful theoretical formalism which will provide semantics for a large class of model representations, and a correspondingly powerful constraint satisfaction system implementation [43].

Partial order programming [69] serves as an initial theoretical formalism. It provides a model-theoretic semantics and a procedural fixpoint-based semantics for model representations. These model representations are composed of partial ordering relations between variables and monotonic functions on variables. That is, models are represented as collections of constraints of the form $x \leq f(y)$, where x is a variable, $\leq$ is a partial order, f is a monotonic function, and y is a vector of variables.

This class of model representations corresponds to a subset of that on which relaxation can be successfully applied. Indeed, in practice, constraint satisfaction system implementations typically employ relaxation, and therefore operate only on restricted classes of all expressible model representations.

Partial order programming is currently being extended to permit some non-monotonic functions on variables, resulting in semantics for a larger class of model representations. This class corresponds to that on which a generalization of relaxation can be successfully applied. Equivalently, it is a class for which each model representation can be transformed to a new representation on which relaxation can be successfully applied. For example, the model representation $x \leq f(y)$, where f is a non-monotonic function, is provided semantics if $x \leq f(y) \Rightarrow x \leq t \bullet f(y)$, t is an invertible transformation, and $t \bullet f$ is a monotonic function [44].

The theorems comprising this extension to partial order programming suggest several constraint satisfaction algorithms. A constraint satisfaction system implementation is being realized as a tool which facilitates experimentation with some of these algorithms. Specifically, it employs a general parameterized algorithm; instantiated instances are various specific algorithms, including relaxation.

Parameters of special interest include

- the strategy for organizing functional dependencies,
- the strategy for detecting transformable constraints,
- the strategy for applying transformations, and
- the strategy for decomposing sets of constraints into approximating sets of constraints.

Some testbed model representations for the constraint satisfaction system implementation

include computer network configurations, solid-body animation sequences, and music compositions.

4.4. Abstract Interpretation (Arman Bostani, Stott Parker)

A lot of research has been done on the derivation and proof of properties of Prolog programs. This work has been fueled by the desire to close the performance gap that exists between imperative and logic programming languages running on conventional hardware. Several reasons have been cited for this performance gap. Firstly, procedures in logic programs are quite versatile and the code can be general enough to be used in many different ways. Secondly, in imperative programs, memory usage is explicitly controlled by the programmer; this not only saves memory, but also time since it avoids copying whole data structures when only slight modifications are made. Finally, type information that is available in an imperative language allow a better implementation of the programmers' data structures.

In the attempt to close the performance gap many researchers have implemented various automated inference systems which can generate polymorphic typing, mode declarations, determinancy information, etc. To formalize this work on Prolog programs, researchers have adapted the ideas of Cousot and Cousot [30] on the abstract interpretation of imperative programs to the field of logic programming. Since a characterization of the exact behavior of a program is in most cases computationally intractable, we are forced to interpret our programs abstractly. That is, the program is thought of as executing in an abstract domain where less information about the data items is accounted for. Results of the computation in this abstract domain then reflect the properties of programs operating in the exact model.

Abstract interpretation of Prolog programs has been used in several applications:

- Automatic inference of polymorphic types [25].
- Automatic mode inference of Prolog predicates. Debray and Warren [34] describe a data flow analysis which is more powerful than previous approaches which solely relied on purely syntactic information.
- Detection of determinancy. Mellish [59] discusses a method for detecting 'determinate' Prolog code (i.e., finding those predicates that never return more than one solution).
- Global optimization of Prolog programs. A general theoretical framework is provided by Bruynooghe [21] with which an optimizing Prolog compiler may use abstract interpretation for efficient code generation.

The main focus of our research has been to devise a system that will be able to derive various properties of prolog programs such as mode, type, aliasing and predicate success information. Previous research has shown, however, that purely syntactic analysis of programs is insufficient for the derivation of truly useful information. Thus, from the very beginning, our goal has been to create a simple formalism under which we can represent both syntactic and semantic information in a unified manner.

To capture the semantics of Prolog programs, we use *'inexact success models'*. These success models provide information on the function of a predicate by classifying its actions based on the

types of arguments with which the predicate is invoked. Two elements determine the 'function' of a predicate:

- Instantiation of variables
- Success of the predicate

Information on the instantiation of variables can be used in the derivation of mode characteristics of the predicates. Also, the type classification above is similar to conventional polymorphic type inference. Determination of the success of predicates, however, affects the derivation of mode and type information. The idea of using success information in abstract interpretation is relatively new [21] and unexplored.

Our success models provide bounds on the number of choice points created by a predicate, and whether or not it will 'fault'*. Thus, our system will be able to detect possible fault conditions before the execution of a program. The task of determinancy detection [59] is also easily performed with this system (e.g., if a predicate generates only 0 or 1 choice points, it is determinate).

The design phase of the system for the inference of these success models is almost finished. Our next step will be to implement the success model derivation system. The system can then be used with applications such as optimizing compiler, interactive debugging tool, performance analysis, etc.

4.5. Graphics (Ted Kim, Stott Parker)

Many modeling applications require graphics. One approach would be to use a declarative graphics interface [41] based on formal picture description grammars. The formal framework offers some attractive parallels to proof systems. This paradigm offers the capability to describe, compose and generate pictures as well as 'prove' or recognize pictures. While we are pursuing some research in this area, we feel that this graphics description is too low-level for general use.

Our main effort is to provide a higher level constraint-based graphics interface [19]. The constraint-based orientation provides a declarative style of programming graphics. Our system uses the X Window system [75] to provide network graphics and windowing capabilities. X also offers the advantage of being a de facto standard. At the lowest level, our design will provide support for graphics by making the X library functions accessible in Prolog without requiring extensive changes to Prolog (e.g., no asynchronous operation).

On top of this layer, we provide a constraint-based graphics toolbox. This toolbox provides common primitives for design of graphical applications as well as support of constraint systems. Speed is very important to graphical applications. The challenge here is to provide general enough constraint solvers that are also fast. Towards this end, our design includes a notion of graphical state. The system responds to perturbations made to a current constraint solution, while attempting to resolve as few constraints as possible. The state from the previous solution is used as much as possible in forming the new solution. This allows local changes to be quickly solved.

*In most Prolog systems a program may terminate unexpectedly due to a fault. A fault may be caused, for example, by passing arguments of incorrect type to a system predicate.

Typically, the toolbox would be used to build graphical displays of modeling solutions. These displays would change automatically in response to perturbations of modeling data. This provides the basis for interactive model management, involving storage, retrieval, query, display, and editing of models [50, 58, 82]. Animation can also be provided in this fashion.

Continuity is also important in graphics, especially in animation. Changes in animation should be smooth and continuous. The graphical state helps here, but is not sufficient. Out of the possible space of solutions for the constraints, we would like to pick a 'good' solution. For the animation problem, 'closeness' of a new solution to an old one is the issue. For the automated display problem [56], the criteria could be such things as expressiveness or color choice. More generally, we can cast the problem as optimization. To address these issues, we are planning to include an optimization mechanism for our graphics constraint solver.

Graphics is also important in the presentation of refinement graphs of data flows in co execution models. With the concurrent execution model, user annotations of programs can be used by the system to generate diagrams of refined data flows implied by the annotations. These diagrams allow the user to spot potential problems with his annotations.

5. Stream Data Processing

In this section we describe projects concerning the *Tangram Stream Processor (TSP)* [71], a stream data management system. It is an extensible system based on a functional sublanguage of Prolog that provides a programmable stream processing capability with a number of interesting characteristics.

5.1. Streams and Data Processing

Relational databases are founded on set theory: all relations are viewed as sets of tuples. Many developments have encouraged generalization of this model to one of *ordered sets*. For example, ordered data can be processed much more efficiently than unordered data. In fact, many standard query evaluation techniques are described in terms of operators (filters, mappings, actors) acting on ordered sequences of tuples. Moreover, temporal query processing seems to necessitate some kind of ordering if important kinds of queries are to be efficiently answerable. Also of course, the order of the tuples in relations is important in presentation of the relations to users.

In many important situations, then, it is advantageous to generalize the set foundation of the relational data model to an ordered set model. We call ordered sets *streams*. Stream-oriented processing is certainly not a new subject, although it has only recently come into its own right as a programming paradigm.

An area where streams are inherent to query processing is for temporal data, data with explicit or implicit time ordering. The analysis of streams has been done for many years as 'time series analysis'. Recently, the subject of time in databases has gotten increasing attention as more applications requiring temporal reasoning have been uncovered [3, 4, 12, 18, 27], and many interesting systems handling temporal queries in novel ways have been developed [2, 33, 49, 51, 52, 55, 77, 84, 85].

Previous research has concentrated either on database processing, or on representational issues and generality of modeling. Two important database systems include:

(1) TQuel [84], a relational query language with embedded time primitives, is an extension of Quel, both syntactically and semantically. TQuel is essentially a relational query language, resting on the relational model.

(2) The Time Sequence approach of Shoshani [77, 79] characterizes properties of temporal data and temporal operators without restriction to the relational model. Data are organized into *Time Sequence Collections (TSCs)*, which can take both relational and stream-like representations. Five basic operators provide an algebra working on TSCs.

These systems emphasize performance and complete handling of a well-defined set of query operators. Other researchers in temporal query processing have worked at more complex modeling, combining work on temporal logic and existing representational systems to define new approaches. Sadri [73] reviews three general recent approaches to temporal reasoning:

(1) The 'event calculus' of Kowalski [49] is an approach for reasoning about events and time within a logic programming framework.

(2) Allen's approach [3, 4] is similar to the event calculus, defining a set of binary predicates giving basic relationships among time intervals (whether they overlap, one precedes the other, etc.).

(3) Lee, Coelho and Cotta [52] present a temporal system for representing and reasoning about time-dependent information and events, specifically for business database applications.

In these approaches it is peculiar that stream processing has not been emphasized more heavily for temporal query processing, as well as for basic relational query processing. Tangram's stream processing approach permits it to handle queries definable under each of the systems listed here.

5.2. Streams and Parallel Processing

Concurrent, object-oriented, functional, and logic programming paradigms all intersect elegantly in the abstraction of streams: the general architecture of cooperating parallel actors transforming streams of events has found its way into many programming systems that have been proposed in the past few years. For example, many parallel logic programming systems have been developed essentially as stream processing systems. Typically, these systems fall into one of several camps:

(1) They resemble PARLOG [26, 39] and the other 'committed choice' parallel programming systems [78] (Concurrent Prolog, GHC, etc.).

(2) They introduce '*parallel and*' or '*parallel or*' operators into ordinary Prolog [53].

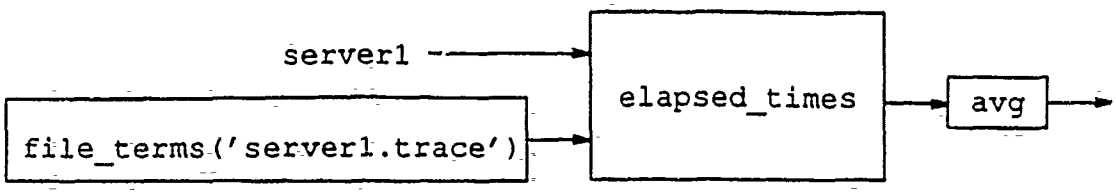
(3) They are extended Prolog systems that introduce streams by adding functional programming constructs [32, 48, 54, 62, 86]. The thrust of this introduction is to make Prolog more like either Lisp or Smalltalk or both.

TSP has drawn on the designs of a number of previous systems which have included stream concepts. These include FAD [6], various dataflow database systems [7, 8, 9, 15, 38], and LDL [11, 89].

After some experience with the tuple-at-a-time and whole-query-at-a-time (embedded query language) Prolog/DBMS interfaces that have been developed to date, we feel a better way to integrate Prolog and databases is through streams [65]. Only minor extensions to Prolog are sufficient to provide fairly efficient stream processing [72]. A stream interface offers an effective medium between these two alternatives, uniformly integrating bulk operations at the DBMS end with incremental evaluation at the Prolog end. Prolog stream processing avoids backtracking through a database, using efficient iterative (tail recursive) processing instead. It is a natural approach for applications like analysis of modeling data.

5.3. The Tangram Stream Processor

The Tangram Stream Processor is founded on the abstraction of transducers. A transducer is a mapping from some number of input streams to one or more output streams. Thus, a transducer may be viewed as an automaton. However, a transducer can take parameters, and as such need not have only a finite number of states. Thus, it is better initially to view transducers as mappings instead, and diagrams of transducer networks resemble dataflow diagrams:



Transducers are the basic building blocks of TSP, and are maintained in an (extensible) library. Since arbitrary transducers are permitted, the expressive power of TSP is equivalent to that of any general programming language. Consequently, the stream-based transducer model is more general than many previous approaches: it is capable of handling traditional database queries and non-traditional queries that reason about time in event databases.

TSP has several further unique aspects:

(1) TSP permits operation on general stream structures, including for example both lists and array models of data. It supports definition of and parallel evaluation of operators on these stream structures, including the operator families of the APL programming language, NIAL [57], and the Nested Array model of data upon which both are based [60, 61]. This includes the ability to define *higher-order operators* on streams, such as aggregate operators (*min*, *max*, *sum*, etc.), APL's reduction operator, LISP's *maplist*, etc. In addition, it permits us to define many useful statistical operators on streams, as in the *S* data analysis system [10].

(2) TSP permits operation on *infinite streams*. A stream may represent a non-terminating sequence of values. This is not permitted, for example, by APL.

(3) TSP permits both *lazy and eager evaluation* of streams. Lazy evaluation permits efficient evaluation of some kinds of queries.

(4) TSP transducers are naturally implemented as concurrent processes. These transducers provide opportunities to place natural boundaries on parallelism, a feature not enjoyed by some parallel Prolog systems.

The resulting system may be used for '*database-flow*' processing, a combination of 'dataflow' and database processing, as well as general feature extraction and data reduction operations that fit in a pipeline structure.

Execution of queries in TSP is quite efficient, in the common situation that the input streams are sorted properly. In fact, TSP query processing can be considerably more efficient than that in relational DBMS. For many TSP queries a single scan of the input streams is sufficient, requiring linear time and constant space, while relational DBMS approaches require significantly more resources. Also, TSP can handle kinds of queries not easily handled by relational query processing systems, including the following:

1. Sliding window queries [77]
2. Event calculus queries [49]
3. Pattern matching queries
4. Abstracting state information from event data
5. Reasoning about time.

As an example of a query that reasons about time, consider asking about what investment strategy would have been optimal over a given period of stock market history. This requires innovative accumulation of dividends, interest rates and rules for compounding interest, days which are holidays, and many other important details. These *'hindsight queries'* illustrate the potential of stream processing in database analysis.

5.4. Log(F)

The Tangram Stream Processor is based on Log(F), an integration of Prolog with a functional language called F*, developed by Sanjai Narain at UCLA [63, 64]. Log(F) is the integration with Prolog of a functional language in which one programs using rewrite rules. This section reviews the major aspects of Log(F), and describes its advantages for stream processing.

F* is a rewrite rule language. In F*, all statements are rules of the form

LHS => *RHS*

where *LHS* and *RHS* are structures (actually Prolog terms) satisfying certain modest restrictions summarized below.

A single example shows the power and flexibility of F*. Consider the following two rules, defining how lists may be appended:

```
append([],W) => W.
append([U|V],W) => [U|append(V,W)].
```

Like the Prolog rules for appending lists, this concise description provides all that is necessary.

Log(F) is the integration of F* with Prolog. In Log(F), F* rules are compiled to Prolog clauses. The compilation process is straightforward. For example, the two rules above are translated (partially evaluated) into something functionally equivalent to the following Prolog code:

```
reduce(append(A,B),C) :- reduce(A, []), reduce(B,C).
reduce(append(A,B),C) :- reduce(A, [D|E]), reduce([D|append(E,B)],C).
reduce( [], [] ).
reduce( [X|Y], [X|Y] ).
```

An important feature of F* and Log(F) is the capability for *lazy evaluation*. With the rules above, the goal

```
?- reduce( append([1,2,3],[4,5,6]), X ).
```

yields the result

```
X = [1|append([2,3],[4,5,6])].
```

That is, in one `reduce` step, only the head of the resulting appended list is computed. The tail, `append([2,3],[4,5,6])`, can then be further reduced if this is necessary. Demand-driven computation like this is referred to as lazy evaluation or delayed evaluation, and is basic to stream processing [1].

Log(F) is a superior formalism for stream processing, and thus for database query processing. From the example above, it is clear that the rules have a functional flavor. Stream operators are easily expressed using recursive functional programs. The syntax is convenient, and can be considered a useful query language in its own right.

It turns out furthermore that Log(F) has a formal foundation that captures important aspects of stream processing:

(1) Determinate (non-backtracking) code is easily detected through syntactic tests only. This avoids the overhead of 'distributed backtracking' incurred by some parallel logic programming systems.

(2) Log(F) assumes that stream values are *ground terms*, i.e., Prolog terms without variables. Again this avoids problems encountered by other parallel Prolog systems which must attempt to provide consistency of bindings to variables used by processes on opposing ends of streams.

These features of Log(F) make it a nicely-limited sublanguage in which to write high-powered programs for stream processing and other performance-critical tasks. Special-purpose compilers can be developed for this sublanguage that produce highly-optimized code.

We must stress strongly that Log(F) is an *extension* of Prolog. The Log(F) code shown above runs as shown. The issues here are not so much language design issues as in developing compilers for Log(F) that generate fast code. Where speed is not critical, the full power of Prolog is available to its users now, in a stable development environment.

5.5. Stream Transducers and Log(F) (Lewis Chau, Dick Muntz, Stott Parker)

In [71] we show how transducers can be written in Log(F) to solve both traditional and very novel query processing problems. It is easy to develop significant stream transducers with compact sets of rewrite rules. We currently have an implementation of Log(F) in Prolog that performs all standard database query processing primitives, and many nonstandard ones as well. For example, transducers can be developed that manipulate streams of 'elapsed time' data, or streams of 'service time' data. We can define transducers as follows:

```
% Elapsed times for first-come, first-served discipline
fcfs_e([],_) => [].
fcfs_e([a(T)|L],State) => fcfs_e(L,append(State,[T])).
fcfs_e([d(T)|L],[T0|S]) => [T-T0|fcfs_e(L,S)].

% Elapsed times for last-come, first-served discipline
fcfs_e([],_) => [].
lcfs_e([],_) => [].
lcfs_e([a(T)|L],State) => lcfs_e(L,[T|State]).
lcfs_e([d(T)|L],[T0|S]) => [T-T0|lcfs_e(L,S)].
```

```
% Service times for first-come, first-served discipline
fcfs_s([],_) => [].
fcfs_s([a(T)|L],(N,T0)) => if(N==0, [fcfs_s(L,(1,T))],
                                    [fcfs_s(L,(N+1,T0))]
                                ).
fcfs_s([d(T)|L],(N,T0)) => [T-T0|fcfs_s(L,(N-1,T))].

% Service times for last-come, first-served discipline
lcfs_s([],_) => [].
lcfs_s([a(T)|L],[]) => lcfs_s(L,[(T,0)]).
lcfs_s([a(T)|L],[(T0,T1)|S]) => lcfs_s(L,[(T,0),(_,T-T0+T1)|S]).
lcfs_s([d(T)|L],[(T0,T1),(T2,T3)|S]) => [T-T0+T1|lcfs_s(L,[(T,T3)|S])].
lcfs_s([d(T)|L],[(T0,T1)]) => [T-T0+T1|lcfs_s(L,[])].
```

These transducers may appear a little forbidding. We can however make these available in a simpler and more natural form by introducing 'higher level' transducers:

```
elapsed_times(Server,S) => policy_elapsed_times(type(Server),S).
policy_elapsed_times( [fcfs], S) => fcfs_s(S,[]).
policy_elapsed_times( [lcfs], S) => lcfs_s(S,[]).

service_times(Server,S) => policy_service_times(type(Server),S).
policy_service_times( [fcfs], S) => fcfs_s(S,(0,_)).
policy_service_times( [lcfs], S) => lcfs_s(S,[]).
```

Interesting statistics (e.g. mean, standard deviation, etc) can then be calculated from this output stream by applying further aggregate operators. For example, the average elapsed times and maximum service times at Server 1 can be obtained with:

```
avg( elapsed_times(server1, file_terms('server1.trace')) )
max( service_times(server1, file_terms('server1.trace')) ).
```

Here `file_terms('server1.trace')` produces a stream of arrival and departure terms

```
a( ArrivalTime )
d( DepartureTime )
```

that are tallied by the transducers defined here. Also, `type(server1)` reduces either to `[fcfs]` or to `[lcfs]`, according to the type of the server.

5.6. Pattern Matching against Streams (Lewis Chau, Stott Parker)

In [71], we illustrate how Log(F) makes a powerful language for expressing transductions of streams. In this section we show how, specifying patterns with *grammars*, it also makes an expressive language for pattern matching against streams.

In order to match patterns against streams, the approach taken in Tangram is to let users specify patterns with *grammars*, which are compiled into efficient transducers. Moreover, users can express their patterns using a library of grammars. For example, regular expressions and, more generally, path expressions, can be easily defined with grammar rules:

```
(X+) => X.
(X+) => (X, (X+)).
(X*) => [].
(X*) => (X, (X*)).
(X;Y) => X.
(X;Y) => Y.
(X,Y) => append(X,Y).
skipto(X) => X.
skipto(X) => ([_], skipto(X)).
```

These Log(F) rules behave just like the context free grammars they resemble.

Pattern matching is signaled explicitly with the match transducer, which takes as its first argument a functional grammar term describing the starting symbol(s) of some grammars used for the match, and as its second argument a Log(F) term that produces a stream. For example,

```
match(([net_failure]+, [cpu_failure]), file_terms('experiment1.output')).
```

matches the pattern 'one or more copies of net_failure followed by a cpu_failure' to the stream of events in the file `'experiment1.output'` into a stream of event types.

The rules for pattern matching are very simple. The basic definition is as follows:

```
match([],S) => S.
match([X|L],[X|S]) => match(L,S).
```

The result of matching a pattern against a stream is what is left of the stream after pattern matching completes, i.e., the remainder of the stream that is not matched by the pattern. Simultaneously, arguments of the nonterminals are bound to values resulting from parsing the input stream. With this definition of `match`, we can immediately define grammars using rewrite rules. We call a collection of these rules a *functional grammar*.

In the section above we showed how transducers can manipulate streams of 'elapsed time' data. It is also possible to specify it with a functional grammar:

```
fcfs_e(Result) => fcfs_e([],[],Result).
fcfs_e(_,Result,Result) => [end_of_file].
fcfs_e(State,Current,Result) => [a(T)],
        fcfs_e(append(State,[T]),Current,Result).
fcfs_e([T0|S],Current,Result) => [d(T)],
        {T1 is T-T0}, fcfs_e(S,[T1|Current],Result).
```

The main issue here is finding a way to compile functional grammars and *match* into efficient transducers, then we can define efficient classes of functional grammars. Not surprisingly, deterministic tail-recursive grammar rules which do not construct large structures for their state can be compiled to efficient transducers. These grammars are much like classical right linear regular grammars, and like DCGs that are actually written in practice. See [24].

5.7. Distributed/Parallel Processing (Brian Livezey, Dick Muntz)

Log(F) benefits from two aspects of distributed processing. First, computation can be performed in parallel. There are many opportunities for computation concurrency in a stream-based programming environment. The execution behavior of sequential Log(F) is analogous to a pipeline in which only one stage is active at any given time. By placing different stages of the pipeline on different processors and replacing lazy evaluation by eager evaluation and stream flow, we can achieve a state in which all stages of the pipeline are active simultaneously. Second, in addition to this 'pipeline concurrency', we can achieve other forms of parallelism. AND-parallelism is achieved by producing all input streams to a given transducer simultaneously (i.e. concurrent reduction of all arguments to a function). OR-parallelism results from having different portions of the same stream being produced simultaneously on different processors.

The performance of queries on distributed databases is greatly affected by the relative locations of the data and the processes that operate on that data. Many techniques exist for optimizing queries on distributed databases. Distributed Log(F) provides the ability to exploit these techniques. By providing the ability to subdivide queries and specify the processor upon which each portion is to execute, Log(F) allows programmers to express efficient distributed queries.

Many stream-based concurrent programming systems [78] are designed for shared-memory multiprocessors and therefore attempt to exploit a very fine granularity of concurrency. In non-shared memory environments, where communication and processes are not cheap, any performance gained through concurrency will be lost in overhead. For such environments, it is necessary to allow a much coarser granularity of concurrency.

Ideally, in non-shared memory environments, the compiler should be able to recognize potential concurrency, balance it against the overhead, and decide how to distribute a given program. While such compilers exist for distributed database queries, no such compilers exist for arbitrary distributed programs. Therefore, we must initially require that the programmer specify how a given program is to be distributed. However, the programmer should not have to define the distribution of the program when designing the logic. Instead, the programmer should write the entire program first and then specify concurrency without changing the program's semantics.

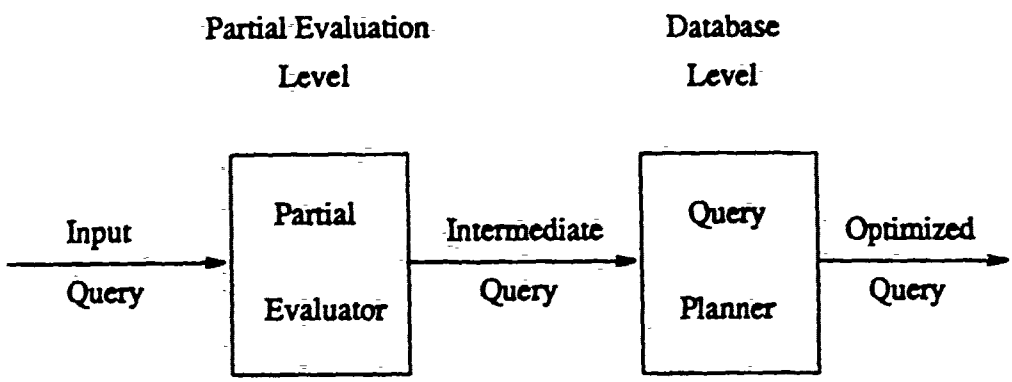
We intend to provide two interfaces to distributed Log(F) to facilitate the composition of distributed programs. First, we provide simple annotations which allow the programmer to indicate transducers which are to reside on remote processors. Second, we intend to provide a graphics interface to Log(F). Programmers will select transducer icons from a toolbox and connect them together to form larger transducers. Process boundaries will be indicated by surrounding portions of the resulting graph with boxes to indicate that that portion is to be run on one processor. Boxes will be annotated to indicate which processor they should be run on.

Ultimately, we will not require the programmer to specify how to distribute his program; we will only require that the programmer assign weights to each of the elementary transducers used in his program. Weights will be a function of the computational expense for each input element as well as the ratio of input elements to output elements. Weights of larger transducers will be determined by appropriately combining the weights of the elementary transducers that compose them. These weights will be used by the compiler to determine how best to distribute the program.

5.8. Program/Query Optimization (Lewis Chau, Cliff Leung, Dick Muntz, Stott Parker)

Partial evaluation is a special kind of program transformation for the purpose of optimization. This optimization is accomplished mainly via instantiation of parameters of a program by propagating values for top-level formal arguments through the program (execution of the unification at compile time), and reduction of the number of logical inferences by opening calls. It is similar in many ways to macro expansion.

A query that is a Prolog goal can be executed in many different ways. *Two-level optimization* is introduced as a means of finding a way to execute the query that is computationally as efficient as possible. *First-level optimization* applies partial evaluation as an alternative to compiling queries. With this approach, the partial evaluation system expands an input Prolog query, generating a conjunction of calls to the extensional database (EDB) intermixed with calls to built-in predicates. Redundant goals can be eliminated to some extent at this level. The partial evaluation system then transmits conjunctions of calls which can be the subject of further optimization by a database system. *Second-level optimization* (query planning) then optimizes a query at the database level. Its purpose is to analyze and improve queries based on straightforward information about the Prolog program underlying the query and the EDB itself. See [23].



Two-level Query Optimization

We are interested in applying partial evaluation to query optimization. A knowledge base is composed of a set of rules and ground facts. We can treat the set of rules simply as a logic program and the set of ground facts as a conventional relational database. Answering a query is equivalent to partially evaluating the query by an interpreter (partial evaluator), and executing the resulting conjunction of calls to the database. Optimization can be applied both to the partial evaluation level and to the database level. Currently, we have implemented a partial evaluator for full Prolog programs.

The output of partial evaluation is a conjunction of calls to the extensional database intermixed with calls to builtin predicates. In second level optimization, our major concern is the design of a query plan such that the resulting query will be executed more efficiently. In Prolog, the ordering of clauses in a program, and the ordering of goals in the right-hand side of a clause, is important control information that helps to determine the way a program is executed. This control information permits generation of an efficient query plan. The control information that is critical

to the query planner is described in [23]. Currently, we are applying the two-level optimization technique to the Tangram Stream Processor.

5.9. The Synopsis of Database Responses (Chung-Dak Shum, Dick Muntz)

Conventional responses in database systems, usually given as lists of atomic objects, although sufficient to serve the purpose of conveying information, do not necessarily provide efficient and effective communications between a user and the system. Recently, new notions of answers to queries have been receiving more research interest. For example, in [45], an answer to a query is expressed in terms of both atomic facts and general rules; in [29], intensional descriptions or concepts are being used as part of an answer. This latter notion of answers is particularly helpful when the number of entities or objects which satisfy the query is very large.

In [29], the answer to a query is expressed not as a set of individuals, but as a set of concepts or predicates, whose extensions may not be explicitly represented. Now suppose that we have retrieved a set of individuals as the answer to a conventional database query, and we want to re-express the answer in terms of a set of concepts. Those concepts must, of course, be pre-defined; otherwise, the user may not be familiar with them and the answer in terms of those concepts thus will not make too much sense. One of the immediate drawbacks to such an approach of expressing answers is that the extensions of the pre-defined concepts often do not satisfy the query conditions as a whole. As a result, we cannot express answers the way we want except in very rare cases.

We consider expressions for answers in terms of concepts and individuals [80]. Exceptions within individual concepts are allowed. Two criteria are defined as measures of the *goodness* of such expressions: (i) minimizing the total number of terms; (ii) minimizing the number of exceptions. Expressions satisfying these two criteria are called optimal expressions. We have shown that, under a strict taxonomy of concepts, any two optimal expressions for an extensional answer share the same set of terms. The inductive proof also leads to an algorithm for obtaining such expressions. Generalizing the strict taxonomy of concepts to a join-semilattice of concepts eliminates the term uniqueness property and also makes the problem of finding an optimal expression intractable. The problem under multiple taxonomies, although it involves a restricted type of join-semilattice, remains intractable.

One of the motivations behind our interest in different forms of answers is their *conciseness*. However, if there is a large number of individuals within a concept and approximately half of the individuals satisfy the query. Using an expression of concepts and individuals, no longer are we able to express our answer concisely. If we insist on concise answers, one possible 'solution' is to sacrifice *preciseness* for conciseness [81]. For each concept, we associate a count of its individuals and a count of qualified individuals which satisfy the query and refer to them, collectively, as a quantified concept. An aggregate or imprecise response is just an expression of quantified concepts. We study the tradeoff between conciseness and preciseness. Conciseness is measured by the length or the number of quantified concepts in an expression, and preciseness is measured by the entropy or the amount of uncertainty associated with the expression. Given its length, an expression with the minimum amount of entropy is considered optimal. Under a one-level taxonomy with the same cardinalities for all leaf concepts, the problem of finding an optimal expression can be solved inexpensively. An efficient heuristic is also proposed for the

general one-level taxonomy. For a taxonomy of more than one level, an algorithm is suggested. Although it is does not always lead to an optimal expression, it avoids the combinatorial explosion associated with the problem and appears to lead to good solutions.

Our work on imprecise responses is closely related to

- Statistical Databases
- Categorical Databases

Studies on statistical database management systems [40] suggests the definition summary tables in databases which are physically stored and maintained as redundant data as well as the original global database. Making use of such summaries, a large class of queries can be answered without extensively accessing data from the global databases. Currently, we are interested in the representations of such summaries under the general context of information abstraction.

6. Tangram Industrial Strength Prolog

Prolog goes a long way toward providing the kind of declarative modeling functionality we desire, and has the added benefit that it has a strong connection (from its logical foundation) with relational database systems. Still, to provide the 'industrial strength' environment we require for Tangram, a number of extensions to Prolog must be made:

- Module management
- Prolog database management
- Object management

Tangram Prolog is an extended Edinburgh-style Prolog system, augmented with a development environment and modules of low-level primitives for the functions listed above. Modules permit selective access to subsets of these primitives to individual Prolog processes.

6.1. Module Management (Tom Page, Dick Muntz)

The concept of reducing software complexity through modularization is well known and essential. Conventional languages have employed procedures and abstract data typing techniques to achieve modularity. Program modules can be constructed independently and composed to form larger systems. Access to a module is available only via its published interface. Internal data structures and procedures are invisible outside the module. The design and implementation process can be facilitated by transparently replacing initial, simple implementations of data structures or services with more sophisticated versions which maintain the same well-defined interface.

By contrast relatively little work has been done on modularization in logic programming systems. Tangram Prolog subdivides the name space of procedures so that each module has its own complete name space. Different parts of a system can be written without knowledge about the local names of other parts. Modules can be collected into libraries to group independent subsystems. Libraries are modules of modules which have their own published interfaces and hide the interfaces of internal modules.

6.2. Prolog Database Management (Tom Page, Dick Muntz)

There is considerable interest in combining database and logic programming technologies [66, 46, 90, 93, 94]. The motivation stems primarily from appreciation of the complementary benefits of the two technologies which were developed largely independently [92, 83, 76]. Our modeling environment requires more sophisticated interpretation of data than current database systems provide as well as efficient access to larger volumes of data than current Prolog implementations afford.

Many attempts at connecting Prolog with Relational DBMSs have been documented over the past few years [17, 94]. However, simply connecting the two systems via an interface is woefully naive [22]. Current Prolog implementations were designed to provide very fast unification over small atom spaces. The problem is that all data that is brought into the Prolog workspace becomes tightly intertwined in order to optimize unification, the performance bottleneck in Prolog. Atoms are not easily garbage collected or dropped from the workspace when they are no longer of interest. The volume of data in database applications quickly swamps current Prolog

system tables. The size of applications envisioned taxes the Prolog programming environment beyond its limits.

Our assertion is that the database should be viewed as a huge virtual workspace for Prolog. This workspace must cache units of data that are persistently stored in the database.

The workspace management replacement algorithm must handle three patterns of database access.

(1) Large flat relations may be accessed via the stream interface. Streams are sequences of terms that flow into transducers. An input stream can be discarded after processing by the transducer.

(2) Large complex objects may be integrated into the Prolog workspace by 'opening' them, and then operating upon them as though situated in memory. When they are 'closed', they are either discarded or rewritten in the database.

(3) Modules of code stored in the database may be brought into the Prolog workspace. Code in the modules may then be run until the modules are discarded.

Standard Prolog implementations lack the ability to deal with code or data as modules. Each clause is independent in Prolog; relationships or structure among clauses are not expressed but rather established operationally via inference [20]. While Prolog programs exhibit little locality when viewed from this perspective of extreme modularity, modules provide a logical bundle of code/data within which locality is expected. Thus, we propose a new organization for the Warren Abstract Machine [36, 91] in which separate atom and functor tables are built for each module. If locality exists, program execution will cross module boundaries infrequently relative to intra-module unifications. Within each module, atom unification will be very fast at the expense of translating arguments across module boundaries.

6.3. Object Management (Tom Page, Dick Muntz)

One characteristic of a logic programming language is that the same call with the same arguments returns the same results in any context [28]. All of the information needed to perform an operation must be present in the arguments and not recorded in the state of the program. This makes it very difficult to achieve the important principle of data abstraction in a logic programming language. We must have a way to represent persistent state information in Prolog, especially if we are to provide a transparent database model.

Tangram Prolog will provide an object-oriented programming module which adds the notion of persistent objects to the language. Objects may be managed by the database and become active when they are addressed with messages which they support. Name binding issues with respect to inherited methods will be explored.

7. Future Projects

The list of open projects is vast. For example, translation is a pervasive problem. The multitude of differing tool input/output formats requires good translation tools, and mapping among different kinds of models is an important related problem. The SARA project at UCLA has dealt heavily with translation issues, and it seems likely that some SARA tools can be adapted here.

Several important projects that we have begun to address are listed here:

- Estimation of time/space requirements to find solutions of models
- Automated use of tools/testbeds given an experiment
- Automated sensitivity analysis
- Automated explanation
- Induction of analytic models from behavior (learning)
- Paradigm systems for
  - Production Systems/Triggers
  - Petri nets

> Those who sell electronic gadgetry would have us believe that the computer age will be a new era for scientific thought and humanity; they might also point out the basic problem, which lies in the construction of models.
>
> – Rene Thom [88]

References

1. Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs*, pp. 242-292, MIT Press, Boston, MA, 1985.
2. Adiba, M. and N.B. Quang, "Historical Multimedia Databases," *Proc. Twelfth Intnl. Conf. on Very Large Data Bases*, pp. 63-70, Kyoto, Japan, 1986.
3. Allen, J.F., "Maintaining Knowledge about Temporal Intervals," *CACM*, vol. 26, no. 11, pp. 832-843, November 1983.
4. Allen, J.F., "Towards a General Theory of Action and Time," *Artificial Intelligence*, vol. 23, pp. 123-154, 1984.
5. Atzeni, P. and D.S. Parker, "Set Containment Inference and Syllogisms," Technical Report CSD-880022 , UCLA Computer Science Dept., June 1987, issued March 1988. To appear, *Theoretical Computer Science*
6. Bancilhon, F., T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, a Powerful and Simple Database Language," *Proc. Thirteenth Intnl. Conf. on Very Large Data Bases*, Brighton, England, 1987.
7. Batory, D.S. and T.Y. Leung, "Implementation Concepts for an Extensible Data Model and Data Language," Tech. Report TR-86-24, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, 1986.
8. Batory, D.S., "A Molecular Database Systems Technology," Tech. Report TR-87-23, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, 1987.
9. Batory, D.S., T.Y. Leung, and T. Wise, "Implementation Concepts For an Extensible Data Model and Data Language," *ACM Trans. Database Systems*, to appear.
10. Becker, R.A. and J.M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, Inc., Belmont, CA, 1984.
11. Beeri, C., S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur, "Sets and Negation in a Logic Database Language (LDL1)," *Proc. Sixth ACM Symp. on Principles of Database Systems*, pp. 21-37, San Diego, March 1987.
12. Ben-Zvi, J., "The Time Relational Model," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1982.
13. Bennett, J.S., L. Creary, R. Engelmore, and R. Melosh, "SACON: A knowledge-based consultant for structural analysis," Technical Report HPP 78-23, Computer Science Dept., Stanford University, 1978.
14. Berson, S., E. de Souza e Silva, and R.R. Muntz, "An Object-Oriented Methodology for the Specification of Markov Models," Technical Report CSD-870030, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, July 1987.
15. Bic, L. and R.L. Hartmann, "AGM: A Dataflow Database Machine," Technical Report, Dept. of Information and Computer Science, Univ. of California at Irvine, February 1987.
16. Bobrow, D.G., "If Prolog is the Answer, What is the Question?," *Proc. Intnl. Conf. on Fifth Generation Computer Systems*, pp. 138-145, ICOT, Tokyo, November 1984.
17. Bocca, Jorge, "On the Evaluation Strategy of EDUCE," in *Proceedings SIGMOD 1986*, pp. 368-378, 1986.

18. Bolour, A., T.L. Anderson, L.J. Dekeyser, and H.K.T. Wong, "The Role of Time in Information Processing," *ACM SIGMOD Record*, vol. 12, no. 3, pp. 27-50, 1982.

19. Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, October 1981.

20. Brodie, Michael L. and Matthias Jarke, *On Integrating Logic Programming and Databases*, pp. 40-62, Computer Corporation of America, Cambridge, Massachusetts.

21. Bruynooghe, M., G. Janssens, A. Callebaut, and B. Demoen, "Abstract Interpretation: Towards the Global Optimization of Prolog Programs," *Proc. Fourth International Symposium on Logic Programming*, pp. 192-204, IEEE Computer Society, 1987.

22. Ceri, Stefano, Georg Gottlob, and Gio Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," in *Proceedings 2nd Expert Database Systems Conference*, pp. 141-153, 1986.

23. Chau, L., "Two-Level Query Optimization," Draft, UCLA Computer Science Dept., July 1987.

24. Chau, L., "Functional Grammars and Stream Pattern Matching," Draft, UCLA Computer Science Dept., March 1988.

25. Chou, C., "Relaxation Processes: Theory, Case Studies and Applications," Report CSD-860057 (M.S. Thesis), UCLA Computer Science Dept., 1986.

26. Clark, K. and S. Gregory, "Notes on the Implementation of PARLOG," *J. Logic Programming*, vol. 2, no. 1, pp. 17-42, 1985.

27. Clifford, J. and D.S. Warren, "Formal Semantics for Time in Databases," *ACM Transactions on Database Systems*, vol. 8, no. 2, pp. 214-254, June 1983.

28. Conery, J. S., "Object Oriented Programming with Horn Clause Logic," Draft, University of Oregon, July 1987.

29. Corella, F., "Semantic Retrieval and Levels of Abstraction," in *Expert Database Systems*, ed. L. Kerschberg, Benjamin-Cummings, New York, 1985.

30. Cousot, P. and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Conf. Rec. 4th ACM Symp. on Princ. of programming languages*, pp. 238-252, 1977.

31. Dantzig, G., "Mathematical Programming and Decision Making in a Technological Society," Tech. Report SOL 82-11, Systems Optimization Laboratory, Dept. of Operations Research, Stanford Univ., August 1982.

32. DeGroot, D. and G. Lindstrom, *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, 1986.

33. Dean, T.L. and D.V. McDermott, "Temporal Data Base Management," *Artificial Intelligence*, vol. 32, pp. 1-55, 1987.

34. Debray, S. K. and D. S. Warren, "Automatic Mode Inference for Prolog Programs," *Proc. Third International Symposium on Logic Programming*, pp. 78-88, IEEE Computer Society, 1986.

35. Dolk, D.R., "A Generalized Model Management System for Mathematical Programming," *ACM Trans. Math. Software*, vol. 12, no. 2, pp. 92-126, June 1986.

36. Gabriel, John, Tim Lindholm, E.L. Lusk, and R.A. Overbeek, "A Tutorial on the Warren Abstract Machine for Computational Logic," ANL-84-84, pp. 1-52, Argonne National Laboratory, Argonne, Illinois, June 1985.

37. Gale, W.A., "REX Review," in *Artificial Intelligence & Statistics*, ed. W.A. Gale, Addison-Wesley, 1986.

38. Golshani, F., "The Basis of a Dataflow Model for Query Processing," *Proc. Eighteenth HICSS*, Honolulu, January 1985.

39. Gregory, S., *Parallel Logic Programming in PARLOG: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1987.

40. Hebrail, G., "A Model of Summaries for Very Large Databases," *Proc. of the 3rd Int. Workshop on Statistical and Scientific Database Management* , Luxembourg, 1986.

41. Helm, R. and K. Marriott, "Declarative Graphics," *Proc. Third Intnl. Conf. on Logic Programming*, pp. 512-527, Springer-Verlag, London, 1986.

42. Hoffmann, C.M. and J.E. Hopcroft, "Simulation of Physical Systems from Geometric Models," Preprint, Dept. of Computer Science, Cornell University, Ithaca, NY, 1986.

43. Huntsinger, R., "On Constraint-Oriented Environments for Continuous System Simulation," CSD-880020, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

44. Huntsinger, R., "Representation Transformation in Constraint Satisfaction Systems," CSD-880018, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

45. Imielinski, T., "Intelligent Query Answering in Rule Based Systems," *J. Logic Programming*, vol. 4, no. 3, September 1987.

46. Jarke, Matthias, Jim Clifford, and Yannis Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System," in *Proceedings ACM SIGMOD*, pp. 296-306, 1984.

47. Joyce, J., G. Lomow, K. Slind, and B. Unger, "Monitoring Distributed Systems," *ACM Trans. Computer Systems*, vol. 5, no. 2, pp. 121-150, May 1987.

48. Kahn, K., "A Primitive for the Control of Logic Programs," *Proc. Symp. on Logic Programming*, pp. 242-251, IEEE Computer Society, Atlantic City, 1984.

49. Kowalski, R.A., "Database Updates in the Event Calculus," Research Report 86/12, Department of Computing, Imperial College, London, 1986.

50. Kurose, J.F., K.J. Gordon, R.F. Gordon, E.A. MacNair, and P.D. Welch, "A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queueing Package (RESQ)," *Proc. Fall Joint Computer Conference*, pp. 709-718, IEEE Computer Society #743, 1986.

51. LeDoux, C.H., "A Knowledge-Based System for Debugging Concurrent Software," Technical Report CSD-860060 (Ph.D. Dissertation), UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1986.

52. Lee, R.M., H. Coelho, and J.C. Cotta, "Temporal Inferencing On Administrative Databases," *Information Systems*, vol. 10, no. 2, pp. 197-206, 1985.

53. Li, P-Y.P. and A.J. Martin, "The Sync Model: A Parallel Execution Method for Logic Programming," *Proc. Symp. on Logic Programming*, pp. 223-234, IEEE Computer Society, Salt Lake City, 1986.

54. Lindstrom, G. and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. Symp. on Logic Programming*, pp. 168-176, IEEE Computer Society, Atlantic City, 1984.

55. Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, "Designing DBMS Support for the Temporal Dimension," *Proc. ACM SIGMOD Conference on Management of Data*, pp. 115-130, June 1984.

56. Mackinlay, J., "Automating the Design of Graphical Presentations of Relational Information," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 110-141, April 1986.

57. McCrosky, C.D., J.J. Glasgow, and M.A. Jenkins, "Nial: A Candidate Language for Fifth Generation Computer Systems," *Proc. ACM'84 Annual Conference*, pp. 157-166, San Francisco, October 1984.

58. Melamed, B., "The Performance Analysis Workbench: An Interactive Animated Simulation Package for Queueing Networks," *Proc. Fall Joint Computer Conference*, pp. 729-740, IEEE Computer Society #743, 1986.

59. Mellish, C. S., "Abstract Interpretation of Prolog Programs," *Proc. Third Intnl. Conf. on Logic Programming*, pp. 463-474, Springer-Verlag, London, 1986.

60. More, T., "Axioms and Theorems for a Theory of Arrays," *IBM J. Res. Develop*, vol. 17, no. 2, pp. 135-175, 1973.

61. More, T., "The Nested Rectangular Array as a Model of Data," *Proc. APL79*, pp. 55-73, May 1979.

62. Naish, L., "All Solutions Predicates in Prolog," *Proc. Symp. on Logic Programming*, pp. 73-77, IEEE Computer Society, Boston, 1985.

63. Narain, S., "LOG(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation," Technical Report CSD-870027, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

64. Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1988.

65. Page, T.W., "Prolog Basis for a Data-Intensive Modeling Environment," Dissertation Prospectus, UCLA Computer Science Department, Los Angeles, CA 90024-1596, March 1988.

66. Parker, D.S., M. Carey, F. Golshani, M. Jarke, E. Sciore, and A. Walker, "Logic Programming and Databases," in *Proceedings First International Workshop on Expert Database Systems*, Kiawah Island, SC, October 1984. (also in *Expert Database Systems*, L. Kershberg, ed. 1985).

67. Parker, D.S. and M. Matsuo, "Incompleteness in Conceptual Modeling," *Proc. Advanced Database Symposium*, Tokyo, August 1986.

68. Parker, D.S., "The Modulus Knowledge Representation System," Draft, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

69. Parker, D.S., "Partial Order Programming," Technical Report CSD-870067, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

70. Parker, D.S. and R.R. Muntz, "A Theory of Directed Logic Programs and Streams," Technical Report CSD-880031, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, April 1988.

71. Parker, D.S., R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," Technical Report CSD-880025, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

72. Parker, D.S., T. Page, and R.R. Muntz, "Improving Clause Access in Prolog," Technical Report CSD-880024, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

73. Sadri, F., "Three Recent Approaches to Temporal Reasoning," Research Report 86/23, Department of Computing, Imperial College, London, Nov. 1986.

74. Sagie, I., "Computer-Aided Modeling and Planning (CAMP)," *ACM Trans. Math. Software*, vol. 12, no. 3, pp. 225-248, Sept. 1986.

75. Scheifler, R.W. and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986.

76. Sciore, E. and D.S. Warren, "Towards an Integrated Database-Prolog System," in *Expert Database Systems*, ed. Larry Kerschberg, pp. 293-305, Benjamin/Cummings, Menlo Park, CA, 1986.

77. Segev, A. and A. Shoshani, "Logical Modelling of Temporal Data," Tech. Rep. LBL-22636, Computer Science Research Department, Lawrence Berkeley Laboratory, Mar. 1987.

78. Shapiro, E.Y., *Concurrent Prolog: Collected Papers*, MIT Press, Cambridge, MA, 1987.

79. Shoshani, A. and K. Kawagoe, "Temporal Data Management," *Proc. Twelfth Intnl. Conf. on Very Large Databases*, pp. 79-88, Kyoto, Japan, August 1986.

80. Shum, C-D. and R. Muntz, "Implicit Representation of Extensional Answers," *Proc. on 2nd International Conference on Expert Database Systems*, 1988.

81. Shum, C-D. and R. Muntz, "An Information-Theoretical Study on Aggregate Responses," Technical Report , UCLA Computer Science Dept., Los Angeles, CA 90024-1596 , 1988.

82. Sinclair, J.B. and S. Madala, "A Graphical Interface for Specification of Extended Queueing Network Models," *Proc. Fall Joint Computer Conference*, pp. 709-718, IEEE Computer Society #743, 1986.

83. Smith, J.M., "Expert Database Systems: A Database Perspective," in *Proceedings First Int. Workshop on Expert Database Systems*, 1984.

84. Snodgrass, R., "The Temporal Query Language TQuel," *ACM Transactions on Database Systems*, vol. 12, no. 2, pp. 247-298, June 1987.

85. Studer, R., "Modeling Time Aspects of Information Systems," *Proc. Second Intnl. Conf. on Data Engineering*, Los Angeles, CA, 1986.

86. Subrahmanyam, P.A. and J-H. You, "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming," *Proc. Symp. on Logic Programming*, pp. 144-153, IEEE Computer Society, Atlantic City, 1984.

87. Takeuchi, A., "Affinity between Meta Interpreters and Partial Evaluation," Technical Report TR-166, ICOT, Tokyo, April 1986.

88. Thom, R., *Structural Stability and Morphogenesis*, Wiley, 1975.

89. Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data Language," *Proc. Twelfth Intnl. Conf. on Very Large Data Bases*, pp. 33-41, Kyoto, Japan, 1986.

90. Ullman, J., "Implementation of Logical Query Languages for Databases," in *Proceedings ACM SIGMOD Conference on Management of Data*, 1985.

91. Warren, David H.D., "An Abstract Prolog Instruction Set," Technical Report 309, SRI International, Menlo Park, CA 94025, October 1983.

92. Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects," in *Proceedings Very Large Data Bases*, pp. 458-469, Stockholm, Sweden, 1985.

93. Zaniolo, C., "Prolog — A Database Query Language for All Seasons," in *Expert Database Systems*, ed. Larry Kerschberg, pp. 219-232, Benjamin/Cummings, Menlo Park, CA, 1986.

94. Zaniolo, C., "Safety and Compilation of Non-Recursive Horn Clauses," *Expert Database Systems*, pp. 167-178, April, 1986.

Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

THE TANGRAM PROJECT: PUBLICATIONS 1987-88

Richard R. Muntz
D. Stott Parker
Gerald J. Popek

January 1989
CSD-890003

The Tangram Project: Publications 1987-88

Richard R. Muntz
D. Stott Parker
Gerald J. Popek

Computer Science Department
University of California
Los Angeles, CA 90024-1596

The Tangram Project at UCLA is aimed at the development of an environment for modeling of dynamic systems. It is an integration of DBMS and KBMS technology with distributed processing techniques. It is supported by DARPA, as contract F29601-87-C-0072. This is a summary of technical reports issued over the first year of the project,

Table of Contents

1. OVERVIEW 2
2. STREAM DATABASE PROCESSING 3
3. LANGUAGE SUPPORT 10
4. COMPUTER SYSTEM PERFORMANCE MODELING 14
5. CONSTRAINT-BASED MODELING 18

1. OVERVIEW

TANGRAM: PROJECT OVERVIEW
Richard R. Muntz, D. Stott Parker
CSD-880032 (39pp.)
April 1988

Today, most computers are used for the modeling of real-world systems. Demands on the extent and quality of the modeling are growing rapidly. There is an ever-increasing need for environments in which one can construct and evaluate complex models both quickly and accurately.

Successful modeling environments will require a cross-disciplinary combination of different technologies:

System modeling tools
Database management
Knowledge base management
Distributed computing

None of these technologies by itself provides all that is needed. A modeling environment must offer high-speed retrieval and exploration of knowledge about systems, as well as integration of diverse information sources with existing modeling tools.

Tangram is a distributed modeling environment being developed at UCLA. It is an innovative Prolog-based combination of DBMS and KBMS technology with access to a variety of modeling tools.

---

2. STREAM DATABASE PROCESSING

THE TANGRAM STREAM QUERY PROCESSING SYSTEM
D. Stott Parker, Richard R. Muntz, Lewis Chau
CSD-880025 (28pp.)
March 1988

Tangram is an environment for modeling. It supports development and management of models, simulation of models, analysis of simulation output and analysis of models in general. Its current focus is on computer system performance modeling.

Modeling applications routinely generate large quantities of simulation data, and analysis of this data requires a system that differs in significant ways from existing database systems. The data often takes the form of time series, and therefore query processing requires both stream processing techniques and heavy numerical computations (e.g., basic statistical and time series analysis) beyond ordinary aggregates.

One of the driving concepts behind Tangram has therefore been the combination of large-scale data access and data reduction with a powerful programming environment. The Tangram environment is based on Prolog, extending it with a number of features, including process management, distributed database access, and generalized stream processing.

This paper describes the *Tangram Stream Processor (TSP)*, the part of the Tangram environment performing query processing on large streams of data. The paradigm of transducers on streams is used throughout this system, providing a *'database-flow' (database dataflow) computation capability.*

shorter version in *Proceedings of the Sixth International Conf. on Data Engineering,* Los Angeles, CA, February, 1989.

A THEORY OF DIRECTED LOGIC PROGRAMS AND STREAMS
D. Stott Parker, Richard R. Muntz
CSD-880031 (31pp.)
April 1988

For some time it has been recognized that logic programmers commonly write *directed predicates,* i.e., predicates supporting only certain input and output patterns among their arguments. In many logic programming implementations,

programmers are encouraged to use 'mode declarations' to announce this directedness, both as a matter of style and as a directive for compiler optimization.

A common application of directed programming is stream or list processing. Programs that operate on streams or lists usually have specific input and output arguments. More generally, directed predicates can represent functions, with specific inputs and outputs.

We present a new declarative formalism for directedness in logic programming systems. The formalism is based on the use of partial ordering constraints rather than unification. Semantics of the resulting system are rigorously definable, and extend ordinary logic program semantics in a natural way.

The approach to directed logic programs presented here will probably provide higher performance than is possible with undirected programs. Furthermore, the approach provides perspective relating diverse concepts such as predicate 'modes', functional computation, constraint processing, and stream processing.

shorter version in R.A. Kowalski and K.A. Bowen (eds.), *Logic Programming,* MIT Press, 1988, pp. 620-650.

IMPLICIT REPRESENTATION FOR EXTENSIONAL ANSWERS
Chung-Dak Shum, Richard Muntz
CSD-880067 (17pp.)
August 1988

An exhaustive list of atomic objects is not always the best means of information exchange. This paper concerns the implicit representation of extensional answers. Expressions for answers are given in terms of concepts and individuals. Exceptions within individual concepts are allowed.

Two criteria are defined as measures of the *goodness* of such expressions: (i) minimizing the number of terms; (ii) positive terms preferred over negative terms. Expressions satisfying these two criteria are called optimal expressions. It is shown that under a strict taxonomy of concepts, any two optimal expressions for an extensional answer share the same set of terms. The inductive proof elicits an algorithm for obtaining such expressions.

Generalizing the strict taxonomy of concepts to a join-semilattice of concepts eliminates the term uniqueness property and also makes the problem of finding an optimal expression intractable. The problem under multiple taxonomies, although it involves a restricted type of join-semilattice, remains intractable.

in L. Kerschberg (ed.),*Expert Database Systems*, Benjamin Cummings, 1989, pp. 497-522.

AN INFORMATION-THEORETIC STUDY ON AGGREGATE RESPONSES
Chung-Dak Shum, Richard Muntz
CSD-880068 (12pp.)
August 1988

An enumeration of individual objects is not always the best means of information exchange. This paper concerns the problem of providing aggregate responses to database queries.

An aggregate response is an expression whose terms are quantified concepts. The tradeoff between the conciseness and preciseness of an aggregate response is studied. Conciseness is measured by the length (the number of terms) of an expression, and preciseness is measured by the entropy or the amount of uncertainty associated with the expression. For a given length, an expression with the minimum amount of entropy is called optimal.

Under a one-level taxonomy with the same cardinalities for all leaf concepts, the problem of finding an optimal expression can be solved inexpensively. An efficient heuristic is also proposed for the general one-level taxonomy. For a taxonomy of more than one level, an efficient heuristic is suggested which experiments indicate yields good solutions.

in *Proc. International Conf. on Very Large Databases,* Los Angeles, CA, August 29-September 1, pp. 479-490, 1989.

ASPEN: A STREAM PROCESSING ENVIRONMENT
Brian K. Livesey, Richard R. Muntz
CSD-880080 (26pp.)
October 1988

In this paper, we describe ASPEN, a concurrent stream processing system. ASPEN is novel in that it provides a programming model in which programmers use simple annotations to exploit varying degrees and types of concurrency. The degree of concurrency to be exploited is not fixed by the program specification or by the underlying system. Increasing or decreasing the degree of concurrency to be exploited during execution does not require rewriting the entire program, but rather, simply re-annotating it.

Examples are given to illustrate the varying types of concurrency inherent in programs written within the stream processing paradigm. We show how programs may be annotated to exploit these varying degrees of concurrency. We briefly describe our implementation of ASPEN.

ASPEN: A STREAM PROCESSING ENVIRONMENT
Brian K. Livesey
CSD-880098 (120pp.)
December 1988

Stream processing is an ideal paradigm for data-intensive applications. The solutions to a rich and varied set of problems that are, at best, awkward to express in other paradigms, can be expressed elegantly within the stream processing paradigm. Furthermore, stream processing presents an execution model in which such problems can be solved efficiently.

This thesis describes ASPEN, a stream processing environment. A programming language called Log(F) is extended to make it an appropriate language for expressing stream processing programs. The thesis focuses on those extensions that provide support for concurrent processing and access to distributed data.

The approach is novel in that the programming model allows the determination of the granularity of concurrency to be separated from the actual coding of the program. The degree of concurrency to be exploited is not fixed by the program specification or by the underlying system. Simple annotations allow the programmer to specify varying degrees of concurrency. Increasing or decreasing the degree of concurrency exploited during execution does not require rewriting the entire program, but rather, simply re-annotating it.

Several examples are given to illustrate the varying types of concurrency inherent in programs written within the stream processing paradigm. Examples are given which demonstrate how programs may be annotated to exploit these varying types and degrees of concurrency. The implementation of ASPEN is also described.

STREAM DATA ANALYSIS IN PROLOG
D. Stott Parker
CSD-890004 (54pp.)
January 1989

Today many applications routinely generate large quantities of data. The data often takes the form of a time series, or more generally just a *stream* – an ordered sequence of records. Analysis of this data requires stream processing techniques, which differ in significant ways from what current database query languages and statistical analysis tools support today. There is a real need for

better stream data analysis systems.

Stream analysis, like most data analysis, is best done in a way that permits interactive exploration. It must support 'ad hoc' queries by a user, and these queries should be easy to formulate and run. It seems then that stream data analysis is best done in some kind of powerful programming environment.

A natural approach here is to analyze data with the stream processing paradigm of transducers (functional transformations) on streams. Data analyzers can be composed from collections of functional operators (transducers) that transform input data streams to output streams. A modular, extensible, easy-to-use library of transducers can be combined in arbitrary ways to answer stream data analysis queries of interest.

Prolog offers an excellent start for an interactive data analysis programming environment. However most Prolog systems have limitations that make development of real stream data analysis applications challenging.

We describe an approach for doing stream data analysis that has been taken in the Tangram project at UCLA. Transducers are implemented not directly in Prolog, but in a functional language called Log(F) that can be translated to Prolog. With Log(F), stream processing programs are straightforward to develop. A by-product of this approach is a practical way to interface Prolog and database systems.

STREAM PROCESSING: AN EFFECTIVE WAY TO INTEGRATE AI AND DBMS
D. Stott Parker
CSD-890005 (11pp.)
January 1989

We present a novel approach for integrating AI systems with DBMS. The 'impedance mismatch' that has made this integration a problem is, in essence, a difference in the two systems' models of data processing. Our approach is to avoid the mismatch by forcing both AI systems and DBMS into the common model of *stream processing.*

By a *stream* here we mean an ordered sequence of data items. Stream processing is a well-known AI programming paradigm in which functional operators (which we call 'transducers') are combined to obtain arbitrary mappings from streams to streams. The stream processing paradigm can be, and has been, applied equally well as an AI programming model and as a query processing model in databases.

We argue first that, in practice, the relational model of data is actually the stream

model. The pure relational model cannot capture important aspects of relational databases such as column ordering, duplicate tuples, tuple ordering, and access paths, while the stream model does so naturally.

We then describe the approach taken in the *Tangram* project at UCLA, which integrates Prolog with relational DBMS. Prolog is extended to a functional language called Log(F) that facilitates development of stream processing programs. The integration of this system with DBMS is simultaneously elegant, easy to use, and relatively efficient.

shorter version in *Proceedings of the Sixth International Conf. on Data Engineering,* Los Angeles, CA, February, 1989.

STATISTICAL RULES: A NOTION OF DATABASE ABSTRACT AND ITS ROLE IN QUERY PROCESSING
Chung-Dak Shum, Richard Muntz
CSD-890007 (25pp.)
January 1989

A database instance is not an arbitrary collection of data, but rather many correlations exist among data items. The notion of *statistical rules* is introduced as a means of expressing such relationships. We demonstrate that statistical rules can be utilized in the query optimization process. In selectivity factor estimation, for example, statistical rules can actually be used to introduce relevant attributes the same manner as exact rules in semantic query optimization. Other uses of statistical rules include the enhancement of parallelism in database machines, and providing incomplete/quick answers as well as more informative responses.

We quantify the notion of how to measure the "inexactness" of a statistical rule using an entropy measure. The lower the entropy or uncertainty of a rule, the better the rule is. Based on such a measure, we show that constructing statistical rules using a "greedy" algorithm will result in a reasonable, although perhaps not optimal rule.

3-WAY HASH JOIN QUERY PROCESSING IN DISTRIBUTED RELATIONAL DATABASE SYSTEMS
Scott E. Spetka, Gerald J. Popek
CSD-890008 (17pp.)
January 1989

Initial distribution of relations as well as storage structures and organization have

an important impact on performance and the appropriate choice of processing techniques for database operations. Consideration of data distribution for partitioned relations used in hash join processing lead us to experiment with a new algorithm for processing 3-way join queries in a distributed system.

Database cacheing is also important for performance of distributed database management systems. An important goal is to provide an algorithm that can complement existing algorithms to provide sufficient generality to operate in a network transparent environment where the location of available resources may be changing, and to use those resources effectively. We present a new algorithm for processing 3-way join queries that can take advantage of cacheing by providing improved performance when data is not ideally distributed for some other algorithms.

3. LANGUAGE SUPPORT

LOG(F): A NEW SCHEME FOR INTEGRATING REWRITE RULES, LOGIC PROGRAMMING AND LAZY EVALUATION
Sanjai Narain
CSD-870027 (18pp.)
July 1987

We present LOG(F), a new scheme for integrating rewrite rules, logic programming, and lazy evaluation. First, we develop a simple yet expressive rewrite rules system F* for representing functions. F* is non-Noetherian, i.e., an F* program *can* admit infinite reductions. For this system, we develop a reduction strategy called *select* and show that it possesses the property of reduction-completeness. Because of this property, *select* exhibits a weak form of lazy evaluation.

We then show how to implement F* in Prolog. Specifically, we compile rewrite rules of F* into Prolog clauses in such a way that when Prolog intereprets these clauses, it directly simulates the behavior of *select*. Since it is not necessary to change Prolog, it is possible to do lazy evaluation efficiently. Since Prolog is already a logic programming system, a combination of rewrite rules, logic programming and lazy evaluation is achieved.

IMPROVING CLAUSE ACCESS IN PROLOG
D. Stott Parker, Thomas W. Page, Richard Muntz
CSD-880024 (7pp.)
March 1988

One of the weakest aspects of Prolog is in its access to clauses. This weakness is lamentable as it makes one of Prolog's greatest strengths, its ability to treat programs as data and data as programs, difficult to exploit. This paper proposes modifications to Prolog and shows how they circumvent important problems in Prolog programming in a practical way. For example, the proposed modifications permit Prolog programs that perform efficient database query (join) processing, coroutining, and abstract machine interpretation. These modifications have been used successfully at UCLA, and should be easy to implement within any existing Prolog system.

LOG(F): AN OPTIMAL COMBINATION OF LOGIC PROGRAMMING, REWRITING, AND LAZY EVALUATION
Sanjai Narain
CSD-880040 (176pp.)
June 1988

A new ap; ;h for combining logic programming, rewriting, and lazy evaluation is described. It rests upon *subsuming* within logic programming, instead of upon *extending* it with, rewriting, and lazy evaluation.

A non-terminating, non-deterministic rewrite rule system, F* and a reduction strategy for it, select, are defined. F* is shown to be reduction-complete in that select simplifies terms whenever possible. A class of F* programs called Deterministic F* is defined and shown to satisfy confluence, directedness, and minimality. Confluence ensures that every term can be simplified in at most one way. Directedness eliminates searching in simplification of terms. Minimality ensures that select simplifies terms in a minimum number of steps. Completeness and minimality enable select to exhibit, respectively, weak and strong forms of laziness.

F* can be compiled into Horn clauses in such a way that when SLD-resolution interprets these, it directly simulates the behavior of select. Thus, SLD-resolution is made to exhibit laziness. LOG(F) is defined to be a logic programming system augmented with an F* compiler, and the equality axiom X=X. LOG(F) can be used to do lazy functional programming in logic, implement useful cases of the rule of substitution of equals for equals, and obtain a new proof of confluence for combinatory logic.

EXECUTABLE TEMPORAL SPECIFICATIONS WITH FUNCTIONAL GRAMMARS
H. Lewis Chau, D. Stott Parker
CSD-880046 (20pp.)
June 1988

The *Stream Pattern Analyzer (SPA)* is one part of the Tangram Stream Query Processing System being developed at UCLA. It uses *functional grammars* to specify pattern analysis for streams of data.

Parallel execution events in a distributed system may be captured in an event stream for analysis. Given a set of functional grammar rules, SPA can analyze arbitrarily complex behavior patterns in this stream. At the same time a SPA grammar can act as a declarative specification of valid event histories.

We define a simple but powerful scheme that coroutines recognition of multiple patterns in an event stream. Propositional temporal logic queries can be

expressed in SPA in terms of predefined temporal operators such as *eventually, implies, not_until,* etc. Thus complex history-oriented specifications can be developed easily.

Functional grammar rules by themselves act as pattern generators or specifiers, and can be used to develop parsers by compilation to Log(F). Log(F) is a combination of Prolog and a functional language called F*. We describe a simple algorithm to compile functional grammars to Log(F), and prove its correctness.

PX REFERENCE MANUAL, VERSION 0.2
Ted Kim
CSD-880079 (47pp.)
October 1988

This manual describes an interface to the X Window System for Prolog. The X Window System is a network-based window system providing a desktop style of user interface and graphics. PX provides a low level interface to X for Prolog similar to that provided by "Xlib" for the C language. PX is designed for use with version 11 of the X Window System. Higher level interfaces (such as toolkits) are built on top of this one and are outside the scope of this document.

PX is implemented in the C language using the C language foreign function interface from Quintus Prolog. Almost any Prolog which supports the Quintus style interface can use this package with few restrictions. In particular, SICStus Prolog was used in the development of this system. This document is a reference manual. As such, it is not a tutorial or user's guide to X or Prolog.

FUNCTIONAL LOGIC GRAMMAR:
A NEW SCHEME FOR LANGUAGE ANALYSIS
H. Lewis Chau, D. Stott Parker
CSD-880097 (16pp.)
December 1988

We present a new kind of grammar. It combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as Definite Clause Grammar (DCG). We call it *Functional Logic Grammar.*

A functional logic grammar is a finite set of rewrite rules. It is efficiently executable, like most logic grammars. In fact, functional logic grammar rules can be compiled to Prolog and executed by existing Prolog interpreters as generators or acceptors. Unlike most logic grammars, functional logic grammar also permits

higher-order specification and modular composition.

This paper defines functional logic grammar and compares it with the successful and widely-used DCG formalism in logic programming. We show that pure DCG can be easily translated into functional logic grammar. Functional logic grammar enjoys the advantages of DCG, as well as its first-order logic foundation. At the same time, functional logic grammar ranks higher in aspects such as expressiveness and modularity, and permits lazy evaluation.

4. COMPUTER SYSTEM PERFORMANCE MODELING

A NOTE ON THE COMPUTATIONAL COST OF THE LINEARIZER ALGORITHM FOR QUEUEING NETWORKS
Edmundo de Souza e Silva and Richard R. Muntz
CSD-870025 (15pp.)
July 1987; revised February 1988

Linearizer is one of the best known approximation algorithms for obtaining numeric solutions for product form queueing networks. In the original exposition of Linearizer, the computational cost was stated to be $O(MK^3)$ for a model with M queues and K job classes. We show in this note that with some straightforward algebraic manipulation Linearizer can be modified to require only $O(MK^2)$ computational cost.

We also discuss the space requirements for Linearizer and show that the space can be reduced to $0(MK)$ but with some increased computational cost.

To appear, *IEEE Transactions on Computers*, 1989.

AN OBJECT ORIENTED METHODOLOGY FOR THE SPECIFICATION OF MARKOV MODELS
Steven Berson, Edmundo Silva, Richard Muntz
CSD-870030 (23pp.)
July 1987

Modelers wish to specify their models in a symbolic, high level language while analytic techniques require a low level, numerical representation. The translation between these description levels is a major problem.

We describe a simple, but surprisingly powerful approach to specifying system level models based on an object oriented paradigm. This basic approach will be shown to have significant advantages in that it provides the basis for modular, extensible modeling tools. With this methodology, modeling tools can be quickly and easily tailored to particular application domains. An implementation in Prolog, of a system based on this methodology and some example applications are given.

ANALYTIC MODELING METHODOLOGY FOR EVALUATING THE PERFORMANCE OF DISTRIBUTED, MULTIPLE-COMPUTER SYSTEMS
Alex Kapelnikov
CSD-870061 (201pp.)
November 1987

In this dissertation, we describe an analytic modeling methodolcgy for evaluating the performance of distributed, multiple-computer systems. The concepts and techniques of this methodology are useful for the approximate analysis of a wide range of distributed computing environments and communication networks. The main strategy of our approach is to segregate, as much as possible, the model of the "logical" behavior of an application (a program or a process) from the model of its underlying execution environment. For representing program behavior, graph-based techniques are used, while extended queueing networks are utilized for modeling system architectures. The solutions of both types of models are combined to estimate the performance of a distributed system in executing some selected applications.

To illustrate the practical application of the methodology introduced in this dissertation and provide an indication of its expected accuracy level, we have included two case studies.

A MODELING METHODOLOGY FOR THE ANALYSIS OF CONCURRENT SYSTEMS AND COMPUTATIONS
Alex Kapelnikov, Richard R. Muntz, and Milos D. Ercegovac
in M.H. Barton, E.L. Dagless, G.L. Reijns (eds.), *Distributed Processing,* Elsevier Science Publishers, pp. 465-479, 1988.

In this paper, we describe a novel modeling methodology for evaluating the performance of distributed, multiple-computer systems. Our approach employs a set of analytic tools to obtain an estimate of the average execution time of a parallel implementation of a program (or transaction) in a distributed environment. These tools are based on an amalgamation of queueing network theory and graph models of program behavior. Hierarchical application of heuristic optimization techniques facilitates the analysis of large and complex programs. A realistic example is used to illustrate the practical application of our methodology.

A DISTRIBUTED ALGORITHM TO DETECT A GLOBAL STATE OF A DISTRIBUTED SIMULATION SYSTEM
Behrokh Samadi, Richard R. Muntz, D. Stott Parker
in M.H. Barton, E.L. Dagless, G.L. Reijns (eds.), *Distributed Processing,* Elsevier Science Publishers, pp. 19-34, 1988.

In this paper, we describe a novel modeling methodology for evaluating the performance of distributed, multiple-computer systems. Our approach employs a set of analytic tools to obtain an estimate of the average execution time of a parallel implementation of a program (or transaction) in a distributed environment. These tools are based on an amalgamation of queueing network theory and graph models of program behavior. Hierarchical application of heuristic optimization techniques facilitates the analysis of large and complex programs. A realistic example is used to illustrate the practical application of our methodology.

DISTRIBUTED SHARED MEMORY IN A LOOSELY COUPLED DISTRIBUTED SYSTEM (EXTENDED ABSTRACT)
Brett D. Fleisch
in *Proceedings COMPCON Spring 88*, San Francisco, CA, February-March 1988, pp.182-184.

In this work we describe new implementation experiences with a distributed shared memory system implemented in a loosely coupled distributed system. Our goal was to investigate the feasibility of distributed shared memory (dsm) in an operating system kernel. Li (1986) demonstrated the feasibility of such a system outside of the kernel with a number of numeric applications, but it remained a relatively open question as to how well dsm performs for a variety of non-numeric applications and what the effects of dsm are on other kernel services. The organization of dsm we describe resembles a cross-processor segmented paging system. Our talk relates implementation experiences and preliminary performance results. We plan to report results from experiments with symbolic computation, which emphasizes rearrangement of data, where often the sequence of operations is highly data dependent and less amenable to compile time analysis than numerical computation. One general goal of this work is to describe a set of software primitives and to identify hardware features that can be used to support the conversion of applications from nondistributed shared memory to distributed shared memory. These features may include hints, user advice, control primitives, and architectural modifications that will improve functionality and performance.

BOUNDING AVAILABILITY OF REPAIRABLE COMPUTER SYSTEMS
Richard Muntz, Edmundo Silva, A. Goyal
CSD-880070 (26pp.)
September 1988

Markov models are widely used for the analysis of availability of computer/communication systems. Realistic models often involve state space cardinalities that are so large that it is impractical to generate the transition rate matrix let alone solve for availability measures. Various state space reduction methods have been developed, particularly for transient analysis. In this paper we present an approximation technique for determining steady state availability. Of particular interest is that the method also provides bounds on the error. Examples are given to illustrate the method.

5. CONSTRAINT-BASED MODELING

SET CONTAINMENT INFERENCE AND SYLLOGISMS
Paolo Atzeni, D. Stott Parker
CSD-870022 (34pp.)
March 1987

Type hierarchies and type inclusion (*isa*) inference are now standard in many knowledge representation schemes. In this paper, we show how to determine consistency and inference for collections of statements of the form

mammal isa vertebrate.

These *containment* statements relate the contents of two sets (or types). The work here is new in permitting statements with negative information: disjointness of sets, or non-inclusion of sets. For example, we permit the following statements also:

mammal isa **non**(*reptile*)
non(*vertebrate*) *isa* **non**(*mammal*)
not(*reptile isa amphibian*)

Binary containment inference is the problem of determining the consequences of positive constraints P and negative constraints **not**(P) on sets, where positive constraints have the form P: $X \subseteq Y$. Negations of these constraints therefore have the form **not**(P): $X \cap$ **non**(Y) $\neq \emptyset$, so positive constraints assert containment relations among sets, and negative constraints assert that two sets have a non-empty intersection.

We show binary containment inference is solved by rules essentially equivalent to Aristotle's *Syllogisms.* Necessary and sufficient conditions for consistency, as well as sound and complete sets of inference rules, are presented for binary containment. The sets of inference rules are compact, and lead to polynomial-time inference algorithms, so permitting negative constraints does not result in intractability for this problem.

To appear, *Theoretical Computer Science,* 1988.

PARTIAL ORDER PROGRAMMING
D. Stott Parker
CSD-870067 (80pp.)
December 1987

We introduce a programming paradigm in which statements are constraints over partial orders. A *partial order programming problem* has the form

$$\begin{array}{ll} \text{minimize} & u \\ \text{subject to} & u_1 \sqsupseteq v_1, u_2 \sqsupseteq v_2, \cdots \end{array}$$

where u is the *goal,* and $u_1 \sqsupseteq v_1, u_2 \sqsupseteq v_2, \cdots$ is a collection of constraints called the *program.* A solution of the problem is a minimal value for u determined by values for u_1, v_1, etc. satisfying the constraints. The domain of values here is a *partial order,* a domain D with ordering relation $\sqsupseteq$.

The partial order programming paradigm has interesting properties:

(1) It generalizes mathematical programming, dynamic programming, and computer programming paradigms (logic, functional, and others) cleanly, and offers a foundation both for studying and combining paradigms.

(2) It takes thorough advantage of known results for continuous functionals on complete partial orders, when the constraints involve expressions using only continuous and monotone operators. These programs have an elegant semantics coinciding with recent results on the relaxation solution method for constraint problems.

(3) It presents a framework that may be effective in modeling of complex systems, and in knowledge representation for cognitive computation problems.

ON CONSTRAINT-ORIENTED ENVIRONMENTS FOR CONTINUOUS SYSTEMS SIMULATION
Richard A. Huntsinger
CSD-880018 (10pp.)
March 1988

Sets of simultaneous differential equations and sets of queries on those equations are naturally expressible as *constraint networks* in the *constraint satisfaction* modeling paradigm. Further, relaxation enhanced to exploit *typed valued* constraints provides a procedural semantics for such constraints which in the best case reduces to propagation, and in the worst case performs comparably to other paradigms. Accordingly, constraint satisfaction is advocated as the paradigm of choice on which to base continuous systems simulation environments.

Examples are presented illustrating constraint network characterizations of continuous systems models, and their corresponding procedural semantics.

REPRESENTATION TRANSFORMATION IN CONSTRAINT SATISFACTION SYSTEMS
Richard Huntsinger
CSD-880020 (11pp.)
March 1988

A practical class of constraint satisfaction systems operate on *relaxable representations* of the form $N = f(N)$, where N is a set of variables, and the declarative semantics is the set of instantiations of N which preserve the equality. In general, relaxation provides a complete procedural semantics for only a subset ρ of such representations. Of interest, then, is the set of *transformable* representations $\alpha \supset \rho$ in which for each representation $M_r \in \alpha$ there exists a determinable transformation $T: \alpha \rightarrow \rho$ such that the declarative semantics of M_r is identical to that of $T(M_r)$.

Relaxable representations for which $f(N)$ is a polynomial are transformable, each corresponding to a transform of the form $N = (f(N)N^n)^{1/(n+1)}$, where n is a function of the degree and coefficients of the polynomial. This observation provides some intuition about more general transformations, applicable to the implementation of powerful (complete over a superset of ρ) constraint satisfaction systems.

PARTIAL ORDER PROGRAMMING: EXTENDED ABSTRACT
D. Stott Parker
CSD-880086 (7pp.)
October 1988

We introduce a programming paradigm in which statements are constraints over partial orders. A *partial order programming problem* has the form

minimize	u
subject to	$u_1 \sqsupseteq v_1,\ u_2 \sqsupseteq v_2,\ \cdots$

where u is the *goal,* and $u_1 \sqsupseteq v_1,\ u_2 \sqsupseteq v_2,\ \cdots$ is a collection of constraints called the *program.* A solution of the problem is a minimal value for u determined by values for u_1, v_1, etc. satisfying the constraints. The domain of values here is a *partial order,* a domain D with ordering relation $\sqsupseteq$.

The partial order programming paradigm has interesting properties:

(1) It generalizes mathematical programming and also computer programming paradigms (logic, functional, and others) cleanly, and offers a foundation both for studying and combining paradigms.

(2) It takes thorough advantage of known results for continuous functionals on complete partial orders, when the constraints involve expressions using only continuous and monotone operators. The semantics of these programs coincide with recent results on the relaxation solution method for constraint problems.

(3) It presents a framework that may be effective in modeling, or knowledge representation, of complex systems.

in *Proceedings of the Sixteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* Austin, Texas, January 11-13, 1989.

OPTIMIZATION BY NON-DETERMINISTIC, LAZY REWRITING
Sanjai Narain
CSD-880092 (19pp.)
November 1988

Given a set S and a condition C we address the problem of determining which members of S satisfy C. One useful approach is to set up the generation of S as a tree, where each node represents a subset of S. If from the information available at a node, we can determine that no members of the subset it represents satisfy C, then the subtree rooted at it can be pruned, or not generated. Thus, large subsets of S can be quickly eliminated from consideration. We show how such a tree can be simulated by interpretation of non-deterministic rewrite rules, and its pruning simulated by lazy evaluation.