January 1995 Report No. STAN-CS-TR-95- 1538

# 1994 Publications Summary of the Stanford Database Group

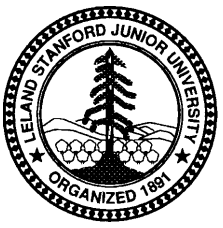
**by**

**Joachim Hammer**

**Department of Computer Science**

**Stanford University**

**Stanford, California 94305**



# 1994 Publications Summary for the Stanford Database Group

*Department of Computer Science*
*Stanford University*
*Stanford, CA 94305-2140 USA*

*Contact:*
*Marianne Siroker*
*e-mail:* siroker@db.stanford.edu
*phone:(415)723-0872*

This Technical Report contains the first four pages of papers written by members of the Stanford Database Group during 1994. We believe that the first four pages convey the main ideas behind each paper better than a simple title and abstract does.

Readers interested in a full paper can fetch an electronic copy via FTP over the Internet. The procedure is as follows:

(1) Call FTP by issuing the command ftp db.stanford.edu

(2) At the prompt for user name type anonymous

(3) At the prompt for password, type in your full name

(4) Go to the appropriate directory by typing cd directory-name. The directory name is listed in the box on top of page 1 of each paper. (For example, for the first paper, type cd /pub/adelberg/1994.)

(5) Get the file you want by typing get file-name. The file name is the last component of the file name given in the box on page 1 of each paper. (For example, for the first paper, type get priority1.ps.)

(6) Quit FTP by typing quit.

(7) In your current directory you now have a copy of the paper. To print it, send the file to a postscript printer. (For example, in UNIX, type lpr priority1.ps.)

We thank you for your interest in our work, and hope that you find it useful to browse through our publications in this fashion.

Sincerely,

*The Members of the Stanford Database Group*

Paper 1

| | |
|---|---|
| FILE: | /pub/adelberg/1994/priority1.ps |
| FILE: | /pub/adelberg/1994/priority2.ps |
| COMMENT : | Synopsis of short (conference version) appeared in RTSS 1994 |

# Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers *

Brad Adelberg† Hector Garcia-Molina‡ Ben Kao§

April 25, 1994

**Abstract**

Real-time scheduling algorithms are usually only available in the kernels of real-time operating systems, and not in more general purpose operating systems, like Unix. For some soft real-time problems, a traditional operating system may he the development platform of choice. This paper addresses method:, of emulating real-time scheduling algorithms on top of standard time-share schedulers. We examine (through simulations) three strategies for priority assignment within a traditional multi-tasking environment. The results show that t lie emulation algorit hms arc comparable in performance to the real-time algorithms *anti* in some instances outperform them.

Keywords: soft real-time, priority assignment, schedu ling.

## 1 Introduction

In this paper we focus on "soft real-time" applications. which have the following characteristics:

- tasks have real-time deadlines:
- missing some task deadlines is acceptable:
- the goal is to minimize the number of missed deadlines:
- task arrival and system load are unpredictable.

A typical soft real-time application is telecommunications. Here a missed deadline might correspond to a dropped call. Another example is program trading for financial markets. Missing a. deadline will correspond to missing a trading opportunity. In both cases missed deadlines arc to he avoided but will not cause calamitous results. Real-time systems typically run on top of an operating system that provides basic services such as scheduling and memory management The choice of operating system for a. soft real-time operating system is an important issue. Previously, most researchers assumed a dedicated machine running a real-time operating system (RTOS). Now, however, there is interest in techniques for developing real-time applications on general purpose operating systems (GPOSs), like Unix. At least one implementation study [MT89] has demonstrated that using a GPOS eases an application's implementation and results in extremely high quality rode. This is especially promising since in practice. general purpose operating systems (GPOSs) are much more common than RTOSs.

While there will probably always be a need for RTOSs, there are at least three reasons to believe that GPOSs will become more popular for soft real-time applications:

- As real-time system design moves out of its closed community and into the general computing population, programmers will want to develop on the platforms which they're familiar with. Traditional operating systems. like Unix. have accumulated a large suite of development tools. In addition. programs written in a. GPOS are more portable than programs written for proprietary operating systems.
- Many applications will he split across the real-time/hatch processing boundary. For example, investment hankers may interface with a real-time system to monitor trading opportunities, while at the same time using spreadsheets, news readers, and other non-critical applications. Also, the inclusion of multimedia will introduce time constraints (20 rames/sec)in 6 otherwise non-real-time applications. Running all components of an application under a single OS will ease development as compared to a heterogeneous approach.
- Using only a. GPOS will reduce total system cost. Economy of scale has driven the price of a GPOS much lower than that, of a real-time kernel. Increased speed of development and rode quality will also reduce product expense.

This is why we believe that soft real-time processing must be intergrated into traditional OSs. In fart, the research we report on here was motivated by our implementation of a real-time database at Stanford [AGMK94]. We do not have the resources to purchase a real-time OS. nor the staff to maintain it.. Hence, we are implementing our database system on a conventional Unix system. HP-UX from Hewlet t Packard. We suspec t many other users of soft real-time systems will be in the same sit uation.

*This work was supported by the Telecommunications Center at Stanford University and by Hewlett Packard Company.

†Stanford University Department of Computer Science. e-mail: adelberg@cs.stanford.edu

‡Stanford University Department of Computer Science. e-mail: hector@cs.stanford.edu

§Princeton University Department of Computer Science. Current Address: Stanford University Department of Computer Science. e-mail: kao@cs.stanford.edu

In this paper we study the *real-time emulation* (RTE) problem, which we define as how to build soft real-time scheduling on top of a traditional OS. In Section 3, we look at. three approaches to RTE and settle on one: design an algorithm to assign a priority to a new process according to its real-time constraints in such a way that the priority scheduling done by the GPOS mimics that of a. real-time scheduler (such as earliest deadline first. least slack first). We call this type of algorithm a *priority assignment algorithm* because it must. use the real-time information about a task (i.e. deadline, slack, . . .) to determine which OS priority level to assign to it.

To illustrate the difficulties we will face in emulating a. real-time scheduler, suppose we have an OS with 5 priority levels, numbered 0 to 1, with level 0 having the highest priority. Initially, the system is idle, and a. task A arrives. What priority do we run it. at,? Well, suppose we run it at a middle priority, in this case 2. While A is running, task B arrives with an earlier deadline than A. Since the OS should schedule B first., we will assign it. a. priority of 1. Of course, if a new task arrives with a deadline in between those of A and B, we are stuck since there is no priority to assign to it. The algorithms we will present will have to cope with situations like this. We will also evaluate the performance of our new algorithms, and compare them against conventional real-time ones. As we will see, not only do the emulation algorithms perform well, in some cases they outperform the real-time algorithms.

The priority assignment algorithms which we develop will have applicability in other scenarios. For instance, designers trying to interface real-time systems to token-ring networks need to assign priorities. 'Tasks in the real-time system have deadlines associated with them, but the token-ring only supports message priorities, usually 8 levels. Somehow a message's priority must he assigned based on the deadline of the task that sends it,. This is similar to the RTE problem applied to earliest deadline first scheduling studied in this paper, although the token-ring problem requires extensions for distributed scheduling.

To study the RTE problem. we assume that, we have no apriori knowledge of real-time task arrival patterns or execution requirements. Given the application areas outlined above, we expert that little will he known about the real-time requests that will he made. Unlike hard real-time systems used in control applications. where tasks are periodic and of known execution time, our soft. real-time system will probably he used in less structured situations, with tasks being event driven and unpredictable. We assume a task receives its deadline just before it is submitted for execution.



The rest of this paper is organized as follows. In Section 2, we mention some related work. Next, in Section 3 we explore the possible approaches to the priority assignment problem and focus on one. Section 4 describes the logical base model for our study. Different priority assignment strategies are introduced in Section 5. A brief description of our simulation experiments is contained in Section 6. In Section 7 we display and analyze the results of our experiments. Finally. in Section 8 we present conclusions.

## 2 Related Work

A lot of research has been done on real-time scheduling in various environments. he it I/O scheduling. processor scheduling, or transaction scheduling [AGM90, AGM88a, AGM88b, LL73, HTT89, CW90]. Through this work. the behavior and properties of earliest arrival (EA) first. earliest deadline (ED) first, and least slack (LS) first have been delineated. These studies all assume an infinite range of priorities and a custom scheduler. The problem of scheduling with a limited number of priority levels was studied in [SLR86], hut centered on rate monotonic scheduling for periodic tasks in hard real-time systems.

Some researchers have studied how to develop real-time systems on traditional operating systems. [FPG+89], [Wel93], [Cra88], and [MT89] have all studied the suitability of Unix for real-time applications. [FPG+89] identifies two properties as essential for an operating system which is to support real-time applications: *Performance* and *Determinism*. The paper then shows that REAL/IX, a. fully preemptive Unix, compares favorably to a real-time OS based on the standards above. [MT89] studies a different. real-time Unix. RX-UX 832, and comes to similar conclusions. Finally, [Wel93] examines two other GPOSs, SCO XENIX System V and OS/2, and concludes that both may he viable for real-time applications. with OS/2 being particularly well suited due to its high predirta.hi1it.y. While these studies demonstrate that a GPOS ran be used in many real-time applications. none address the problem of priority assignment. It is implicitly assumed that process priorities ran he determined during the design of the application, probably by a variant of the general rate monotonic algorithm.

## 3 General Approaches

Our goal is to emulate a real-time scheduler on top of a GPOS scheduler. Solutions to this problem vary depending on the accuracy of the emulator that is desired, the amount of total coding complexity that is tolerable, and the distribution of new rods between applications and

| | |
|---|---|
| FILE: | /pub/adelberg/1994/updates1.ps |
| FILE: | /pub/adelberg/1994/updates2.ps |
| COMMENT: | Short version to appear in SIGMOD '95 |

Paper 2

# Applying Update Streams in a Soft Real-Time Database System

Brad Adelberg* Hector Garcia-Molinat Ben Kao‡

**Abstract**

Many papers have examined how to efficiently export a materialised view but to our knowledge none have studied how to efficiently import one. To import a view, i.e., to install a stream of updates, a real-time database system must process new updates in a timely fashion to keep the database "fresh," but at the same time must process transactions and ensure they meet their time constraints. In this paper, we discuss the various properties of updates and views (including staleness) that affect this tradeoff. We also examine, through simulation, four algorithms for scheduling transactions *and* installing updates in a soft real-time database.

Keywords: soft real-time, temporal databases, materialised views, updates.

## 1 Introduction

The problem we study in this paper arose during the on-going implementation of the STRIP real-time database system.[1] This system [AGMK94b] provides traditional database services (e.g., SQL, indexing, recovery) with real-time facilities (e.g., transaction deadlines, triggers). It is a soft real-time system where the exact resource requirements are not known in advance; the system minimizes the number of timing constraints violated (as opposed to a hard system that guarantees all are always met). An application we are targeting for is program trading, where financial instruments and currencies are examined to discover trading opportunities. A typical transaction might compare the price of German marks in London to the price in New York and if there is a significant difference, rapidly perform a trade.

The heart of such a real-time database system (RTDB) is the scheduler, which has to allocate scarce resources (mainly CPU cycles) to meet the time constrained service requests. All RTDB scheduling algorithms we are aware of (see [KGM93, Ram93] for RTDB overviews) assume that service requests exclusively come from transactions. Hence, the scheduling problem is simply to decide what transaction to run next.

However, we have discovered that in almost all RTDB applications there is another very significant component: managing the data input streams and applying the corresponding database updates. In our program trading application, consider that there are currently over three-hundred thousand financial instruments world-wide that could be tracked. The update stream can be up to 500 updates/second during peek time [CB94]. (The update streams are provided by several commercial companies such as Reuters.) In other applications, input streams report data from sensors (e.g., in an industrial control system) or service requests (e.g., call requests in a telecommunications system), or reflect the state of other databases.

The key issue is that these database updates should not be handled either as separate transactions with individual deadlines (overhead would be too great), or as a single transaction (it is a continuous process without a single deadline). Instead we will argue that they must be handled by a single process, and that the scheduler must *very* carefully balance the requirements of transactions and their deadlines, against the need to keep the part of the database that reflects the outside world up-to-date. If updates are executed with higher priority so as to maintain "external consistency," the system may be left with no time to meet transaction deadlines. On the other hand, if transactions are given preference, they may read stale data. Neither of these outcomes is desirable. In our example, missing deadlines represents missed opportunities, and operating on stale data means we may be making the wrong decisions.

Notice that our input stream management problem can be viewed as an instance of the *materialized view* problem. In other words, the portion of the real-time database that is being externally updated can be thought of as a materialized views of some external databases (in our example, the databases at the New York Stock Exchange, the Tokyo Exchange, and so on). Although the question of how to efficiently *export* a materialized view has been studied in great detail [Han87, LHM+86, RK86, SP89, ALSO, BLT86, SG90], the question of how to efficiently import a view has not been studied at all. It is usually assumed, often implicitly, that the problem of incorporating view updates is straightforward and that the resource demand it represents is trivial. We believe this is definitely not true, especially in real-time applications where the view importation must be done in a timely fashion.

Incidentally, one way to "solve" the problem we are addressing is to limit the size of the materialized view so that the number of updates per second is so low that the resource demands for importing them are indeed trivial. In our sample application, this would correspond to selecting in advance a small subset of financial instruments to track. While this may be acceptable in some cases, it also severely restricts the types of application processing that can be done (e.g., the trades one may consider). We believe it is much better to develop efficient mechanisms for installing large volumes of updates, without interfering with the deadlines of transactions.

The contributions of this paper are the following:

- We present RTDB scheduling algorithms that consider both the deadlines and value of transactions and the requirements of the update installation process As mentioned earlier, current algorithms only consider the former.
- We outline strategies for the update installation process. The strategies include, for instance, what order to install queued updates: and when (and how) to discard updates (because they are stale or there is a queue overflow).
- We propose metrics for evaluating system performance. Traditional metrics only focus on transactions (e.g., number of missed deadlines) Here we extend them to include, for example, data staleness, and the fraction of transactions that read stale data. We also propose several ways to measure data staleness.

*Stanford University Department of Electrical Engineering. e-mail: adelberg@cs.stanford.edu
†Stanford University Departments of Computer Science and Electrical Engineering.
‡Princeton University Department of Computer Science. Current Address: Stanford University Department of Computer Science.

[1] STRIP stands for STanford Real-time Information Processor.

- We give the results of a detailed performance evaluation that compares the various algorithms. These results make it possible to select good scheduling policies to implement in a system such as STRIP.

The rest of the paper is organized as follows. Section 2 outlines relevant properties of updates and further develops the notion of data timeliness. In Section 3, the conceptual model for this study is described. Four algorithms are then defined for transaction/update scheduling in Section 4. Section 5 describes the decisions made and the parameters used to implement the conceptual model. The results of the simulations are discussed in Section 6. Finally, Section 7 summarizes the contributions of this work and presents ideas for future study.

## 2 Properties of Updates and Views

Since a major concern of this paper is how to maintain external data consistency, we start by investigating properties of updates and views, including the notion of view staleness. In general, updates can be characterized by the following two properties:

- *Periodic* versus *aperiodic updates.* With periodic updates, the current value of a data object is provided at periodic intervals regardless of whether it has changed or not. Aperiodic updates do not occur at times predictable by the database system, and usually only occur when the value of the data object changes. In this paper we focus on aperiodic updates.
- *Complete versus partial updates.* In a complete update, the value of every attribute of an object is provided with each update, even if the value has not changed since a previous update. Partial updates provide only values for attributes that have changed since the previous update. In this paper we mainly focus on complete updates.

The database views (i.e., the database portion updated from external sources) may also have a variety of properties, including:

- *Snapshot* versus *historical views.* Snapshot views only maintain the value of an object at one point in time (typically the current time). Updating an object causes the previous attribute values to be lost forever. Historical views provide support for maintaining not only the current attribute values of an object, but its past values as well. In this paper we only consider snapshot views.
- *View complexity.* In some cases, the updates are applied to the view by simply storing the new values. In other cases, the update values must be transformed or combined with other values before being stored. For example, company names may have to be changed to match local conventions, and running averages may have to be computed. Hence, the cost of installing a single update can vary from say a single SQL command (if that is the database interface) to executing several commands and application programs.
- *View* staleness. The application semantics typically determine when view data is considered up-to-date or stale. There are actually several options for defining staleness.

One option is to define staleness based on the time when update values were generated. In this case, updates arrive at the RTDB with a timestamp giving the generation time at their external source. A database value is then considered stale if the difference between the current time and its generation timestamp is larger than some predefined maximum age $\Delta$.[2] We call this definition *Maximum Age (MA).* Notice that with *MA,* even if a view object does not change value, it must still be periodically updated, or else it will become stale. Thus, *MA* makes more sense in applications where all view data is periodically updated and/or where data that has not been recently updated is "suspect." For example, *MA* may be used in a plant control system where sensors generate updates on a regular basis, or in a military scenario where an aircraft position report is not very useful unless it is quite recent.

Another option is to be optimistic and assume that a data object is always fresh unless an update has been received by the system (put into the update queue) but not yet applied to the data. We will refer to this definition *as Unapplied Update* ($UU$). A problem with $UU$ is that it ignores delays that may occur before an update is handed to the RTDB system Hence, $UU$ is more useful in applications with fast and reliable delivery of updates. For example, $UU$ may be used in a telecommunications RTDB server because delays outside the system may be irrelevant and because we do not want the extra traffic associated with *MA.* (Example: if a call is on-going, we do not want to be periodically notified that it is still going on.)

There are several variations on these two basic definitions. For example, in the *MA* staleness definition we could replace generation time by arrival time at the RTDB. This would mean that updates for every database object should be received at least every A units. We could also combine *MA* and $UU$. Here, an object would be considered stale if it were stale under either definition. Due to space limitations, in this paper we only consider *MA* (generation time) and $UU$ (separate from *MA)* as the possible staleness criteria. Readers are referred to [SL92, KM93] for some other interesting discussion on data timeliness.

Once staleness is defined, we need to discuss what transactions do when they encounter stale data. One option is for them to abort. For example, in the program trading application it may be dangerous to commit a transaction that made its decisions based on out-of-date information. Another option is to complete such transactions, but to raise some warning indicator. For instance, in a plant control system, it may be better to operate with stale data than to do nothing at all, as long as a "red light" goes on in the control room. A third option is to complete transactions that read stale data, without any special action. In this last case, the notion of staleness is only used to compare scheduling algorithms, but is not implemented in the system.

Notice that if transactions need to know at run time if their data is stale, detection mechanisms must be implemented, and they will have a cost. In particular, for MA all view objects must have a timestamp associated with them. For $UU$, for every object read we must check for relevant updates in the unapplied update queue (see Section 3). Also notice that for $UU$, aborting transactions that read stale data is probably not a good strategy. If the data is stale, we have already identified some updates that need to be applied, so it seems best to apply the updates and resume the transaction, instead of aborting it. Under *MA* however, stale data will only become fresh if an external update arrives, so aborting the transaction is a reasonable option. In Section 6 we return

[2] For simplicity we assume synchronized physical clacks.

FILE: /pub/keller/1994/diana-mobidata-short.ps
COMMENT: Appeared in MOBIDATA: Workshop on Mobile and Wireless Information Systems, Rutgers, NJ, November 1994



# The DIANA Approach to Mobile Computing

**Tahir Ahmad, Stanford University**
**Mike Clary, Sun Microsystems**
**Owen Densmore, Sun Microsystems**
**Steve Gadd, Sun Microsystems**
**Arthur M. Keller, Stanford University**
**Robert Pang, Stanford University**

## Abstract

This paper presents a new application architecture to solve two major difficulties in developing software for mobile computing — diversity of user interface and varied communication patterns. Our architecture achieves display and network independence by decoupling the user interface logic and communication logic from the processing logic of each application. Such separation allows applications to operate in the workplace as well as in a mobile environment in which multiple display devices are used and communication can be synchronous or asynchronous. Operation during disconnection is also supported.

## 1. Introduction

People would like to compute not only while in the workplace but also while they travel. Although much current computer software can provide sophisticated functionality to ordinary users, it rarely addresses the special needs of those mobile users.

A number of issues limit the usability of software for mobile users. We consider two particular issues: user interface and communication. Conventional software is usually developed with implicit assumptions about the kinds of display devices users have and the communcation media available between them and the users. For example, applications written for the X-wmdows protocol can only be used by those users who run X-wmdows across high-speed computer networks. Such software is useless to mobile corn puter users with a different kind of environment.

Various computing devices tailored for mobile user are currently available, such as notebook computers palmtop computers and personal digital assistant (PDA). The user interface on these devices differ greatly. To make software accessible to thes devices, application developers currently have to cus tomize their software for each individual type o device. We anticipate an even greater variety of sucl mobile devices in the future and such diversity ca put a significant burden on software developers.

On the other hand, the communication network available to mobile users are still relatively slow an unreliable as compared to the high-speed local are. networks that connect workstations and mainframes Also, various communication protocols are used fo mobile computing and handling all these differen protocols can be a significant software developmen cost. Moreover, mobile users may need to communi cate intermittently because of the high cost o unavailability of communication.

To overcome these two hurdles in software develop ment for mobile computing, we propose a new appii cation architecture for mobile applications. Th DIANA architecture — Display Independence an Asynchronous Network Access — de-couples th display and communication logic of application from their processing logic. The resultant appiica tions will enjoy the benefits of being display an transport independent. Not only will new applica





tions be able to take advantage of this architecture, but also we envision that existing applications designed for directly connected users on particular devices will migrate to our approach.

### 1.1 Background

Our work on this project started by looking at workflow systems closely and trying to understand why people had failed to use them extensively in the industry. Our study raised the following issues as hindrances in the applicability of such systems.

**Platform Bindings.** One desirable aspect of these systems is that they should be able to make legacy applications work together in a broader context than they were originally designed for. However, most of these legacy applications are bound very strictly to the platform for which they were designed. Porting these applications to multiple platforms incurs significant development and maintenance costs.

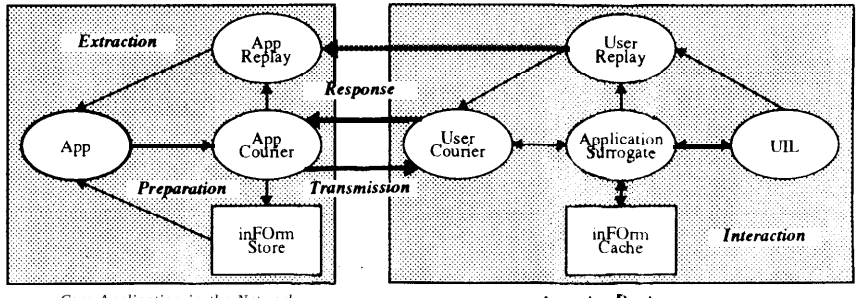
**Network Management.** Both the legacy applications and the workflow management systems have embedded assumptions about the underiymg networks and the available communication protocols that they will be using. These assumptions provide significant inflexibiiities and limitations. Furthermore, networking across hardware platforms introduces complications such as security. Also, there may be the unavailability of similar operating system features on different platforms, e.g., a workflow system relying on RPC (remote procedure call) to implement triggers becomes ineffective if one participant system does not provide support for RPC.

**Design Complexity.** An inherent problem with workflow systems is the complexity of designing working workflow models. Most of the systems, in their bid for providing the designers maximum flexibility, put a lot of responsibilities on the designers. Designers have to manage the conceptual entities in their workflow model, such as roles, tasks, jobs. Designers also have to be aware of the heterogeneity of the physical resources their system will be using and the restrictions this heterogeneity will impose on their model.

**Different Data Representation.** Multiple applications comprising a workflow system often have different views of data, which makes it more difficult to develop a generic solution for inter-application workflow.

To address these issues, we developed the DIANA architecture. DIANA uses a universal, dynamic language understood by all the entities in the system (applications, users, and system components) that IS based on the semantics of interaction. This language can be interpreted on multiple platforms, so that the

**FIGURE 1. DIANA — Overall Architecture and inFOrm Life Cycle**



same core application becomes accessible from a variety of heterogeneous devices. The network components of DIANA are capable of switching between direct connect (synchronous) and e-mail based (asynchronous) communication modes. Applications can access the data by using a simple information-level Application Programming Interface. By providing a single DIANA client to legacy applications, they can be made to interact with all other entities using the DIANA architecture.

### 1.2 Related work

Considering display and network independence, we observe that Mosaic and Teiescript [Caruso93] have close similarity with DIANA.

DIANA has a significant overlap with Mosaic as far as the interface is concerned. Mosaic uses Fill-Out Forms to express user interactions and query certain information sources distributed throughout the network. We have implemented the sample travel authorization application using Fill-Out Forms for a comparison between DIANA and Mosaic. From our experience of using Fill-Out Forms, we observe that one key difference between the DIANA approach and the Mosaic approach is that DIANA puts forward an interface language which is based completely on the semantics of the user interaction. On the other hand, Fill-Out Form language, combined with HTML, still assumes the presence of an access device with certain layout characteristics. This semantic representation of interaction in DIANA makes the system extensible to non-conventional access devices like telephones. However, we are looking into the possibility of enriching FDL by incorporating some HTML layout features without sacrificing display independence.

In reference to network communication, Mosaic adopts stateless communication between users and the application. The transaction states are kept by the application. On the other hand, the Courier in DIANA provides communication using a connection paradigm, as it keeps the states for the client applications. We believe this statefui communication can not only reduce setup time and bandwrdth requirements but also make it possible to cache informatiot locally. In addition, the Courter provides both synchronous and asynchronous commumcation whil Mosaic provides only synchronous communicatio currently. Future Mosaic may include communica tion between users and applications by e-mail. W have yet to see its implementation.

General Magic's Teiescript promises network an device independence. Unfortunately, very little infor mation about Telescript is publicly available to per mit a meaningful comparison with DIANA. A simplistic comparison is that DIANA handles th user interface like Mosaic and the network interfac like Teiescript.

[Imielinski93] identities some of the issues invoivec in mobile wireless computing, including iocatioi management, configuration management, dtsconnec tion, cache-consistency, recovery, scale, efficiency security and integrity. The paper also presents model of a system to support Mobility. In this sys tern. Mobile Units are assumed to have wireless e mail access to Mobile Support Stations (fixed net work hosts with a wireless e-mail interface to corn municate with the mobile units). Most of ou discussion assumes the presence of a similar systen for the purposes of wireless e-mail communication.

The Coda File System [Satya93] is a highly availabl file system designed to suit distributed and mobil computing. It uses an optimistic replica control strat egy to provide high availability and relies on dynamic cache manager to provide disconnecte operation. DIANA proposes to use a similar cachin strategy to minimize the network utilization an improve efficiency in addition to support discon nected operation. Details of various aspects of th Coda File System are available in [Satya90a] [Satya90b] and [Steere90].

### 1.3 Outline

In this paper, we will describe the proposed architec ture and explore the advantages of using DIAN architecture in mobile computing. We will also repor on the status of our implementation and our expert





ence in using DIANA. The organization of the rest of this paper is as follows. Section 2 describes the overall architecture of DIANA along with an illustrative application. Section 3 discusses how DIANA addresses the dtspiay independence problem. Section 4 focuses on the connectivity issues in DIANA. Section 5 discusses the current implementation of DIANA. Section 6 discusses future issues followed by concluding remarks in Section 7.

## 2. DIANA -The Overall Architecture

This section presents an overview of the DIANA architecture. We define the key components of the system, describe their responsibilities and explain how they work together. Subsequent sections of the paper discuss these components in further detail.

### 2.1 User interface Logic

In order to de-couple the user interface logic from the processing logic of applications, we define a *User Interface Logic* (UIL) to handle user interface operation for applications. As a component of the DIANA system, this UIL is independent of the client applications. There will be a different UIL for each different type of display device. Each different UIL will implement the semantics of forms by its own display characteristics appropriate for the display device. Client applications will be able to communicate with the UIL for a X-windows display device as they will do so with the UIL for a PDA. The end result will be that users can use different display devices to access the same application without requiring the application to handle those devices separately.

In order to allow applications to communicate with the UIL in a generic way, we design a *Form Description Language* (FDL) for applications to express the types of user interface to UILs. This language is based on the form paradigm and focuses on the semantics of the information exchange between the applications and the users rather than its aesthetics. A Form or script written in FDL is called *an inFOrm* (standing for *info-form*, hence the capitals). Since FDL focuses on the semantics of information exchange rather than the characteristics of display devices, applications which use FDL to describe their user interface can truly be display independent.

### 2.2 The Courier

The network manager m DIANA is called the *Courier*. It supports network Independence through the support of multiple network protocols and both synchronous and asynchronous modes. The synchronous communication mode represents the direct communication with potentially high bandwidth and low latency. On the other hand, the asynchronous communication is the situation when the communication is intermittent and with high latency. For asynchrony, a store-and-forward information exchange paradigm is more appropriate. Courter makes this kmd of asynchronous communication transparent to applications. We will discuss these modes in more details ma subsequent section.

There are two components of the Courier: one resides on the network with the application (Application Courier) and the other resides on the access device (User Courter). The purpose of having these couriers is to provtde an encapsulation of the underlying communication medium so that both users and applications can have the same processing logic independent of whether they are communicating in synchronous or asynchronous modes.

### 2.3 The Application Replay

In asynchronous communication mode, users may work asynchronously with applications during disconnection. In order to deliver inFOrms from users to applications in the sequence as they are generated, we define *Application Replay*, an agent on the applications' side that presents the inFOrms that were collected during disconnection to applications as if they were generated while continuously connected.

### 2.4 Other components

in order to reduce communication traffic, we add an inFOrm cache on the user device to store those

# An Algebraic Approach to Rule Analysis in Expert Database Systems+

Elena Baralis† Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
{baralis,widom}@cs.stanford.edu

**Abstract.** Expert database systems extend the functionality of conventional database systems by providing a facility for creating and automatically executing Condition-Action rules. While Condition-Action rules in database systems are very powerful, they also can be very difficult to program, due to the unstructured and unpredictable nature of rule processing. We provide methods for static analysis of Condition-Action rules; our methods determine whether a given rule ret is guaranteed to terminate, and whether rule execution is confluent (has a guaranteed unique final state). Our methods are based on previous methods for analyzing rules in active database systems We improve considerably on the previous methods by providing analysis criteria that are much less conservative: our methods often determine that a rule set will terminate or is confluent when previous methods could not. Our improved analysis is based on a "propagation" algorithm, which uses a formal approach baaed on an extended relational algebra to accurately determine when the action of one rule can affect the condition of another. Our algebraic approach yields methods that are applicable to a broad clan of expert database rule languages.

## 1 Introduction

In the past decade there has been a surge of interest in adding rule processing to database systems. *Deductive database systems use* logic rules to provide an expressive query facility [9, 24]. *Active database systems use* Event-Condition-Action rules to provide reactive behavior [15]. In this paper we focus on what we refer to as *expert database systems*. An expert database system is a conventional database system extended with a facility for creating and automatically executing Condition-Action rules. Expert database systems originated by coupling a rule processor for a production rule language such as *OPS5* [7] to a conventional DBMS; this approach is taken in, e.g., [23]. More recently the prevalent approach has been to build rule processing directly into the database system. Examples of recent or ongoing projects in expert database systems are [6, 11, 12, 13, 21]. Note that some systems described as active database systems actually use the Condition-Action rule paradigm, and hence fall into the class of expert database systems as we use the term here; examples of such systems are [14, 22]. Since expert database systems evolved from production rule systems such as OPS5 and are closely related to active and deductive database systems, the techniques presented in this paper certainly can be adapted for other database rule paradigms.

While expert database systems are very powerful, developing even small applications can be a difficult task, due to the unstructured and unpredictable nature of rule processing. During rule processing, rules can activate and deactivate each other, and the intermediate and final states of the database can depend on which rules are activated and executed in which order. It is highly beneficial if the rule programmer can predict in advance some aspects of rule behavior. This can be achieved by providing a facility that statically analyzes a set of rules, before installing the rules in the database [1]. Static rule analysis can form the basis of a design methodology and programming environment for expert database systems.

As has been observed in the past [1,17,25], two important and desirable properties of rule behavior are *termination* and *confluence*. A rule set is guaranteed to terminate if, for any database state and set of modifications, rule processing cannot continue forever (i.e. rules cannot activate each other indefinitely). A rule set is confluent if, for any database state and set of modifications, the final database state after rule processing is independent of the order in which activated rules are executed.

In this paper we propose a generally applicable algorithm for dctcrmining when the action of one rule can affect the condition of another rule. The algorithm uses an extension of relational algebra to model rule conditions and actions. Essentially, the algorithm "propagates" one rule's action through another rule's condition to determine how the action may affect the condition; hence, *we call* it the *Propagation Algorithm*. The Propagation Algorithm is useful for analyzing termination since it can determine when one rule may activate another rule. The Propagation Algorithm also is useful for analyzing confluence since it can determine when the execution order of two rules is significant. The Propagation Algorithm determines these properties much more accurately than previous methods, e.g [1,16]. In addition, since we take a general approach based on relational algebra, our method is applicable to most expert database systems that use the relational model.

### 1.1 Previous Related Work

In traditional expert systems, i.e. production rule systems such as OPS5 [7], predicting properties such as termination and confluence is of less importance than in the database environment. Consequently, to our knowledge there has been little work on rule analysis in traditional expert systems.

In the database context, [16,26] give methods for analyzing Condition-Action rules that are similar to the rules we consider. However, the goal of their work is to impose restrictions on rule sets so that confluence (a "unique fixed point" in their model) is guaranteed; we instead provide techniques for analyzing the behavior of arbitrary rule sets. In addition, the methods in [ 16,261 have been shown to be weaker than the methods in [ 1], which in turn are weaker than the methods we present here. The methods in [I] arc developed in the context of the Starburst Rule System, which uses an Event-Condition-Action (active database) rule model. Their technique for analyzing rule interaction relies on a shallow comparison of the actions performed by one rule and the events and conditions of another rule. We improve on this approach significantly by using a formal algebraic model that allows us to accurately analyze the intcraction between rules using the semantics of rule conditions and actions. In an initial report we applied our approach to termination only [3]; here we refine the techniques in [3] and propose a general framework for analysis of both termination and confluence.

In other related work, [25] analyzes rule behavior in the context of object-oriented active database systems. Their work focuses on differences between *instance-oraenfed* and *sef-orienfed* rules (we consider only set-oriented rules in this paper) and on dccidability properties for rule analysis. Their rule model is rather restricted, in that rule actions (methods) can only modify data selected by the corresponding rule condition, and deletions and insertions seem to be disallowed. The properties of confluence and of termination within some fixed number of steps arc shown to be decidable using an approach based on "typical databases"; a typical database contains all possible data instances that could affect the outcome of rule processing The rule set is "run" over the typical database and the outcome is checked for the desired properties. This approach is clearly infeasible in practical applications, so lower complexity algorithms are proposed, but the details and applicability of these algorithms are not clarified

A rather different approach to *rule* analysis is taken in a recent paper [17], where Event-Condition-Action rules are reduced to *term rewriting systems*, and known analysis techniques for termination and confluence of term rewriting systems are applied. The rule model they use is quite different from ours, and it is unclear whether a general relational rule model such as ours can be expressed as a term rewriting system. However, in the future we plan to explore the relationship between these different approaches.

Our Propagation Algorithm is closely related to the problem of *independence of queries and updates*, addressed in, e.g., [18]. [18] gives an algorithm for detecting if the outcome of a query, expressed as a Datalog program, can be affected by a given insertion or deletion For analyzing expert database rules, we need a somewhat stronger technique: when a query and update arc not independent, we need to know whether the update adds to, removes from, or modifies the result of the query. Furthermore, while the algorithm presented in [18] applies to more general queries than we consider here (e.g recursive queries), their model for database updates is considerably simpler than ours.

Finally, our Propagation Algorithm is somewhat related to *incremental evaluation*, as in (4, 19, 20]: both problems address the effect of a database modification on a relational expression However, incremental evaluation techniques are designed for run time, when the actual modifications are known, while our techniques apply at compile time, when the modifications are expressed as database operations.

### 1.2 Outline of the Paper

In Section 2 we present our algebraic Condition-Action rule language and provide several examples that are used throughout the paper. Section 3 contains the Propagation Algorithm, examples of its application, and a correctness Theorem In Sections 4 and 5 we apply the algorithm to the analysis of termination and confluence, respectively; again, several examples arc included In Section 6 we draw conclusions and outline future work

## 2 Algebraic Rule Language

A rule in our language has a *condition* and an *action*. Rule conditions are expressed as queries over the database; rule actions are database modifications. We use a language in which conditions and actions are both represented by relational algebra expressions. In this section we describe the extensions to relational algebra that are required to represent general rule conditions and actions. Then we specify the syntax of our rule language using this algebra, and we describe the semantics of rule processing in our model. Finally, we give several examples of how Condition-Action rules may be represented in our algebraic language

### 2.1 Algebraic Operators

Based on [8], we define an extension to relational algebra that allows us to represent any queries that are express

Thh work was partially performed while the authors were at the IBM Almaden Research Center, San Jose, CA. At Stanford this work was supported by the Reid and Polly Anderson Faculty Scholar Fund and by equipment grants from Digital Equipment Corporation and IBM Corporation.

Permanent address: Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi, 24-10129 Torino, Italy.



**Proceedings of the 20th VLDB** Conference, **Santiago, Chile, 1094.**

| Operator | Description |
|---|---|
| $\ltimes_p$ | semijoin with predicate p |
| $\ltimes_{\exists p}$ | not-exists semijoin with predicate $p$ |
| $\alpha_{A_1;A_2}$ | attribute rename |
| $\mathcal{E}[X = expr]$ | attribute extension and expression evaluation |
| $\mathcal{A}[X = a(A); B]$ | attribute extension and aggregate function evaluation |

Table 1: Additional algebraic operators

ible in SQL (or Quel), with the exception of the handling of duplicates and ordering conditions. We also introduce an extension that allows us to represent the SQL data modification operations **insert, delete**, and update.

Our extended relational algebra includes the basic relational algebra operators select ($\sigma$), ***project*** ($\pi$), ***cross-product*** (x), ***natural join*** ($\bowtie$), union (u), and ***difference*** ($-$), which we do not elaborate on here; see [24]. The first two lines of Table 1 present useful operators derived from the basic operators, while the next three lines present additional operators that we use In the table, X and $A$ denote attributes, $B$, $A_1$, and $A_2$ denote attribute lists, $a$ is an aggregate function, aud $expr$ is an expression (explained below). In liie 1, $E_1 \ltimes_p E_2 = \pi_{schema(E_1)}(\sigma_p(E_1 \times E_2))$; in line 3, $\alpha$ renames the attributes in list $A_1$ as $A_2$. In the remainder of the paper, we adopt the shorthand notation $E_1 \ltimes E_2$ and $E_1 \ltimes_{\mathcal{I}} E_2$ to denote $E_1 \ltimes_p E_2$ and $E_1 \ltimes_{\exists p} E_2$ when predicate p equates all attributes in both $schema(E_1)$ and $schema(E_2)$ (similar to the natural join). We now discuss the other operators in more detail, then we present the modification operations.

### 2.1.1 Not-Exists Semijoin

The ***not-exists semijoin operator,*** $\ltimes_{\exists p}$, is introduced to concisely express ***negative subqueries*** as they are expressed in SQL (e.g. not exists); negative subqueries appear frequently in rule definitions [10]. The not-exists semijoin operator is defined as:

$$E_1 \ltimes_{\exists p} E_2 = E_1 - (E_1 \ltimes_p E_2)$$

Note that we could instead define the relational difference operator in terms of not-exists semijoin: $E_1 - E_2 = E_1 \ltimes_{\mathcal{I}} E_2$ (with renaming of attributes in $E_1$ and $E_2$ as necessary). Hence, for convenience, we consider only the not-exists semijoin and not the difference operator in the remainder of the paper.

### 2.1.2 Aggregate Functions and Expression Evaluation

The ***attribute extension*** operators allow us to extend a relational expression $E$ with a new attribute; this approach is used for *aggregate* functions and for modification operations. We have:

- The $\mathcal{E}$ operator, which computes expressions applied to each tuple of $E$
- The $\mathcal{A}$ operator, which computes aggregate functions (e g mar, min, avg, sum, cnt) over partitions of $E$

$\mathcal{E}$ is a unary operator applied to a relational expression $E$ producing a result with schema $schema(E)$ u {X}. Recall from Table 1 that the $\mathcal{E}$ operator is expressed as:

$$\mathcal{E}[X = expr]E$$

$expr$ is an expression evaluated over each tuple $t$ of $E$ (a conventional expression involving attributes of $t$ and constants) yielding one value for each tuple; this value is entered into the new attribute X for each tuple of $E$. For details of similar operators see [8]; examples are given in later sections.

A is also a unary operator applied to a relational expression $E$ producing a result with schema $schema(E)$ U (X). Recall from Table 1 that the A operator is expressed as:

$$\mathcal{A}[X = a(A); B]\ E$$

$B$ defines a *set* of attributes on which the result of $E$ *is* partitioned; each group in the partition contains all the tuples with the same $B$ value. a is an aggregate function that is applied to the (multiset of) values contained in the projection of each partition on attribute A, yielding one value for each partition; this value is entered into the new attribute X for each tuple of the partition. The attributes $B$ are optional: when $B$ is omitted, no grouping is performed, and the aggregate function a is applied to the entire result of $E$, yielding one value; that value is entered into the new attribute X for each tuple of $E$. For details see [8].

### 2.1.3 Modification Operations

We represent data modification operations in relational algebra by characterizing the operations in terms of the database state they produce. Table 2 presents inserts, deletes, and updates by indicating the algebraic expressions that are used *to* denote the operations, and the way in which these expressions are applied to a relation $R$ to produce a new value for $R$. In the table, $A_u$ denotes the attributes of $R$ that are updated, $A'_u$ denotes primed versions of these attributes (explained below), and $A_r = schema(R) - A_u$.

**Insert operation.** An insert operation is denoted by a relational expression $E_{ins}$. $E_{ins}$ produces the tuples to be inserted (either a set of constant tuples or the result of an algebraic expression). The schema of $E_{ins}$ must coincide with the schema of $R$.

**Delete operation.** A delete operation is denoted by a relational expression $E_{del}$. $E_{del}$ produces the tuples to be deleted The schema of $E_{del}$ must coincide with the schema of $R$

| Operation | Algebraic expression | New database state |
|---|---|---|
| insert | $E_{ins}$ | $R \cup E_{ins}$ |
| delete | $E_{del}$ | $R \ltimes_{\mathcal{I}} E_{del}$ |
| update | $E_{upd}$ | $(R \ltimes_{\mathcal{I}} E_{upd}) \cup \alpha_{A'_u;A_u}(\pi_{A_r,A'_u} E_{upd})$ |

Table 2: Algebraic description of insert, delete, and update operations

**Update operation.** An update operation is denoted by a relational expression $E_{upd}$. $E_{upd}$ has schema $schema(R)$ u $A'_u$, where attributes $A'_u$ contain the new values for the updated attributes $A_u$. As convention, the new values for the updated attributes are always assigned the corresponding "primed" attribute names. That is, if attribute $A \in A$, is updated, then the new value for $A$ is assigned to attribute A'. A typical way to express $E_{upd}$ is:

$$E_{upd} = \mathcal{E}[A'_{u1} = expr_1]\ \mathcal{E}[A'_{u2} = expr_2]\ \cdots\ \mathcal{E}[A'_{un} = expr_n]\ E_c .$$

where $E_c$ is an expression producing the tuples to be updated (i.e. the "selection condition" of the update operation). The schema of $E_c$ must coincide with the schema of $R$. $\mathcal{E}[A'_{ui} = expr_i]$ evaluates expression $expr_i$ on each tuple of $E_c$ and assigns the result to the new attribute $A'_{ui}$. Although this is a useful form, in its generality $E_{upd}$ can be any relational expression with schema $schema(R) \cup A'_u$.

As specified in Table 2, the new state of $R$ after the update operation is the union of two terms:

1. The first term $R \ltimes_{\mathcal{I}} E_{upd}$ includes in the result all tuples in $R$ that are not modified by the update operation.
2. The second term $\alpha_{A'_u;A_u}(\pi_{A_r,A'_u} E_{upd})$ includes in the result the original values for the non-updated attributes of the modified tuples and the new values for the modified attributes, with the primed attribute names replaced by the original attribute names.

Given a relational expression $E$ (say) with schema $schema(R) \cup A'_u$, we often need the corresponding expression that is compatible in schema with $R$ and contains either the pre-updated (old) or the updated (new) values for the modified attributes. For convenience we will use the abbreviations $\rho_{old}(E) = \pi_{schema(E)-A'_u} E$ and $\rho_{new}(E) = \alpha_{A'_u;A_u}(\pi_{schema(E)-A_u} E)$.

## 2.2 Rule Syntax and Semantics

A Condition-Action rule in our language is defined as:

$$E_{cond} \rightarrow E_{act}$$

where:

- $E_{cond}$ states the rule's condition as an expression in our extended relational algebra.
- $E_{act}$ states the rule's action as a data modification operation expressed using $E_{ins}$, $E_{del}$, or $E_{upd}$ as given in Table 2. [1]

[1] For simplicity, we consider rules with a single action here, although many expert database systems allow rules with a sequence of actions. Our methods easily extend to multiple actions, usually simply by applying the method once for each action [2]

When this rule is evaluated, the condition $E_{cond}$ is true if and only if $E_{cond} - E^{old}_{cond} \neq 0$, where $E^{old}_{cond}$ denotes the result of $E_{cond}$ the last time the rule was evaluated during rule processing. If the rule has not previously been evaluated, then $E^{old}_{cond} = 0$. That is, informally, the condition is true whenever the query produces "new" tuples. This is identical to the interpretation of conditions in the Condition-Action rules of, e.g., *Ariel* [14], *RPL* [11], and set-oriented adaptations of OPS5 [13]; it also is similar to the way many Event-Condition-Action rules appear to be programmed in practice [10].

The action $E_{act}$ is a normal data modification operation executed on the current database state. In some expert database systems, e.g. [13,14], a rule's action implicitly operates only on the data "selected" by the condition, rather than on the entire database. We could use a similar rule model here, but it would complicate the syntax and semantics and has no bearing on our analysis methods: see Section 6 for further discussion.

Rule processing is invoked after some set of user or application modifications to the database The basic algorithm for rule processing is:

```
repeat until no rule has a true condition:
    select a rule r with a true condition;
    execute r's action
```

In this paper, we do not consider the effect of a *conflict resolution policy* for selecting among multiple rules with true conditions [15]. However, as an extension to our framework we plan to incorporate conflict resolution using rule priorities; see Section 6. Note also that the "granularity" of rule processing invocation with respect to database modifications [15] is irrelevant here in the context of rule analysis.

## 2.3 Examples

In this section we give the algebraic representation of five rules. These rules will be used as examples throughout the paper. All five rules refer to the following relations.

```
ACCT(num,name,bal,rate)
CUST(name,address,city)
LOW-ACC(num,name,date)
```

Relation ACCT contains information on a bank's accounts, while relation CUST contains information on the bank's customers. Relation LOU-ACC contains all accounts with a low balance, including the date on which the balance became low. We assume that the first attribute is a key for each relation, although our method does not rely on this assumption.

Paper 5

| | |
|---|---|
| FILE: | /pub/brin/1994/copy-detect.ps |
| COMMENT: | To appear in SICMOD '96 |

# Copy Detection Mechanisms for Digital Documents *

Sergey Brin, James Davis, Hector Garcia-Molina
Department of Computer Science
Stanford University
Stanford, CA 94305-2140
e-mail: sergey@cs.stanford.edu

October 31, 1994

Abstract

In a digital library system, documents are available in digital form and therefore are more easily copied and their copyrights are more easily violated. This is a very serious problem, as it discourages owners of valuable mformation from sharing it with authorized users. There are two mam philosophies for addressing this problem: prevention and detection. The former actually makes unauthorized use of documents difficult or impossible while the latter makes it easier to discover such actwity.

In this paper we propose a system for registering documents and then detecting copies, either complete copies or partial copies. We describe algorithms for such detection, and metrics required for evaluating detection mechanisms (covering accuracy, efficiency, and security). We also describe a working prototype, called COPS, describe implementation issues, and present experimental results that suggest the proper settings for copy detection parameters.

## 1 Introduct ion

Digital libraries are a concrete possibility today because of many technological advances in areas such as storage and processor technology, networks, database systems, scanning systems, and user interfaces. In many aspects, building a digital library today is just a matter of "doing il." However, there is a real danger that such a digilal library will either have relatively few documents of interest, or will be a patchwork of isolated systems that provide very restricted access.

The reason for this danger is that the electronic medium makes il much easier lo illegally copy and distribute information. If an information provider gives a document lo a customer, the customer can easily distribute il on a large mailing lisl or can post il on a bulletin board. The danger of illegal copies is not new, of course; however, it is much more time consuming lo reproduce and distribute paper. CDs or videotape copies than on-line documents.

Current technology does not strike a good balance between protecting the owners of intellectual properly arid giving access lo those who need the information. Al one extreme art! the open sources on the Internet, where everything is free, bul valuable information is frequently unavailable because of the dangers of unauthorized distribution. [1] At the other extreme are closed systems, such as the one that the IEEE currently uses to distribute is papers in CD-ROM. This a completely stand-alone system where users can look for specific articles, view them, and print them, but cannot move any data in electronic form oul of the system, and cannot add any of his or her data.

Clearly, one would like lo have an infrastructure that gives users access 10 a wide variety of digilal libraries and information sources, bul that al the same lime gives information providers good economic incentives for offering their information. In many ways, we believe this is *the* central issue for future digilal information and library systems.

In this paper we present one component of the information infrastructure that addresses this issue. The key idea is quile simple: provide a *copy detection service* w here original documents can be registered, and copies can be detected. The service will delecl not jusl exact copies, bul also documents that overlay is significant ways. The service can be used (see Section 2) in a variety of ways by information providers and communications agents lo delecl violations of intellectual properly laws. Although the copy detection idea is simple, there are several challenging issues we address here involving performance, storage capacity, and accuracy that need lo be resolved. Furthermore, copy detection is relevant lo the "database community" since its central component is a large database of registered documents.

We stress that copy detection is not the complete solution by any means: il is simply d helpful 1001. There are a number of other important "tools" that will also assist in safeguarding inlelleclual property. For example, good encryption and authorization mechanisms are needed in some cases. It is also important lo have mechanisms for charging for access lo information. The articles in [5, 7, 9] discuss a variety of other topics related lo intellectual properly. These other tools and topics will not be covered in this paper.

In the following section we will briefly discuss some of the options for safeguarding iulelleclual property, and will argue that copy detection is a very promising approach. In Section 3 we define the basic terms and evaluation metrics for copy detection. Then in Section 5 we describe our working prototype, COPS, and report on some iuilial experiments. A sampling technique that can reduce the storage space of registered documents or can speed up checking time is presented and analyzed in Section 6. Finally, some security considerations are discussed in Section 3.3.

## 2 Safeguarding intellectual property

How can we ensure that a document is only seen and used by a person who is authorized (e.g., has paid) lo see il? Lel us illustrate the possibilities and the problems by suggesting two particular techniques.

The first technique i s based on the notion of a *secure printer*.Such a printer is sealed and cannot be opened b y its owner. It contains a public key encryption device [11], where the private hey is unique lo this printer and is only known lo the printer itself. The printer's public key and the name of the owner are registered in a database provided by the trusted printer manufacturer. When the owner requests a document from an information vendor, the vendor first ensures the owner is authorized (e.g., has paid), then il fetches the public key of the printer from the registry, il encrypts the document using the public key and sends the result. When the owner receives the data, he can send il lo the printer which can then decrypt and print the document. However, the electronic data cannot be used for anything else, as only this one printer can decrypt il. The data can be resent 10 the printer 10 create another paper copy, so the document can be reproduced in this way. However, illegally reproduced paper copies is a "previously unsolved problem" that this scheme does not address.

The main problem with this scheme is that it is too restrictive. It is more of an "electronic paper delivery system" than anything else. Users cannot browse through documents before buying, and cannot use parts of the document in others, e.g., for quotes. Furthermore, it requires special purpose hardware. However, it may still be useful in conjunction with other schemes. For example, perhaps

*This research was sponsored by the Advanced Research Projects Agency (ARPA) of the Depar tment of Defense under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI) The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endor sement, either expressed or implied, of ARPA, the U S. Government or CNRI.

[1] As jut one example, Knight-Ridder Tribune recently ( June 23, 199 1) ceased publishing on ClariNet the Dave Barry and the Mike Royko columns because subscribers re distributed the ar ticles on large mailing lists .

users can be allowed to browse through low-resolution copies of documents, or through documents that have hey components missing. Once the user decides he wants to read the document. he can purchase the a high quality copy llial can be delivered via the secure printer. The scheme can also be adapted for a "secure computer" instead of a printer.

The second technique we wish to illustrate is that of an *active document* (suggested in [6]). The idea is that an information vendor does not send oul a documents: instead il sends out programs lhdl can generate documents. When a user receives one of these programs, call il $P$. he can run it on his local machine. Embedded within $P$ and ils data structures is the encrypted document: as $P$ runs. il displays the document. However. before or during display, $P$ sends a message lo the vendor. informing il that il is being run, and wails for a response. This way. the vendor can charge each lime $P$ runs. or can limit the number of times $P$ runs.

This scheme also has ils drawbacks. A user cannot read the document through his favorite viewer. The vendor must know the architecture of the user's machine in advance. lo generate appropriate code. The user cannot see the document if the vendor's machine is unavailable on the network. Finally, the scheme is not bullet proof, since the user could run $P$ in an software emulator of his machine that could record the characters of the document as they are displayed.

While we have only given two examples, we believe that they illuslrale a common problem with document *protection* techniques: they are often cumbersome arid usually gel in the way of users. The alternative is lo use *detection* techniques. That is, we assume most users are honest, allow them access lo the documents. and focus on detecting those that violate the rules. Many software vendors have found this approach lo be superior (protection mechanisms gel iii the way of honest users. and sales may actually decrease).

One possible direction is lo incorporate a "watermark" into a document llial identifies ils origin [12, 3. 4. 2]. For example, if we think of the documents as images, we may encode the watermark into a small number of random bits throughout the image. The users would be unaware of w here the watermark bits were, bul the information vendor that originally provided the document could exlracl them lo determine who the document was sold lo originally. If the document is possessed by a different person or organization, then a violation is detected. The main weakness of approaches such as these is that users may destroy the watermark by processing the document. For instance. passing the (image) document through a noise filter or lossy compression algorithm could easily change enough bils (without really altering the image) lo destroy the watermark.

A second approach, and one llial we advocate iii lliis paper (for lexl documents), is that of a *copy detection server* [1, 10]. The basic idea is as follows: When an author creates a new work, he or she registers il al the server. The server could also be the repository for a copyright recordation and registration system, as suggested in [8]. As documents are registered, they are broken into small units, for now say sentences. Each sentence is hashed and a pointer lo il is stored in a large hash table.

Documents can be compared lo existing documents in the repository, lo check for plagiarism or other types of significant overlap. When a document is lo be checked, il is also broken into sentences. For each sentence. we probe the hash table lo see if that particular sentence has been seen before. If the document and a previously registered document share more than some *threshold* number of sentences. then a violation is flagged. The threshold can be set depending on the desired checks. smaller if we are looking for copied paragraphs, larger if we only want lo check if documents share large portions. A human would then have lo examine both documents lo see if il was truly a violation.

Unlike the case with watermarks. il is not easy for a user lo automatically subvert the system. i.e.. to make an undetectable copy. For example, if the decomposition units are sentences. a user would have lo change a large number of sentences in the document. This involves more than just adding a blank space between words (assuming that the hashing scheme ignores spaces). Of course. d determined user could change all sentences. bul our goal is lo make il hard lo copy documents. not lo make il impossible. This makes il hard lo rapidly distribute copies of documents.

The copy detection server can be used in a variety of ways. For example, a publisher is legally liable for publishing materials the author does not have copyright on; thus. il may wish lo check if a soon-to-be-published document i s actually an original document. Similarly. bulletin-board software may automatically check new postings in this fashion. An electronic mail gateway may also check the messages that go through (checking for "transportation of stolen goods"). Program committee members may check if a submission overlaps loo much with an author's prev ious paper. Lawyers may want lo check subpoenaed documents lo prove illegal behavior. (Copy detection can also be used for computer programs [10], bul we only locus on lexl iii this payer.) The are also applications that do not involve detection of undesirable behavior. For example. a user that is retrieving documents from an information retrieval system or who is reading electronic mail. may want lo flag duplicate items (with a given overlap threshold). Here the "registered" documents are those that have been seen already; the "copies" represent messages that are retransmitted or forwarded many times. different editions or versions of the same work, and so on. Of course. potential duplicates should 1101 be deleted automatically; il is up lo the user lo decide if he wants lo view possible duplicates.

In summary, we think that detecting copies of lexl documents is a fundamental problem for distributed information or database systems. And there are many issues that need lo be addressed. For instance, should the decomposition units be paragraphs or something else instead of sentences? Should we take into account order of the units (paragraphs or sentences). e.g.. by hashing sequences of units? Is il feasible to only hash a fraction of the sentences of registered documents? This would make the hash table smaller. hopefully slill making il very likely that we will catch major v iolations. If the hash table is relatively small, il can be cloned. Our mail gateway above could then perform ils checks locally. instead of having lo contact a remote copy detection server for each message. There are also implementation issues that need lo be addressed. For example. how are sentences extracted from say Latex or Word documents? Can one extract them from Postscript documents. or from bil maps v id OCR?

These and other questions will be addressed in the resl of this paper. We start in Sections 3 and 4 b y defining the basic terms. evaluation metrics. and options for copy detection. Then in Section 5 we describe our working prototype, COPS. and report on some initial experiments. A sampling technique llial can reduce the storage space of registered documents or can speed up checking time is presented and analyzed in Section 6.

## 3 General Concepts

Ill lliis section we define some of the basic concepts for copy detection and for evaluating mechanisms that implement il. (As far as we know. lexl copy detection has 1101 been formally studied. so we slarl from basics.) The starting point is the concept of a *document*. a body of lexl from which some structural information (such as word and sentence boundaries) can be extracted. In an initial phase. formatting information and non-textual components are removed from documents (see Section 5). The resulting *canonical form* document consists of a string of ascii characters with whitespace separating words. punctuation separating sentences and possibly a standard method of marking the beginning of paragraphs.

A *violation* occurs when a document infringes upon another document in some way (e.g.. by duplicating portions of lexl). There are a number of violation types which can occ ur including plagiarism of a few sentences. exact replication of the entire document. and many steps in between. The notion of checking for a particular type of violation between two documents is captured by a *violation test*. If $\ell$ i s a violation test d u d $\ell(d, r)$ holds. then document $d$ violates document $r$ according lo the particular test. For example, $Plagiarism(d, r)$ is 11 ue if document $d$ has plagiarized from document I . We alsb extend this notation lo include checking against a set of documents:

# Deriving Incremental Production Rules for Deductive Data*

Stefano Ceri
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32
I-20133 hlilano, Italy
ceri@ipmel2.elet.polimi.it

Jennifer Widom
Department of Computer Science
Stanford U niversity
Stanford, CA 93405-2140
U SA
widom@cs.stanford.edu

**Abstract**

We show that the production rule mechanism provided by active database systems can be used to quickly and easily implement the logic rule interface of deductive database systems. Deductive rules specify derived relations using Datalog with built-m predicates and stratified negation; the deductive rules are compiled automatically mto production rules. We present a materialized approach, in which the derived relations are stored in the database and the production rules automatically and incrementally propagate base relation changes to the derived relations. We also present a non-materialized approach, in which the production rules compute the derived relations on demand.

## 1 Introduction

A considerable amount of research has focused on adding rules lo database systems. This work is divisible into two areas: *deductive* database systems and *active* database systems. In deduclive database systems, logic programming style rules are used lo provide a more powerful user interface than that provided by most database query languages [CGT90,Min88,Ull89]. In active database systems, production style (forward-chaining) rules are used lo provide automatic execution of database operations in response to certain events and/or conditions [DHW94,HW93].

With the rapid emergence of production rule capabilities in research prototypes, in a number of commercial database systems, and in the upcoming SQL3 standard, we believe llial active databases will become widely available in practice. However, we also believe that produclion rules in database systems should be treated as a low-level mechanism. used lo implement higher-level functionality [Cer92]. In lliis payer we show llial produclion rules can be used lo easily implement the high-level interface provided by deduclive databases. Hence, our work will make it possible lo quickly support deduclive capabilities in widely available systems.

In deductive databases. there is a distinction between *base* (or *extensional*) data and *derived* (or *intensional*) data. In lliis paper. we assume all data is stored in *relations*. Base relations are created and manipulated by users in llie usual way. Derived relations are Mined by deduclive rules specified in a language analogous to *Datalog* [Ull89], which in our framework includes buill-in predicates and stratified negation. Derived relations can be *materialized*. in w hich case they are stored in the database and kept consistent with the base relations over which they are defined. or they can be *non-materialized*, in which case they are not stored in the database but are computed from llie base relations as needed.

In our framework. each deduclive rule is compiled automatically into d set of production rules. Our main result concerns materialized derived relations. In lliis context. the generated production rules are triggered by changes lo base relations llial may affect the value of derived relations. arid the productions rules automatically modify the derived relations accordingly. The modifications are performed *incrementally*, thus exploiting available information about base relation changes and avoiding complete recomputation of derived relations. We also adapt our approach lo non-materialized derived r&lions. In this context. the generated produclion rules are triggered by queries referencing derived relations. and the yroduclion rules automatically compute llie necessary derived relations. Known optimization techniques for the computation, such as *magic sets* [BR86], can be incorporated into the generated production rules. In both the materialized and the non-materialized approach, the generated production rules encode a specific strategy for computing derived relations; however, our framework can easily be adapted for different strategies.

There is a strong similarity between the strategy used by the generated production rules and *semi-naive evaluation* [CGT90,Ull89]. Semi-naive evaluation is an incremental method that relies on *deltas*—changes between a previous database state and the current state. Most database production rule languages provide a mechanism for accessing deltas directly and efficiently [DHW94]. Hence, one of the contributions of our work is to provide an automatic method for exploiting the access to deltas provided by production rules. eliminating llie need lo "hard code" lliis access for deductive rule processing.

### 1.1 Related Work

Although there are substantial bodies of work in both active and deduclive databases, only limited research lo dale has focused on connecting the two approaches. Some work has shown how deductive databases can be extended lo support active behavior. e.g [BJ93, CCCR+90, HD93b, SKdM92,Tan91]; this relates lo the converse of the problem considered here. In the *RDL1* system, a forward-chaining production style of rule processing is used to implement d deductive database language [KdMS90]. The goal of RDL1 is an efficient deductive database system. Hence. although the underlying implementation is based on rule triggering, the rule language is deduclive. In our case. we illuslrale how deductive rules can be supported using the existing facilities of an active database system. This paper considerably improves upon and extends our own initial work in this area [Wid91].

In the approach we take to materialized derived relations, production rules propagate base relation changes to derived relations incrementally. Other research that has considered the problem of incrementally maintaining derived data includes [AP87, DT92, GMS93, HD93a, Kuc91, NY83, UO92,WDSY91]. Similar approaches have been applied to integrity constraint maintenance, e.g. [Nic82,UKN92,CFPT94,CW90], where the incremental nature of the computation results from the

*This work was partially performed while both authors were at the IBM Almaden Research Center, San Jose, CA. In Milano, this work was partially supported by Esprit Project P6333, "Idea". At Stanford, this work was partially supported by the Reid and Polly Anderson Faculty Scholar Fund and by an equipment grant from IBM Corporation.

assumption that the previous database state satisfies the constraints.

Pioneering work in incremental maintenance of deductive data [NY83] proposes an approach to materialized derived relations in which a counter is maintained for each derived tuple, encoding its number of derivations. Algorithms are given that use the counters to efficiently maintain the derived relations when base relations are modified. A similar approach based on derivation counting is described in [GMS93]. In [DT92,Kuc91,WDSY91], the actual derivations of tuples are encoded in auxiliary information stored with the derived relations. Changes to base relations are first considered with respect to the auxiliary information, then propagated to the derived relations. In contrast to all of these approaches, our method does not require any auxiliary information—derived relations are maintained incrementally using only the behavior provided by production rules. The absence of auxiliary information enables us to maintain derived relations using active database rules only, without any modifications to the underlying database system.

Methods in [GMS93, HD93a] also do not require auxiliary information; both papers propose algorithms that effectively "undo" and then "redo" derivations, which is similar to the effect of our production rules for materialized derived relations without unique derivations. In [UO92], special *internal events* are defined that describe transitions on derived relations, with rules for incremental evaluation of these events. In [AP87], a technique is presented based on *belief revisions*: procedures are defined for deducing the effect of insertions or deletions of data or rules on a given database that represents the current *belief*. Although the context and procedures in [AP87] are very different from our approach, there are some similarities (such as the treatment of stratification).

In [CW91], we describe a method for deriving production rules that incrementally maintain materialized views, where views are specified using a subset of SQL. Since there is an overlay between SQL views and derived relations specified using Datalog, there also is an overlay between the methods in [CW91] and the methods given here. In particular, views that are defined to be *safe* in [CW91] correspond to derived relations with *unique derivations* in this paper, so simplifications introduced for safe views in [CW91] carry over: see Section 6. In this paper, however, we handle derived relations that are defined recursively and may be defined through multiple deductive rules. Furthermore, we give a fully incremental approach, even for derived relations with multiple derivations, thereby generalizing and improving on the results in [CW91] considerably. Finally, note that much of the mechanism in [CW91] involves handling the complexities and limitations of views defined using SQL: these complexities and limitations are not an issue here.

### 1.2 Outline of Paper

Section 2 describes the initial language we use for deductive rules (excluding negation) and presents a running example for the payer. Section 3 describes the language we use for production rules—the rule language of *Starburst*.[1] Section 4 contains the first core results of the paper: a method for translating deductive rules in our initial language into production rules that incrementally maintain materialized derived relations, and a proof that the method is correct. Section 5 extends these results for deductive rules with stratified negation. Section 6 explains how the generated production rules can be simplified when derived relations are known to have unique derivations. Section 7 describes our approach for non-materialized derived relations. Section 8 concludes and describes future work.

[1] We use the Starburst rule language since we have developed and can experiment with it, but the same approach can be applied using other database production rule languages.



## 2 Deductive Rules

The initial language we use for deductive rules is based on Datalog with hill-in predicates but without negation. (We extend our results to stratified negation in Section 5.) The description given here is brief and somewhat informal; for thorough treatments of deductive database rule languages see, e.g., [BR86,CGT90,Ull89].

Let the database consist of a set of relations, some designated as base relations and others designated as derived relations.[2] Users may perform arbitrary retrieval or modification operations on base relations (e.g. SQL **select, insert, delete, update**).[3] Users may perform only retrieval operations on derived relations—data in derived relations is specified exclusively by deductive rules. The general form of a deductive rule is:

$$\begin{aligned} \textit{rule-name}: D(E_1, \ldots, E_l) \text{ :- } & R_1(X_1^1, \ldots, X_{j_1}^1), \quad \ldots \quad R_n(X_1^n, \ldots, X_{j_n}^n), \\ & P_1(E_1^1, \ldots, E_{k_1}^1), \quad \ldots, \quad P_m(E_1^m, \ldots, E_{k_m}^m) \end{aligned}$$

with the following requirements:

- $D$ names a derived relation.
- $R_1, \ldots, R_n$ name base or derived relations.
- $P_1, \ldots, P_m$ are hill-in predicates. Any predicate evaluable by the database query language is allowed. (Since we will be considering the Starburst database system [H+90], these are the standard SQL predicates along with user-defined predicates.)
- All X's on the right-hand-side (RHS) of the rule are either constants (specified by value) or variable names.
- All E's on the left-hand-side (LHS) and RHS of the rule are either constants, variable names that also appear at least once on the RHS as an $X$, or expressions over constants and such variables. Any expression evaluable by the database query language is allowed. (Again, in Starburst these are the standard SQL expressions along with user-defined expressions.)
- The number and type of arguments to $D$ and $R_1, \ldots, R_n$ match the schema of the relations.

[2] Although the only derived data we consider in this paper is relations, a similar approach could certainly be used to support, e.g., derived values stored as relational attributes.

[3] For simplicity in this paper we treat update operations as deletes followed by inserts (see Section 1), but an extension to directly handle updates is straightforward.

Paper 7

| | |
|---|---|
| FILE: | /pub/chawathe/1994/autonomous-cm.ps |
| FILE: | /pub/chawathe/1994/cm.ps |
| COMMENT: | Technical report |

# Flexible Constraint Management for Autonomous Distributed Databases*

Sudarshan S. Chawathe, Hector Garcia-Molina and Jennifer Widom
Computer Science Department
Stanford University
Stanford, California 94305-2140
E-mail: {chaw,hector,widom}@cs.Stanford.edu

## 1 Introduct ion

When databases inter-operate, integrity constraints arise naturally. For example, consider a flight reservation application that accesses multiple airline databases. Airline A reserves a block of X seats from airline *B*. If *A* sells many seats from this block, it tries to increase X. For correctness, the value of X recorded in *A*'s database must be the same as that recorded in B's database; this is a simple distributed copy constraint. However, the databases in the above example are owned by independent airlines and are therefore autonomous. Typically, the database of one airline will not participate in distributed transactions with other airlines, nor will it allow other airlines to lock its data. This renders traditional constraint management techniques unusable in this scenario. Our work addresses constraint management in such autonomous and heterogeneous environments.

In an autonomous environment that does not support locking and transactional primitives, it is not possible to make "strong" guarantees of constraint satisfaction, such as a guarantee that a constraint is always true or that transactions always read consistent data. We therefore investigate and formalize weaker notions of constraint maintenance. Using our framework it will be possible, for example, to guarantee that a constraint is satisfied provided there have been no "recent" updates to pertinent data, or that a constraint holds from 8am to 5pm everyday. Such weaker notions of constraint satisfaction requires modeling time, and consequently, time is explicit in our framework.

Most previous work in database constraint management has focused on centralized (for e.g., [?]) or tightly-coupled and homogeneous distributed databases (for e.g., [1], [2], [3], [4]). The multi-database transaction approach to constraint management weakens the traditional notion of correctness of schedules [5], [6]. This approach cannot, however, handle a situation in which different databases support different interfaces. In modeling time, our work has similarities with some work in temporal databases [7] and temporal logic programming [8]. 0ur approach is closer to the event-based specification language in RAPIDE [9].

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and diitribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation

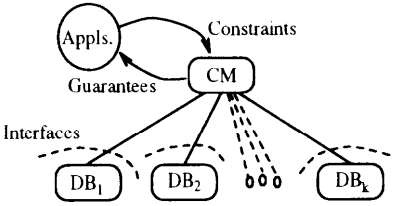



Figure 1: Constraint Management Architecture

In this paper, we give a brief overview of our formal framework for constraint management in autonomous systems and describe the constraint management toolkit we are building. The details of the underlying execution model, semantics of events, and syntax and semantics of the rule language may be found in [10].

## 2 Formal Framework

In this section, we present an outline of our formal framework for constraint management. Our framework assumes the simplified system architecture shown in Figure 1.[1] Each database chooses the interface it offers to the constraint manager (CM) for each of its data items (involved in an inter-database constraint). The interface specifies how each data item may be read, written and monitored by the CM. Applications inform the CM of constraints that need to be monitored or enforced. Based on the constraint and the interfaces available for the data items involved in the constraint, the CM decides on the constraint management strategy it executes. This strategy tries to monitor or enforce the constraint as well as possible using the interfaces offered by the local databases. The degree to which each constraint is monitored or enforced is formally specified by the *guarantee* We describe interfaces, strategies and guarantees below.

### 2.1 Interfaces

The interface for a data item involved in a constraint describes how that data item may be read, written, and/or monitored by the constraint manager. Interfaces are specified using a notation based on events and rules. For example, consider a simple write interface for a data item X, This interface promises to write the requested value to X within, say, 5 seconds. We express this as $W\ R(X, b)@t \rightarrow W_g(X, b)@[t, t + 5]$. Here $WR(X, b)@t$ represents a "write-request" event which requests the operation $X \leftarrow b$, and which occurs at some time $t$. The rule says that whenever such an event occurs, a "write"[2] event, $W_g(X, b)$ occurs at some time in the interval $[t, t + 5]$. The interfaces for the data items involved in inter-database constraints are specified by the database administrator of each database, based on the level of access to the database he or she is willing to offer the CM. Currently, we rely on the users of our framework to verify that the interfaces specified do faithfully represent the actual systems.

[1] Note that we assume a centralized constraint manager for simplicity in presentation only; the constraint manager is actually distributed.

[2] $W_s()$ is a *generated* write event which occurs as the result of CM activity, to be distinguished from spontaneous write events, $W_s()$, which occur due to user/application activity in the underlying database.

### 2.2 Strategies

The strategy for a constraint describes the algorithm used by the constraint manager to monitor or maintain the constraint. Like interfaces, strategies are specified using a notation based on events and rules. In addition to performing operations on data items involved in a constraint, strategies may evaluate predicates over the values of data items (obtained through database read operations) and over private data maintained by the Constraint Manager.

As a simple example, consider the following strategy description, which makes a write request to Y whenever it receives a "notify" event from $X$:[3] $N(X,b)@t \rightarrow WR(Y,b)@[t, t + 7]$.

Once our framework has been used to specify a strategy, and to verify the correctness of a guarantee, then the rule-based strategy specification is implemented using the host language of the Constraint Manager. This is a simple translation, and may be done using a rule engine.

### 2.3 Guarantees

A guarantee for a constraint specifies the level of global consistency that can be ensured by the Constraint Manager when a certain strategy for that constraint is implemented. Typically a guarantee is *conditional*, e.g., a guarantee **might** state that if no updates **have** recently been performed then the constraint holds, or that if **the value** of a CM data item is true then the constraint holds. Guarantees are specified using predicates over values of data items and occurrences of certain events. For example, consider the following guarantee for a constraint X = Y: (Flag = **true**)$@t \Rightarrow$ (X = $Y$)$@@[t - \alpha, t - \beta]$. This guarantee states that if the (Boolean) data item Flag is true at some time $t$, then X = Y (at all times) during the interval $[t - \alpha, t - \beta]$. Note that this guarantee is weaker than a guarantee that X = Y always, which is very difficult to make in **the** heterogeneous, autonomous environments we study.

## 3 A Constraint Management Toolkit

We are building a toolkit that will permit constraint management across heterogeneous and autonomous information systems. For example, this toolkit will allow us to maintain a copy constraint spanning data stored in a Sybase relational database and a file system, or an inequality constraint between a *whois*-like database and an object-oriented database. We give a brief overview of this toolkit in this section.

### 3.1 Architecture

Figure 2 depicts the architecture of our constraint management toolkit. The Raw Information Source (RIS) is what is already present at each site (for example, a relational database, a file system, or a news feed.) The **RISI** is the interface offered by each **RIS** to its users and applications. For example, for a Sybase database, the **RISI** is based on a particular dialect of SQL, and includes details on how to connect to the server.

The CM-Translator is a module that implements the CM-Interface (the interface discussed in Section 2.1) using the **RISI**. The CM-RID is a configuration file used to specify (1) which types of CM-Interface (selected from a menu of pre-compiled interface types) are supported by the **CM-Translator**, and (2) how these interfaces are implemented using the underlying **RISI**.

[3] A notify event is an event type representing the database containing X notifying the CM of a write to X. Thus N( X, 5) means that a write X ← 5 occurred.



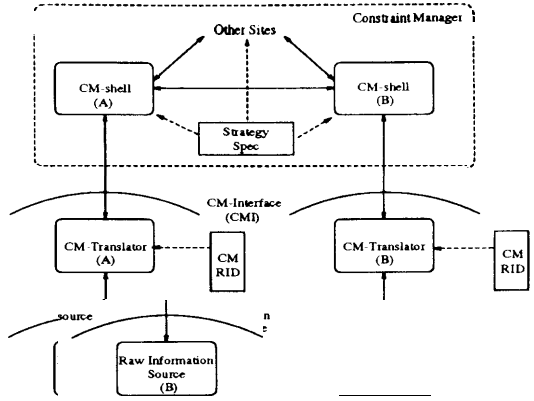



Figure 2: Constraint Management Toolkit Architecture

The CM-Shell is the module that executes the constraint management strategies described in Section 2.2. Since we specify strategies using a rule-based language, the CM-Shells are distributed rule engines that are configured by a Strategy Specification.

### 3.2 Application

We now describe how database administrators would use our toolkit to set up constraint management across autonomous systems. The database administrators at each site first decide on **the** CM-Interfaces **they** are willing to offer, selected from menu of pre-compiled interfaces provided by the toolkit. For example, if the underlying RIS provides triggers, then a notify interface may be offered (where the CM is notified of updates); if not, perhaps a read/write interface can be offered. The choice also depends on the actions the administrator wants to allow. For instance, even if the **RIS** allows database updates, the administrator may disallow a write interface that lets the CM make changes to the local data.

Each CM-RID file records the interfaces supported, as well as the specification of the **RIS** objects to which **the** interface applies. The CM-RID is also **the** place where site-specific translation information is stored. This includes, for example, the name of the Sybase data server that holds the data and its port number, how a read request from the CM is translated into an SQL query, how a request to set up a notify interface is translated to commands that set **up** a trigger, and so on.

Next, the administrator uses a Strategy Design Tool (not shown in Figure 2) to develop the CM strategy. This tool takes as input the inter-database constraints, and based on the available interfaces, suggests strategies from its available repertoire. For each suggested strategy, the design tool can give **the** guarantee that would be offered. The result of this process is the Strategy Specification file, that is then used by the CM-Shells at run time. (Knowledgeable administrators **can** write their Strategy Specifications, bypassing the Design Tool.)

At run-time, the CM-Shells execute the specified strategy based on the event-rule formalism. The CM-Translators take care of translatmg events to site-specific operations, and vice versa.

Paper 8

FILE: /pub/chawathe/1994/tsimmis-overview.ps
COMMENT: Appeared in 1994 IPSJ Conference (Tokyo)

# The TSIMMIS Project: Integration of Heterogeneous Information Sources'

**Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, Jennifer Widom**

**Department of Computer Science**
**Stanford University**
**Stanford, CA 94305-2140**

*last-name@cs.stanford.edu*

## Abstract

**The goal of the Tsimmis Project is to develop took that facilitate the rapid integration of heterogeneous information sources that may include both structured** and **unstructured data. This paper gives an overview of the project, describing components that extract propertia from unstructured objects, that translate information into a common object model, that combine** information **from several** sources, that **allow browsing of information, and that** *manage* **constraints across heterogeneous sites. Tsimmis is a joint project between Stanford and the IBM Almaden Research Center.**

## 1 Overview

A common problem facing many **organizations** today is that of multiple, **disparate** information sources **and** repositories, including **databases,** object stores, knowledge bases, file systems, digital libraries, information retrieval systems, and electronic mail systems. Decision makers often need information from multiple sources, but are unable to get and fuse the required information in **a timely** fashion **due to the** difficulties of accessing the **different** systems, and due **to** the fact that the **information** obtained can be inconsistent **and** contradictory.

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339 The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government This work was also supported by the Reid and Polly Anderson Faculty Scholar Fund, the Center for Integrated Systems at Stanford University, and by Equipment Grants from Digital Equipment Corporation and IBM Corporation*

The goal of the TSIMMIS' project **is to provide** tools for accessing, in an integrated fashion, **multiple** information sources, **and to ensure that** the information obtained is consistent. Numerous other recent projects **have** similar goals, of course. Before describing the differences between Tsimmis and other data integration projects, let us give an overview of the Tsimmis architecture, describing the functions of the **various** components **and** the philosophy of our approach. Refer to Figure I.

### 1.1 Translators and Common Model

Figure I shows a collection of (disk-shaped) heterogeneous information sources. Above each source is a ***translator*** *(or wmpper)* that logically converts the underlying **data objects to** a **common** information model. To do this logical translation, the translator **converts** queries over information in the common model into requests **that the** source can execute, and it converts the **data** returned by the source into **the** common model.

For the Tsimmis project we **have** adopted a simple ***self-describing*** **(or lagged) object** model. Similar models have been in use for years; we call our version the ***Object Exchange*** Model, or *OEM*. OEM allows simple nesting of objects, and a complete specification is given in Section 2. The fundamental idea is that all objects, and their subobjects, have ***labels*** that describe their meaning. For example, the following object represents a Fahrenheit temperature of 80 degrees:

(temp-in-Fahrenheit, int, 80)

where the string "temp-in-Fahrenheit" is a **human**-readable label, "int" indicates an integer value, and "80" is the value itself. If we wish to represent a complex **object,** then each component of the object has **its own** label. For example, an object representing a set of two temperatures may look like:

[1] As an acronym, TSIMMIS stands for 'The Stanford-IBM Manager of Multiple Information Sources" In addition, Tsimmis is • Yiddish word for • stew with "heterogeneous" fruits and vegetables integrated into • surprisingly tasty whole

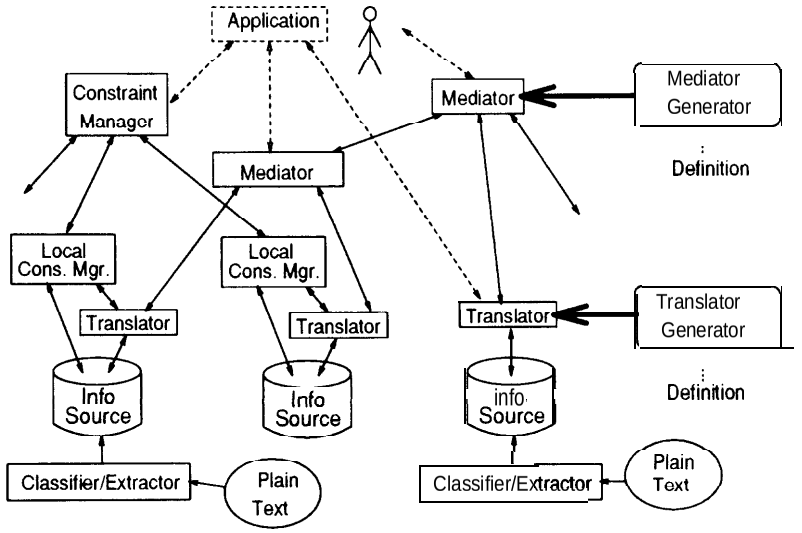



Figure 1: Tsimmis Architecture

(set-of-temps, set, $\{cmp_1, cmp_2\}$)
$cmp_1$: (temp-in-Fahrenheit, int, 80)
$cmp_2$: (temp-in-Celsius, int, 20)

OEM is very simple, while providing the expressive power and flexibility needed for integrating information from disparate sources. We also **have** developed a query language, *OEM-QL*, for requesting **OEM objects.** OEM-QL is an SQL-lie language extended to **deal** with labels and object nesting; see Section 2.

### 1.2 Mediators

Above the translators in Figure **1** lie **the** ***mediators***. A mediator is **a** system that refines in some way **information** from one or more sources [31]. A mediator embeds the knowledge that is necessary for processing a specific type of information. For example, a mediator for "current events" might know that relevant information sources **are** the AP Newowire and the New York Times database. When the mediator receives a query, say for **articles on** "Bosnia," it will know to **forward** the query to those sources. The mediator may also pro **cess** answers before forwarding them to the user, say by converting **dates** to a common format, or by **eliminating** articles that duplicate information. **While the** task of converting dates is probably straightforward, the task of **eliminating** duplicate information could be very complex,-figuring out that two articles written by different authors say "the same thing" requires real intelligence In Tsimmis we are focusing on relatively simple mediators based **on** patterns or rules. Still, even simple mediators can perform very useful information processing and merging tasks.

Implementing a mediator can be complicated and time-consuming, but we believe that much of the coding involved in mediators can be automated. Hence, one important goal of the Tsimmis project is to automatically or **semi-automatically** generate mediators from high level descriptions of the information processing they need to do This is illustrated by the ***mediator generator*** box on the **right** side of Figure **1. Similarly, we** provide a ***translator generator*** that can generate OEM translators based on a **description** of the conversions that need to take place for queries received and results returned. This component, also illustrated in Figure **1,** significantly facilitates the **task** of implementing a new translator.

### 1.3 System and User Interfaces

Mediators export **an** interface to their clients that is identical to **that** of translators. Both translators and mediators take **as** input OEM-QL queries and return OEM objects. **Hence**, end users and mediators **can**

obtain their information either from translators and/or other mediators. This approach allows new sources to become useful as soon as a translator is supplied, it allows mediators to access new sources transparently, and it allows mediators to be "stacked," performing more and more processing and refinement of the relevant information.

End users (top of Figure 1) can access information either by writing applications that request OEM objects, or by using one of the generic browsing tools we have developed. Our most recent browsing tool provides access through Mosaic or other *World-* Wide- *Web* viewers [4,29]. The user writes a query on an interactive world-wide-web page, or selects a query from a menu. The answer is received as a hypertext document. The root of this document shows one or more levels of the answer object, with hypertext links available to take the user to portions of the answer that did not appear on the root document. This tool provides a mechanism for exploring heterogeneous information sources that is easy to interact with and that is based on a commonly used interface. The browser is described in more detail in Section 3.

### 1.4 Labels end Mediator Processing

It is important to note that there is no global database schema, and that mediators can work independently. For instance, to build a mediator it is only necessary to understand the sources that the mediator will use. In fact, it is not even necessary to fully understand the sources used. For example, returning to our "current events" mediator, suppose one source exports objects with subobjects labeled by *title*, *dale*, *author*, and *country*. The mediator might always pass the author and country subobjects to its client with no additional processing. Now suppose a second source provides topic and *date* subobjects. The mediator might convert the dates from both sources into a common format, and it will know how to convert a mediator query about the subject of an article into the appropriate *topic* or *title* queries to be sent to the sources.

When a mediator simply passes subobjects to its clients (as in author and *country* above), it might append the *source* name to the labels so that the client can interpret the objects correctly. For example, a mediator subobject might have label *NYTimes.author*, indicating that this author is from the New York Times *source* and follows its conventions for authors. Another object might have the label *AP.author*. (Of course, the mediator could also make the formats consistent and export subobjects with label author, but here we are illustrating a simple mediator that does not do such processing.)

The key points are that a mediator does not need to understand all of the data it handles, and no person or software component needs to have a global view of all the information handled by the system.

### 1.6 Constraint Management

Another important component in the Tsimmis architecture is constraint management, illustrated in Figure 1 by a *Constraint* Manager and two *Local Constraint Managers*. *Integrity* consfminfs specify semantic consistency requirements over stored information; such constraints arise even when the information resides in loosely coupled, heterogeneous systems. For example, a construction company keeps data about a building under construction. This data must be consistent with the architect's design (e.g., walls must be in the same places), which may be stored in an entirely different system. Constraint management in the distributed, heterogeneous environments addressed by Tsimmis is a much more difficult and complex problem than constraint management in centralized systems: Transactions across multiple information sources usually are not provided, and each information source may support different capabilities for accessing and monitoring the data involved in a constraint.

In current environments, constraints across heterogeneous information sources usually are monitored or enforced by humans, in an ad-hoc fashion (or, frequently, not checked at all). For example, an architect may freeze the building design and send the latest specifications to the construction company so that consistency is "guaranteed." Of course, it is clear that these ad-hoc mechanisms do not work well in general; in our example, it is likely that the building may eventually not meet its specifications.

Since in a loosely coupled environment it is generally not possible to guarantee that every user or application sees consistent data every time it interacts with the system, the Tsimmis constraint manager enforces constraints with weaker guarantees than what a centralized system may provide. Tsimmis makes "relaxed" guarantees, e.g., a constraint is true from 8am to 5pm every day, or a constraint is true if some "Flag" is set. Ensuring relaxed consistency is especially challenging because one now has to deal with the timing of actions and of guarantees. However, the advantages of being able to handle relaxed guarantees in heterogeneous systems are significant; knowing precisely what holds and what does not hold, and when, will clearly lead to more trustworthy systems.

The Tsimmis constraint manager supports the definition of the *interfaces* that a source supports for the information involved in a constraint (e.g., can a trigger be set on a data item?), specification of the desired constraint (e.g., two items should have the same value), and specification of the *strategy* that is to be followed for enforcing the constraint or for detecting violations. The Local Constraint Managers in Figure 1 are responsible for describing and supporting interfaces, while the Constraint Manager processes constraints and executes strategies. Note that the Constraint Manager actually is not centralized as illustrated in Figure 1, but rather is a set of distributed components that jointly manage constraints. Constraint management is described in more detail in Section 4.

### 1.6 Classification end Extraction

The final component of the Tsimmis architecture consists of the *Classifier/Extractors* shown at the bottom of Figure 1. Many important information sources are completely unstructured, consisting of plain files or incoming bit strings (e.g., from a newswire). Often it is possible to automatically classify the objects in such sources (e.g., is the file an email message, a text file, or a gif image?), and to extract key properties (e.g., creation date, author). The Classifier/Extractor performs this task, based on identifying simple patterns in the objects. The information collected by the Classifier/Extractor can then be exported (via a translator, if necessary) to the rest of the Tsimmis system, together with the raw data. The Classifier/Extractor component is based on the *Rufus* system developed at the IBM Almaden Research Center [25] and is not discussed further in this paper.

### 1.7 Related Work

There are a number of differences between integration of information sources in the Tsimmis project and other database integration efforts (e.g. [2,13,18,28] and many others):

- Tsimmis focuses on providing integrated access to very diverse and dynamic information. The information may be unstructured or semi-structured, often having no regular schema to describe it. The components of objects may vary in unpredictable ways (e.g., some pictures may be color, others black and white, others missing, some with captions and some without). Furthermore, the available sources, their contents, and the meaning of their contents may change frequently.
- Tsimmis assumes that information access and integration are intertwined. In a traditional integration scenario, there are two phases: an integration phase where data models and schemas (or parts thereof) are merged, and an access phase where data is fetched, In our environment, it may not be clear how information is merged until samples are viewed, and the integration strategy may change if certain unexpected data is encountered.
- Integration in our environment requires more human participation. In the extreme case, integration is performed manually by the end user For example, a stock broker may *read* a report saying that IBM has named a new CEO, then retrieve recent IBM stock prices from a database to deduce that stock prices will rise. In other cases, integration may be automated by a mediator, but only after a human studies samples of the data, determines the procedure to follow, and develops an appropriate specification for the mediator generator.

In summary, the Tsimmis goal is *nof* to perform fully automated information integration that hides all diversity from the user, but rather to provide a framework and tools to assist humans (end users and/or humans programming integration software) in their information processing and integration activities

Regarding the constraint management aspects, there has been substantial prior work on database constraints, focusing on centralized databases (e.g., [14]), tightly-coupled homogeneous distributed databases (e.g., [12, 26]), or loosely-coupled heterogeneous databases with special constraint enforcement capabilities (e.g., [8,24]). The multidatabase transaction approach weakens the traditional notion of correctness of schedules (e.g., [5, 10]), but this approach cannot handle a situation in which different databases support different capabilities In its modeling of time, our work has some similarity to work in temporal databases [27] and temporal logic programming [1], although our approach is closer to the event-based specification language in RAPIDE [19]

### 1.8 Remainder of Paper

In the rest of this paper we provide additional details on some of the Tsimmis components. In Section 2 we describe the OEM object model and its query language In Section 3 we present the Tsimmis/Mosaic object browser. In Section 4 we outline the main components of the constraint management toolkit. In Section 5 we conclude, describe the status of the Tsimmis prototype and discuss future directions of our work.

## 2 Object Exchange

As described in Section 1.1, our Object Exchange Model (OEM) is used as the unifying object model for information processed by Tsimmis components Note that information *need* not actually be stored using OEM, rather OEM is used for the processing of logical queries, and for providing results to the user.

Each object in OEM has the following structure

# Scheduling Problems in Parallel Query Optimization (Extended Abstract)

CHANDRA CHEKURI*
Department of Computer Science
Stanford University
Stanford, CA 94305-2140

WAQAR HASAN†
Department of Computer Science
Stanford University
Stanford, CA 94305-2140
*and*
Hewlett-Packard Laboratories

RAJEEV MOTWANI‡
Department of Computer Science
Stanford University
Stanford, CA 94305-2140

**Abstract**

We introduce a class of novel multiprocessor scheduling problems that arise in the optimization of SQL queries for parallel machines. These consist of scheduling a tree of inter-dependent operators while exploiting both inter and intra-operator parallelism. We focus on the specific problem of scheduling a *Pipelined Operator Tree* in which all operators run in parallel using inter-operator parallelism. Weights associated with nodes and edges represent respectively the cost of operators and communication. Communication cost is incurred only if adjacent operators are assigned different processors. The optimization problem is to assign operators to processors so as to minimize the maximum processor load. We develop two approximation algorithm.4 for this NP-hard problem. The faster algorithm has a performance ratio of 3.56 while the slower algorithm has a ratio of 2.88.

*E-mail: chekuri@cs.stanford.edu. Supported by an OTL grant and NSF Young Investigator Award CCR 9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

†E mail: hasan@cs.stanford.edu.

‡E mail: rajeev@cs.stanford.edu. Supported by an IBM Faculty Development Award, an OTL grant, and NSF Young Investigator Award CCR 9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

## 1 Introduct ion

Exploiting parallel execution [1, 16] to speedup database queries presents a parallelism-communication trade-off. While work is divided among processors. the concomitant communication increases total work itself [6, 12]. We ran represent the task to be scheduled as a *weighted operator tree* in which nodes represent atomic units of execution (operators) and directed edges represent the flow of data as well as timing constraints between operators. Weights associated with nodes and edges represent respectively the cost of operators and communication between them. A class of novel multi-processor scheduling problems arise since edges can represent either pipelining or precedence constraints, and both inter-operator and intra-operator parallelism may he exploited.

We discuss several problems but focus on the specific problem of scheduling a *Pipelined Operator Tree* (POT scheduling) in which edges represent parallel constraints, i.e., all operators run in parallel. A schedule assigns operators to processors. Since edge weights represent the cost of remote communication. this cost is saved if adjacent operators share a processor. Given a schedule. the load on a processor is the sum of the weights of nodes assigned to it plus the weights of all edges that connect nodes on the processor to nodes on other processors. The response time (makespan) of a schedule is the maximum processor load. The optimization problem is to find schedules with minimal response time.

The POT scheduling problem is NP-hard since the special case in which all communication costs are zero is classical MULTIPROCESSOR SCHEDULING [S]. We will measure the quality of algorithms by their *performance ratio* which is the ratio of the response time of the generated schedule to that of the optimal. Since the problem is intractable, our goal is to develop *approximation algorithms* that run in polynomial time and guarantee small hounds on the performance ratio.

This paper develops two approximation algorithms. The faster algorithm has a. performance ratio of 3.56 while the slower algorithm has a ratio of 2.88. Our earlier work [7] developed algorithms for this problem that are only known to have hounded performance ratios when the shape of the tree is a path or a star.

The scheduling problems discussed in this paper differ in several ways from classical scheduling problems [5]. Firstly. operators have *distinct* weights since parallel database systems exploit *coarse-grained* parallelism. Secondly, since data is transmitted in long streams. the important aspect of communication is the cpu overhead of sending/receiving messages and not the delay for signal propagation (see [11, 10] for models of communication as delay). 'Thirdly. edges represent two kinds of timing constraints parallel and precedence. Finally. the set oriented nature of queries has led to intra-operator parallelism (relations are horizontally partitioned and a clone of the operator applied to each partition) in addition to inter-operator parallelism [1].

Section 2 provides an overview of parallel query optimization and develops a model for the scheduling problems. Section 3 investigates the POT scheduling problem anti develops two approximation algorithms. Section 4 characterizes some open problems.

## 2 A Model for the Problem

Figure 1 shows the two-phase approach [9, 8] for parallel query optimization. The first phase. similar to conventional query optimization [14], produces an annotated join tree that fixes aspects such as the order of joins and the strategy for computing each join. The second phase. *parallelization*, converts a join tree into a parallel plan. Parallelization itself has two steps [7]. The first converts the annotated join tree to an operator tree [2, 8, 13]. The second schedules the operator tree on a parallel machine.

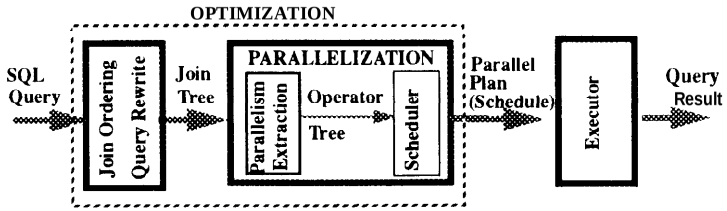



Figure 1: Software Architecture for Parallel Query Processing

### 2.1 Forms of Parallelisttt

Parallel database systems speed-up queries by exploiting *independent* and *pipelined* forms of inter-operator parallelism as well as intra-operator or *partitioned* parallelism. Independent parallelism simultaneously runs two operators with no dependence between them on distinct processors. Pipelined parallelism runs a consumer operator simultaneously with a producer operator on distinct processors. Partitioned parallelism uses several processors to run a single operator. It `exploits` the set-oriented nature of operators by partitioning the input data and running a `copy` of the operator on each processor.

### 2.2 Operator Trees

Available parallelism is represented as an operator tree $T = (V, E)$ with $V = \{1, \ldots, n\}$. Nodes represent operators. Functionally, an operator takes zero or more input sets and produces a single output set. Physically, it is a. piece of code that is *deemed* to he atomic. Edges between operators represent the flow of data as well as timing constraints. As argued in our earlier work [7], operators may be *designed* to ensure that any edge represents either a parallel or a precedence constraint.

**Definition 1** *A* pipelining *edge from operator $i$ to $j$ represents a parallel constmint that requires $i$ and $j$ to start at the same time and terminate at the same time. A* blocking *edge from $i$ to $j$ represents a* precedence constraint *that requires $j$ to start after $i$ terminates.*

A pipelining constraint is symmetric in $i$ and $j$. The direction of the edge indicates the direction in which tuples flow but is immaterial for timing constraints.

**Example 2.1** Figure 2 shows a join tree and the corresponding operator tree. 'Thin edges are pipelining edges. thick edges are blocking. A simple hash join is broken into `Build` and `Probe` operators. Since a hash tahle must he fully built before it. can he probed. the edge from `Build` to `Probe` is blocking. A sort-merge join sorts both inputs and then merges the sorted streams. The merging is implemented by the `Merge` operator. In this example. we assume the right input of sort-merge to he pre-sorted. The operator tree shows the sort required for the left input broken into two operators `FormRuns` and `MergeRuns`. Since the merging of runs can start only after run formation. the edge from `FormRuns` to `MergeRuns` is blocking. I

The operator tree exposes the available parallelism. Partitioned parallelism may he used for any operator. Pipelined parallelism may be used between two operators connected by a pipelining edge Two subtrees with no (transitive) timing constraints between them may run independently (eg: subtrees rooted at `FormRuns` and `Build`).

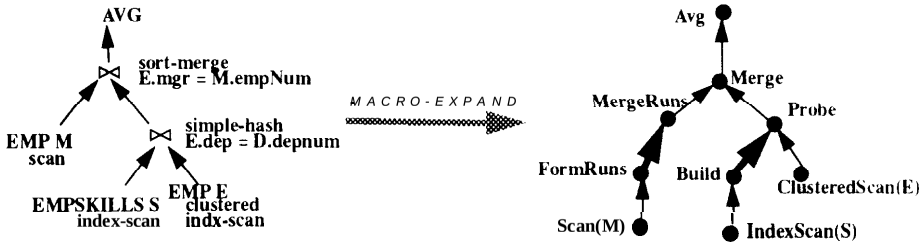
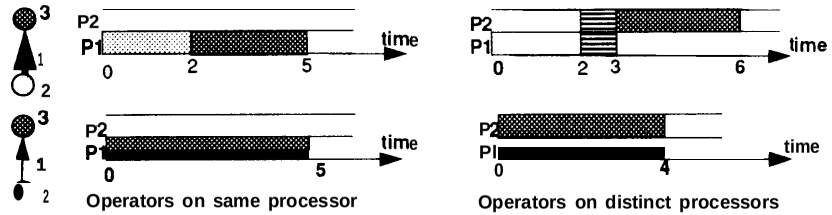



Figure 2: Macro-Expansion of Join Tree to Operator 'Tree (Parallelism Extraction)




Fignrc 3: Communication Costs for Blocking and Pipelining Edges: Gantt Charts

### 2.3 Model of Communication

The weight $t_i$ of node $i$ in an operator tree is the time to run the operator in isolation assuming all communication to he local. The weight $c_{ij}$ of an edge from node $i$ to $j$ is the *additional* cpu overhead due to inter-processor communication. This overhead is incurred at hoth the sender and the receiver processor. A specific schedule incurs rommnniration overheads only for the *fraction* of data that it actually communicates across processors.

Figure 3 shows the extreme cases of rommnniration costs of blocking and pipelining edges. Communication is saved when the two operators are on the same processor and totals to twice the edge weight when they are on distinct processors. For a blocking edge the rommnniration occurs after the producer terminates and before the consumer starts. For a pipelined edge the communication is spread over the execution time of the entire operator. Note that since all operators in a pipeline start and terminate simultaneously, cheaper operators consume a smaller percent age of the processor they run on.

As discussed in [7], conventional cost models (such as System R [11]) that estimate the sizes of the intermediate results may he easily `adapted to` estimate node and edge weights.

### 2.4 Scheduling Problems

We assume a parallel machine to consists of $p$ identical processors. A schedule assigns operators to processors. We model partitioned parallelism as permitting processors to execute *fractions* of an operator. Depending on whether partitioned parallelism is allowed or not, the assignment of operators to processors is a fractional' or O-I assignment Since the goal of a parallel database system is to speedup queries, we are interested in finding schedules with minimal response time

Paper 10

| | |
|---|---|
| FILE: | /pub/widom/1994/book-chapter.ps |
| COMMENT: | Appeared in book edited by Won Kim |

# Active Database Systems

**Umeshwar Dayal**
**Hewlett-Packard** Laboratories
1501 **Page Mill Road**
**Palo** Alto, CA 94303
*dayal@hplabs.hp.com*

**Eric N. Hanson**
**Dept. of Computer** and **Information Sciences**
University **of Florida**
**Gainesville, FL** 32611
*hanson@cis.ufl.edu*

**Jennifer Widom**
**Dept. of Computer Science**
**Stanford** University
**Stanford,** CA 94305-2140
*widom@cs.stanford.edu*

**Abstract**

Integrating a production rules facility into a database system provides a uniform mechanism for a. number of advanced database features including integrity constraint enforcement, derived data maintenance, triggers, alerters. protection, version control, and others. In addition. a database system with rule processing capabilities provides a useful platform for large and efficient knowledge-base and expert systems. Database systems with production rules are referred lo as *active database systems*, and the field of active database systems has indeed been active. This chapter summarizes current work in active database systems; topics covered include active database rule models and languages, rule execution semantics. and implementation issues.

## 1 Introduction

Conventional database systems are *passive*: they only execute queries or transactions explicitly submitted by a user or an application program. For many applications, however. it is important lo monitor situations of interest. and lo trigger a timely response when the situations occur. For example, an inventory control system needs lo monitor the quantity in stock of items in the inventory database, so that when the quantity in stock of some item falls below a threshold. a reordering activity may be initiated. This behavior could be implemented over a passive database system in one of two ways, neither of which is satisfactory. First, the semantics of condition checking could be embedded in every program that updates the inventory database. but this is a poor approach from the software engineering perspective. Alternatively, an application program can be written lo poll the database periodically lo check for relevant conditions. However. if the polling frequency is too high. this can be inefficient. and if the polling frequency is loo low. conditions may not be detected in a timely manner.

An *active database system*, in contrast. is a database system that monitors situations of interest. and when they occur. triggers an appropriate response in a timely manner. The desired behavior is expressed in *production rules* (also called *event-condition-action rules*), which are defined and stored in the database. This has the benefit that the rules can be shred by many application programs, and the database system can optimize their implementation.

The production rule paradigm originated in the field of Artificial Intelligence ( A I ) with expert systems rule languages such as OPS5 [Brownston et al. 1985]. Typically. in AI systems. a production rule is of the form:

$$condition \rightarrow action$$

An inference engine cycles through all the rules in the system, *matching* the condition parts of the rules with data in working memory. 01' all the rules that match (the *candidate set*), we is selected using some *conflict resolution policy*, and this selected rule is *fired*, that is. ils action part is executed. The action part may modify the working memory, possibly according lo the matched data. and the cycle continues until no more rules match.

This paradigm has been generalized to event-condition-action rules for active database systems. These are of the form:

**on** *event*
**if** *condition*
**then** *action*

This allows rules lo be triggered by events such as database operations, by occurrences of database states. and by transitions between slales (among other things), instead of being evaluated by an inference engine that cycles periodically through the rules. When the triggering event occurs. the condition is evaluated against the database; if the condition is satisfied, the action is executed. Rules are defined and stored in the database. and evaluated by the database system, subject lo authorization, concurrency control, and recovery.

Such event-condition-action rules are a powerful and uniform mechanism for a number of useful database tasks: they can enforce integrity constraints, implement triggers and alerters. maintain derived data. enforce access constraints, implement version control policies. gather statistics for query optimization or database reorganization, and more [Eswaran 1976, Morgenstern 1983, M. Stonebraker 1982]. Previous support for these features, when present, provided lillle generality and used special-purpose mechanisms for each. In addition, the inference power of production rules makes active database systems a suitable platform for building large and efficient knowledge-base and expert systems.

While the power of active database systems was recognized some lime ago, a true research lield did not emerge until relatively recently [Dayal 1988]. However, the lield has quickly blossomed. and i l currently enjoys considerable activity and recognitionA number of powerful research prototypes have been built [Hanson 1992] [Chakravarthy et al. 1989] [Dayal et al. 1988] [McCarthy and Dayal 1989] [Gehani and Jagadish 1991] [Stonebraker et al. 1990] [Stonebraker and Kemnitz 1991] [Delcambre and Etheredge 1988] [Widom el al. 1991] [Widom and Finkelstein 1990] [Beeri and Milo 1991] [Cohen 1989] [Diaz el al. 1991] [Kotz et al. 1988] [Schreier el al. 1991] [Simon el al. 1992] [Buchmann 1990] [E. Anwar 1993] [S. Gatziu 1991]. In this chapter, we will illustrate the features of active database systems using *Ariel* [Hanson 1992], *HiPAC* [Chakravarthy el al. 1989. Dayal el al. 1988, McCarthy and Dayal 1989], *POSTGRES* [Stonebraker el al. 1990, Stonebraker and Kemnitz 1991], and *Starburst* [Widom el al. 1991. Widom and Finkelstein 1990] as representative of the lield. Limited production rule capabilities are now appearing in commercial database products such as *Ingres* [INGRES 1992], *InterBase*, *Oracle* [ORACLE 1992], *Rdb* [Rdb 1991], and *Sybase* [Howe 1986], and in the emerging SQL2 and **SQL.?** standards.

This chapter provides a broad survey of current work in active database systems. The discussion is divided into three technical areas. Rule models and language design are discussed in Section 2. rule execution semantics in Section 3, and implementation issues in Section 4. Section 5 concludes and discusses areas for future research.

## 2 Rule Models and Languages

This section describes the issues involved in designing a database production rule language and explains how those issues have been addressed in various active database systems. We also describe the rule language features proposed for SQL2 and SQL3, which are indicative of the state of commercial practice.

Some of the differences among rule languages stem from differences in the underlying data models supported by the different systems. In relational systems **such** as Ariel. POSTGRES. Starburst. and the commercial products, rules are defined (and named) as metadata in the schema, together with tables, views. integrity constraints. and the like. As with other metadata. operations **are** provided **lo add. drop.** or modify **rules.** In object-oriented systems such as HiPAC, rules are treated as first class objects that are instances of rule types defined in the schema. These rule types are subtypes of a generic type **rule.** Rules are structured objects, having events. conditions. and actions as their components. Like any object, rules can be created. deleted. or modified. In addition, rule objects have some special operations, including: fire. which causes a rule **lo** be triggered; **enable,** which causes **a** rule lo be activated: **disable.** which causes a rule lo be deactivated (so that il won't be triggered even if ils triggering event occurs).

Database rule languages vary considerably in the complexity of specifiable events. conditions. and actions. In some languages, the triggering event may be implicit—any relevant change **lo** the database that can **cause** the condition to become true is treated as a triggering event. However. a key advantage of making events explicit is the flexibility gained in expressing transitions. For example, suppose il is desired lo keep the salaries of two employees, Alice and **Bob.** the same. Suppose further that the following semantics are desired: if the constraint is violated because Alice's salary is changed by a user transaction. then change **Bob's** as well: however. if the constraint is violated because Bob's salary is changed by a user transaction. then abort the transaction. With explicit events. it becomes possible lo specify these separate transitions.

I I I addition. some languages prov ide mechanisms w hereby data (parameters) can be *bound*

# Precision and Recall of *GlOSS* Estimators for Database Discovery

Luis Gravano Héctor García-Molina Anthony Tomasic

Computer Science Department
Stanford University
Stanford, CA 94305-2140

{gravano,hector,tomasic}@cs.stanford.edu

## 1 Overview

On-line information vendors offer access to multiple databases. In addition, the advent of a variety of INTER VET tools [1, 2] has provided easy, distributed access to many more databases. The result is thousands of text databases from which a user may choose for a given information need (a user query). This paper, an abridged version of [3], presents a framework for (and analyzes a solution to) this problem, which we call the *text-database discovery problem* (see [3] for a survey of related work).

Oitr solution to the text-database discovery problem is to build a service that can suggest potentially good databases to search. A user's query wilt go through two steps: first,. the query is presented to our server (dubbed *GlOSS*, for **Glossary-Of-Servers Server**) to select a set o f promising databases to scarrh. During the second step, the query is actually evaluated at the chosen databases. *GlOSS* gi ves a hint of what databases might, tic useful for the user's query, based on word-frequency information for each database. This information indicates, for each database and each keyword in the database vocabulary, how many documents at, that database actually contain thr keyword, for each field designator (Sections 2 and 3). Fnr example, a Computer-Science library could report that *"Knuth" (keyword)* occurs a.9 an *author (field designator)* in 180 documents, the keyword *"compute r,"* in the *title* of 25.548 documents, and so on. This information is orders of magnitude smaller than a full index (see [4]) since for each keyword *firld*-designation pair we only need to keep its frequency, tint the identities of the documents that contain it

To evaluate the set of databases that. *GlOSS* returns for a given query, Section 4 presents a framework based on the precision and recall metrics o f information-retrieval theory In t h a t theory, for a given query $q$ and a given set $S$ o f relevant documents for $q$, *precision* is the fraction of documents i n the answer to $q$ that are in $S$, and *recall* i s the fraction of $S$ itt the answer to $q$. We borrow these notions to define metrics for the text-database discovery problem: for a given query $q$ and a given set of "relevant databases" $S$, $P$ is the fraction of databases in the attswrr to $q$ that. are in $S$, and $R$ is the fraction of $S$ i n the answer to $q$. We further extend our framework by offering different definitions for a "relevant database" (Section 4 ) . We have performed experiments using query traces from the FOLIO library information-retrieval system at. Stanford Universi ty, and i nvolving six databases available through FOLIO. As we will see, the results obtained for different variants of *GlOSS* arc very promising (Section 5). Even though *GlOSS* keeps a small amount of information about the contents of the available databases, this information proved to be sufficient to produce very useful hints on where to scarrh.

## 2 *GlOSS*: Glossary-Of-Servers Server

Consider a ***boolean "and"*** query $q$ [1] that we want to evaluate over a wt. of databases DR. *GlOSS* selects a subset of $DB$ consisting of "good candidate" databases for actually submitting $q$. To make this selection, *GlOSS* has available the following information:

- $DBSize(db)$, the total number of documents in database $db$, $\forall\ db \in$ DR. and
- $freq(t, db)$, the number of documents in $db$ that contain $t$, $\forall\ db \in DB$, and for all keyword field-designation pairs $t$. Note that *GlOSS* does not have available the actual "inverted l i s t s ' corresponding to each keyword-field p a i r and each database, but just the length o f these inverted lists.

[1] We can generalize this paper's approach to "or" queries (see [1]) and to the vector-space retrieval model [5].

This information is extracted from each database by a *collector* program (not discussed further here) that forwards it periodically to *GlOSS*.

To assess how good each database is for a given query, *GlOSS* uses an *estimator*: an estimator $EST_\epsilon$ (for some fixed $0 \le \epsilon \le 1$) consists of a function $\mathsf{ESize}_{EST_\epsilon}$ t h a t predicts t h r result size o f a query in a database, and a "matching' function that uses these estimates t o select the "good' databases ($Chosen_{EST_\epsilon}$). Onrr $\mathsf{ESize}_{EST}(q, rfh)$ has been defined, we can determine $Chosen_{EST_\epsilon}(q, DB)$ i n the following way:

$$Chosen_{EST_\epsilon}(q, DB) = \{db \in DB \mid \mathsf{ESize}_{EST}(q, tfb) > 0 \wedge \frac{\mathsf{ESize}_{EST}(q, db) - best}{best} \le \epsilon\} \quad (1)$$

where $best = \max_{db' \in DB} \mathsf{ESize}_{EST}(q, db')$.

Users ran set the value for $\epsilon$ according to the query semantics they arc interested in: in general, higher values for $\epsilon$ make the $Chosen_{EST_\epsilon}$ set "larger:' i f $\epsilon = 0$, only those databases containing the strictly highest non-zero estimates will belong to $Chosen_{EST_\epsilon}$, whereas if $\epsilon = 1$, all databases with a non-zero estimate will belong to $Chosen_{EST_\epsilon}$.

## 3 The *Ind*$_\epsilon$ estimators

Thr *Ind*$_\epsilon$ (for "independence' ) estimators [4] arc built upon the (unrealistic) assumption that keywords appear in the different documents of a database following independent and uniform probability distributions [2]. Under this assumption, given a database $db$, any n keyword field-designation pairs $t_1, \ldots t_n$, and any document $d \in db$, the probability that $d$ contains all of $t_1, \ldots t_n$ is

$$\frac{freq(t_1, db)}{DBSize(db)} \times \ldots \times \frac{freq(t_n, tfh)}{DBSize(db)}$$

So, according to *hf*.. thr estimated number of documents in $db$ that will satisfy the query *"find* $t_1$ A. $\wedge t_n$' is [6]:

$$\mathsf{ESize}_{Ind_\epsilon}(find\ t_1 \wedge \ldots \wedge t_n, db) \frac{\prod_{i=1}^{n} freq(t_i, db)}{DBSize(db)^{n-1}}$$

The $Chosen_{Ind_\epsilon}$ set i s then computed with Equation 1, for any value of $\epsilon$

## 4 Evaluatiou metrics

Let $DB$ be a set of databases and $q$ a query. In order to evaluate an estimator $EST$ (e.g . $EST$=hf.).

[2] Even though this assumption is unrealistic, we will see that the *Ind*$_\epsilon$ estimators work surprisingly well.

we need to compare its prediction **against.** what actually i s $Right(q, DB)$, the "right subset" n f $DB$ to query. There arc several notions of **what the** right subset means, depending on the semantics the query submitter has in mind. In t h i s paper, we will consider two of these notions. for which the goodness of a database $db$ w i t h respect to a query $q$ will be determined by thr number of documents that $db$ returns when presented w i t h $q$ [3].

0111 first definition for $Right(q, DB)$ is $Matching(q, DB)$, the wt. of all databases in $DB$ containing at least one document that matches query $q$ More for mally,

$$Right(q, DB) = Matching(q, DB) = \{db \in DB \mid \mathsf{RSize}(q, db) > 0\}$$

where $\mathsf{RSize}(q, db)$ is the actual result size o f query $q$ in database $db$. There a r c (at least) two types o f users that may specify $Matching(q, DB)$ as their right set of databases One is users t h a t want an exhaustive answer to their query. They a r c n o t willing to miss any o f the matching documents We will refer to these users as "recall-oriented' users O t t the other h a r d . "precision-oriented' users may be in "sampling' mode: they simply want to obtain *some* matching documents without searching useless databases.

Our second definition for $Right(q, DB)$ is, for a fixed $0 \le \delta < 1$. $Best_\delta(q, DB)$, the s r t of those databases containing the most matching documents More formally,

$$Right(q, DB) = Best_\delta(q, DB) = \{db \in DB \mid \mathsf{RSize}(q, tfb) > 0 \wedge \left|\frac{\mathsf{RSize}(q, db) - hreal}{hreal}\right| \le \delta\}$$

where $hreal = \max_{db' \in DB} \mathsf{RSize}(q, db')$. Parameter $\delta$ is not a parameter of our estimators, but of our evaluation metrics the submitter of a query does not give a $\delta$ value to *GlOSS*. Higher values for 6 yield more comprehensive $Best_\delta$ sets Therefore, parameter $\delta$ should be fixed according to the desired "meaning' for $Best_\delta$ For example, suppose that we are evaluating *Ind*$_\epsilon$ for a user that wants to locate *Best* databases, but 1s willing to search at sites that have 90% or more *of* the number of matching documents than the overall *Best* sites have Then, the experimental results t h a t are relevant to this user are those obtained for 6 = 0 1

Again, users that define $Best_\delta(q, DB)$ as their right set of databases for query $q$ might be classified as being "recall oriented" or "precision oriented ' "Recall-oriented' users a r c willing to miss some databases,

as long as they are not the best ones. These users want to ensure that at least those databases having the highest payoff (i.e., the largest number of documents) are searched. On the other hand, "precision-oriented" users want to examine (some) best databases. Due to limited resources (e.g., time, money) the users only want to submit their query at databases that will yield the highest payoff.

Once we have defined the *Right* set for a query $q$ and a database set $DB$, we evaluate how well $Chosen_{EST}(q, DB)$ approximates $Right(q, DB)$ by adapting the well-known *precision* and *recall* parameters from information-retrieval theory [5] to the text-database discovery framework. If we regard *Right* as the set of "items" (databases in this context) that are relevant to a given query $q$, and $Chosen_{EST}$ as the set of items that is actually retrieved, we can define the following functions $P_{Right}$ and $R_{Right}$, based upon the precision and recall parameters:

$$P_{Right}(q, DB) = \begin{cases} \frac{|Chosen \cap Right|}{|Chosen|} & \text{if } |Chosen| > 0 \\ 1 & \text{otherwise} \end{cases}$$

$$R_{Right}(q, DB) = \begin{cases} \frac{|Chosen \cap Right|}{|Right|} & \text{if } |Right| > 0 \\ 1 & \text{otherwise} \end{cases}$$

where $Chosen = Chosen_{EST}(q, DB)$ ($EST$ is a fixed estimator for $GlOSS$) and $Right = Right(q, DB)$.

Intuitively, $P$ is the fraction of selected databases that are *Right* ones, and $R$ is the fraction of the *Right* databases that are selected. "Precision oriented" users will be interested in high values of $P$, while "recall oriented" users will be interested in high values of $R$.

Section 5 evaluates different estimators in terms of the average value, over a set of user queries, of the $P$ and $R$ parameters defined above, for different *Right* sets of databases.

## 5 $Ind_\epsilon$ results

In order to evaluate the performance of the $Ind_\epsilon$ estimators according to the $P$ and $R$ parameters of Section 4, we performed experiments using 6897 queries and six databases available through the FOLIO library information-retrieval system at Stanford University. Real users issued the queries to the INSPEC database through the FOLIO system. In [3], we describe the query trace and the experiments. We also study how well $\mathsf{ESize}_{Ind}$ approximates RSize.

Figure 1 shows the average values of the $P$ and $R$ parameters for $Ind_0$, for growing values of $\delta$. Our estimator remains fixed (since $\epsilon = 0$) for the different values of $\delta$, and so does $Matching$. This is why

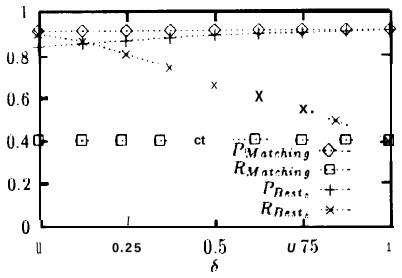



Figure 1: The average $P$ and $R$ parameters as a function of $\delta$, for the $Ind_0$ estimator ($\epsilon = 0$).

the curves corresponding to $P_{Matching}$ and $R_{Matching}$ are flat. In particular, $P_{Matching}$=0.9126: this means that, for the average query, 91.26% of the databases in $Chosen_{Ind_0}$ have matching documents. In contrast, $R_{Matching}$=0.4044, meaning that, for the average query, $Chosen_{Ind_0}$ includes only 40.44% of the $Matching$ databases. $Ind_0$ chooses only the most promising databases, not all of the ones that might contain matching documents. Higher values of $\epsilon$ address this problem (see below).

On the other hand, the set of best databases, $Best_\delta$, varies as $\delta$ does. In Figure 1 we see that parameter $R_{Best_\delta}$ worsens as $\delta$ grows, since $Best_\delta$ tends to contain more databases, while $Chosen_{Ind_0}$ remains fixed. This is exactly why $P_{Best_\delta}$ improves with higher values of $\delta$. Note that for $\delta = 1$, $Best_1 = Matching$, and so, $P_{Matching}$ and $R_{Matching}$ coincide with $P_{Best_\delta}$ and $R_{Best_\delta}$, respectively.

Figure 2 shows the average values of the $P$ and $R$ parameters for $Ind_\epsilon$, for growing values of $\epsilon$. For all these results, $\delta = 0$ (i.e., the "best" set of databases is fixed to $Best_0$). Since $Chosen_{Ind_\epsilon}$ tends to cover more databases as $\epsilon$ grows, $R_{Matching}$ and $R_{Best_0}$ improve for higher values of $\epsilon$. For $\epsilon = 1$, $R_{Matching} = R_{Best_0} = 1$, since $Chosen_{Ind_1}$ contains all of the potentially matching databases. This is also why $P_{Best}$ worsens as $\epsilon$ grows. Parameter $P_{Matching}$ remains basically unchanged for higher values of $\epsilon$, but worsens for $\epsilon$ close to one, for the same reasons $P_{Best_0}$ gets lower.

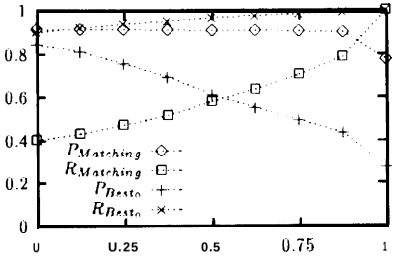



Figure 2: The average $P$ and $R$ parameters as a function of $\epsilon$, for the $Ind_\epsilon$ estimator ($\delta = 0$).

| *Right* | $P_{Right}$ ($Min_0$) | $R_{Right}$ ($Min_0$) | $P_{Right}$ ($Ind_0$) | $R_{Right}$ ($Ind_0$) |
|---|---|---|---|---|
| *Matching* | 0.9077 | 0.4031 | 0.9126 | 0.4044 |
| $Best_0$ | 0.8356 | 0.8938 | 0.8438 | 0.9010 |

Figure 3: The average $P$ and $R$ parameters for the $Min_0$ estimator. The last two columns show the corresponding values for the $Ind_0$ estimator.

## 6 Other results

The $Ind_\epsilon$ estimators are based upon the assumption that the occurrence of query keywords in documents follows independent and uniform probability distributions. We can build alternative estimators by departing from this assumption. For example, we can adopt the "opposite" assumption, and assume that the keywords that appear together in a user query are strongly correlated. So, we define another family of estimators for $GlOSS$, $Min_\epsilon$ (for "minimum"), by letting:

$$\mathsf{ESize}_{Min}(find\ t_1 \wedge \ldots \wedge t_n, db) = \min_{i=1}^{n} freq(t_i, db)$$

$\mathsf{ESize}_{Min}(q, db)$ is an upper bound of the actual result size of query $q$: $\mathsf{RSize}(q, db) \leq \mathsf{ESize}_{Min}(q, db)$. $Chosen_{Min_\epsilon}$ follows from the definition of $\mathsf{ESize}_{Min_\epsilon}$ using Equation 1. As Figure 3 shows, the results we obtained for the $Min_0$ estimator, using the INSPEC queries, are very similar to those we obtained for the $Ind_0$ estimator.

To analyze how dependent the results are on the trace used, we ran our experiments using a different query trace, consisting of 2404 real-user queries.

| *Right* | $P_{Right}$ (ERIC) | $R_{Right}$ (ERIC) | $P_{Right}$ (INSPEC) | $R_{Right}$ (INSPEC) |
|---|---|---|---|---|
| *Matching* | 0.8960 | 0.4621 | 0.9126 | 0.4044 |
| $Best_0$ | 0.8498 | 0.9384 | 0.8438 | 0.9010 |

Figure 4: The average $P$ and $R$ parameters for the $Ind_0$ estimator, using the ERIC queries. The last two columns show the corresponding values for the INSPEC queries.

Real users issued these queries to the ERIC database through Stanford's FOLIO system. Figure 4 shows the results corresponding to these queries, for the different instances of the $P$ and $R$ parameters. The results obtained differ only slightly from the ones for the INSPEC queries, which suggests that our results are not sensitive to the type of trace used.

## Acknowledgments

This research was sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). This work was supported by an equipment grant from Digital Equipment Corporation. We would also like to thank Jim Davis and Daniela Rus for their helpful comments on an earlier version of this paper.

## References

[1] Michael F. Schwartz, Alan Emtage, Brewster Kahle, and B. Clifford Neuman. A comparison of INTERNET resource discovery approaches. *Computer Systems*, 5(4), 1992.

[2] Katia Obraczka, Peter B. Danzig, and Shih-Hao Li. INTERNET resource discovery services. *IEEE Computer*, September 1993.

[3] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. Precision and recall of *GlOSS* estimators for database discovery. Technical Report STAN-CS-TN-94-010, Stanford University, July 1994. Available by anonymous `ftp` from `db.stanford.edu` in `/pub/gravano/1994/stan.cs.tn.94.010.ps`.

[4] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. The effectiveness of *GlOSS* for the text-database discovery problem. In *Proceedings of the 1994 ACM SIGMOD Conference*, May 1994. Also available by anonymous `ftp` from `db.stanford.edu` in `/pub/gravano/1994/stan.cs.tn.93.002.sigmod94.ps`.

[5] Gerard Salton and Michael J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, 1983.

[6] G. Salton, E. A. Fox, and E. Voorhees. A comparison of two methods for boolean query relevance feedback. TR 84-564, Cornell University, July 1983.

| | |
|---|---|
| FILE: | /pub/widom/1994/cc-federated.ps |
| COMMENT: | Technical report |

Paper 12

# Integrity Constraint Checking in Federated Databases*


Paul Grefen
Department of Computer Science
University of Twente
7500 AE Enschede, The Netherlands
grefen@cs.utwente.nl

Jennifer Widom
Department of Computer Science
Stanford University
Stanford. CA 94305-2140 USA
widom@cs.stanford.edu



**Abstract**

We st udy the problem of monitoring integrity constraints in loosely-couplet1 federated databases. where global queries, global transaction mechanisms, and global concurrency control are unavailable. A family of constraint checking protocols for federated databases is developed and analyzed. The differences across protocols are with respect. to the requirements of the underlying systems, the level of constraint checking guaranteed by the protocols, and t tie processing and communication costs. Thus, we offer a suite of options from which a protocol can be chosen to suit the capabilities and requirements of a particular federated database application.


## 1 Introduction

The integration of multiple, autonomous database systems is fast becoming an important topir in hoth the research and commercial communities. Information servers are being developed that provide integrated access to multiple data sources. Legacy database systems and applications are being coupled to form enterprise-wide information systems. Large workflow management applications require the routing of information through multiple. autonomous local systems. The integration of autonomous database systems into loosely-coupled federations requires the development of novel database management techniques specific to these environments. Many concepts and techniques from centralized or tightly-coupled distributed databases simply are not usable in a federated architecture.

One important issue in federated datahase systems is checking integrity *constraints* over data from multiple sites in the federation. Integrity constraints specify those states of the (global) database that arc considered to he semantically valid. In a federated environment, integrity constraints might specify that replicated information is not rontradirtory. that information is not duplicated. that certain referential integrity constraints hold, or that some other condition is true over the data in multiple databases. Below, we describe an example scenario involving interconnected hospital information systems with rross-system constraints.

In centralized or tightly-coupled distributed databases, (distributed) transactions are rentral to integrity constraint cherking: Before a transaction commits. it ensures that all integrity constraints are valid. If a constraint is violated. then the transaction may he ahorted. the ronstraint may he corrected automatically, or some error condition may he raised [GA93]. Unfortunately. the lark of inter-site transaction mechanisms in federated databases renders traditional constraint cherking mechanisms unusable. This paper describes and analyzes a family of constraint checking protocols. all of which are designed for the federated datahase system environment. The differences across protorols are with respect to the requirements of the underlying systems, the level of constraint checking guaranteed by the protocols. and the processing and communication costs. By providing a family of protocols. rat her than a single protocol, we permit a protocol in the family to be rhosen and tailored for the capabilities and requirements of a particular federated datahase application.


*This work was supported at Stanford by ARPA Contract F33615-93-1-1339, by the Anderson Faculty Scholar Fund. and by equipment grants from Digital Equipment Corporation and IBM Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements of the U.S. Government.


### 1.1 Related Work

Most work addressing the prohlem of integrity constraint cherking in multidatabase environments has ronsidered tightly-coupled distributed databases in which global queries, global transactions. and global concurrency control are present? e.g. [GA90, Qia89, SV86]. Since these approaches rely on global services that typically are unavailable in federated databases. they are inappropriate for the environment we ronsider. Note that some approaches focus on relaxing the traditional notion of transaction serializability for constraints in distributed environments. e.g. [BGMS92, Elm91], hut some level of locking and global query facilities is still expected.

A few recent papers have addressed the issue of monitoring constraints in loosely-coupled. distrihuted, and sometimes heterogeneous database environments. One class of work involves *local constraint checking* deriving tests whose success over one database implies the validity of a multidatabase constraint [BGM92, GSUW94, GW93]. Local tests optimize the constraint checking process. hut they still require a conventional (non-local) method when the local test fails. As will he seen, in this paper we develop protocols that integrate local checking with non-local methods. In [CGMW94], a framework is defined for constraint management in loosely-coupled, highly heterogeneous environments. Thr fo( us in [CGMW94] IS on maintaining constraints across systems that have varying capabilities and varying "willingness" to partiripate in constraint checking protocols, on describing the timing properties associated with constraint checking, and on more "relaxed" notions of constraint consistency than we consider here.

The issue of maintaining consistency of replicated data across loosely-coupled, semantically heterogeneous databases has been considered in. e.g. [CW93, RSK91]. Our ronstraint checking prohlem can he seen as a special case (or first step) of the consistency maintenance problem. In [CW93], a method is described that relies on active rules and persistent queues. The approach is similar to the simplest in our family of protocols. Similar issues are addressed in [RSK91], hut no specific protocols are provided. A related problem is that of maintaining views over distrihuted data in loosely-coupled systems. addressed in [ZGMHW94] in the context of *data warehousing*. In [ZGMHW94], algorithms are presented for handling the anomalies that arise when materialized views are refreshed in an asynchronous manner. The algorithms in [ZGMHW94] rely on view definit ions and compensating *queries*, and t hus are not directly applicable to our problem. However, the results in [ZGMHW94] may become relevant as we extend our protocols to handle more complex constraints or to incorporate constraint repair.

Finally. we note that in the field of distributed (operating) systems there has been considerable aork in the area of *snapshots* and *consistent global* states: see e.g. [CI 85, BM93]. Although this work appears highly related to the problem we are addressing. there are two significant differences: (1 ) The conditions to he evaluated in the distributed system setting are *stable*. meaning that

once the condition becomes valid. it stays valid. This property is not true of database integrity constraints: in fart. our goal is to track constraints as they move from validity to invalidity and vice-versa. (2) Protocols for the distributed system setting are designed to ohtain some (any) global state. hut not to ohtain all global states. In contrast, to effectively monitor database ronstraints it is necessary to monitor all global states, or at least a subset of those states corresponding to consistency checkpoints (see Section 2.1).

### 1.2 Structure of the Paper

Section 2 provides preliminary concepts and definitions, the basic system architecture we address, and the class of integrity ronstraints we deal with. **In** addition, a simple example application is introduced. Section 3 is the core of the paper it presents the family of constraint checking protocols we have developed. Section 4 analyzes the family of protocols presented in Sertion 3. First, the "design space" for protocols is further inspected. Then the various protocols are compared with respert to their requirements, properties, and costs. Finally, in Section 5 we conclude and discuss future work.

## 2 Preliminaries

This section presents the necessary preliminary material for the remainder of the paper. First, the concepts used in our work are defined formally. Next, we present the basic architecture we consider: a federation of two autonomous, relational databases, each with a component for constraint checking. Next, we formally describe the class of integrity constraints we consider. This section ends with the description of an example federated database application that is used to motivate our work.

### 2.1 Concepts and Definitions

**Definition 1.** A *fedemted relational database system* $F$ is a set of $n$ interronnerted autonomous datahase systems $\{S_1 \ldots S_n\}$. Each autonomous database system $S_i \in F$ hosts a local database $D_i$ with schema $\mathcal{D}_i$. A local database $D_i$ ronsists of relations $R_1^i, \ldots, R_{n_i}^i$ with schemata $\mathcal{R}_1^i, \ldots, \mathcal{R}_{n_i}^i$. The set of all relation schemata $\mathcal{R}_j^i$ in $F$ is called the glohal datahase schema $\mathcal{G}$ of $F$. □

In the following, we assume a global clock so that we ran refer to glohal times in defining certain concepts. Note that the glohal clock is used for concept definition only it is not a requirement of the federated database systems we consider.

**Definition** 2. A *global state* $G$ of a global datahase schema $\mathcal{G}$ at global time $t$ is the set of the states of all relations in $\mathcal{G}$ at global time $t$. □

Since in prartire it ran he diffirult or impossihle to observe the state of a glohal database at a single time. an application may observe a state that has never actually existed. We call such an observed global state a *phantom state.*

**Definition** 3. A *phantom state* $\Phi$ of a global datahase srhetna $\mathcal{G}$ observed by application $A$ at time $t_2$ as a consequence of a request by $A$ at time $t_1$ is a set of states of all relations in $G$ such that there exists no $t_3$ where $t_1 < t_3 < t_2$ and the global state of $G$ at time $t_3$ is $\Phi$. □

That is, a phantom state is a state observed hy an application such that the state never actually existed between the "request for observation" and the answer.

**Definition 4.** A **global** *integrity constraint* I is a hoolean expression over a glohal datahase schema $\mathcal{G}$, i.e. a function $I$: $\mathcal{G} \rightarrow \{true, false\}$. A global integrity ronstraint cannot he expressed over a local database schema $\mathcal{D}_i \in \mathcal{G}$ (otherwise the ronstraint would he local, not global) *constraint checking protocol* for I is an algorithm for evaluating $I$. □

In a centralized database system or a distributed database system supporting glohal transactions. the transitions from one database state to another are determined by transactions: transactions are treated as atomic operations whose intermediate states have no semantics beyond the transaction. Consequently, in traditional database environments. integrity constraints generally are required to hold in the states immediately preceding and following each transaction. Since federated environments consist of multiple. autonomous database systems with no global transactions. we must rely on other concepts to determine the glohal datahase states that should satisfy the integrity constraints. We define a notion of global states in which the federated system is "at rest." These states correspond roughly to the before and after transaction states in traditional datahase systems, and they are the states in which we want to ensure that our integrity constraints are not violated.

**Definition** 5. A federated datahase system $F$ is in a *quiescent state* at time $t$ if all updates submitted before $t$ have completed. and all constraint rherking protocols triggered hy any updates before $t$ also have completed. □

Now we define two important properties of ronstraint rherking protocols: *safety* and *accuracy.*

**Definition** 6. Consider a global database schema $\mathcal{G}$ and a global integrity ronstraint $I$ over $\mathcal{G}$. A ronstraint checking protocol for $I$ is *safe* if the transition from a quiescent global database state $G_0 \in \mathcal{G}$ that satisfies $I$ to a quiescent state $G_1 \in \mathcal{G}$ that does not satisfy $I$ always results in the protocol raising an alarm.' □

That is. a protocol is safe if it detects every transition to a quiescent state in which the constraint becomes violated. We assume that safety is required of any ronstraint checking protocol that will he useful in prartire.

**Definition** 7. Consider a global datahase srhema $\mathcal{G}$ and a glohal integrity ronstraint $I$ over $\mathcal{G}$. A constraint rherking protocol for $I$ is **accurate** if. after a quiescent global database state $G_0$ at time $t_0$, a protocol-generated alartn at time $t_2$ implies the existence of a glohal database state $G_1$ at time $t_1$ such that $t_0 < t_1 \leq t_2$ and $G_1$ does not satisfy $I$. □

That is. a protorol is accurate if. whenever an alarm is raised. there is indeed a state in which the constraint is violated. Although accuracy is a desirable feature of a constraint checking protocol.

[1] In this paper we do not consider the reaction to constraint violations, although thia is an important area of future work Rather, we focus on thr detection of constraint violations, and we say that when a protocol detects a constraint violation it raises an "alarm."

Paper 13

FILE: /pub/gupta/1994/ppcp.ps
COMMENT: Appears in The Second Workshop on the Principles and Practice of Constraint Programming

# Efficient and Complete Tests for Database Integrity Constraint Checking*

Ashish Gupta
Yehoshua Sagiv†
Jeffrey D. Ullman
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
{agupta,sagiv,ullman,widom}@cs.stanford.edu

## 1 Introduction

An important feature of modern database management, systems is the automatic checking of *integrity constraints*. An integrity constraint, is a predicate or query such that if the predicate holds on a state of the data. or equivalently if the query produces an empty answer. then the database is considered *valid*. When an integrity constraint. is *violated*. i.e.. when the predicate does not hold or the query produces a non-empty answer. then the update creating the undesirable database state must be rejected or some other compensating action must be taken.

We are interested in efficient methods for checking integrity constraints (hereafter called constraints) as a database is updated. Here. general efficiency is measured both in the amount of data that needs to he accessed in order to check a constraint,. and in whether the check can he performed by submitting a query to the database system (rather than running an algorithm directly on the data). In terms of complexity. we are not interested in methods that are exponential in the size of the data. or in the number of constraints. but we are willing to accept, methods that arc exponential in the size of the constraints themselves since. in databases. constraints tend to be short,.

Suppose that we have a constraint C. and a database update occurs. We need to ensure that $C$ still holds after the update. Assume that we have available at least. the update itself and the definition of $C$. In addition to this information. there are three levels in the amount of data. we might use to check the constraint: *none*. *some*. or *all*. Using none of the data corresponds to the *query* independent of *update* problem. which has been studied in its generality in [BT88. Elk90. LS93] and with respect to constraints by us in [GSUW93]. Using all of the data amounts to efficient. evaluation of predicates or queries over the database [BC79. GMS93. HD92. Nic82. QW91. UO92]. We study the case where *some* of the data is used to check the constraint. This scenario arises whenever certain data involved in the constraint, is very expensive or impossible to access. such as in distributed database systems or collaborative design [TH93. GT93]. Hereafter we refer to the portion of the data used to check a constraint as *accessible* data. and we refer to the portion of the data involved in a constraint but not used to check the constraint. as *inaccessible* data.

*Research sponsored by NSF grants IRI-91-16646 and IRI-92-23405, by ARO grant DAAL03-91-G-0177, by ARPA contract F33615-93-1-1339, and by a grant of Mitsubishi Electric Corp.

†Permanent address: Department of Computer Science, Hebrew University, Jerusalem, Israel.

Note that. unless all of the relevant data is accessible. our constraint checking methods will be conservative. That. is. by looking at only some of the data. we may be able to determine that the constraint still holds. or we may determine that it is necessary to look at all of the data to check the constraint. A check is *correct* if. whenever it determines that a constraint still holds. then indeed the constraint holds. We also want our checks to be *complete*. but completeness is with respect to the accessible data. A check is complete in this sense if. whenever we determine that a constraint may not hold. there is some configuration of the inaccessible data for which the constraint indeed does not hold.

In the remainder of this short paper we outline the languages we have been considering for database constraints and we solidify the notion of using *some* of the data to check a constraint. We then give several examples that illustrate when and how our constraint checking methods apply. (Due to spare limitations. complete technical results are not included.) The examples serve to bring out a number of problems we have not, yet, solved. which are enumerated at the end of the paper.

## 2 Problem Definition

We consider relational databases where relations are modeled as predicates and queries are expressed as logical rules that derive a. *result* predicate. as in. e.g.. *Datalog* [Ull88]. Examples are given below. A *constraint* is expressed as a. query whose result. is a special 0-ary predicate that we call panic. If the query produces $\emptyset$ on a given database I). then the constraint holds for $D$. If the query produces panic then the constraint is violated. The difficulty of checking constraints depends on the language that we use to express constraint, queries. Examples of interesting languages for expressing constraint queries are:

1. Conjunctive queries [CM77].
2. Nonrecursive Datalog, or unions of conjunctive queries [SY80].
3. Conjunctive queries with arithmetic comparisons [Klu88].
4. Datalog with negation [Ull88].
5. Recursive Datalog, possibly with arithmetic comparisons and/or negation and/or arithmetic operators [Ull88].

For some combinations of a language above and an amount of information used (*none*, *some*, or *all*), the constraint, checking problem can be reduced to other problems that have been studied in the literature: see [GSUW93] for a discussion. For instance. conjunctive query containment results ran be used to check constraints when only updates and constraint definitions are used [LS93].

In this paper we focus our discussion on constraints expressed as conjunctive queries with arithmetic comparisons. we suppose the only accessible relation is the updated relation. and we consider updates that are insertions of a single tuple. The general form of a conjunctive query constraint, is:

$$\textbf{panic} : - l \,\&\, r_1 \,\&\, \ldots \,\&\, r_n \,\&\, c_1 \,\&\, \ldots \,\&\, c_k$$

Here. l is the predicate for which the corresponding relation $L$ is accessible. the relation $R$, for each of the $r_i$'s is inaccessible. and each $c_i$ is an arithmetic comparison involving one of $<, \leq, >, \geq, =$.[1]

[1]The use of $L$ and $R$ refers to the fact that, in distributed databases, the "Local" data is accessible and the "Remote" data is inaccessible.

Let tuple $t$ be inserted into relation $L$ and assume constraint $C$ holds before the insertion. We want to use $L$, $C$, and $t$ to infer that $C$ is not violated after the insertion. We derive a condition that relation $L$ needs to satisfy in order for $t$ not to violate $C$. We refer to this condition as the test condition. If the test condition is satisfiable, then relations $R_1, \ldots, R_n$ do not need to be accessed. The test condition is obtained by reducing the problem outlined above to the problem of checking if a conjunctive query is contained in a union of conjunctive queries; details are in [GSUW93].

## 3 Examples

EXAMPLE 1. Consider an employee-department relational database with two relations:

emp($E$, $D$, $S$) % *employee number $E$ in department $D$ has salary $S$*
dept($D$, MS) % *some manager in department $D$ has salary MS*

Let the constraint assert that every employee earns less than every manager in the same department. This constraint is expressed as a conjunctive query $C$ such that if $C$ **produces panic** then the constraint is violated:

**C: panic** : emp($E, D, S$) & dept($D, MS$) & $S \geq MS$.

Let relation EHP for predicate emp be accessible and relation DEPT be inaccessible. Suppose tuple emp($e$, $d1$, 50) is inserted into relation **EMP**. Constraint $C$ will be violated if department $d1$ has a manager whose salary is $\leq$ 50. However, suppose department $d1$ already has an employee who.. salary is 100. Since constraint, $C$ is not, violated before the insertion, we can infer that, no manager in $d1$ earns as little as 100, and therefore emp($e$, $d1$, 50) does not violate constraint, $C$.

The above inference procedure can be formalized by specifying a test condition on the relation **EMP** and the inserted tuple, such that if **EMP** satisfies the test condition, then the inserted tuple does not violate the constraint. For constraint $C$, the test condition is the following Datalog query that derives `insertion-ok` if and only if the inserted tuple does not violate $C$, independent of the value of relation DEPT.

`insertionok` : `inserted`($E, D, S$) & emp($X, D, Y$) & $Y \geq S$.

Relation INSERTED contains only the inserted tuple and EHP does not, contain the inserted tuple. This test is complete with respect, to the accessible data, as defined in Section 1. □

Note, the test condition in Example 1 is a single Datalog rule and was derived without considering the actual value of the inserted tuple. We now give two examples that illustrate the complexity that simple arithmetic comparison operators $<, >, \leq, \geq$ introduce. Example 2 shows that the complete test could be a recursive Datalog program. The constraint, in Example 3 also has a complete test in the form of a recursive Datalog program, but illustrates the computational complexity of evaluating the test.

EXAMPLE 2. We shall refer to this example as *forbidden intervals*.

**C: panic** : l($X, Y$) & r(Z) & $X \leq Z \leq Y$.

Each pair in the accessible relation L can be thought of as the ends of an interval that no $Z$ in the inaccessible relation R may occupy.

Suppose relation L has the tuples (3, 6) and (5, 10). The tuples of relation R that violate the constraint given tuple l(3, 6) lie in the interval [3, 6] and similarly, the tuples of relation R that violate the constraint given tuple l(5, 10) lie in the interval [5, 10]. If the constraint is not violated then we can infer that the tuples of the inaccessible relation lie outside the *forbidden intervals* [3, 6] and [5, 10] and therefore outside the combined forbidden interval [3, 10].

Let tuple l($a$, $b$) be inserted into relation L. If $a \geq 3$ and $b \leq 10$, then the forbidden interval for l($a$, $b$) is contained in the union of the forbidden intervals of one or more existing tuples, and relation R need not be accessed in order to infer that constraint $C$ is not violated. Note, the complete test may need to access multiple existing tuples in order to make the above inference. An incomplete, but sufficient, test would be to check that the forbidden interval for some single existing tuple contains the forbidden interval for the inserted tuple. That corresponds to using a single tuple in the accessible relation as opposed to using an arbitrary number of tuples, and was the approach taken in our initial work [GW93]. The sufficient test is linear in the number of tuples in L whereas the complete test could be exponential, if implemented naively. With some preprocessing, the complete test can also be evaluated in linear time [GSUW93].

The complete test for this example is the following recursive Datalog program that derives `insertionak` if and only if the inserted tuple does not, violate $C$, assuming that $C$ was not violated before the insertion.

`insertionak` : `inserted`($A, B$) & `forbidden_int`($C, D$) & $A \geq C$ & $B \leq D$.
`forbidden_int`($C, D$) : l($C, D$).
`forbidden_int`($C, D$) : `forbidden_int`($C, X$) & `forbidden_int`($Y, D$) & $X \geq Y$.

□

EXAMPLE 3. Consider a constraint $C$ that, involves two variables in the inaccessible relation:

$C$: **panic** : l($U, V, W, Z$) & r($X, Y$) & $U \leq X \leq V$ & $W \leq Y \leq Z$.

Intuitively, the above constraint is the *forbidden interval* constraint in two dimensions. A tuple in relation R defines a point in a two dimensional space and a tuple in relation L defines a rectangular region in this 2-D space. Constraint $C$ requires that all the points defined by the inaccessible relation R lie outside every rectangular region defined by the accessible tuples. Therefore, an inserted tuple l($a, b, c, d$), does not violate $C$ if the rectangle defined by l($a, b, c, d$) is contained in the union of the rectangles defined by the existing tuples in L. The test for determining when a w-tangle is contained in a set of other rectangles can still be represented as a recursive Datalog program. However, building the program is not as straightforward as in Example 2. In addition, the complexity of the test is high even with preprocessing. Without preprocessing the test is exponential in the number of tuples in L □

## 4 Discussion

In [GSUW93] we identify some subclasses of conjunctive query constraints with arithmetic comparisons for which the complete test is a (recursive) Datalog program. We also identify some subclasses where the complete test does not need to consider multiple tuples from the accessible relation, but can consider tuples one at a time (i.e there is no need to consider combinations of tuples, as in Example 2). The test condition for conjunctive query constraints including the subclasses, is NP-complete. However, in at least some cases, the exponential behavior is only in the size of the constraint specifications, which we believe will be relatively small. In other cases the tests may be exponential in the size of the database or the number of constraints in the system. In such cases sufficient tests, instead of complete tests, may be preferable.

For conjunctive query constraints that use function symbols like +, − (instead of only arithmetic-comparisons) the complete test is an implication condition where both sides of the implication use

FILE: /pub/hammer/1994/hicss.ps
COMMENT: In Proceedings of the 27th Hawaii International Conference on System Sciences, pages 389-407, Computer Society of the IEEE, University of Hawaii, Hawaii, USA, January 1994

Paper 14

# An Intelligent System For Identifying and Integrating Non-Local Objects In Federated Database Systems*

Joachim Hammer, Dennis McLeod and Antonio Si

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA

## Abstract

*Support for interoperability among autonomous, heterogeneous database systems is emerging as a key information management problem for the 1990s. A key challenge for achieving interoperability among multiple database systems is to provide capabilities to allow information units and resources to be flexibly and dynamically combined and interconnected while at the same time, preserve the investment in and autonomy of each existing system. This research specifically focuses on two key aspects of this: (1) how to discover the location and content of relevant, non-local information units, and (2) how to identify and resolve the semantic heterogeneity that exists between related information in different database components. We demonstrate and evaluate our approach using the Remote-Exchange experimental prototype system, which supports information sharing and exchange from the above perspective.*

## 1 Introduction

The past several decades have witnessed the emergence of powerful general-purpose database management facilities, as well as the proliferation of databases constructed with those facilities. A central current problem in database research is to support the effective sharing and exchange of information among various database systems, while at the same time respecting the autonomy of those systems. Further, recent advances and wide acceptance of communication networks have provided foundational mechanisms to support the interconnection of related existing database systems. Such a collection of cooperating but heterogeneous, autonomous database systems may be termed a *federated database systems* or *federation* for short. each individual database system in a federation is termed a *component database system* or *component*.

While the interoperability problem is largely addressed in the communication network environment [22], it only provides limited support for the interoperation of heterogeneous database systems. A main reason for this limitation stems from the difficulties in overcoming the problem of unifying heterogeneous information. This is commonly known as the *semantic heterogeneity problem*[9].

In light of this observation, various architectures have been proposed to address the database interoperability problem. Such architectures range from tightly-coupled composite approaches in which component databases are integrated into a centralized global database [2], to loosely-coupled federated environments wherein information is shared among component database systems while retaining their autonomy [11]. A key challenge for all these environments is to provide capabilities to allow information units and resources to be flexibly and dynamically combined and interconnected, while at the same time preserving the investment in and autonomy of each individual database component.

The *Remote-Exchange* federated architecture and experimental system extends the traditional federated architecture in supporting two key aspects of this. (1) how to discover the location and content of related, non-local information units (simple objects, types of object, units of behavior, etc.), and (2) how to identify and resolve the semantic heterogeneity that exists between related information in different database components. The goal of this paper is to demonstrate the architecture of this system and explain key implementation issues faced. Our approach to the two key problems cited above can be utilized by a variety of intelligent and cooperative information systems (ICISs) [17, 19].

The remainder of this paper is organized as follows. In Section 2, we review related work. In Section 3, we introduce a typical sharing scenario in a federated database environment, wherein individual components can share and exchange information units (objects). Section 4 describes the architecture of the Remote-Exchange experimental system in detail and illustrates the ideas of our approach to information sharing. In Section 5, we examine key issues in the design and implementation of the Remote-Exchange prototype using a collection of Omega-based database systems [8]. Finally, Section 6 contains concluding observations with a critical evaluation of our results and their potential impact on future work.

*This research was supported in part by NSF grant IRI-9021028.

## 2 Related Work

The term "heterogeneous databases" was originally used to distinguish work that included database model and conceptual schema heterogeneity from work on "distributed databases"[1] which addressed issues solely related to distribution.[1] Recently, there has been a resurgence in research in the area of heterogeneous database systems (HDBSs). Work in this area can be characterized by the different levels of integration of the component DBSs and by different levels of global (federation) services. In Multibase [2], for example, which is considered a tightly-coupled HDBS, component database schemas are integrated into one centralized global schema with the option of defining different user views on the unified schema. While this approach supports pre-existing component databases, it falls short in terms of flexible sharing patterns. Furthermore, the integration process is expensive and difficult, and tends to be hard to change.

The federated architecture proposed in [11], which is similar to the multidatabase architecture of [16], involves a loosely-coupled collection of database systems, stressing autonomy and flexible sharing patterns through inter-component negotiation. Rather than using a single, static global schema, the loosely-coupled architecture allows multiple import schemas, enabling data retrieval directly from the exporter and not indirectly through some central node as in the tightly-coupled approach.

One common approach to schema integration is to reason about the meaning and resemblance of heterogeneous objects in terms of their structural representation. In Larson et al. [15], the meaning of an attribute is approximated in terms of its value type (set of possible values), cardinality constraints, integrity constraints, and allowable operations. However, one can argue that any such set of characteristics does not sufficiently describe the real-world meaning of an object, and thus their comparison can lead to unintended correspondences or fail to detect important ones. Other promising methodologies that have been developed include heuristics to determine the similarity of objects based on the percentage of occurrences of common attributes [10, 23]. More accurate techniques use classification for choosing a possible relationship between classes [21].

Whereas most of previous methods primarily utilize schema knowledge, techniques utilizing semantic knowledge (based on real-world experience) have also been investigated [5, 12]. These approaches usually assume the existence of a real world knowledge base which serves as a global schema to which every local schema is mapped. However, we believe that a useful approach is for the centralized knowledge base to only contain information actively used to support sharing within the federation and thus, as illustrated in our mechanism, should be incrementally built tailored to the federation. Furthermore, these approaches lack the ability of tailoring the identification process to the context of user request.

A different approach proposed by Kent [14] uses an object-oriented database programming language to express mappings among different similar concepts that allow a user to view them in some integrated way. It of course remains to be seen if a language that is sophisticated enough to meet all of the requirements given by Kent in his solution can be developed in the near future.

A very recent approach to interoperability by Mehta et al. [18] uses so-called path-methods to explicitly create inter-component and inter-object mappings between source and target object classes in order to retrieve and update related data objects. The obvious drawback of this approach is the large overhead in calculating and maintaining the mappings, which may be impractical for large federations with extensive and/or dynamic sharing patterns.

## 3 The Federated Database System Context

Consider the following scenario involving a group of molecular biologists whose databases form a collaborative Federation Of Macromolecular Databanks (FOMD), as depicted in Figure 1. The goal of FOMD is to share and exchange macromolecular information between individual institutes and researchers [7, 20]. For instance, a researcher could efficiently reuse protein data that had already been sequenced (decoded) by other components for his/her experimental work.

Figure 1 also shows a snapshot of the information stored in FOMD at a given point in its lifetime. Since each macromolecular database is managed by a different organization (institute), the contents of their databases reflect their different foci and interests. We can see, for example, that institute *B* is the only component with information on both genetic and protein sequences. All other components maintain either genetic or protein information with various levels of detail.

A difficulty in finding a solution to the problem of achieving interoperability in FOMD and similar federations stems from the conflicting nature of sharing and autonomy. On the one hand, an institute would like to share information with other components of FOMD. On the other hand, the same component would also like to exercise autonomy over its own database with respect to organization, administration and sharing, e.g., control over the information it is willing to "export" to the other components[2]. In consequence, we assume that when a component agrees to join the federation, information to be made available to other institutes is stored in a specially "marked" schema called an *export schema*.

The level of interoperability that can be achieved within such a federation depends largely upon two key

[1] The term distributed database is used here as it has been mainly used in the literature, denoting a relatively tightly-coupled, homogeneous system of logically centralized, but physically distributed component databases.

[2] This capability of exporting only a specific portion of a component's database is especially important in the FOMD environment where a research institute would probably not willing to release any information that has not been published or fully validated.

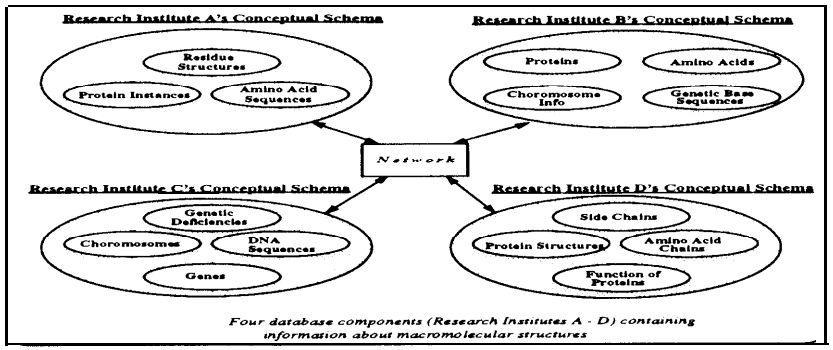



Figure 1: A conceptual overview of FOMD and its components

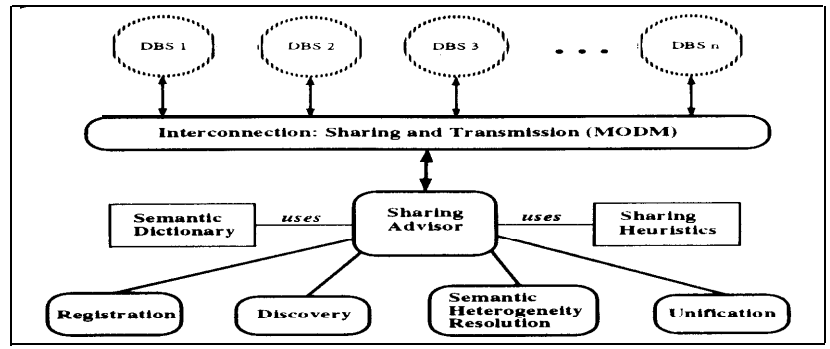



Figure 2 The Remote-Exchange architecture

capabilities:

1. the ability of a component to identify and locate potentially appropriate non-local information with respect to its needs (the *discovery problem*); and

2. as requested remote information is identified and located, the ability to fold it into the local system framework (the *unification problem*).

The focus of this work is to address the above two problems; several other important issues such as security (access control) and automatic update of shared data are not directly addressed here.

## 4 Achieving Interoperability in Rernote-Exchange

In order for any coll aboration to take place among the heterogeneous components of a federation, some common model for describing the sharable data must be utilized. One may of course argue as to the nature of this "lingua franca". We believe that this model should be semantically expressive enough to capture the intended meanings of conceptual schemas which may reflect essential kinds of heterogeneity (as will be discussed below). Further, this model must be simple enough so that it can be readily understood and implemented. To this end, we have chosen to use a Minimal Object Data Model (MODM) as the common database model for describing the structure, constraints, and operations for sharable data.

### 4.1 MODM

MODM is a generic functional object database model, which supports the usual object-based constructs. In particular, it draws upon the essentials of functional database models, such as those proposed in Daplex, Iris, and Omega [8]. MODM contains the basic features common to most semantic and object-oriented models [13]. The model supports complex objects (aggregation), type membership (classification), subtype to supertype relationships (generalization), inheritance of properties (attributes) from supertype to subtypes, and user-definable functions (methods).

In the context of MODM, interoperation among components, generally, is possible at many different levels of abstraction and granularity, ranging from factual information units (data objects), to meta-data (conceptual schema), to behavior. For this paper, we limit our investigation to the sharing of type objects, a process we term *type-level* sharing. Sharing of individual instances (*instance-level* sharing) and sharing of behavior or functions (*function-level* sharing) are preliminarily examined in [4].

### 4.2 Remote-Exchange Experimental System

Figure 2 illustrates the architecture of the Remote-Exchange experimental system. In this figure, the *sharing advisor* is a special component that manages knowledge about existing type objects that components export; this knowledge resides in the *semantic dictionary*. The sharing advisor provides four "intelligent" services to the components of the federation: *Registration*, *Discovery*, *Semantic Heterogeneity Resolution* and *Unification*. These services address different aspects of the problem of type-level sharing, ranging from detecting similarity during the registration process to determining precise relationship between objects during the unification process. While it is nearly impossible to completely automate these services, our approach provides substantially useful functionality.

### 4.3 Registration

Registration allows a (new) component to inform the sharing advisor about any information it is willing to share with other components in the federation. In a sense, it establishes an initial sharing context within the federation by logically connecting the exported information to the semantic dictionary via the sharing advisor. Incremental registration allows a component to augment its export schema with new information.

Figure 3 shows two partial conceptual schemas of component *A* and component *D*, respectively. Let us assume that component *A* has already registered three type objects **Protein Instances** and **Amino Acid Sequences** with thr sharing advisor. This is reflected in the semantic dictionary shown in Figure 4a. When component *D* registers type object **Protein Structures**, the sharing advisor can use its existing knowledge (in this case the information obtained from component *A*) to determine if **Protein Structures** has any similarities with previously registered types, i.e., **Protein Instances** and **Amino Acid Sequences**. User interaction may be necessary to instruct the sharing advisor in case the information in the semantic dictionary is insufficient to detect (dis)similarities among type objects automatically. The newly acquired knowledge and the newly registered information are stored in the semantic dictionary for future consultation; see, for example, contents of thr semantic dictionary in Figure 4b after the registration of the type object **Protein Structures**. In a sense, the semantic dictionary represents a dynamic *federated knowledge base* about sharable information in the federation.

In the remainder of this section, we illustrate how the sharing advisor can effectively utilize thr semantic dictionary as a means of organizing various registered type objects in order to accommodate information sharing. We also introduce a set of sharing heuristics that guide the sharing advisor through registration.

#### 4.3.1 The Semantic Dictionary

In the semantic dictionary, types determined to be similar by the sharing advisor are classified into a collection called a *concept*, within which subcollections called *subconcepts* can be further classified. This generates a *concept hierarchy*. Naturally, the relationships expressed in this concept hierarchy can only be approximations of the true, real-world relationships that rxisl between the exported types of different components; additional mechanisms are needed to establish more exact relationships (see Section 4.5).

Figure 4 shows two snapshots of thr concept hierarchy in the semantic dictionary taken at different times during the life-time of our example federation. Figure 4a indicates the concept hierarchy after types **Protein Instances** and **Amino Acid Sequences** of component *A* have been registered. Figure 4b shows the corresponding hierarchy after component *D* has registered type **Protein Structures**. The hierarchy in Figure 4b also indicates that **Protein Instances** and **Protein Structures** are representing similar information, since they belong to the same classification, called **Protein Information**. In addition, types **Protein Instances** and **Protein Structures** have properties that distinguishes one type from another. Hence, they are also created as subconcepts of **Protein Information** in order to express their dissimilarities. By contrast, **Amino Acid Sequences** is similar to neither **Protein Instances** nor **Protein Structures**, and thus appears as a separate concept in the hierarchy. Similarity and dissimilarity of this kind is detected by the sharing advisor based upon sharing heuristics (described below), with user input as required.

The advantage of organizing type objects in a concept hierarchy is that a hill climbing technique can be used to place newly registered type objects. For example, consider a type object being registered which is determined to be unrelated to members of concept **Protein Information** in Figure 4b; in this case, no further comparisons with members of subconcepts of **Protein Information** are necessary.

Generally, a type object represents a specific view

FILE: /pub/harinarayan/1994/icdt.ps
COMMENT: To appear in ICDT 1995

# Optimization Using Tuple Subsumption

Venky Harinarayan[1] and Ashish Gupta[2]

[1] Department of Computer Science, Stanford University, CA 94305-2140
(venky@cs.stanford.edu)
[2] Department of Computer Science, Stanford. *and* IBM Almaden Research Center

**Abstract.** A tuple $t_1$ of relation $R$ *subsumes* tuple $t_2$ of $R$, with respect to a query $Q$ if for every database, tuple $t_1$ derives all, and possibly more, answers to query $Q$ than derived by tuple $t_2$. Therefore, the subsumed tuple $t_2$ can be ignored with respect to $Q$ in the presence of tuple $t_1$ in relation $R$. This property finds use in a large number of problems. For instance: during query optimization subsumed tuples can be ignored thereby avoiding the computation of redundant answers; the size of cached information in distributed and object oriented systems can be reduced by omitting subsumed tuples; constraints need not be checked and rules need not be recomputed when provably subsumed updates are made. We give algorithms for deciding efficiently when a tuple subsumes another tuple for queries that use arbitrary mathematical functions. We characterize queries for which, whenever a set of tuples $\mathcal{T}$ subsumes a tuple $t$ then one of the tuples in $\mathcal{T}$ also subsumed $t$, yielding efficiently verifiable cases of subsumption.

## 1 Introduction

We discuss and formalize the property of *subsumption* in this paper. Subsumption, intuitively, is the identification of the tuples of a database that do not "contribute" to the result of a query. Subsumption is a powerful optimization technique for a variety of problems. In the introduction we give some examples to illustrate and motivate the use of subsumption. The first example illustrates the use of subsumption to optimize integrity constraint checking.

*Example 1.* This example is from [GW93]. Consider an employee-department relational database with two relations:

**EMP**$(E, D, S)$ % *employee number $E$ in department $D$ has salary $S$*
**DEPT**$(D, MS)$ % *some manager of department $D$ has salary $MS$*

Consider a constraint on the database which asserts that every employee earns less than every manager in the same department. This constraint is expressed as a conjunctive query $C$ [Ull89] such that if $C$ derives **panic** the constraint is violated:

$C$: **panic** :- **emp**$(E, D, S)$ & **dept**$(D, MS)$ & $S \geq MS$.

We use upper case letters to refer to the relation corresponding to a particular predicate. For instance, **EMP** refers to the relation corresponding to predicate **emp**. Suppose tuple **emp**$(e1, d1, 50)$ is inserted into relation **EMP**. Constraint $C$ will be violated if department $d1$ has a manager whose salary is $\leq 50$. However, suppose department $d1$ already has an employee whose salary is 100. Since constraint $C$ is not violated before the insertion, we can infer that no manager in $d1$ earns as little as 100, and therefore **emp**$(e1, d1, 50)$ does not violate constraint $C$. We say tuple **emp**$(e2, d1, 100)$ subsumes tuple **emp**$(e1, d1, 50)$ with respect to constraint query $C$.

Consider a scenario where the relation **DEPT** is expensive to access or not accessible at all. Let tuple $\mu$ be inserted into **EMP**. If some existing tuple in **EMP** subsumes $\mu$ then constraint $C$ can be checked using only relation **EMP** without accessing **DEPT**.

[GW93, GSUW94] built a theory that uses subsumption to verify integrity constraints that are expressable as Select-Project-Join statements that use arithmetic inequalities. The techniques developed there often lead to more efficient constraint checking than naive strategies that do not exploit subsumption. This is especially true in distributed systems and heterogeneous systems where some relations may be very expensive or impossible to access. However, the results in those papers do not extend to queries that use arbitrary mathematical expressions. In addition, in this paper we characterize subsumption in a noncomputational way that yields insight into the problem and opens new avenues for identifying classes of constraints and updates for which subsumption holds. Now we illustrate other applications of subsumption.

*Example 2.* Consider a distributed database where the relation **EMP** is on site 1 and **DEPT** is on site 2. Let site 2 use view **baddept** defined as follows:

$C$: **bad_dept**$(D)$ :- **emp**$(E, D, S)$ & **dept**$(D, MS)$ & $S \geq MS$.

Let site 2 cache the relation **EMP** for answering queries on view **baddept**. If relation **EMP** has tuples **emp**$(e1, d1, 100)$ and **emp**$(e2, d1, 50)$ then both tuples need not be cached in order to answer queries on view **baddept**. In particular emp$(e2, d1, 50)$ can be dropped from the cache. In general, all subsumed tuples can be dropped from the cache resulting in a smaller cached relation. The tuples that are not subsumed by any other tuple constitute the *representative* relation. The representative relation for emp will be referred to as **emp**$^{\mathcal{R}}$. Additionally, if relation **EMP** is updated on site 1 and if a subsumed tuple is inserted or deleted, then the cache on site 2 need not be updated!

*Example 3.* Consider view **baddept** defined in Example 2 Note, the view has the same body as the constraint in Example 1 except that the head has arity 1 anti not 0. Just as in Example 1, it is straightforward to observe that tuple **emp**$(e1, d1, 100)$ subsumes tuple **emp**$(e2, d1, 50)$ with respect to view **baddept**. Hence, if tuple **emp**$(e2, d1, 50)$ is inserted into relation **EMP** and if the relation has a subsuming tuple, then the view update decision can be made without accessing relation **DEPT**.

Finally, tuple subsumption can be used in query optimization. The notion of subsumption can be pushed into a query at optimization time. Hence, each

relation can be reduced to its representative relation before the relation participates in a join. Techniques currently used, like pushing selections down the query tree and the use of semi-join algorithms are instances of this general concept of subsumption. We can do more: in Example 3 the property of subsumption can be used to replace relation $\mathtt{EMP}(E, D, S)$ by

Define $\mathtt{emp}^r$ as $\mathbf{select}\ D, \mathbf{max}(S)\ \mathbf{from}\ \mathtt{emp}\ \mathbf{groupby}(D)$

and to replace relation $\mathtt{DEPT}(D, MS)$ by

Define $\mathtt{dept}^r$ as $\mathbf{select}\ D, \mathbf{min}(MS)\ \mathbf{from}\ \mathtt{dept}\ \mathbf{groupby}(D)$ .

The representative relations can be used instead of the original relations, to compute baddept. Note, the representative relations $\mathtt{emp}^r$ and $\mathtt{dept}^r$ need not be explicitly defined or materialized; rather the aggregation can be pushed into the original query yielding an alternative query plan for the optimizer to choose from. Thus, subsumption based query rewriting may *introduce* aggregate predicates in a query that originally did not use aggregation. This rewrite can be done automatically using the theory of subsumption. It has been shown in [CS94] that pushing aggregate subgoals down a query tree often results in more efficient plans than the original plan. Thus, we have reason to believe that introducing aggregate subgoals can result in significant savings; often reducing a $O(n^2)$ time query to an $O(n)$ query.

We formally define subsumption later in the paper. For now it is sufficient to define it as follows: consider a query $Q$ on a database consisting of a know-n set of tuples (t;) in relation $R$ and some unknown relations $S$. Now let us augment the relation $R$ with a tuple $t$. If the answer to query $Q$ does not change, independent of the values of the relations $S$, we say that tuple $t$ is subsumed by the sat of tuples $\{t_i\}$ with respect to query $Q$.

*STS Property* Consider a query $Q$ and tuple $t$ such that the following property holds: $t$ is subsumed by a set of tuples $\{t_i, 1 \leq i \leq n\}$ if and only if $t$ is subsumed by one of the tuples $t_j$, $1 \leq j \leq n$ in the set. For such cases we say that single tuple subsumption is the *strongest* possible check and that the tuple $t$ has the STS property with respect to query $Q$. Or in the language of [GSUW94] we say that single tuple subsumption is the "complete local check"[3]. In the examples given earlier all the tuples had the STS property with respect to the queries considered. The following example is an instance where the STS property does not hold. In such cases we say that multiple tuple subsumption (MTS) holds.

*MTS Property* Consider a database that stores points (on the number line) in relation P and line segments in relation L. Let view free-point contain those points that are not part of any line segment. Let a line segment $l1$ he inserted into relation L. If l1 is a subsegment of some other segment l2 that is already in relation L then we can infer that $l2$ subsumes $l1$ with respect to view freepoint and that the view stays unchanged. We can make the same inference if $l1$ is a subsegment of (is subsumed by) the union of *two* line segments l2 and $l3$. In general, $l1$ could be a subsegment of the union of an arbitrary number of existing lines segments without being a subsegment of any one of them. Note, single tuple subsumption is a degenerate case of MTS and is always a *sufficient* check though possibly not the strongest (*complete*) check. The STS property refers to cases where single tuple subsumption *is* the complete check, *i.e.*, the "if and only if" condition stated before holds.

Inferring subsumption by multiple tuples is computationally more expensive than inferring subsumption by a single tuple: checking for MTS may be exponentially more expensive than checking for STS. More importantly, the MTS check may not be expressable in the query language of the database. For instance, for the point-line example above, the general MTS check may use an arbitrary number of tuples and thus checking MTS may involve iterating/recursing over the entire database. Such a check cannot be expressed in a first order language and thus cannot be written as a SQL query (unless some recursive or iterative construct is used). On the other hand, a single tuple check (be it just sufficient or complete) can be written as a Angle SQL query. We believe STS will lead to etticient optimizations for the applications given earlier.

Thus it is useful to identify classes of queries for which the STS property holds and can be evaluated efficient ly.

We consider integrity constraints, 0-ary queries and for t hese queries we char acterize STS in terms of the existence of a "distinguished point". This characterization is a powerful result as it reduces the problem of determining if a k-dimensional unsafe region is contained in a union of $m$ unsafe regions to the problem of determining if a *point* in k-dimensions is contained in one of $m$ k-dimensional regions.

The above characterization enables us to itlentify classes of queries that use arbitrary mathematical functions as subgoals and for which it is computationally feasible to check STS. The existential characterization holds for a larger class of constraints than those for which w-e can compute the distinguished point [GSUW94] considered the problem of checking the STS property for Select-Project-Join statements that use arithmetic comparison operators $<, \leq, >, \geq, =$. The results in [GSUW94] are an example of the more powerful abstraction of the distinguished point, introduced in this paper. Also [GSUW94] considered the restricted case where the STS property holds for a given constraint but does not consider constraints where the STS property may hold for a constraint and a particular tuple but not for the same constraint and some other tuple.

The results of this paper enable subsumption to be checked for queries that use functions like distance, volume, and other more complicated mathematical functions. Such functions appear often in constraint and query specifications [TH93]. Therefore, we need a general theory that deals with subsumption in the presence of arbitrary arithmetic constraints. For instance, consider the following example:

*Example 4.* Consider a factory floor with a robotic arm that is hinged at one end and rotates in a two dimensional plane. The free end of the arm defines a circular locus. However, the arm does not have 360° rotational freedom. The

[3] *Completeness* is formally defined in [GSUW94]

Paper 16

FILE: /pub/harinarayan/1994/sigmod.ps
COMMENT: Technical report

# Generalized Projections: a Powerful Query-Optimization Technique *

Venky Harinarayan Ashish Gupta†

**Abstract**

In this paper we introduce *generalized projections* (GP). GPs capture aggregations, groupbys, conventional projection with duplicate elimination (**distinct**), and duplicate preserving projections. We develop a technique for pushing GPs down query trees of Select-project-join queries that may use aggregations like **max**, **sum**, *etc.* and that use arbitrary functions in their selection conditions. Our technique pushes down to the lowest levels of a query tree aggregation computation, duplicate elimination, and function computation. The technique also *creates* aggregations in queries that did not use aggregation to begin with. Our technique is important since applying aggregations early in query processing can provide significant performance improvements. In addition to their value in query optimization, generalized projections unify set and duplicate semantics, and help better understand aggregations.

## 1 Introduction

A fundamental problem in query processing is deciding when to drop rows (tuples) and columns (attributes) of relations that do not, contribute to the final result of a query. Query optimizers try to drop irrelevant rows and columns as early as possible in query processing. Removing irrelevant. tuples and attributes early leads to smaller intermediate relations thus reducing the cost of the query. However there is also a. cost associated with discovering and discarding irrelevant tuples and attributes. Query optimizers in commercial systems handle this trade-off in many ways. Some optimizers seek to minimize cost by using heuristics. others by exhaustively evaluating numerous alternatives in a cost based manner [S79]. Yet others use a combination of heuristics and cost based evaluation. Techniques which remove irrelevant tuples and attributes early are thus of considerable importance in query processing. A commonly used method for removing irrelevant tuples early is to push selections down the query tree [Ull89]. The further down a. tree we can push a selection. the earlier in processing we discard irrelevant tuples. Thus far. pushing projections down a query tree has been used only in removing irrelevant attributes early in query processing. In this paper, we show how projections can be used to remove irrelevant tuples as well and summarize the relevant information contained in a relation.

We introduce the notion of a *generalized projection* that unifies duplicate eliminating projections (corresponds to the SQL **distinct** adjective). duplicate preserving projections. groupbys. and aggregations. in a common framework. We develop an algorithm for pushing GPs down query trees. 'Thus. we are able to push duplicate elimination. aggregation. and duplicate preservation in a uniform way resulting in query execution strategies that not derivable using existing optimization techniques. Though selections and generalized projections both eliminate tuples they do so in different ways. A selection compares each tuple of the relation with the selection predicate and discards those that do not satisfy the selection predicate. A generalized projection does not work at the level of tuples but rather at the level of a relation. The end result though is similar: both can reduce the size of relations considerably and can thus give big performance gains if applied early in query processing. As we shall see in the following sections. a generalized projection can be expressed using SQL aggregation-groupby operators. Therefore. by pushing GPs down trees. we are able to create aggregate subqueries in queries that did not originally use aggregation.

The examples below give an indication of the wide range of SQL queries to which our technique applies and illustrate both pushing down and creation of aggregations. The optimized query trees in the examples are produced using the algorithm given in this paper.

**EXAMPLE 1.1** Consider the following schema that models an automobile manufacturer's database

```
cost(M#, CP)                    % "$CP" is the base cost price for model "M#"
factor(M#, State, Factor)       % Model "M#" has a overhead of "Factor" in "State"
sales(Vid, M#, SP, Dealer, State)
% car with ID "Vid" and model "M#" was sold in "State" by "Dealer" for a sales price "$SP"
```

The underlined columns state the key for each relation. On the above schema. consider a query $Q1$ that computes for each model the profit made by the car manufacturer. The following query tree represents this query. The topmost. projection. not shown. outputs $M\#$ and the corresponding value for [sum($SP$) – sum($CP$* ***Factor)]***.

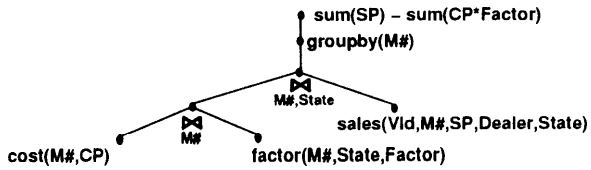



Figure 1: Find profit for company using query $Q1$

In the left branch of query $Q1$ relations **factor** and **cost** are joined to associate the multiplicative factor for each state with the base cost price of the appropriate model. The resulting intermediate relation $R(M\#, State, CP, Factor)$ is joined with relation sales and the total revenue is computed by summing ***SF.*** The total cost incurred by the manufacturer is computed by summing $CP*Factor$ for each car sold.

The query tree of Figure 1 does the aggregation step after all the joins are done. This is the normal way of doing aggregations in relational systems. A better option for evaluating $Q1$ is first to compute the revenues in each state for each model by aggregating relation sales over $(M\#, State)$. The resulting aggregate relation would be much smaller than the initial sales history table because there are far fewer states and model numbers than the total number of cars sold. The query tree for the rewritten query is in Figure 2.

Doing an early aggregation helps reduce the size of (and time taken in) subsequent joins and thus reduces the overall execution cost of, the query.

Now we illustrate how aggregations can be created as a result of pushing GPs. Consider query $Q2$ executed by the market research department of the car manufacturer $Q2$ finds all those models

*Work was supported by NSF grants IRI-91-16646 and IRI-92-23405, by ARO grant DAAL03-91-G-0177, and by Air Force Grant F33615-93-1-1339 Authors' address: Department of Computer Science, Stanford University, Stanford, CA 94305-2140. Email: {venky,agupta}@cs.stanford.edu

†Current address: IBM Almaden Research Center, San Jose

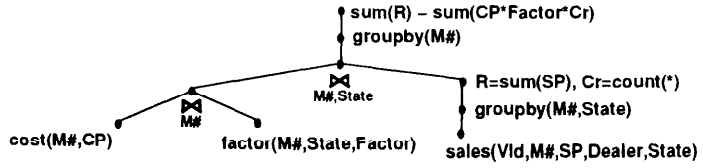



Figure 2: Query to find profit after pushing down aggregations

whose cost price after factoring in the state overhead. is more than 85% of the sales price of some car of that model sold in that state. Such models are considered low-profit.. Query $Q2$ is represented by the following query tree:

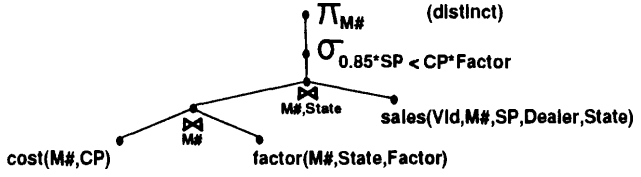



Figure 3: Find all low-profit models using query $Q2$

Query $Q2$ uses no aggregation. However, aggregation can be introduced at. very low levels in the query in order to reduce the size of joins. Let us see how relation sales might be aggregated without affecting the answer of query $Q2$. The point to note is that all tuples of sales are not relevant to the query. For instance consider two tuples $t1 = (V1, miata, 14K, John, CA)$ and $t2 = (V2, miata, 13K, Sam, CA)$. Both tuples refer to cars of the same model sold in the same state. Say that $(CP * Factor)$ for a "miata" in CA is $13K$. Both tuples $t1$ and $t2$ imply that "miata" is a low-profit model. However, whenever tuple $t1$ causes "miata" to become a low-profit model. then tuple $t2$ also allows the same inference because the sales price in $t2$ is less than the sales price in $t$ 1. Thus. we can discard tuple $t1$ without affecting the answer to $Q2$. That is. we can introduce an aggregation operation above relation sales to compute for each state and each model. the minimum value of $0.85 * SP$. This minimum value determines if a. model is low-profit. Figure 4 shows the query tree for query $Q2$ after the aggregation has been created. The aggregated sales relation has as many tuples as the product of the number of states and number of models supplied by the manufacturer. Note. a function computation has also been pushed down to a base relation.

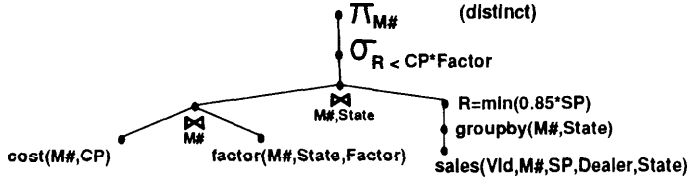



Figure 1: Query to compute low-profit cars after pushing down aggregation

$Q2$ shows that aggregations can be introduced gainfully in queries that do not use any aggregation. Queries $Q1$ and $Q2$ are representative of queries in decision support systems. These systems typically involve massive tables. The queries compute aggregate values of some attributes grouped by other attributes that denote some financial or sociological class.

This example illustrated three optimixations: pushing aggregations introducing new aggregations in queries that did not have aggregations. and pushing function computations. All three are inferred in a uniform manner by the GP pushing algorithnr we present in this paper. □

Aggregations are a. way of eliminating some tuples of a relation based on some other tuples. Current optimizers do not use tuples in a relation $R$ to discard other tuples in $R$. Duplicate elimination is another instance of such an optimization. Currently, **distinct** queries are computed by doing duplicate elimination at the very top of a query tree. **distinct** computations c-an be pushed down query trees using the algorithm we develop. We also show how to optimize standard duplicate preserving queries with no aggregations by introducing the count aggregate operator.

**EXAMPLE 1.2** Consider query $Q3$ that for all models with base $CP > 15K$. returns the model number of the car and names of dealers who sell cars of that model. Query $Q3$ preserves duplicates. The query is represented by the left half of the following figure. This query is a simple Select-Project-Join query.

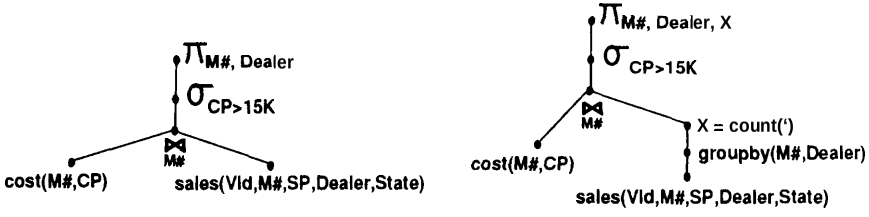



Figure 5: Compute $M\#, Dealer$ such that $Dealer$ sells cars of model $M\#$ and $M\#$ has $CP > 15K$

The optimized query is as shown in the right half of the above figure. In the final answer. counts represent repeated tuples. If the answer requires duplicates as output. we need to output each tuple as many times as indicated by its count. In general. if duplicates are needed in the answer then generalized projection pushing involves keeping a count of the number of duplicates at only some intermediate nodes. Our rewrite algorithm allow us to carry counts as an extra attribute for all intermediate computations. Thus. in all intermediate steps. the cost of joins and selections will be reduced because multiple tuples will be replaced by a single tuple that has an extra field. It is important to note that keeping a count does not require altering the way joins are done.

Consider a database where relation sales has 1 million tuples. that involves 1000 distinct dealer names. and 10 model numbers. In the optimized tree for query $Q3$. the size of the aggregated sales relation is at most 10. 000 assuming that each dealer sells every model. Even with this conservative assumption. the number of tuples after the join is reduced by a factor of 100 □

Aggregations are an integral part of SQL [LCW93] and are very commonly used in decision support systems. Efficient implementations of aggregation and groupby operators exist in most commercial systems. The performance impact of doing aggregations early has been recognized For example. in the Tandem optimizer. single table aggregations are done at the disk process level whenever possible [HJT91]. While efficient single-t able aggregation implementations are present. such features are not used in queries where the aggregation is done after a join The algorithm presented in this paper allows us to push aggregations down a query tree. enabling the use of efficient single-table aggregation operators before joins are taken. as well as reducing the size of

Paper 17

FILE: /pub/hasan/1994/color.ps
COMMENT: Technical report

# Coloring Away Communication in Parallel Query Optimization


WAQAR HASAN
Stanford University & HP Labs
hasan@cs.stanford.edu

RAJEEV MOTWANI*
Stanford University
rajeev@cs.stanford.edu



**Abstract**

We address the problem of finding parallel plans for SQL queries using the two-phase approach of *join ordering and query rewrite* (JOQR) followed by *parallelization*. We focus on the JOQR phase and develop optimization algorithms that account for the communication overhead of repartitioning data. Taking queries that include operations such as grouping, aggregation, intersection and set difference in addition to joins, we model the problem of minimizing repartitioning cost as a *query tree coloring* problem and devise an efficient algorithm. We also show that repartitioning costs impact the choice of join predicates and the order of joins. We develop a generalized abstraction for a System R style dynamic programming algorithm and use it to develop a *colored join ordering* algorithm that finds a join tree with minimal total cost while accounting for repartitioning.



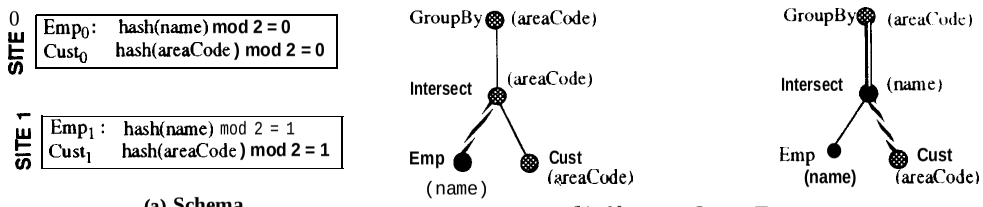
**Keywords: Parallel Query Optimization, Algorithms**


## 1 Introduction

An important challenge in parallel database systems [DG92, Val93, BCC+90, DGG+86] is *parallel query optimization*. This is the problem of finding optimal *parallel* plans for *decision-support* queries that include operations such as aggregation. grouping, union. intersection, set difference and calls to external functions in addition to joins. The problem may he broken into two phases [HS91]: (1) join ordering and query rewrite (JOQR) followed by (2) parallelixation. This paper focuses on the JOQR phase and develops optimization algorithms that account for the communication overhead [PMC+90, Gra88] of exploiting parallelism.

Partitioned parallelism [DG92] which exploits horizontal partitioning of relations is an important way of reducing the response time of queries. This may require data to be *re-partitioned* among sites thus incurring substantial communication overhead.


*Supported by an IBM Faculty Development Award an OTL grant. and NSF Young Investigator Award CCR 9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation


(a) Schema (b) Alternate Query Trees

Figure 1: Query Trees: Hatched edges show repartitioning

**Example 1.1** Assume tables `Emp (enum, name, areaCode, number)` and `Cust (name, areaCode, number)` are horizontally partitioned on two sites on the underlined attribute. Suppose we want to determine the number of employees who are also customers and group the result by `areaCode`. After deciding it reasonable to guess an employee and a customer to he the same person if they have the same name and phone number, we may write the following query in SQL2 [X3H92]:

```
Select areaCode, Count(*)
From (Select name, areaCode, number From Emp) Intersect Cust
GroupBy areaCode
```

Figure 1 shows two query trees that differ only in how data is repartitioned. Since tuples with thr same `areaCode` need to come together. `GroupBy` is partitioned by `areaCode`. However, `Intersect` may he partitioned on *any* attribute. If we choose to partition it by `areaCode`, we will need to repartition the (projected) `Emp` table. If we partition by `name`, we will need to repartition the `Cust` table as well as the output of the intersection. Thus one or the other query tree may be better depending on the relative sizes of the tables. I

We address the problem of choosing the partitioning attributes in a query tree so as to minimize repartitioning overhead. By regarding partitioning attributes as colors, we model it as a *query tree coloring* problem in which repartitioning cost is saved when adjacent operations have the same color. This optimization problem is related to the classical problem of Multiway Cuts [DJP+92] and has been encountered in computational biology [ES92, ES94] and network design [CR91]. We develop an elegant $O(nc)$ algorithm ($n$ is number of nodes in query tree and $c$ the number of distinct pre-existing partitioning attributes) that is simpler and more efficient than previously known algorithms.

We show that repartitioning costs impact the choice of join predicates as well as the order of joins. We develop a general understanding of the cost and applicability of "System R style" dynamic programming algorithms [SAC+79] to join ordering This allows us to to clearly understand the effects of physical properties of tables (indexes. sort-orders) on join ordering and to incorporate partitioning as a new physical property. Another contribution of the abstraction is to show that dynamic programming is applicable irrespective of the complexity of cost formulas.

Our work is in contrast to the use of a conventional query optimizer by Hong and Stonebraker [HS91, Hon92] as the JOQR phase in XPRS. However, it should he noted that Hong [Hon92] conjectured

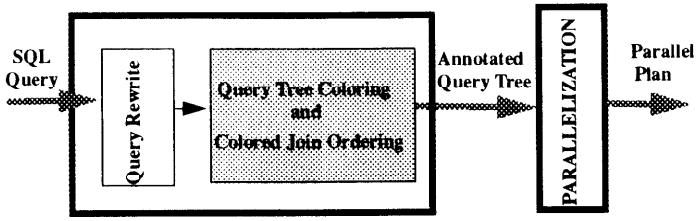



Figure 2: Two-Phase Query Optimization (Algorithms developed in this paper are shaded)

the XPRS approach to he inapplicable to architectures such as shared-nothing that have significant communication costs. Other work on parallel query optimization [SE93, LST91, SYT93, CLYY92, HLY93, ZZBS93] also ignored modeling communication overheads of parallelism.

'Though query processing in parallel and distributed databases [CP84, OV91, YC84] is fundamentally similar. repartitioning intermediate results to reduce response time did not receive much attention until the appearance of parallel machines. Shasha and Wang [SW91] investigated algorithms for join ordering taking repartitioning cost into account. However, they developed heuristics assuming the cost of a join to be proportional to the sum of the sizes of operands. This excludes common join methods that. use indexes or sorting whose cost formulas are non-linear. In contrast, the dynamic programming approach permits rich cost models and guarantees optimality with respect to the cost model.

Section 2 discusses the architecture of a two-phase parallel query optimizer and shows how our algorithms may he used. It defines the *Query Tree Coloring* and *Colored Join Ordering* optimization problems that are solved in this paper. Section 3 develops an efficient algorithm for query tree coloring and shows several extensions. Section 4 develops a. general abstraction for understanding the applicability of dynamic programming to join ordering and shows how reparitioning cost may he taken into account. It. also sketches the design of an integrated algorithm for query tree coloring as well as join ordering. Section 5 summarizes our contributions and discusses future work.

## 2 A Model for the Problem

### 2.1 Software Architecture of a Parallel Query Optimizer

We adopt a two-phase approach to parallel query optimization: *JOQR* followed by *parallelization*. JOQR is similar in functionality to a conventional query optimizer. Given an SQL query, it produces an annotated join tree that fixes the order of operations and other procedural derisions such as the strategy for each join. This phase minimizes the *total* cost for computing the query. The parallelization phase constructs a parallel plan (i.e. a schedule) for the annotated query tree so as to minimize *response time*. It uses a detailed cost model that incorporates timing constraints between operators and makes derisions about allocation of resources.

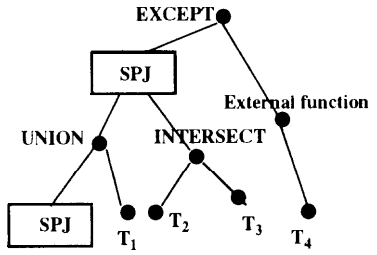



Figure 3: A Complex Query as select-project-join expressions connected by other operations

A conventional optimizer may he regarded as consisting of two subphases [HFLP89]: *query rewrite* followed by *join ordering*. Query rewrite transforms a complex query into a more efficient form hy applying rewrite rules. It also separates the query into select-project-join (SPJ) sub-expressions ("select boxes" i n [PHH92]) erconnected by other operations (see Figure 3). The join ordering phase optimizes each SPJ expression separately. The algorithms developed in this paper may be used as shown in Figure 2.

### 2.2 Partitioning

**Definition** 2.1 A *partitioning* is a pair $(a, h)$ where $a$ is an attribute and $h$ is function that maps values of $a$ to non-negative integers. □

Given a table $T$, a partitioning produces fragments $T_0, \ldots, T_k$ such that a tuple $t \in T$ occurs in fragment $T_i$ if and only if $h(t.a) = i$. For example. the partitioning of `Emp` table in Example 1.1 is represented as (`name, hash(name) mod 2`). The function `hash(name) mod 2` is applied to each tuple of `Emp` and the tuple is placed in fragment $\texttt{Emp}_0$ or $\texttt{Emp}_1$ depending on whether the function returns 0 or 1.

Partitioning provides a source of parallelism since the semantics of most database operations allows them to he applied in parallel to each fragment. Suppose $S_0, \ldots, S_k$ and $T_0, \ldots, T_k$ are fragments of tables $S$ and $T$ produced by the *same* partitioning $a = (a, h)$

**Definition 2.2** A unary operation $f$ is *partitionable* with respect $a$ iff $f(S) = f(S_0) \cup \ldots \cup f(S_k)$. A binary operation f is *partitionable* with respect to $a$ iff $f(S,T) = f(S_0, T_0) \cup \ldots \cup f(S_k, T_k)$ □

**Example 2.1** Suppose each of tables `Emp'(name, areaCode, number)` and `Cust(name, areaCode, number)` is partitioned across two sites using the hash function `hash(areaCode) mod 2` Since the tables have the *same* partitioning. $\texttt{Emp}' \cap \texttt{Cust} = (\texttt{Emp}'_0 \cap \texttt{Cost},) \cup (\texttt{Emp}', \cap \texttt{Cust}_1)$. This permits $\texttt{Emp}' \cap \texttt{Cust}$ to he computed by computing $\texttt{Emp}'_0 \cap \texttt{Cust}_0$ and $\texttt{Emp}'_1 \cap \texttt{Cust}_1$ in parallel I

# Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism

WAQAR HASAN
Stanford University
*and*
Hewlett-Packard Laboratories
hasan@cs.stanford.edu

RAJEEV MOTWANI*
Department of Computer Science
Stanford University
Stanford, CA **94305**
rajeev@cs.stanford.edu

## Abstract

We address the problem of finding parallel plans for SQL queries using the two-phase approach of join ordering followed by parallelization. We focus on the parallelization phase and develop algorithms for exploiting pipelined parallelism. We formulate parallelization as scheduling a weighted operator tree to minimize response time. Our model of response time captures the fundamental tradeoff between parallel execution and its communication overhead. We assess the quality of an optimization algorithm by its *performance ratio* which is the ratio of the response time of the generated schedule to that of the optimal. We develop fast algorithms that produce near-optimal schedules – the performance ratio is extremely close to 1 on the average and has a worst case bound of about 2 for many cases.

## 1 Introduction

We address the problem of *parallel query optimization*, which is to find optimal parallel plans for executing SQL queries. Following Hong and Stonebraker [HS91], we break the optimization problem into two phases: join ordering followed by parallelization. We focus on the parallelization phase and develop optimization algorithms for exploiting pipelined parallelism.

Our model of parallel execution captures a fundamental tradeoff – *parallelism has a price* [Gra88, PMC+90]. Two communicating operators may either incur the communication overhead and run on distinct processors, or share a processor and save the communication overhead.

We model parallelization as scheduling an operator tree [GHK92, Hon92b, Sch90] on a parallel machine. We represent the resource requirements of operators and communication as node and edge weights respectively. Our optimization objective is to find a schedule (i.e. a parallel plan) with minimal response time.

This paper concentrates on algorithms for exploiting pipelined parallelism. In general, pipelining is a useful supplement to partitioned parallelism [DG92] but is sometimes the only way of speeding up a query. Consider decision support queries that join a large number (say 10) of relations and apply external functions, grouping and aggregation. Selection predicates may localize access to single partitions of each relation. If each reduced relation is small, partitioned parallelism ceases to be a viable option and pipelined parallelism is the only source of speedup. Algorithms for managing pipelined parallelism are thus an *essential* component of an optimizer.

To the best of our knowledge, scheduling theory does not provide algorithms that handle communication costs. Scheduling pipelines constitutes a nontrivial generalization of the classical problem of multiprocessor scheduling [GLLK79]. Prior work in parallel query optimization [Hon92a, SF93, LCRY93, LST91, SYT93, SYG92, TWPY92, WFA92] ignored the communication overhead of exploiting parallelism. This was sometimes justified by restricting to situations where communication overhead is low shared-memory architectures with omission of pipelined parallelism.

Brute force algorithms are impractical for scheduling pipelines due to the extremely large search space. A query that joins 10 relations leads to an operator tree with about 20 nodes. The number of ways of scheduling 20 operators on 20 processors exceeds $5 \times 10^{43}$.

Algorithms that simply ignore communication overhead are unlikely to yield good results. We show that one such naive algorithm produces plans with twice the optimal response time on average, and is arbitrarily far from optimal in the worst case.

We will measure the quality of optimization algorithms by their *performance ratio* [GJ79] which is the ratio of the response time of the generated schedule to that of the optimal. Our goal is to develop algorithms that are near-optimal in the sense that the average performance ratio should be close to 1 and the worst performance ratio should be a small constant.

We develop two algorithms called *Hybrid* and *GreedyPairing*. We experimentally show that both algorithms, on the average, find near-optimal plans for small operator trees. Experiments with larger operator trees proved impossible for the very reason that it required the practically infeasible task of computing the optimal schedule.

Thus, one motivation for our worst-case analytical results was the need to provide performance guarantees independent of the size of the operator tree or the number of processors. Another motivation was to provide guarantees that do not depend on the choice of the experimental benchmark and are valid across all database applications. Finally, worst-case bounds on the performance ratio apply to each schedule and thus guarantee that the scheduling algorithm will *never* produce a bad plan.

We show, for $p$ processors, the performance ratio of *Hybrid* is no worse than $2 - 1/p$ for operator trees with arbitrary paths and no worse than $2 + 1/p$ for stars. We also show the performance ratio of *GreedyPairing* to be no worse than $2 - \frac{2}{p+1}$ when communication costs are zero.

*Hybrid* outperforms *GreedyPairing* almost uniformly but the difference is significant only when operator trees are large and communication costs are low. On the other hand, *GreedyPairing* is a simple algorithm which can be extended naturally to take data-placement constraints into account.

Section 2 develops a cost model for response time and provides a formal statement of the optimization problem. Section 3 summarizes our approach and discusses why a natural algorithm called *Naive LPT* does not perform well.

Section 4 develops the *GreedyChase* algorithm to identify parallelism that is simply not worth exploiting irrespective of the number of processors. Use of *GreedyChase* as a *pre-processing* step leads to improved schedules. The *Modified LPT* algorithm is developed and shown to be near-optimal for star-shaped operator trees.

Section 5 focuses on the restricted class of *connected* schedules. We show the optimal connected schedule to be a near-optimal schedule for path-shaped operator trees. We develop a fast polynomial algorithm called *BalancedCuts* to find the *optimal* connected schedule.

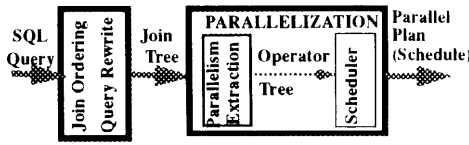



Figure 1: Two-phase Parallel Query Optimization

Section 6 develops algorithms for the general problem. The *Hybrid* algorithm is devised by combining the best features of connected schedules and the *modified LPT* algorithm. Finally, the *GreedyPairing* algorithm is developed and experimentally compared with *Hybrid*. Section 7 summarizes our contributions and provides directions for future work.

## 2 A Model for the Problem

Figure 1 shows a two-phase approach for compiling an SQL query into a parallel plan. The first phase is similar to conventional query optimization, and produces an annotated join tree that fixes aspects such as the order of the joins, join methods and access methods. The second phase is a parallelization phase that converts a join tree into a parallel plan.

We define a parallel plan to be a schedule consisting of two components: (1) an *operator tree* that identifies the atomic units of execution (operators) and the timing constraints between them; and, (2) an allocation of machine resources to operators.

We consider parallelization itself to consist of two steps. The first step translates an annotated join tree to an operator tree. The second step is a scheduling step that allocates resources to operators.

Section 2.1 refines prior notions of operator trees [GHK92, Hon92b, Sch90] by reducing timing constraints between operators to parallel and precedence constraints. Section 2.2 shows how resource requirements of nodes and edges may be derived from conventional cost models. Section 2.3 describes a cost model for response time. Finally, Section 2.4 provides a formal statement of the optimization problem addressed in this paper. A full description of the cost model and its justification can be found in [Has94a, Has94b].

### 2.1 Operator Trees

An operator tree is created as a "macro-expansion" of an annotated join tree (Figure 2). Nodes of an operator tree represent operators. Edges represent the flow of data as well as timing constraints between operators.

An operator is an atomic piece of code that takes zero or more input sets and produces a single output set. Operators are formed by appropriate factoring of the code that implements the relational operations specified in an annotated join tree. A criteria in

*Supported by NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation and Xerox Corporation.



**Proceedings of the 20th VLDB Conference**
**Santiago, Chile, 1994**

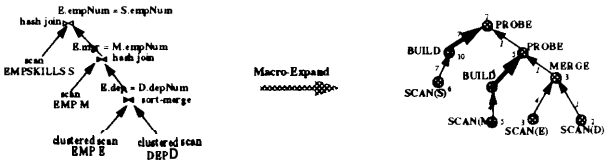



Figure 2: Macro-expansion of a Join Tree tn a Weighted Operator Tree

designing nprratnrs is to reduce inter-operator timing constraints to simple forms, i.e. parallel and precedence constraints.

It. is often possible to run a consumer operator in parallel with an operator that produces its input. This is termed ***pipelined parallelism.*** In order t n identify such opportunities for speedup, wc classify the inputs/outpu t of operators into two idealized categories:

- ***blocking***: the set of tuples is produced/consumed as *a whole set.* A Mocking output produces the entire output set at. the instant the operator terminates. An operator with a blocking input requires the entire input set before it can start.
- ***pipelining***: the set is produced/consumed "tuple at a time" and the tuples are uniformly spread over ov er the *entire* execution time of the operator.

Opportunities fnr pipelined parallelism rxist only a l o n g edges that connect a pipelining output to a pipelining input.

**Definition 2.1** If an edge connects a pipelining output to a pipelining input, it is a ***pipelining edge***; otherwise it is a ***blocking edge.*** □

In Figtirr 2 blocking edges are shown as thick edges.

Pipelined execution i s typically implemented using a ***flow control*** mechanism (such as a table queue [PMC+90]) to ensure t h a t, a fixed amount of memory suffices for the pipeline. This constrains all operators in a pipeline tn run concurrently - the pipeline executes at the pace of the slowest nprratnr. Thus, pipelining edges represent parallel constraints. and Mocking edges represent precedence constraints.

**Definition 2.2** Given an edge from nprratnr ***i*** tn j. a ***parallel constraint*** requires *i* and j to start at the same time a n d terminate at the same time. A ***precedence constraint*** requires j to start after *i* terminates. □

Note that. a pipelining constraint. is symmetric in *i* a n d j. The direction of the edge indicates the direction in which tuples flow but is immaterial for timing constraints.

The code for join and access methods is expected to be broken down into operators such that **inputs/outputs** are easily classifiable as blocking or pipelinable. For example a simple hash join may be broken into **build** and **probe** operators. The **build** operator produces a "whole set," i.e., the hash table while **probe** produces output "tuple at. a time." If the inputs/output are not easily classifiable, the operator should be further broken down. As an example consider the merge-sort algorithm that consists of repeatedly forming larger runs followed by a merge o f the find set of runs. The output is neit her produced at the instant. the operator terminates nor is it spread out over the entire execution time nf the operator. T h i s situation is h andled by breaking the algorithm into two operators: ***form-runs*** that produces a blocking output, i.e. the final rttns. and merge-runs that takes a blocking input and produces a pipelining output.

### 2.2 Resource Requirements AS Weights

The weight $t_i$ of node $i$ is the time to run the nprratnr in isolation assuming all communication to be local. The dominant. overhead of using distinct. processors for two communicating operators is the additional instructions that must be executed fnr communicating data [Gra88, PMC+90]. The weight $c_{ij}$ o f an edge between nodes $i$ and j is the ***additional*** execution timr incurred by both operators if they are assigned distinct processors.

Conventional models (such as System R [SAC+79]) use statistical models to estimate the sizes o f the intermediate results. These sizes a r r then used to determine the CPU, IO and communication requirements of relational operations such a.. pins. Since relational operations arr broken down into nprratnrs, the same formulas are easily extended for nprratnrs.

Given the CPU, IO a n d communication requirements o f a n operator, its execution time may be estimated by a formula that combines the individual resource requirements using functions such as *weighted sum* and *max*. The exact formula depends on the hardware and software architecture oft he system in consideration. The nature of the formula ts not important in solving the parallelization problem, it suffices that such formulas exist.

### 2.3 Response Time of a Schedule

***In the rest of this paper, we shall restrict ourselves to operator trees with only pipelining edges and these edges will be considered undirected since the direction is immaterial for the timing constraints.*** For example,if the indexes (hash tables) pre-exist, the operator tree of Figure 2 would have no blocking edges and would reduce to the one shown in Figure 3(a).

4 schedule (t c parallel plan) allocates operators to processors. Since we assume processors t o be identical, a schedule may be regarded as a partition of the set of operators.

**Definition 2.3** Given $p$ processors and a u operator tree $T = (V, E)$, a *schedule* is a partition of 1'. the set of nodes, into $p$ sets $F_1, \ldots, F_p$. □

Suppose $F$ is the set of operators allocated to some processor. The cost of executing $F$ is the cost of executing all nodes in $F$ plus the overhead for communicating with nodes on other processors. It is thus the sum of the weights of all nodes in $F$ and the weights of all edges that connect a node within $F$ to a node outside. Fnr convenience, we define $c_{ij} = 0$ if there is no edge from $i$ to j.

**Definition 2.4** If $F$ is a set of nprratnrs, $cost(F) = \sum_{i \in F}[t_i + \sum_{j \notin F} c_{ij}]$. □

**Definition 2.5** If $F$ is thr set of operators allocated to processor $p$, $load(p) = cost(F)$. □

Since our goal i s to minimize the response time $R$ o f a parallel p l a n, we now derive a formula for $R$ given an operator tree $T = (V, E)$ and a schedule $S = F_1, \ldots, F_p$.

Thr pipelining constraint forces all nprratnrs in a pipeline to start simultaneously (titiir U) and terminate simultaneously at time $R$. Fast operators are forced to "stretch" over a longer time period by the slow operators. Suppose operator i is allocated to processor $p_i$ a n d uscs fraction $f_i$ o f the processor. The pipelining constraint is then

$$f_i = \frac{1}{R}[t_i + \sum_{j \notin F_{p_i}} c_{ij}] \quad \text{for all nprratnrs } i \in V \quad (1)$$

The utilization of a processor is simply the sum of utilizations of the operators executing on it. Since at least one processor must be saturated (otherwise t he piprtinr would speed up):

$$\max_{1 \le i \le p}[\sum_{i \in F_i} f_i] = 1$$

$$\Rightarrow \quad R = \max_{1 \le i \le p} cost(F_i) \qquad \text{using equation (1)}$$

**Definition 2.6** The response time $rt(S)$ of schedule $S = F_1, \ldots, F_p$ is $\max_{1 \le i \le p} cost(F_i)$. □

**Example 2.1** Figure 3(a) shows a schedule for a pipelined operator tree. Notice that the join tree of Figure 2 would expand to exactly this tree if indexes pre-exist. Edges are shown as undirected since the direction is immaterial for the purposes of scheduling. The schedule is shown by encircling the set of operators assigned to the same processor. The cost of each set is underlined. For example ($probe$) costs 8 by adding up its node weight (1) and the weight of the edge (1) connecting tt to its child. Figure 3(b) shows a Gantt chart o f the execution specified by the schedule. T h r fraction of thr processor used by each operator in shown in parent hnis. ■

### 2.4 Formal Problem Statement

The pipelined operator t r r r scheduling problem may now be star ed as follows:

**Input:** Operator Tree $T = (V, E)$ with positive real weights $t_i$ for eacti node $i \in V$ and $c_{ij}$ for each edge $(i, j) \in E$; number of processors $p$

**Output** A schedule $S$ with minimal response time i e., a partition of $V$ into $F_1, \ldots, F_p$ that minimizes $\max_{1 \le i \le p} \sum_{i \in F_i}[t_i + \sum_{j \notin F_i} c_{ij}]$.

This problem is intractable since the special case in which all edge wrights arr zero is the NP-complete problem of ***multiprocessor scheduling*** [GJ79, GLLK79].

Since ttir number of ways of partitioning $n$ elements into $k$ disjoint non-empty sets is given by $\left\{ {n \atop k} \right\}$, which denotes Stirling numbers of the second kind [Knu73], we have

**Lemma 2.1** *The number of distinct schedules for a tree with $n$ nodes on $p$ processors is* $\sum_{1 \le i \le p} \left\{ {n \atop i} \right\}$

This number of schedules is about $1.2 \times 10^5$ for $n = p = 10$, $1.4 \times 10^9$ for $n = p = 15$, and $5 \times 10^{13}$ for $n = p = 20$.

## 3 Overview of Our Approach

Scheduling pipelined operator trees is an intractable problem and the space of schedules is super-exponentially large. Thus any algorithm that finds the *optimal* is likely to be too expensive to be usable. Our approach is to develop ***approximation algorithms*** [GJ79, Mot92], i.e., *fast* heuristics that produce ***near-optimal*** schedules.

We first. discuss our methodology for evaluating algorithms. We then show why a naive a l g o r i t h m does not perform well. Finally, we provide a road map to algorithms in ttir rest of the paper.

The proofs of lemmas and theorems are omitted due to space constraints. The interested reader i s referred to the full version nf this paper [HM94].

### 3.1 Methodology

We will evaluate algorithms based on their ***performance ratio*** which is defined as the the ratio of the response time of the generated schedule to that of the optimal.

Our goal is to devise algorithms that satisfy two criteria. Firstly, the *average* performance ratio should be close to 1. Secondly, the *worst* possible performance ratio should be *bounded*. In other words, the performance

| | |
|---|---|
| FILE: | /pub/keller/1994/CEDB-paper.ps |
| COMMENT: | Technical report, submitted for publication |

a aper 19

# Versions, Configurations, and Constraints in CEDB[1]

H. Craig Howard[2,3]
Arthur M. Keller[4]
Ashish Gupta[4]
Karthik Krishnamurthy[2]
Kincho H. Law2
Paul M. Teicholz[2,5]
Sanjai Tiwari[2]
Jeffrey D. Ullman[4]

## Abstract

The architecture-engineering-construction (AEC) industry is highly fragmented, both vertically (between project phases, e.g., planning, design, and construction) and horizontally (between specialists for the various disciplines at a given project phase, e.g., design). We need software that detects, analyzes, and manages changes efficiently during concurrent distributed design processes. In the CEDB (Collaborative Environment for the Design of Buildings) project, we have developed a model of a combination of versions, configurations, and constraints. Versions are organized into hierarchies of alternatives within a single discipline (e.g., architecture, structural engineering). A configuration is a set of versions, one from each of a number of disciplines, combined with a set of cross-disciplinary constraints to check for violations. Our objective in this integrated model of versions, configurations, and constraints is to assist designers by informing them of the changes by others that affect them and their changes that affect others, in particular, the changes that result in constraint violations. To accomplish this objective, we aim to find those violations as efficiently as possible. We have substantially implemented our model using multiple ORACLE databases.

1 This work is part of the CEDB project (Collaborative Environment for the Design of Buildings, or Civil Engineering DataBase), Jeffrey D. Ullman, Principal Investigator. This effort is funded in part by National Science Foundation grant IRI-91-16646.

2 Department of Civil Engineering, Stanford University, Stanford, CA 943054020

3 Phone: 315-723-5678; e-mail: hch@cive.stanford.edu

4 Department of Computer Science, Stanford University, Stanford, CA 94305-2140

5 Center for Integrated Facility Engineering, Stanford University, Stanford, CA 943054030





## 1. Introduction

The U.S. architecture-engineering-construction (AEC) industry is highly fragmented compared with many of its Asian and European competitors [Howard 89a]. This fragmentation exists both within individual phases of the construction process (e.g., the design phase). and across project phases, from planning through design and construction and into facility maintenance and operation. The problems arising from fragmentation affect productivity and competitiveness throughout the AEC industry. Our goal in this work is to address this fragmentation through change management tools that facilitate and enhance collaboration among multiple designers and contractors.

To provide change management in this environment, we propose a combination of versions, configurations, and constraints. A *version* represents a design checkpoint in a single discipline (e.g., architecture, structural engineering). Versions may be grouped hierarchically within a discipline, with later versions represented as modifications to earlier versions and parallel alternatives represented as branching versions. ***Constraints*** are used to manage interactions across the discipline-specific versions by specifying inconsistent design states-ombinations that aren't allowable. A ***configuration*** represents an "integrated" design, with one version from each of a number of distinct disciplines combined with a set of constraints to be verified. A configuration can be generated for major design review or in response to a "what-if' test by a single designer who wants to compare his/her latest version with existing versions from other disciplines. A configuration results in a list of violations that trigger notifications to the designers involved in the constraints. By providing operators to compare the versions within new configuration to the versions in a previously checked configuration, we can perform incremental constraint checking, thus limiting the volume of changes that need to be tested in each design iteration. Our objective in this integrated model of versions, configurations, and constraints is to notify the appropriate designers of constraint violations and to find those violations as efficiently as possible.

This paper describes the CEDB (Collaborative Environment for the Design of Buildings or Civil Engineering DataBase) project at Stanford University, an interdisciplinary effort between the departments of computer science and civil engineering, that addresses change management for concurrent design. Section 2 discusses some of the related work in the areas of versions, configurations, and constraints. Then Section 3 provides an overview of the AEC domain, including a pair of sample databases and a sample constraint based on a real example. Sections 4 and 5 describe the model in detail and demonstrate the processing of changes using the example defined in Section 3. Finally, we conclude by summarizmg the salient points of the model.

## 2. Related Work

For versions and configurations, there is a useful survey in the area [Katz 90]. Katz considers the following issues in version and configuration management: (i) organizing the version set, (ii) static and dynamic binding mechanisms, (iii) hierarchical compositions, (iv) version grouping mechanisms, (v) change notification and propagation, and (vi) object sharing mechanisms. The following issues addressed by our work are surveyed in detail by Katz.

- **Versions of an individual entity:** Katz et al. have formalized the version derivation history as a hierarchy [Katz 86]. More general structures as a rooted DAG have been proposed in [Klahold 86, Ecklund 87]. The versions are connected by *derived-from* links.
- **Versions of an assembly of entities:** Previous research efforts have defined *configurations* as the version of a composite entity in terms of the versions of its components [Katz, 87, Ketabchi 87, Landis 86. Lorie 83 ].

- **Inheritance among versions:** ***Type-version*** inheritance [Batory 85], as well as ***instance-instance*** inheritance schemes along ***descendant-of, equivalent-to,*** and ***component-of*** relationships [Katz 89] have been proposed.

- **Implementation Schemes:** Implementation of versions in terms of ***deltas*** to support the incremental addition of data to a version has primarily been studied for the software engineering environment [Rochkind 75, Leblang 84]. There has been relatively little effort on developing versioning systems that support the evolutionary nature of the design process. i.e., the incremental addition of data to an individual version.

The subject of constraint management has been a very active area in engineering automation, and we will concentrate here on a few of the civil engineering research projects that have directly influenced our effort. Holtz [Holtz 82] described symbolic algebra and dependency driven expressions to manipulate constraints for consistency management in design applications. Rasdorf and Fenves [Rasdorf 86] emphasized the importance of automated representation and processing of design constraints in relational databases. They proposed a mechanism of augmented relations, where additional attributes were appended to the database relation schemes, Their work assumes a centralized database. The DICE project [Sriram 92] addresses the issues of coordination and communication through a centralized blackboard and a global project database. DICE includes conflict handling in shared transactions, where the participants are notified of each update made within the scope of a transaction. The DICE project studies transaction management and concurrency control for collaborative engineering environments, while we focus on efficiency aspects of real-time constraint management and collaboration.

Most of the work done in integrity constraint management concentrates on centralized databases. However, the issues involved in checking constraints that span multiple databases have not received much attention. Yet distributed constraints are essential in a design environment composed of independent disciplines that need to coordinate.

The architecture proposed in this paper can use most of the existing approaches to constraint management at the local sites. If an underlying system supports local constraint management, then we incorporate that capability into the global validation process. Our constraint management system can be built on top of the existing database systems. Ceri and Widom [Ceri 90] describe a production rule system that allows declaration of rules that are triggered on events and their corresponding actions executed if some conditions are met. Such a system can provide monitoring at any of the underlying sites. In fact, part of our implementation uses their system as the backend. [Qian 88] describes techniques for distributing global constraints among sites in a way that reduces communication at run time. These techniques can be used to preprocess constraints in our architecture. Similarly, efficient constraint checking techniques discussed in [Nicolas 82, Bry 92] can be used by both local constraint managers and the global constraint manager. Distributed constraint checking can also be made more efficient by using the demarcation protocol [Barbara 92] or by generating queries that are sufficient to allow us to infer that a constraint has not been violated [Blakeley 89, Gupta 93b, Levy 93].

## 3. The Problem Domain

To provide the reader with a more tangible sense of the data management issues, Figure 1 illustrates several of the key stages involved in a building project. The figure emphasizes the differing views that the various project participants have about the data describing the process. The following discussion elaborates on the management of data at each stage of the process:

- The *architect* may start with grand ideas for the structure, but must shape these mto a more practical design based upon the constraints (financial, schedule, technical. operational)





generated by the owner, engineers. users and others in the process. The architect is normally the overall coordinator and leader in this multidisciplinary process. and increasingly is likely to use advanced CADD software. The architect also generates many, if not most, of the constraints and criteria that govern others in the process, but currently communicates with them mainly through traditional paper documents and meetmgs.

The ***structural engineer*** takes the architect's ideas and designs the skeleton that allows the building to resist the loads that derive from its function (e.g., furniture and people) as well as the loads caused by the environment (e.g., wind and earthquakes). As with the architect. the operations of the structural engineer may be highly computerized, but the coordination of changes is still currently a manual process.

The ***contractor*** takes the contract plans and specifications that result from all of these prior deliberations, and adds his or her knowledge of costs, schedules, methods and materials. The result is an estimate and resource-based schedule for completing the project most efficiently. The general contractor normally retains specialty subcontractors who, in turn, may further subcontract specific tasks. There is little use of electronic data communication across these boundaries, and coordination of changes is a time-consuming and error-prone process-frequently involving the use of a jackhammer.

The ***owner*** inherits the resulting performance of the structure (which hopefully stands up better than shown in the Figure 1). The owner's facility management must start with an accurate record of what was designed and built. That record must then be maintained over the lifetime of the facility. Many facilities undergo one or more modifications during their lifetime, particularly manufacturing or chemical process facilities such as computer chip fabrication plants, refineries, and space launch facilities. Even more mundane facilities may require renovation for new uses, retrofitting for seismic upgrade, etc.

The intent of this discussion is to highlight the differing views of each of the project participants, to emphasize the current lack of electronic data communications among the participants, and to underscore the potential for automated change management. The followmg section summarizes the requirements for change management in the AEC industry. Section 3.2 then introduces an example from the AEC domain that will be used throughout the rest of the paper.

FILE: /pub/kao/1994/dist_srt.ps

Paper 20

# On Building Distributed Soft Real-Time Systems

Ben Kao* Hector Garcia-Molinat Brad Adelberg‡

Abstract

When building a distributed real-time system, one can either build the whole system from scratch, or from pre-existing standard components. Although the former allows better scheduling design, it is not economical in terms of the cost and time of development. This paper studies the performance of distributed soft real-time systems that use standard components with various scheduling algorithms and suggests ways to improve them.

Keywords: soft real-time, distributed systems, deadline assignment, priority assignment, scheduling.

## 1 Introduction

Consider a distributed system for stock market analysis and program trading. In this application, information on stock prices may be gathered from multiple information sources and piped through a series of filters for refinement. The refined information may then be stored in a database server to be queried by, for example, an expert system that spots trading opportunities. The expert system may then trigger certain buy-and-sell actions to realize a profit.

There are two interesting features to observe. First, the system involves many components of different types: multiple information sources supplying streams of financial data, a network component, a database server (probably I/O bound), and an expert system (probably CPU bound). Second, there are global tasks which consist of multiple stages involving work at multiple components. These tasks are associated with soft deadlines, e.g., a buy-sell action triggered by a stock price update should be implemented within 2 minutes from the time when the update arrives.

When designing a distributed real-time system, such as the program trading one above, one approach is to build it from scratch, coding each component with the algorithms and features that match the applications' needs. However, the development and maintenance cost of such a system will be extremely high, and the development cycle will be very long. Alternatively, one can build the system by piecing together well-tested, easy-to-maintain standard components, such as a commercial database server, a token ring network, etc. The disadvantage of this approach is that standard components are usually not designed to handle real-time tasks. As an example, commercial database systems do not compare the deadlines of conflicting transactions when granting a lock request, which may result in priority inversions. Even if standard real-time components are available (e.g., commercial real-time operating systems), they may vary in their ability to meet tasks' real-time requirements. Also, their schedulers may not provide any external control. This makes global real-time scheduling algorithms impossible because they demand cooperation and coordination among the local schedulers. Given these difficulties, is it possible to construct a distributed system out of a variety of conventional non-real-time and real-time components and still meet real-time constraints? Even if we cannot meet all real-time constraints, can we ensure that "almost all" are met? In this paper we address these questions. Although we will not provide definitive answers, we do present the results of some simple experiments that provide insight into those questions. In particular, the results illustrate the cost (in terms of missed deadlines) of using conventional, heterogeneous components, and suggest strategies for reducing the cost.

For our experiments we consider a distributed system with a number of components Each component employs its own scheduling policy (whether real-time or non-real-time). We study how the local schedulers impact the system's ability to meet task deadlines. We also suggest ways of improving the system performance when some of the components cannot perform real-time scheduling. Before we proceed, we state some premises:

- We focus on *soft* real-time systems. In such systems, a primary performance goal is to meet as many deadlines as possible, but unlike hard real-time systems, there is no absolute guarantee that all deadlines will be met. There are two reasons why we look at soft (instead of hard) real-time systems. First, the kinds of distributed tasks we are looking at are quite complex, involving multiple stages of processing. This means that it is generally hard to get the accurate running time estimates that are required for hard real-time scheduling. Second, in many applications it is undesirable or impossible to place an upper bound on the load. Both problems make hard real-time scheduling impossible to achieve.

- Each component's scheduler is independent. There is no global scheduler that instructs each scheduler what to do. Each scheduler makes decisions based solely on the subtasks that have been presented to it for execution, without consulting other schedulers. We believe that large systems are built out of preexisting components. Each component will have its own scheduling policy and will be unable or unwilling to coordinate or subordinate its scheduling decisions with (or to) others.

*Princeton University Department of Computer Science. Current address: Department of Computer Science, Stanford University., Stanford. CA 94305. e-mail: kao@cs.stanford.edu

†Stanford University Department of Computer Science. e-mail: hector@cs.stanford.edu

‡Stanford University Department of Computer Science. e-mail: adelberg@cs.stanford.edu

- We look at on-line scheduling, as opposed to a priori scheduling when the tasks are defined or first submitted. On-line scheduling is more appropriate for the type of systems we study where the tasks' types or their durations may be unknown in advance. Also, when the system provides distribution transparency (e.g., whether a piece of data item is available locally or remotely), the subtasks to be created are unknown until run-time and thus off-line pre-analysis is not appropriate.

- Each system component is unique. If a task is scheduled to run at a particular component, it must run there. There is no load balancing, i.e., an overloaded component cannot ship tasks to other components.

The rest of this paper is organized as follows. Section 2 describes the model for our study. Section 3 discusses the case when all system components use real-time scheduling such *as* earliest-deadline-first. In Section 4 we study how the system performance degrades when some of the components serve tasks in FCFS order. In Section 5 we discuss the benefits we can gain when a non-real-time component provides static priority scheduling. Finally, we conclude our paper is Section 6.

## 2 The Model

In this section, we describe the task model, the system model, and the simulation model we use for our analysis. We will first define global tasks, and then describe a model of a distributed system on which tasks are mapped for execution. As the reader will notice, our model is quite simple. As mentioned earlier, our goal is to understand the basic tradeoffs, not to realistically evaluate a particular system.

### 2.1 The Task Model

In this paper, we consider serial global tasks that involve work at multiple components in the system. As shorthand, we use the notation $T = [T_1 T_2 ... T_n]$ to represent a global task $T$ that consists of n subtasks, $T_1, T_2, \ldots, T_n$, to be executed in series. A subtask $T_i$ $(i > 1)$ cannot execute before subtask $T_{i-1}$ finishes.

A task X (whether it is a subtask or a global task) has the following attributes:

at(X) = arrival (or submission) time of X,
$dl(X)$ = deadline of X,
$sl(X)$ = slack of X.



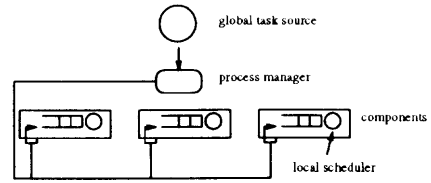



Figure 1: The System Model.

We also define the stage of a subtask X to be its position in a global task For example, if $T = [T_1 T_2 T_3]$, then $stage(T_2) = 2$. We say that a subtask X is an *earlrer-stage* than another subtask $Y$ if they are of the same global task and stage(X) < $stage(Y)$.

Finally, there is the issue of tardy tasks, or *overload management policy*. Suppose a task X has already missed its deadline, but has not completed execution. One option is to abort X as soon as it misses its deadline, under the assumption that whatever it was doing is now useless A second option is to continue to process X, under the assumption "better late than never " Due to space limitations, in this paper, we only focus on the no abortion case.

### 2.2 The System Model

Our model of a distributed real-time system consists of a number of *components* (Figure 1) These components manage different resources like a database, an expert system, or a compute engine. Even the communication network is considered a resource and is subsumed as one or more components. For example, a direct link between two sites is considered as one resource, while a LAN is considered another. Task service order is scheduled by a local scheduler residing at each component. These schedulers are all independent and do not collaborate. The only things that may influence scheduling decision are the real-time attributes associated with each task.

Newly created global tasks are first processed by the process manager. Figure 1 shows a single process manager, but in reality there can be many. Each manager controls one or more global tasks. We assume that certain control information of a global task, such as the precedence relationship among the subtasks, and the end-to-end deadline is available to its process manager. The major functions of the process manager are to submit the subtasks to the appropriate components for execution and enforce the precedence constraints among the subtasks of a global task. In some cases, the process manager has to generate scheduling information for a subtask before submitting it For example, if a component *A* accepts tasks with priority (and schedules them according to the priorities) and another component *B* schedules

FILE: /pub/keller/1994/predicate-cache.ps
FILE: /pub/keller/1994/predicate-cache-long.ps
COMMENT: The short version appeared in Parallel and Distributed Information Systems, September 1994

# A Predicate-based Caching Scheme for Client-Server Database Architectures

Arthur M. Keller*
Stanford University
Computer Science Department
Palo Alto, CA 94305-2140

Julie Basu
Stanford University
and
Oracle Corporation

## Abstract

*We propose a new client-side data caching scheme for relational databases with a central server and multiple clients. Data is loaded into a client cache based on queries, which are used to form predicates describing the cache contents. A subsequent query at the client may be satisfied in its local cache if we can determine that the query result is entirely contained in the cache. This issue is called 'cache completeness'. On the other hand, 'cache currency' deals with the effect of updates at the central database on the client caches. We examine various performance and optimization issues involved in addressing the questions of cache currency and completeness using predicate descriptions. Expected benefits of our approach over commonly used object ID-based caching include lower query response times, reduced message traffic, higher server throughput, and better scalability.*

## 1 Introduction

This paper addresses the issue of data caching in client-server relational databases with a central server and multiple clients that are individually connected to the server by a local area network. The database is resident at the server and transactions are initiated from client sites, with the server providing facilities for shared data access. Dynamic local caching of query results at client sites can enhance the overall performance of such a system, especially when the operational data spaces of the clients are mostly disjoint.

In typical commercial relational databases with client-server configurations [16], caching aims to avoid disk traffic and is done on the server side only, based on buffering of frequently accessed disk blocks or pages. The assumption is that clients are low-end workstations that are likely to be overloaded by local data processing; their function is accordingly limited to transmission of SQL queries across the network to the server, and presentation of the received results to the user. However, with the continuing precipitous drop in the price-performance ratio for workstations, the validity of this assumption comes into question. It is increasingly common to find clients that are high-end workstations with the server being a mainframe or a minicomputer. Such intelligent clients are capable of performing intensive computations on their own, using the database as a remote resource that is to be accessed only when necessary. Increased local functionality and autonomy can potentially lead to less network traffic, improved utilization of local computing power, faster response times and higher server throughput, as well as better scalability.

Several techniques to provide caching facilities at client sites have been recently investigated [21, 22]. All of these schemes use the notion of object IDs to store, retrieve and maintain cached objects at client sites. Such caching schemes can only support ID-based operations like *ReadObject* and *UpdateObject* within transactions; an associative query that accesses database objects using a predicate on a relation, e.g., through a WHERE clause of a SELECT-FROM-WHERE SQL statement, cannot be handled in these schemes. Similar observations can be made for the page-based caching schemes presented in [4, 5, 9] — none of these schemes can reuse locally cached data for associative query execution. We feel this is a major drawback of these caching schemes, since associative queries are very common in relational databases, and are indeed one of the major reasons for their success.

One possible way of supporting associative access in object-ID or page-based caching is to use indexes defined on the database at the server, as is done in some object-oriented databases. If server indexes exist that can determine the target objects or pages for an associative query, the client can fetch and examine the relevant index pages to see if current versions of the desired objects or pages are locally available. However, even in systems that have low to moderate update activity, index pages generally have very high contention, and may be subject to frequent invalidation or update. The need to reference server index pages locally is thus likely to cause increased network traffic and slower response times.

In our approach, queries executed at the srrwr are used to load the client cache, and *predicate descriptions* derived from these queries are stored at both the client and the srrwr to examine and maintain the contents of the cache. If a client determines from its local cache description that a new query is not completely computable locally, then the query (or a part of it) is sent to the server for processing. The result of this query is optionally added to the client cache, whose description is updated appropriately. On the other hand, a locally computable query is executed by the client on its cached data (the effect of such local query evaluation on concurrency control is discussed later). Each cache may also have its own local indexes or access paths to facilitate local query evaluation. To ensure the currency and validity of cached data, predicate descriptions of client cache contents are used by the server to notify each client of committed updates that are possibly relevant for its cache.

Consider, for example, an employee database managed by a central server, in which table EMPLOYEE(*emp_no, name, job, salary, dept_id*) records a unique employee number and other details of each employee. Suppose that a client caches the result of a query for all employees in department 100, along with a predicate description (*dept_id* = 100) for these tuples. Assuming that no update at the server has affected these EMPLOYEE tuples, a subsequent query at the same client for all managers in department 100, i.e., those employees that satisfy ((*dept_id* = 100) ∧ (*job*='manager')), can be answered using the cache associatively, and without referencing server index pages or communicating with the server (except, if deemed necessary, for purposes of concurrency control like locking the accessed objects at the server). Another query represented by the predicate (*job*='manager') asking for all managers can only partially be answered from the cache. In this case, the database could be requested either for all managers, or only for those not in department 100. This choice is an important optimization decision that can potentially speed up data transmission and query processing.

The situation is more complex if the cached data is out-of-date as a result of updates committed at the server. There are several choices for maintaining the currency of data cached at a client: automatic refresh by the server as transactions commit updates, invalidation of appropriate cached data and predicates, or refresh upon demand by a subsequent query. Both automatic and by-demand refresh procedures may again either be recomputations or incremental, i.e., performed either by cached query re-execution or by differential maintenance methods. Which method performs best depends on the characteristics of the database system such as the volume and nature of updates, pattern of local queries, and constraints on query response times. In our scheme, the maintenance method adopted is allowed to vary by client, and also for different query results cached at a client. A client may have results of frequently posed queries automatically refreshed, and may choose to invalidate upon update what is perceived as a random query result.

Examination and maintenance of cached tuples via predicate descriptions entails determining satisfiability of predicates, and concerns about overhead and scalability may naturally arise over reasoning with large numbers of predicates in a dynamic and *real-time* caching environment. In this paper, we attempt to address the practical design issues and trade-offs that pertain to this environment, with the conceptual structure as our primary focus.

To reduce the complexity of the reasoning process, we allow approximate algorithms that might sometimes err causing inefficiency, but can never produce incorrect results. A cache description used for determining *cache completeness* (i.e., whether a query can be completely or partially evaluated locally) need not be exact, but it can be *conservative*. In other words, data claimed to be in the client cache must actually be present in it, so that local query evaluation does not produce incomplete results; it is however not an error if an object residing in the cache is re-fetched from the server. Another description of a client's cache is maintained by the server for alerting the client of changes to its cached objects (the *cache currency* issue). This description can also be approximate, but can only be exact or *liberal*. That is, occasionally notifying the client of an irrelevant update is not a problem, but failure to notify the client that a cached object has changed can result in significant error.

Apart from the above approximation techniques, we investigate several optimizations applicable in our context. *Predicate indexing* mechanisms, simplification of cache descriptions through *predicate merging* and *query augmentation* are used to reduce caching costs (details of these techniques are beyond the scope of this paper). The expected net effect is a decrease in query response times and increase in server throughput compared to other systems, and improved scala-

*Arthur Keller's e-mail address is arkOcs.stanford.edu

bility with respect tn the number of clients

The paper is organized as follows. Section 2 gives an overview of related work. We outline the details of our schrmr in Section 3. Implementation issues and trade-offs are addressed in Section 4. Finally, we summarize our contributions, work in progress, and future plans in Section 5.

## 2 Related work

Our caching scheme is reminiscent of predicate locks used fnr concurrency control [8], where a transaction requests a lock on all tuples of a relation that satisfy a given predicate. Predicate lock implementations have not been very successful, mainly because they are pessimistic in nature and because of their execution cost [11] The pessimism arises because two predicates intersecting in the attribute space but without any tuples in their intersection for the particular database instance will nonetheless prevent two different transactions from simultaneously locking these predicates. This rule protects against phantoms, but reduces the allowable concurrency. Our caching scheme on the other hand can support predicate locks that are more optimistic, in that two transactions at different clients conflict (and arc notified by the server of the conflict) only when a tuple in the intersection nf shared predicates already exists or is actually updated or inserted.

Query containment [18] is a topic closely rrlatrd to the cache completeness question. Query evaluation on a set of *derived relations* is examined in [15] Efficient maintenance of materialized views has also been the subject of much research [2, 6. 10, 14], and is related to t hr issues examined in this paper. For example, our update notification scheme involves detecting whether an update has an effect on a client cache, and this has much in common with the elimination of updates that a r r irrelevant for a view [1]. The above techniques of query containment and materialized view maintenance, though directly applicable in our schrmr. have mostly been designed for relatively static situations with small numbers of queries or pre-defined views. Performance issues in handling large numbers of dynamic queries and views in a client-server environment have not been addressed in these papers.

Among other rrlatrd work, [17] proposes a view caching scheme that uses the notions of *extended logical access path* and *incremental access methods*. Results on the simulated performance of this scheme (and some variations) in a client-server environment arc in [7]. Update logs are maintained by the server(s), and each query against cached data at a client results itr an explicit refresh request to the server(s) to compute and propagate the relevant differential changes from these logs. In contrast, we follow an incremental and flexible notification strategy at the server, and attempt to split the workload of refreshing cached results more evenly amongst the server and the clients

Rule sy stems for triggers and active databases [12, 20] are related to our notification scheme, in the sense that given a database update, efficient identification of applicable rules is desired. This requires determining satisfiability of predicates, and efficiency issues similar to ours arise fnr such systems One difference is that for caching. notification by the server can afford to be approximate as long as it is liberal. Additionally, our notification scheme has t he capability of directly propagating certain update commands to relevant clients for local execution on cached data. instead o f propagating the tuples modified by the update. Therefore, unlike rule systems in active databases, we require that the notification system employed by the server handles not only 'point inputs' corresponding to single tuples, but also general query predicates over the attributes of a relation.

## 3 Our approach

We propose a predicate-based client-side caching scheme that. reduces query response times and network traffic between the clients and the sci ver by attempting to locally answer queries from the cached tuples using associated predicate descriptions. The database is assumed to be resident at. the central server, with users originating transactions from client sites. Each client executes transactions sequentially, with at most one transaction active at any time (concurrency control at. individual clients can be incorporated in our scheme, but is not considered in this paper).

### 3.1 Class of queries

Queries specify their target set o f objects using *query predicates*, as in the WHERE clause of a SELECT-FROM-W HERE SQL query. We allow general SELECT-PROJECT-JOIN queries over one or more relations, with the restriction that the keys o f all relations participating in a query must be included in the query result We feel this is not overly restrictive, since a query posed by the user that does n o t satisfy this constraint may optionally be *augmented* by a client to retrieve these key s from the server I ser-transparent query augment ation (discussed in Section 4) is in many cases a viable technique fnr reducing long-term costs of maintaining cached query results.

Transactions may also execute insert, delete, and update commands on a single relation. Insert commands that use subqueries to specify inserted tuples arc not considered in this paper.

Query predicates specified as above are classified as either *point query* or *range query* predicates. A point query predicate specifies a unique tuple (that may or may not exist) in a single relation, by conjunctively specifying exact values for all attributes that constitute its primary key, and possibly values for other non-key attributes as well. P o i n t query predicates arise frequently during navigation among tuples of different relations using foreign key s, e.g , a query about a tuple i n the relation DEPARTMENT( *dept_id*, *dept_name*, *director*) based on the matching value in the foreign key *dept_id* of an EMPLOYEE tuple. A range query predicate may specify either a single value or a value range for one or more attri butes, and may in general have zero, one, or more tuples in i t s target set. Note that a point query is a special case of a range query; we distinguish between the two only because they are processed differently for efficiency reasons. A general SELECT-PROJECT-JOIN query predicate is treated as a collection of range query predicates on the participating relations with some ranges being *variable*, i.e., specified in terms of join attributes of other relations.

### 3.2 A formal model of predicate-based caching

We now formalize our terminology using the usual predicate calculus notation. Suppose that there arc $n$ clients in the client-server sy stem, with $C_i$ being the $i$th client, $1 \le i \le n$. Let $Q_i^E$ be the number of query predicates such that the results of all queries corresponding to these predicates are cached at client $C_i$, where $Q_i^E \ge 0$. We denote by $P_{ij}^E$ the query predicate corresponding to the $j$th query result cached at client $C_i$ The superscript $E$ in $Q_i^E$ and $P_{ij}^E$ stands for *exact*, to represent the fart that these arc the precise count and exact for ms of cached query predicates respectively. Other information rrlatrd to a query may be associated to its query predicate, e g . the list of *visible* attributes retained aftrr a projection operation on the tuples selected by a W HERE clause condition

**Definition 1:** An *exact cache description* $ECD_i$ for the $i$th client $C_i$ i s defined to be the set o f exact query predicates $P_{ij}^E$ corresponding to all query results cached at the client

$$ECD_i = \{P_{ij}^E \mid 1 \le i \le n,\ 1 \le j \le Q_i^E\}.$$

In a real-life scenario, query predicates may be quite complex, and perfor mance problems may arise if precise reasoning methods are followed. To alleviate such problems, we introduce the notion of *approximate* cache descriptions. A client may use a simpler but *conservative* version of the exact cache description for determining cache completeness. As long as data thought to be in the cache is actually present in it, local evaluation of queries will produce correct and complete answers

**Definition 2:** A *conservative cache description* $CCD_i$ for the $i$th client $C_i$ is a collection of zero or more predicates $P_{ik}^C$ such that the union of these predicates is contained in the union of the predicates ill the exact cache description $ECD_i$ for the client. Let $Q_i^C$ denote the number of predicates in $CCD_i$. Formally,

$$CCD_i = \{P_{ik}^C \mid 1 \le i \le n,\ 1 \le k \le Q_i^C\},$$

where $Q_i^C \le Q_i^E$ and $\bigcup_{1 \le k \le Q_i^C} P_{ik}^C \in CCD_i$
$\Longrightarrow \bigcup_{1 \le j \le Q_i^E} P_{ij}^E \in ECD_i$.

The symbol $\Longrightarrow$ denotes the material implication operator, and in the context of query predicates has the following meaning if $Q \Longrightarrow P$, then the result of the query corresponding to predicate Q i s contained in and is computable from the result of the query corresponding to predicate $P$. Note that the number of predicates $Q_i^C$ in $CCD_i$ i s defined t o be possibly smaller, and never greater, than $Q_i^E$; this restriction corresponds to our informal notion that $CCD_i$ is simpler than $ECD_i$

One example o f conservative approximation o f a query predicate i s i t s simplification by discarding a disjunctive condition. For other possible differences between $ECD_i$ and $CCD_i$, consider some cached EMPLOYEE tuples that were fetched through a number of point queries, as well as through a few range queries $CCD_i$ may consist only o f the range query predicates, whereas $ECD_i$ includes a ll cached predicates, both point and range $CCD_i$ i s thus simpler than $ECD_i$, having eliminated point query predicates The result o f this approximation is that cached results of point queries are not taken into consideration when addressing the cache completeness question fnr range queries, likely speeding up the reasoning process. Thus, if all EMPLOYEE tuples in department 100 have been fetched through separate point queries, a range query on the EMPLOYEE relation with query predicate ($dept\_id = 100$) will result i n re-fetching these tuples from the sci ver Such a remote access is inefficient, but not incorrect in any way.

It is important to note that the conservative approximation pertains to the cache description only, and not to the cache contents. In the above example, single EMPLOYEE tuples cached through point

FILE: /pub/keller/1994/cnet-cat-arch.ps
COMMENT: Accepted for publication in Workshop on Electronic Commerce, December 1994

Paper 22

# A Smart Catalog and Brokering Architecture for Electronic Commerce

Arthur M. Keller
Michael R. Genesereth
Narinder P. Singh
Mustafa A. Syed

We present an architecture for smart catalogs and brokering that supports cross-search of multiple catalogs. Typically, the World Wide Web (WWW) resembles a giant menu system, where each document behaves like a menu of links to other documents that get the user closer to the information desired. For example, in order to find information about a particular product, you must separately visit each vendor of that product, and then navigate through that vendor's Web space to find the desired product. Making such navigation more difficult is that each vendor organizes its Web space differently. Keyword searching techniques, such as WAIS, are only of limited help, as they require that the user phrase the query in the terms of the information, and vendors tend to use different terminology from each other to describe products. Our approach is to support reverse-search of multiple vendor catalogs based on a deeper understanding of the contents of these catalogs.

Our architecture is illustrated in Figure 1. Product data is stored in databases, so that it may be readily searched and maintained. Such data will include structured information, parameters, text, pictures, sound, video, etc. Each product database communicates with a Catalog Agent using its native language, such as SQL. The Catalog Agent performs 3 roles: It advertises the coverage of a product database; it understands queries and translates them into the language of the product database, and it packages answers from the product database in a standard format.

The communication language used by the agents in our architecture is Agent Communication Language (ACL). ACL consists of the Knowledge Query and Manipulation Language (KQML), the Knowledge Interchange Format (KIF), and a set of Ontologies. KQML consists of performatives, such as ask-one, ask-all, and tell, that describe the nature of the action to be taken. KQML has the role of the communication language in CORBA. KIF is based on First-Order Predicate Calculus and is the content language we use with KQML. KIF is powerful enough to contain or to encapsulate any other content language, so that any information may be obtained from the information source, translated to the desired format, and transmitted to the requestor, assuming the necessary components exist. Each product database is described by an Ontology, which defines the database, its structure, the terms used in it, and how they relate to each other. Base ontologies are used to define common terms that may be used by product ontologies. Translation ontologies are used to translate specific terms used in one database to related terms used in another database.

A Facilitator in our architecture acts as a broker. It stores agent-provided advertisements of coverage in a knowledge base along with relevant ontologies. It uses to advertisements to determine which agents can support a particular request. And it translates requests into the language and terms used by a responding agent and also translates responses into the language and terms used by the requesting agent. A Facilitator will decompose requests requiring action by multiple agents and then compose the responses for the requestor.

Someone using a WWW client, such as Mosaic, will connect to a User Agent using the ordinary WWW protocols HTTP and HTML. The user will describe the desired object using an HTML form. For example, the request may be to find color printers for the Macintosh costing under $1000. The User Agent will translate the query into KIF and submit it to a Facilitator. The Facilitator handling the query will consult its knowledge base for the facilitators or agents that can handle this request. For example, the Facilitator may transmit the request to the Catalog Agents for Apple and for Hewlett-Packard. The Catalog Agent will then interrogate the product database and translate the answer into KIF. For example, the Hewlett-Packard Catalog Agent may respond with the description of the HP 560C printer. The Apple Catalog Agent may respond with the Apple Color StyleWriter Pro. The facilitator will then collect these responses for the User Agent, which will package the responses in HTML for the Mosaic client.

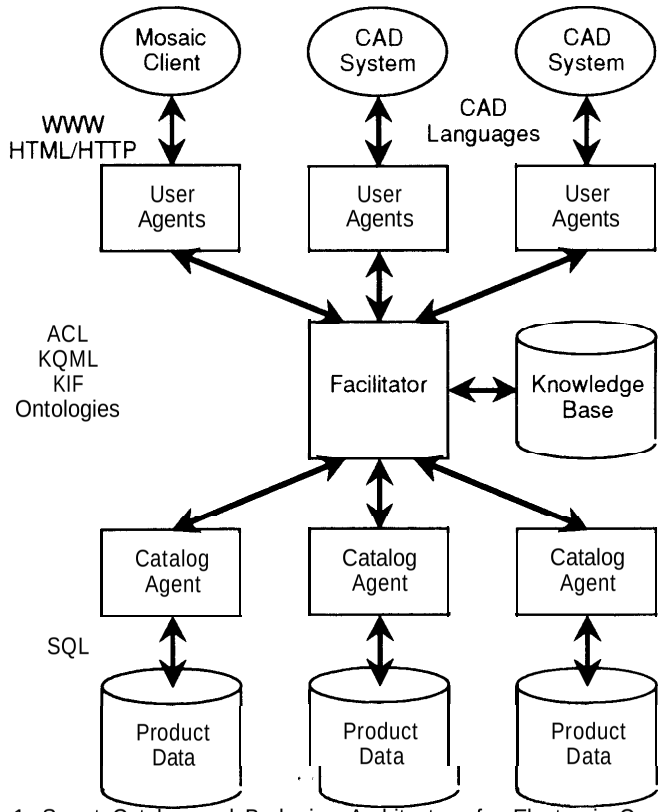



Figure 1. Smart Catalog and Brokering Architecture for Electronic Commerce

This architecture has been demonstrated in the domains of software interoperability and concurrent engineering. It is now being applied to the domain of electronic commerce as part of Stanford's Center for Information Technology's (CIT) efforts on CommerceNet. Please contact Arthur Keller at ark@cs . stanf ord. edu for more information.

# Querying Semistructured Heterogeneous Information*

Dallan Quass, Anand Rajaraman, Yehoshua Sagiv†, Jeffrey Ullman, Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
{quass,anand,sagiv,ullman,widom}@cs.stanford.edu

Abstract

Querying and integrating data from multiple, heterogeneous information sources usually requires dealing with irregular and incomplete data. Data models and query languages designed for well structured data with a fixed schema are inappropriate in this environment. In the TSIMMIS project at Stanford we are using a "lightweight" object model (called OEM) for data exchange and integration. This paper describes the lightweight-object query language (LOHEL) and object repository (LORE) we have developed for the OEM model. In the paper we motivate the need for a lightweight approach, we describe the syntax and informal semantics of LOREL (a complete denotational semantics is available by anonymous ftp), and we present the basic architecture and query processing strategy of LORE.

## 1 Introduction

An increasing amount of information becoming available electronically to the casual user, and the information is managed under an increasing diversity of data models and access mechanisms. There is a strong need to provide a uniform query environment helping users access the information they desire. The goal of the TSIMMIS[1] project at Stanford is to provide a framework and tools to assist in integrating and accessing information from multiple, heterogeneous information sources [7]. In TSIMMIS we have identified three areas of focus:

1. **Information exchange.** A common data model and data exchange format are necessary to exchange information between the various components of the framework [17]. The model must be flexible enough to realize the variety of object relationships and structures supported by the underlying information sources, and it must be able to handle *semistructured* and unstructured data in addition to structured data. By semistructured, we mean that there is no schema fixed in advance, and the information may be irregular or incomplete. The data exchange format in TSIMMIS is *self-describing*, so that data sent between components can be parsed by the receiving component without external context (such as a global schema).

2. **Information discovery and browsing.** A common query language is needed for communicating requests among the components of the framework. While we do not expect every information source to be able to answer every query expressible in the language, we do expect the language to be powerful enough to express most of the kinds of queries that can be evaluated by the information sources. On the other hand, the language should not be so complex that it becomes impractical to translate to source-specific query mechanisms. The language must facilitate retrieval of information through ad-hoc queries from casual users and allow discovery of the structure of the information, in addition to the information itself.

3. **Mediators.** A mediator is a program to provide a more meaningful view of the underlying data by collecting data from one or more sources, processing and combining it, and exporting the resulting information [24].

This paper focuses primarily on the the query language we have developed and the architecture of an object repository, LORE[2], supporting our language. We describe the data model briefly, only to the extent that it is necessary to understand the query language. A complete description of the model and its benefits has been given previously in [17]. Mediators also are described in more detail elsewhere [24].

We realize that (100) many query languages already exist. However, rather than choose an existing language for our task, we have chosen to develop a new one: The requirements for querying semistructured heterogeneous information are sufficiently different from traditional requirements that we feel a new language is justified. What are these new requirements? Clearly a query language for our environment must be able to express relationships among objects, since the relationships are not known by a casual user. Both nested-relation and object-oriented query languages express object relationships easily, but they are designed primarily for access to well structured data whose schema is known. In addition, object-oriented **query** languages focus especially on facilitating programmer access, supporting different kinds of built-in and extensible object structures and methods. We term such languages (and their underlying models) *heavyweight*, in that they enforce strong typing, provide different ways of dealing with sets, arrays, and record object structures, and include other features important for queries embedded in programs but perhaps too strong for casual users.

We propose a *lightweight* object query language, called LOREL (for LORE Language), aimed specifically at semistructured information. Recall that by semistructured, we mean that there is no schema fixed in advance, and the information may be irregular or incomplete. Information integrated from heterogeneous sources almost always is semistructured, because object structure varies between sources, and even sophisticated mediators are unlikely to make all information have identical structure and corresponding contents. We have the following design objectives for our language.

- It should allow the user to specify queries declaratively.
- It should have a well understood formal semantics.
- It should contain the power of typical query languages.
- Queries applied to partial or missing data should not return errors or counterintuitive results.

*This work was supported by ARPA Contract F33615-93-1-1339, by the Anderson Faculty Scholar Fund, and by equipment grants from Digital Equipment Corporation and IBM Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements of the US Government.

†Permanent Address: Department of Computer Science, Hebrew University, Jerusalem, Israel

[1] As an acronym, TSIMMIS stands for "The Stanford-IBM Manager of Multiple Information Sources." In addition, Tsimmis is a Yiddish word for a stew with "heterogeneous" fruits and vegetables integrated into a surprisingly tasty whole.

[2] Lightweight Object REpository; also Data's sinister elder brother, to Star Trek fans, which brings up the question of whether a LORE-base is some kind of evil database.

- The language should treat similar concepts from different information sources similarly, despite minor differences in object structure.
- The language should exploit a few basic concepts that are easy for the casual user to understand. Simplicity will also facilitate its translation to source-specific query mechanisms.
- The language should allow querying when the object structure is unknown or partially known.
- The language should allow schema browsing.

LOREL has been developed specifically to meet these objectives. The following are highlights of our approach.

- We have a simple declarative semantics, which has been defined formally using a denotational approach [23]. The process of formal definition turned out to be very helpful, as it helped uncover several anomalies that might otherwise have gone unnoticed.
- As in [5, 13, 26] we use *path expressions*, but unlike most object-oriented query languages, path expressions automatically "flatten" nested structures [13]. For example, the expression Library. Book. Author evaluates to a single set of authors, instead of a set of author-sets. There is no need to introduce range variables or a "collapse" operator.
- We use heterogeneous sets to model both record structures and homogeneous sets, enabling the same syntax for both single- and set-valued attributes and allowing us to handle missing data without introducing the complexities normally associated with null values.
- There is essentially no type-checking. If a comparison operator such as < is applied to operands that are not comparable, instead of returning an error the predicate just returns false. The absence of type-checking is one example of a language characteristic that is worthwhile for our semistructured environment, but inappropriate for queries over well structured data.
- We allow "wildcards" in path expressions to facilitate queries when the structure of the data is only partially known. The absence of type-checking permits queries with wildcards that would not be legal in a strongly typed language.
- Queries can request either object values or their self-describing *labels*. Querying over object labels permits schema browsing.

We are currently implementing LORE, an object repository supporting our data model and query language. LORE will be a component of several parts of the TSIMMIS framework (see Section 4). With LORE, clients will be able to store query results for later recall, and translators (information source wrappers) and mediators will be able to manipulate intermediate results during query execution. In addition, LORE is a convenient way to bring new data into TSIMMIS. Ultimately, LORE could serve as a *data warehouse*, importing large quantities of (possibly semistructured) data from different sources, and storing it in an integrated form especially suited to querying [12].

### 1.1 Related Work

Several articles have pointed out the need for new data models and query languages to integrate heterogeneous information sources, e.g., [15, 16, 18]. However, most of the research in heterogeneous database integration has focused on integrating data in well structured databases [3]. In particular, systems such as Pegasus [20] and UniSQL/M [14] are designed to integrate data in object-oriented and relational databases. Some systems, such as GAIA [21] and Willow [11], address the other end of the spectrum, providing uniform access to data with minimal structure.

The goal of the TSIMMIS project is to uniformly handle unstructured, semistructured, and well structured data [17]. In this goal our effort is similar to the work on integrating SGML [1] documents with relational databases [4], or with object-oriented databases such as OpenODB [25] or $O_2$ [8]. While their approaches extend existing data models and languages [2, 10], our language and data model are designed specifically for integration. Designing a new language has its advantages and disadvantages, of course. A disadvantage is that we are unable to manage our objects using an existing DBMS. An advantage is that we do not have to work around the limitations of a data model and language designed originally for querying with structured data with a fixed schema.

LOREL is similar to query languages for a number of object-oriented [2, 5, 6] and nested relational [9] systems. Path expressions, for example, have been incorporated into many query languages since their use in the language GEM [26]. However, since our approach emphasizes querying semistructured data, where the schema may be only partially known, our path expressions are more flexible than most. For example, we handle values ranging over an empty set more flexibly than [6]. We allow schema browsing by querying object labels, and we allow wildcards within path expressions to facilitate queries when the object structure is partially known. In this respect we are closest to the query languages of [8, 13], which also allow querying the schema and wildcards within path expressions. The main difference is that the simplicity of our object model yields many fewer concepts in the query language, resulting in a language that we believe is more appropriate for the casual user.

### 1.2 Outline of Paper

Section 2 describes the data model upon which our language, LOREL, is based. An exposition of LOREL using a series of examples appears in Section 3. Section 3 also includes an informal description of LOREL's semantics. Section 4 describes the LORE system and provides an overview of how queries are executed. Conclusions and future work are given in Section 5. The complete LOREL syntax is in Appendix A. Due to space limitations, we have not included the denotational semantics of the language in this paper—it is included in the full version, which is available by anonymous ftp [19].

## 2 Data Model

In the TSIMMIS project we have developed a simple data model called OEM (for Object Exchange Model) [17], based on tagged values. Every object in our model has an *identifier*, a *label*, and a *value*. The *identifier* is some value that uniquely identifies the object among all objects in the domain of interest. The *label* is a string presumably denoting the "meaning" of the object. Labels may be used to group objects by assigning the same label to related objects. The *value* can be of a scalar type, such as integer or string, or it can be a set of (sub)objects. We define *atomic objects* as objects with scalar values, and *complex objects* as objects whose values are sets of subobjects. ***Note***

FILE: /pub/rajaraman/1994/limited-opsets.ps
COMMENT: To appear in PODS '95

Paper 24

# Query Optimization for Limited Operation Sets (Extended Abstract)

Anand Rajaraman
Yehoshua Sagiv†
Jeffrey D. Ullman
*Department of Computer Science*
*Stanford University*

*ABSTRACT*

When integrating heterogeneous information resources, it may be the case that the forms of queries that can be answered by a source is rather limited. If a query is asked of the entire system, we have a new kind of optimization problem, in which we must try to express the given query in terms of the limited query forms that this source can answer. For the case of conjunctive queries, we show how to decide with a nondeterministic polynomial-time algorithm whether the given query can be answered. We then extend our results to allow arithmetic comparisons in the given query and in the query forms.

## I. Motivation

A data-integration system such as Tsimmis (Papakonstantinou, Garcia, and Widom [1994], Chawathe et *al.* [1994]) translates information sources of arbitrary type into a common data model and language. If a source is an SQL database, then its interface with the Tsimmis system is fairly clear: one can ask it SQL queries over its database schema and nothing else. However, the source may be radically different from an SQL database; it could be a collection of ASCII documents, a spreadsheet file, and so on.

In these situations, it is necessary to interface the source with the common language and model. We not only need to relate terms of the global model to terms that appear at the various sources (which we must do even if the sources are SQL databases). We must also describe how queries in the global query language are carried out at each source. The approach taken by Tsimmis is to use a *translator generator*. The input to the generator is a list of rules that associate a parameterized query in the global language with the operation(s) to be carried out at the source.

**Example** 1: Suppose the information source is a genealogy, and the only two query forms it can answer are:

1. Given any individual $C$, find $C$'s parents.
2. Find all the individuals who have parents specified by the information source.

† Permanent address: Dept. of CS, Hebrew Univ., Jerusalem, Israel.

Work supported by NSF grant IRI–92–23405 and USAF grant F33615–93–1–1339.

There are various reasons why the source might be able to respond to only a limited set of query forms. For example, it could be that the source is a conventional relation with an index on children, so it is easy to ask question (1) but hard to ask the inverse question "find the children of a given individual." The same index supports question (2), although not cheaply. As another example, some bibliographic search sources require that at least one argument (e.g., name of author or title of book) be bound.

Now, let us see how we could answer some queries that are not of the form (1) or (2). First consider query "Find the grandparents of individual $a$." This query is answered by:

*i*) Find the set $P$ of parents of $a$, using query (1).

*ii*) For each individual $p$ in set $P$, use query (1) to find the parents of $p$.

*iii*) The answer is the union of all the individuals found in step (*ii*).

Next, consider the query "Find the grandchildren of individual $g$." This query cannot be answered at all if we only have query (1) to work with. However, with the help of query (2) (find all individuals), we *can* answer this query as follows:

*i*) Use (2) to find all individuals.

*ii*) Use (1) to find the parents of the individuals found in (*i*).

*iii*) Use (1) to find the parents of the individuals found in (*ii*).

*iv*) Find those individuals from (*iii*) such that $g$ is one of their grandparents.

Evidently, this solution is much more expensive than we would like, since it examines the entire genealogy instead of proceeding from the given individual. However, with queries (1) and (2) as our only information-gathering options, there is no more efficient way to answer the query EI

## II. A Formal Model

Example 1 suggests the following way to model a set of given query forms and queries that may or may not be answerable from these forms.

1. We assume that there are certain predicates about which queries are posed. These predicates correspond to concepts that are in the shared of global model for the information to be integrated. In Example 1, *parent*($C$, $P$) would be such a predicate.
2. We assume that there are certain given query forms, which we call *views* for consistency with certain other works to be discussed in Section III. A view consists of a "head" and a "body." The *head* consists of:
   a) A predicate denoting the view,
   b) Arguments for the predicate, and
   c) A binding pattern (or adornment, as in Ullman [1989]) indicating which arguments of the predicate are expected to be bound (provided as parameters of the query represented by the view) and which are free (produced as answers to the query).

The *body* of the view is a program in some notation that produces a result to the query.

**Example 2:** Consider Example 1. The two queries we are allowed to perform on the genealogy become two views. The parameters of the queries are the bound arguments, and the results are the free arguments. We assume *parent* is the predicate representing child-parent information. Thus, we canwritequery(1,) as:

$$v_1^{bf}(C,P) \text{ :- parent } (C,P)$$

That is, $v_1$ expects a binding for $C$, which becomes the first argument of *parent*, and it produces values for $P$, the second argument of *parent*.

Similarly, query (2) is written:

$$v_2^{f}(C) \text{ :- parent } (C, P)$$

That is, with no argument, all the values of $C$ that are a firstargument of some *parent* fact are produced. □

### Queries

Now, we must define what it means for a program to be a solution to a given query. A *query* is denoted exactly as a view. That is, it has a head with a binding pattern and it has a body that is a program over the predicates that represent the information of the source.

**Example 3:** The query of Example 1 in which we are asked to find the grandparents of individual $C$ is expressed

$$answer_1^{bf}(C,G) \text{ : - } \text{parent}(C,P) \text{ \& } \text{parent}(P,G)$$

That is, we are given a binding for $C$ and are asked to find the related values of $G$ according to the program in the body that composes *parent* with itself.

The query

$$answer_2^{fb}(C,G) \text{ :- } \text{parent}(C,P) \text{ \& } \text{parent}(P,G)$$

looks similar, but expresses the more difficult (for the views of Example 1) query in which we are given a grandparent and asked to find the grandchildren. Cl

### Valid Solutions

A query is solved by a program that uses views to obtain answers from the external world. In general, the program can create its own data, corresponding to IDB predicates in datalog programs (see Ullman [1988]), and it can use arithmetic comparisons or other predicates that have a meaning independent of any information source.

In most of what follows, we Slllll assume that solutions are conjunctive queries in the form of Chandra and Merlin [1977], although we shall also consider bodies that are datalog programs over the views. For each notation used to describe programs that are solutions, we need to define precisely what it means for the views to be used properly, i.e., for the solution to be *valid*.

### Conjunctive Queries as Solutions

The simplest case is when the views and solutions are conjunctive queries. All subgoals have view names as predicates, and we assume they are evaluated in left-to-right order. There are two conditions a solution must satisfy to be valid.

1. The binding patterns must be appropriate. That is, if the $i$th subgoal has predicate $v_j$, and the binding pattern for the head of view $v_j$ requires a certain argument be bound, then any variable appearing in that argument of the $i$th subgoal must either appear in one of the first $i-1$ subgoals, or it must appear in the query head in a bound argument (see Ullman [1989] for details of how bindings pass from arguments to variables). It is not required that other arguments of the $i$th subgoal be "free."
2. The *expansion* of the solution, in which each subgoal is replaced by the body of the appropriate view, must be equivalent to the given query.

**Example 4:** Consider our running example, with views $v_1^{bf}$ and $v_2^{f}$ from Example 2 and the query $answer_1^{bf}$ from Example 3 (find the grandparents of $C$). An appropriate solution is

$$answer_1^{bf}(C,G) \text{ :- } v_1^{bf}(C,P) \text{ \& } v_1^{bf}(P,G) \qquad (1)$$

This solution satisfies the binding-pattern condition because in the first subgoal, the first argument $C$ is bound by the head,and iii the second subgoal, the first argument $P$ is bound by the first. subgoal.

We must also expand solution (1) to check that it is equivalent to the given query. In this case, we replace $v_1^{bf}(C,P)$ by the definition according to Example 2, giving us *parent*$(C,P)$ in place of the first subgoal. We must replace the second subgoal similarly by *parent*$(P,G)$. The expansion of (1) is thus

$$answer_1^{bf}(C,G) \text{ :- } \text{parent } (C,P) \text{ \& } \text{parent}(P,G)$$

This conjunctive query is identical to the query from Example 3, so surely it is equivalent to that query. We conclude that the proposed solution (1) is valid. □

**Example 5:** Now consider the second query from Example 3, where we are asked to find grandchildren. The following is *not* a valid solution:

$$answer_2^{fb}(C,G) \text{ :- } v_1^{bf}(C,P) \text{ \& } v_1^{bf}(P,G)$$

The reason is that in the first subgoal, the first argument $C$ does not receive a binding from the head, which only binds the second argument $G$. Moreover, reversing the order of the subgoals does not help; the first argument $P$ is then free.

However, using view $v_2$ we can produce a valid solution: {

$$answer_2^{fb}(C,G) \text{ :- } v_2^{f}(C) \text{ k } v_1^{bf}(C,P) \text{ \& } v_1^{bf}(P,G) \qquad (2)$$

Note that the first subgoal, with predicate $v_2$, requires no bound argument but provides a binding for $C$. That binding allows the second subgoal to have its first argument bound, as $v_1^{bf}$ requires. The third subgoal also has its required binding for the first argument. the fact that its second arglument is also bound, because $G$ appears in a bound argument of

FILE: /pub/gio/1994/medont.ps
COMMENT: Proceedings International Symposium on Fifth Generation Computer Systems (FGCS) 94, Workshop on Heterogeneous Cooperative Knowledge-Bases, Vol.W3, pages 33-48, ICOT, Tokyo, Japan, Dec. 1994

Paper 25

# Interoperation, Mediation, and Ontologies

**Gio Wiederhold**
Stanford University
November 9, 1994

**Abstract**

In this paper we address the problem of interoperation at a semantic level. We assume that emerging standards will solve most of the syntactic infrastructure problems that exist today. However, interoperation has to deal with sources from differing domains, and their semantic differences.

We present mediation as the principal means to resolve problems of semantic interoperation. Since the number of domains is large we cannot foresee a global solution to that problem, and hence will have to deal with domain interaction incrementally, and bring domains together as and when needed. To formalize the processing in mediation we will need formal models of the source domains, of the articulation points, and of the customer needs. We expect to capture these models using tools developed in the ARPA knowledge-sharing projects. The basis for these models are domain ontologies. We propose a knowledge-based algebra to manage the interoperation of information from different domains. Several research issues that arise in the formalization of semantic interoperation are indicated.

## 1. Introduction

Several centuries ago, few people traveled. Most worked on a farm or pursued a trade at home, and occasionally walked to the local market. Soldiers marched long distances, mariners sailed, knights rode horses, and only some adventurous merchants and the crusaders used multiple means of transport, as camels, horses, and ships. Today, traveling long distances routinely requires a variety of conveyances, each suited for its particular domain, say buses, trains, planes, shipe, barges, bicycles, cars, trucks, and the like. Vehicles that have to operate in multiple domains, for example, amphibious carriers, tend to be less efficient in each mode than domain-specific vehicles. Even similar vehicles become attached to specific domains; once a tanker truck has carried gasoline, it is no longer a desirable vehicle for transporting milk. Specialization may also be needed to deal with quantitative differences. An old car may well be adequate to go shopping, but using it may be too risky for delivering supplies to customers. Driving a personal car cross-country is possible, but rarely feasible in business; for long distances air travel dominates. To switch among transport modes, interchange hubs are established, where people can wait, obtain tickets, and where goods are repackaged to suit the means of transport and the consumer. Many businesses spring up around the hubs, taking advantage of the traffic and adding value to the goods passing through.

On the information highways, we are encountering diversity of roads and vehicles as well. The diversity becomes a greater concern as we reach out beyond our homes and workplaces. While once computers operated in stand-alone mode, or used simple connections to each other, the information highways we are contemplating are likely to have millions of participating computers, and thousands of interchange points. While the notion of having similar computers, and similar data and information structures everywhere would make life simple, such a coherence is clearly not feasible, just as we could not have a single mode of transport nor a single type of container for all the goods to be shipped. Progress also requires change, and incremental changes also create inconsistencies. While Henry Ford might have been content if we all stayed with the Model T, over time most Model T's were replaced by a variety of faster, bigger, and more colorful vehicles.

## 2. Mediators

The software equivalent of interchange hubs are called mediators. We will summarize here the essential aspects of mediation, an activity which reduces data to information by applying knowledge about resources, search strategies, and user requirements [Wiederhold:92C]. Mediation is an integrating concept, combining a number of current technologies to find and transform data, and making the resulting information available at hubs along the information highways. Mediation recognizes the autonomy and diversity of the data systems and information services that support the hubs, and the user applications that utilize them. The autonomy of the participants enables the overall system to grow, since new sources, new means of transport, and novel information processes can be inserted. Incremental growth only requires that a few mediating hubs be adapted to link the new facilities into the traffic network. As the new facilities become more popular, further mediators will adapt to take advantage of them and the business they represent. Those users that need the new sources will use the adapted mediators; users that don't care remain unaffected.

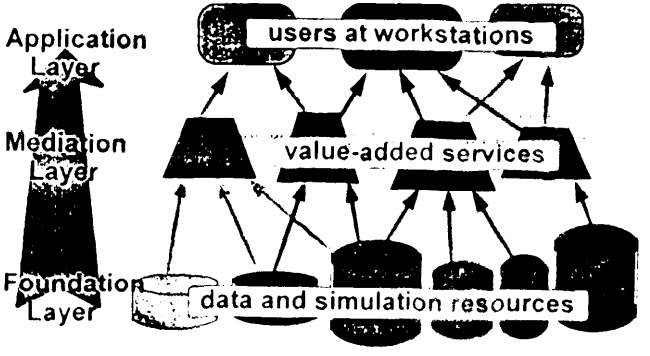



Figure 1. Transforming Data to Information

Value-added services in a mediator include combinations of:

1 Focused access to a variety of resources [Gravano:94]
2 Selection of relevant source material [Salton:90]
3 Resolution of scope mismatches [Wiederhold:92]
4 Abstraction to bring material to matching levels of granularity
5 Integration of material from diverse source domains
6 Assessment of quality of material from diverse sources
7 Ranking in terms of quality or relevance of material from diverse sources
8 Omission of replicated or known information
9 Seeking exceptions from expected values or trends
10 Transformation of material to make presentation effective for the customer
11 Adaptation to the bandwidth and media capabilities of the customer
12 Optimization to provide small response times or low cost

This list is implies a wide range of software technology. Remote, specialized services may be invoked by a mediator, just as transportation hubs encourage specialized factories to provide value-added services. Sometimes intensive processing is needed to abstract and merge data. Such processing may be performed at yet other nodes in the network, including high-performance computers.

The implementation of mediators varies greatly. If knowledge-based processing is crucial, we may find mediators programmed in languages as LISP. If optimization is crucial to processing, the mediators may depend on packages written in FORTRAN. Maintenance would be enhanced by using declarative approaches that could be understood and modified by end-users. Many of our current mediators have been coded in the c language.

Projects that are using mediators today include manufacturing systems, as design of portions of a new fighter aircraft at Lockheed Aeronautical Systems and gimbals for antenna positioning on spacecraft at Lockheed Space Systems. Access to data for healthcare services, managed at the University of Texas in Arlington, is slated to use mediator technology. Early applications have been in the integration of information for flexible military command systerns. The technology for these projects is supplied by a variety of vendors, ISX corporation serves as a contact point for these projects [ISX:94].

Since we assume that most mediation services are performed in autonomous computing nodes, the requirement for interoperation is the ability to communicate according to some standard conventions [GK:94]. To enable a greater variety of communication actions, participants, and representations a *Knowledge Query and Manipulation* Language (KQML) has been developed [FFMM:94]. Heterogeneity of computing platforms, operating systems, and message passing infrastructures is being overcome by concerted efforts in many communities. For mediation we must also consider the terminology and representation conventions.

### 2.1 Domain-specificity

Just as hubs for transport have become specialized to deal with people, mail, goods, foodstuff as fruit, fish, beef, and the like, mediators will also be specialized. Specialization makes maintenance feasible; an expert can focus on one's own domain, without having to consider the different constraints imposed by handling unrelated domains, say, fish versus bicycles [Haddock:94]. Differing domains may be best served by different programming paradigms, say finance versus engineering. Other subsets of domains may use similar technology, but differ in the concepts and structure of their knowledge representations, say electronic versus civil engineering. Most of this paper will deal with the latter issue, namely the management of diverse *ontologies*.

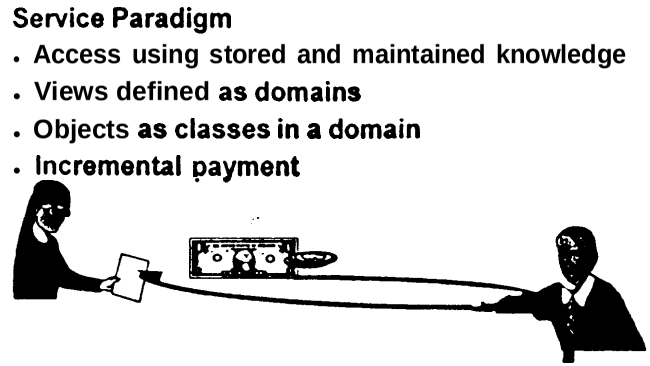



Figure 2. Some Features of Mediators

### 2.2 Distribution

Mediation is achieved by software. A mediator transforms data available on the network to make it more suitable and relevant to the consumer. This software function can be carried out on the computer where the mediation was developed, or can be assigned to other computers on the network. Since software is easy to copy over the digital highways and mediators are not directly bound to large data files, mediators can be rapidly replicated. Software is typically much smaller than to data it processes, so that it is easy to install at any location where compatible computers are available. The ease with which software-based factories can change location implies a much more flexible configuration of the network than seen in traditional manufacturing. A supplier who fails to deliver sufficient added value can be rapidly replaced.

Replication can also enhance the technical performance of mediators. By locating the mediator near the data source communication load can be reduced, since the resulting information is typically smaller. If the processing algorithms used in mediation are costly, the software can be moved to a high performance processor on the network, again improving response time.

If demand for a particular type of mediator is strong, replicates can be distributed to additional sites in the network. Since now the communication links will shorten, response time is enhanced as well.

FILE: /pub/gio/1994/onto.ps
COMMENT: Proceedings of 1994 Monterey Workshop on Formal Methods for Computer-Aided Software Development, Sep. 1994, U.S. Naval Postgraduate School, Monterey CA, pages 66-60

Paper 26

# An Algebra for Ontology Composition

(Abstract)

Gio Wiederhold

ARPA and Stanford University
July 1994

## INTRODUCTION

To compose large scale software there has to be agreement about the terms, since our models depend on symbolic linkages among the components. In modestly-sized systems, we implicitely count on such an agreement. Within a specific domain terms are indeed likely to be consistent, so that specifications can be developed from English (or other natural) language source documents. When combined with a coherent framework we have the underpinnings for a Domain-Specific-Software Architecture (DSSA). In this abstract we propose extending concepts used in object-based structural algebras and DSSA research to a knowledge-based algebra, suitable for composing larger systems that span multiple domains.

The principal operations in the algebras are simple and provide for selection from the objects in the source domain space and placing them into new domains that represent the information needed for the composed results. The algebra enables partitioned management of domain ontologies, assuring scalability.

## 1. BACKGROUND

Divide-and-conquer is an essential approach in science and technology, and in large software systems as well. Early manifestations of division of tasks in software were scientific subroutine libraries, since their development and evaluation required uncommon rigor. These libraries grew to encompass statistical procedures SAS [Rosc:83], and specialized libraries serve diverse domains as planetary navigation NASA [Acton:93] and Graphical User Interfaces. Today commercial firms or funded service agencies provide most of such libraries.

An alternative approach is the development of packages, which are not intended to be integrated into larger suites. These were also popular in the statistical domain, as BMD [Dixon:69], but have been largely supplanted by composable routines, which allow the application of statistics to a variety of application domains.

In this abstract we consider the construction of software from autonomous modules, *megaprogramming* [Wiederhold:92]. We refer to the scope of autonomous modules as their domain, and to the terms used to describe items and their relationships in a domain as their domain ontologies. While the terms in these ontologies are often only manipulated in paper form or perhaps as IDEF [Loomis:87] documents, we believe that ontologies warrant formal manipulation if reliable systems are to be composed.

## 2. DOMAIN SPECIFIC KNOWLEDGE

An attraction of object technology is the incorporation of semantics within the units that the software and retrieval strategies deal with. Object technology rapidly permits software composition by combining well-defined objects into larger objects and programs [Luqi:89]. The definitions that make retrieved objects coherent are particular to a specific application area and its domain. The focus of research in Domain-Specific-Software-Architecture (DSSA) has been the acquisition of knowledge in a specific *domain*. A working definition of a domain is an area of science or products where there is a common, shared *ontology* Gruber:93].

Having a common ontology enables collaborators to work together with minimal risk of misunderstanding each other. When computer systems are used as the intermediaries in collaborative work, the need for a common ontology is even greater because many of the cues that exist in personal face-to-face interaction: the raised eyebrow, the wandering of attention, up to the breaking of pencils, cannot be perceived by one's correspondent.

The architectural aspect of a DSSA approach is that, once enough knowledge has been garnered about a specific domain, the object classes can be defined and placed in an operational relationship to each other [Haddock:94]. Within a domain, we assume consistency, namely, that the terms mean the same thing, i.e., refer to the identical object instances [Wiederhold:91], and have the same relationship.

## 3. DOMAIN DIFFERENCES

Having defined domains by their internal consistency, we must now consider the cases where such consistency does not hold. First of all, different domains will consider different objects. Different domains are likely to have different ontologies. These differences can be simply due to differences in naming and scope, both with respect to the names and semantics of meta-information about attributes that appear in the schema and the names and semantics that appear as values in the content of a database:

1. Naming attribute items differently. This is common, but is also the easiest inconsistency to resolve. A example of this type occurs when employees are named in the payroll domain EMP and in the personnel domain PEOPLE. A simple table can be used to support the desired match and bring the information together.
2. Scope differences are much more insidious, and have to be determined by content analysis. The personnel domain may include assignees from other institutions, who are not listed in payroll. The payroll may include support for student benefits for employee's children, but those children are, appropriatly, not listed in personnel. Resolution of the differences requires establishing, validating, and processing of rules. These rules may have to refer to variables in one or the other source domain that are not part of the intended domain intersection.
3. Encoding differences of values are common as well. When numbers are used, a conversion can be established with a formula, say `meter = foot/0.305`. More complex are differences in dates and identifiers, say ssn with or without hyphens. Here rules have to be introduced as well, but when encodings are irregular, for instance `stock-codes`, tables have to be introduced. Tables dealing with instances of values occuring in databases require ongoing maintenance. We can hope that practical interoperation provides feedback which eventually will encourage coherence among domains.
4. Attribute scopes are often subjective. The term hot has a different meaning in

the **weather** domain than the **truck-engine** or truck-cargo domains. If **hot ueather** can effect the truck-engine. expert knowledge is needed to make the linkage. Differences in scope lead to differences in referencing, which makes their resolution yet more critical. For example, both **patient** s and **nurses** are subsets of people. but their roles in a hospital are quite distinct, so that it would be unwise to create generalized **people** *f* bjects and encapsulate all the differences internally.

No central organization can resolve all these differences, they require knowledge about the source domains and their intersection. Fortunately, not all differences among domains need to be reolved. Only tllf: terms that are needed to connect domains, refered to already as the *intersection* of domains need to be resolved. For retrieval terms in the intersection are used to articulate tllf! domains [Collet:91].

The differences enumerated above can make a once-and-for-all integration of distinct domain infeasible. A dynamic capability is essential if we wish to achieve associative access to multiple domains, since the transformations required to achieve optimization must maintain the correct semantics. For instance, the PENGUIN system constructs objects as needed out of relational databases, given a structural model of connecting references [Barsalou:91].

## 4. DOMAIN MERGING

There are several approaches to dealing with building composed ontologies from domains that have ontological differences:

1. Aggregate the terms from all relevant ontologies, give them to a committee, and ask them to prepare definitions that are acceptable to all. When tllf! definitions are completely documented, release the document and expect that all participants will adjust their usage to conform to the definitions.
2. Assume that terms match, and when mismatches are discovered, make the terms distinct, typically by prefixing them with source or domain identifier. This is the approach used by UMLS [Humphreys:92]; all the sources are labeled to make such distinctions easy, and by CYC, where micro theories can encapsulate differences [Lenat:90]. Over time, the processes of sharing of information encouraged by the availability of the joint ontology will cause convergence of meanings, although coherence can never be assured.
3. Assume that terms never mean the same thing unless explicit ly instructed. Such instructions, encoded as *matching rules*, form a knowledge-base to be managed by collaborators from two or more domains. No restrictions are imposed on the evolution of local terms within a domain. Terms that are covered by matching rules form a new, second layer abstract ontology. Hi&a abstract layers can be defined recursively, abandoning unneeded local terms

WC! ff ocus here on the third alternative.

## 5. An Example for Limited Domain Sharing

A multi-domain algebra needs the knowledge about the domains, specifically about the semantics of the intersecting terms.

We will illustrate the concept with a simple example:

S Domain S is of shoe stores, with objects il.4 shoes to be sold, customers, their feet, sales people, business locations, and suppliers

F Domain F is of shoe factories, with shoes being produced, lasts, materials as leather, glue, nails, and thread, suppliers for tllf! material, employees, and production machinery.

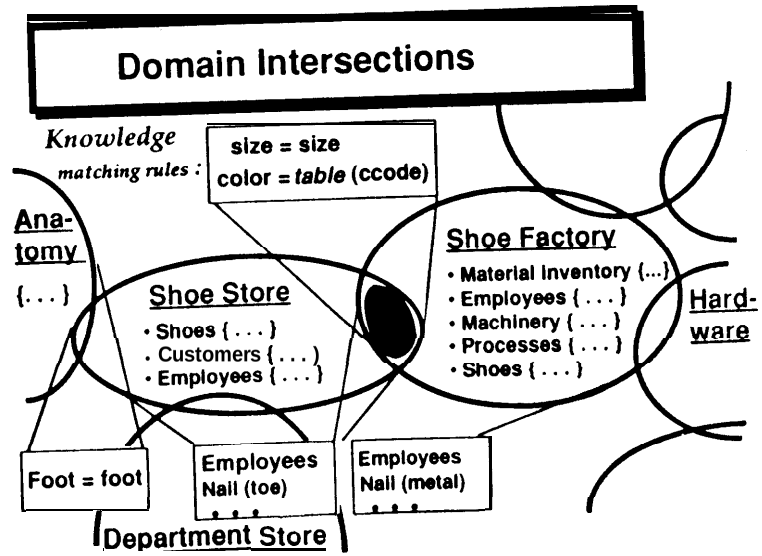
In order to create an information system that combines data from both, it is not necessary to merge both the S and F ontologies completely. Only terms along their connections must be merged, we assume by default that terms do not match. The required knowledge is:

```
S:supplier.name =  F:factory.name
S:shoe.size     =  F:shoe.size
S:shoe.color    =  color-table (F:shoe.color)
```

The color table provides the translation between the colors being attached to sales items, such as **pretty pink** and the color designation used in the factory, say, 13XF3. Sometimes such relationships can be expressed as functions, say, conversions from **cm** to **inches**

Not included in the knowledge-base, and hence not composable is the term **nail**, which in tllf! store domain S is part of the customer's anatomy, and in the factory F designates part of the material used to make shoes. Similarly, the **employees** remain distinct, since the data collected for sales people differ from those in the factory

The attached Figure illustrates the issues.




The income tax domain I will establish other connections between it and the sales and factory domains A department store, incorporating many sales subdomains, will have more semantic connections, but still avoid needing an unconstrained union of all its ontologies

Paper 27

FILE: /pub/yan/1994/dsdi.ps
COMMENT: Appeared in PDIS '94

# Distributed Selective Dissemination of Information

**Tak W. Yan**
**Hector Garcia-Molina**

Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

To help users cope with information overload, Selective *Dissemination of Information (SDI)* will increasingly become an important tool in wide area information systems. In an SDI service, users post their long term queries, called *profiles*, at some *SDI* servers and continuously receive new, filtered documents. To scale up with the volume of information and the size of user population, we need a distributed SDI service with multiple servers.

In this paper we first address the key problem of how to replicate and distribute profiles and documents among SDI servers. We draw the parallel between distributed SDI and the well-studied replica control problem, adapt quorum-based protocols for use in distributed SDI, and compare the performances of the different protocols. Next we address another important problem, that of efficient document delivery mechanisms. We present and evaluate a practical scheme, called *profile* grouping, which exploits the geographical locality of users to cut down network traffic generated by document delivery. Finally, we carry out a sensitivity analysis to determine the parameters that have critical impact on performance, and investigate strategies to cope with the scaling up of those parameters.

## 1 Introduction

The exploding volume and diversity of digital information pose challenging issues in large-scale wide-area information systems. In this dynamic environment it is difficult for the user, equipped with only conventional search capability, to keep up with the fast pace of information generation. Instead of making the user go after the information, information can also selectively flow to the interested users. Traditionally, libraries and databanks provide such kind of *information filtering* service, named *Selective Dissemination of Information (SDI)* [19]. A user expresses his interests in a number of long-term, continuously evaluated queries, called profiles. He or she will then passively receive documents filtered according to the profiles. Such SDI service will become increasingly important and form an indispensable tool for global information systems.

Netnews (see e.g., [10]), a popular information system today, can be regarded as providing some flavors of an SDI service. However, it is not very effective in this respect because it only provides a fixed number of broad categories. Thus a user interested in say "relational database systems" needs to receive and read all articles in the newsgroup `comp.databases`, as well as other related newsgroups. Relevant articles in newsgroups not explicitly subscribed to will be missed. Much recent research [5,6,14,16,22] has focused on providing more fine-grained and effective filtering, using relational, rule-based, information retrieval (IN.), and artificial intelligence approaches Yet little has been done on the efficiency aspect, which is also critical as SDI is going to be used on a large-scale. In [23,24] we studied the use of *profile indexes* to streamline filtering at a central site, assuming IR filtering techniques. However, to really scale up we need a distributed SDI service and the efficiency of such a service is the central topic of this paper.

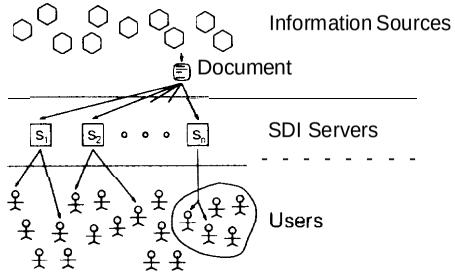



Figure 1: Distributed SDI

Distributed SDI is a problem of remote match-making. On the one hand, we have information providers that look for interested users. On the other hand, we have users who seek relevant information. What forms an efficient and reliable way to perform the matching and establish the flow from the providers to the users? One naive way to achieve this is to have the users post their profiles at each and every source that may generate useful information. The sheer number of sources renders this scheme inhibitly expensive. Further, it would be very difficult to locate all potentially relevant sources. The other extreme is to have the providers send information to each and every user, and have the users screen out irrelevant information Again, this is clearly not scalable. Valuable network bandwidth is wasted to transmit mostly irrelevant information and a lot of wasteful local processing is done. A feasible solution is to have some intermediate third-party *SDI servers* (Figure 1). Such servers, well-known to both providers and users, accept profiles from the users, collect documents and match them against user profiles, and relay relevant information to users.

In this paper we address the key problem of how to replicate and distribute profiles and docu-

ments among SDI servers. To illustrate, suppose a document is sent to every server, and a profile is posted at only one server. The number of profiles at each server is low, but the document arrival rate is high. Further, the availability of the service is low, since if a server goes down, the users it serves miss documents. The opposite way is to replicate a profile at all servers, and send a document to any one of them. This way, availability is high if a server goes down, documents can be rerouted to any other server. However, the number of profiles is high at each server, and the cost of updating a profile is high also.

There are intermediate solutions in between these two. If we denote the set of servers that a profile $x$ is posted at by $P_x$, and the set of servers that a document y is sent to by $D_y$, then to ensure that a profile does not miss a document, we must guarantee that $P_x$ intersects $D_y$ for every $x$ and y. This parallels the idea of *quorum* consensus in replicated data management. In this paper, we formalize the correspondence and, given the options for replication and distribution, we study the tradeoff between communication costs, document delivery times, reliability, and other parameters.

Once a document is matched to a set of profiles at an SDI server, there is the additional problem of sending the document to the users that posted the profiles. In general, there could be a very large number of users that need to receive the document. Without efficient document distribution mechanisms, much wide-area network (WAN) traffic would be generated. (The problem is related to message broadcast, e.g., [4].) To illustrate, say we have a collection of users at an institution that have posted profiles. A straightforward strategy for an SDI server is to send matching documents directly to users (as $S_1$ does in Figure 1). The disadvantage of this approach is that if a given document happens to be of interest to many users at the institution, many copies will be sent over the WAN. An improvement may be to have a *local distribution site* at the institution. Now, if a server (e.g., $S_n$ in Figure 1) discovers that there is a document that matches any profile from that institution, only one copy needs to be sent to the distribution site, which then distributes the document to the relevant user(s) locally. We call this idea of viewing a group of profiles as a single delivery unit *profile grouping*. Our assumption here is that the local distribution can be done without going through the WAN. This way, WAN congestion is reduced, possibly resulting in faster document delivery. On the other hand, it makes the delivery mechanism a bit more complex. In this paper we investigate whether profile grouping provides sufficient savings to be worthwhile.

Incidentally, we have implemented at Stanford two SDI servers, one for disseminating Netnews articles and the other for Computer Science Technical Reports. [1] Recently, we publicized the Netnews server in two newsgroups; within ten days of the announcement, we received well over a thousand profiles, submitted by users from almost every continent. The number of profiles keeps increasing ever since and has now exceeded two thousand. Clearly, a centralized server does not scale with the number of users (or profiles) and provides low availability. We intend to distribute the service, and this motivates the work reported in this paper. We also make use of real statistics collected from the running Netnews server to determine the values of parameters in the performance evaluation model in this paper.

## 2 Quorums for Distributed SDI

Replica control is a well-studied problem. Suppose a data item $x$ has a number of copies. If we execute a read (write) operation on $x$ by accessing copies that make up a read (write) quorum, then to guarantee one-copy equivalence, we must enforce the following [2]:

- **Write-write Intersection Property:** Two write quorums $W$ and $U$ must have a non-empty intersection: $W \cap U \neq 0$.
- **Read-write Intersection Property:** A write quorum $W$ and a read quorum $R$ must have a non-empty intersection: $W \cap R \neq 0$.

In distributed SDI, we have multiple servers available. A document may be sent to more than one servers, and a profile may be posted at more than one servers. If we call the set of servers that a document is sent to a *document quorum*, and the set of servers that a profile is posted at a *profile quorum*, then to guarantee that a profile does not miss a document, the following must be satisfied:

- **Profile-document** Intersection Property: A profile quorum $P$ and a document quorum $D$ must have a non-empty intersection: $P \cap D \neq 0$.

As we can see, the problem is analogous to replica control. In the rest of this section we briefly list the main quorum-based replica control protocols and show how to adapt them for SDI.

### 2.1 Majority Consensus Protocol

Majority consensus is proposed by Thomas [21]. In the distributed SDI scenario, suppose we have n SDI servers. A document is sent to a document quorum formed by $d$ (arbitrary) servers A profile

[1] The reader is encouraged to try out the services. For World-Wide Web access, please connect to http://woodstock.stanford.edu:2000. For email access, send an electronic message to netnews@db Stanford. edu or elib@db.stanford.edu, with the word "help" in the body.

FILE: /pub/yan/1994/openodb-tm.ps
COMMENT: Appeared in VLDB '94 Proceedings

Paper 28

# Integrating a Structured-Text Retrieval System with an Object-Oriented Database System

**Tak W. Yan**
Department of Computer Science
Stanford University
Stanford, CA 94305

**Jurgen Annevelink**
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

**Abstract**

We describe the integration of a structured-text retrieval system (TextMachine) into an object-oriented database system (OpenODB). 0 ur approach is a light-weight one, using the external function capability of the database system to encapsulate the text retrieval system as an external information source. Yet, we are able to provide a tight integration in the query language and processing; the user can access the text retrieval system using a standard database query Language. The efficient and effective retrieval of structured text performed by the text retrieval system is seamlessly combined with the rich modeling and general-purpose querying capabilities of the database system, resulting in an integrated system with querying power beyond those of the underlying systems. The integrated system also provides uniform access to textual data in the text retrieval system and structured data in the database system, thereby achieving information fusion. We discuss the design and implementation of our prototype system, and address issues such as the proper framework for external integration, the modeling of complex categorisation and structure hierarchies of documents [under automatic document schema importation), and techniques to reduce the performance overhead of accessing an external source.

## 1 Introduction

With the advent of document structure markup standards such as SGML [8] and ODA [3], structured documents have recently received a lot of attention. The logical structure of text constitutes important information for querying and/or retrieval, but traditional text retrieval systems typically do not make full use of the document structure. New research proposals and prototypes, such as [4,11,14], and commercial text retrieval systems, such as [1], have emerged to address this problem.

Structured documents also lend themselves to be managed by database systems. A lot of efforts have been made to add text retrieval capability to both relational and object-oriented systems. Past efforts to model structured text in a relational system have met with difficulties [15], and object-oriented database systems have been developed partly with the aim to store and manage text and other complex data. In either case, the database system stores the documents and is extended with text indexing and retrieval capabilities.

In this paper, we present a more light-weight, flexible approach of integrating a text retrieval system (TextMachine [1]) and an object-oriented database system (OpenODB [9]). The text retrieval system retrieves structured documents. It exists alongside with the database system. Documents are stored in, and all indexing and retrieval of documents are performed by, the text retrieval system. The object-oriented database system treats the text retrieval system as an external information source. It communicates with the text retrieval system through the latter's application programming interface, submitting queries and receiving answers. The interface is encapsulated by the so-called external function capability of the object-oriented system, and the underlying processing is transparent to the user; i.e., the user is unaware that the text retrieval system is an external source.

Such integration seamlessly combines the best of both worlds – the efficient and effective indexing and retrieval of structured documents performed by the text retrieval system, and the rich modeling and general-purpose querying capabilities supported by the database system – resulting in an integrated system with querying power beyond those of the underlying systems. The integrated system also provides uniform access, i.e., the ability to uniformly query over both the textual data managed by the text retrieval system and the structured data stored in the database system, thereby achieving information fusion.

To illustrate our claims above, let us consider some sample queries that can be processed by the integrated system. We describe the queries in English here, before we introduce the query languages and models of the two systems. (The actual queries will be presented later in the paper.) Suppose we have a collection of medical progress notes residing in the text retrieval system. Also suppose we have some structured information, including names and specialties, about some physicians stored in the database system. Now consider a user who is interested in retrieving paragraphs in the progress notes that mention "bronchitis" and also the name of a physician whose specialty is "respiratory." The user submits a query, in the database query language, to the integrated system (the actual query is (Q3) in Section 3.3.3). When the query is processed, the names of the physicians with specialty "respiratory" are selected from the database and each name becomes part of a query to the text retrieval system. A number of queries (equal to the number of names selected) are then submitted to the text retrieval system and the combined results are returned to the user This way, the user transparently makes use of structured data to query the text retrieval system Furthermore, the text data retrieved can then be extracted and assimilated into the structured database.

In the next example, we demonstrate that the retrieval power of the integrated system is beyond that provided by the underlying systems. The retrieval model of TextMachine is similar to the tree inclusion primitive, presented recently in [11]. The essential idea of the primitive is to represent documents and queries as trees; a query specifies a partial tree pattern that the retrieved document trees must contain. (We will further discuss this in Section 2.2.2.) In the integrated system, we are able to retrieve text data using queries not expressible with the tree inclusion primitive Query (Q4) in Section 3.3.4 is such an example. The user submitting this query uses two partial patterns to select paragraphs from the text retrieval system He further adds the constraint that a paragraph retrieved by the first pattern must be adjacent to a paragraph retrieved by the second pattern, using

database query language constructs. We cannot express this adjacency constraint in TextMachine or using the tree inclusion primitive alone.

In this paper, we describe the design and implementation of our prototype integrated system. We address the following issues:

- Framework for external integration

  We investigate the use of the database system's external function capability to access the retrieval system. We show what abstraction of the external source is desirable for integration and how document schemas from the text retrieval system are imported. We show how, in spite of the nature of the external integration, we achieve a tight coupling of query language through query translation.

- Modeling text

  We discuss the use of object-oriented concepts – objects, types, and functions – to model complex categorization and structure hierarchies of documents. We address the modeling issues involved when we import document schema from the text retrieval system into the integrated system.

- Performance issues

  A potential drawback of this approach is the performance penalty of accessing an external source. However, we can minimize the overhead in a number of ways, such as efficient techniques to perform conversion between keys from external sources and database object identifiers (OIDs), optimizing the processing of external functions, and so on. In our prototype, we investigate the use of algorithmic OID conversion.

Although the work reported in this paper is on integrating with a particular structured-text retrieval system, the approach and techniques are orthogonal to the underlying retrieval model and language, and can be extended to other text retrieval systems as well. In fact, we have integrated OpenODB with another system, Wide Area Information Servers (WAIS) [10], which is a networked information system This demonstrates the generality of our techniques.

The rest of the paper is organized as follows. In Section 2 we describe the testbed systems of our integration. In Section 3 we present the integrated system. The implementation details of the prototype are covered in Section 4. We survey related work in Section 5. Finally, Section 6 is for conclusion and discussion of future work.

# 2 The Testbed

In this section, we briefly describe the object-oriented database system and the text retrieval system that we have integrated. The scope of our paper limits this to a short description of relevant topics. The reader is referred to [1] and [9] for detailed coverage of the systems.



## 2.1 OpenODB: a Functional Object-Oriented Database

OpenODB is an object-oriented database system developed by Hewlett-Packard. OSQL [2] is a database programming language for OpenODB. It combines the object-oriented features found in such languages as C++ and Smalltalk with a query capability that is a superset of the familiar SQL relational query language.

The OSQL language is centered around three basic concepts: objects, types, and functions Objects represent the real-world entities and concepts from the application domain that the database is storing information about. Each object has a unique object identifier (OID). *Types* are used to classify objects on the basis of shared properties and/or behavior. Types are also used to define the signature of functions, i.e., their argument and result types. Types are related into a subtype/supertype hierarchy that supports multiple inheritance. The type hierarchy enforces type containment, i.e., if an object is an instance of a given type $T$, it must also be an instance of all supertypes of the type *T*. OSQL surrogate objects can be instances of any number of types, even if the types are not related by a subtype/supertype relationship.

*Functions* are used to model attributes of the object, interobject relationships, and arbitrary computations. One of the key distinctions of OSQL as compared to other models, e.g, those inspired by various programming languages, is this unifying notion of a function to model stored and derived attributes, stored and derived relationships and arbitrary computations (behavior). An OSQL function takes one or more objects as arguments and may return an object as a result. OSQL functions can be overloaded, i.e., there can be multiple functions with the same name but different argument types. Functions have extensions. The extension of a function is the mapping from its argument(s) to its results. Function extensions can be explicitly stored, or they can be computed. Functions whose extension is computed can be implemented either as an OSQL expression, or as a program (subroutine, procedure) written in a general purpose programming language (e.g., C). These latter are called external functions and give OSQL a unique form of extensibility by allowing the encapsulation of (entry points in) external libraries, to access information, data, and functionality outside of an OpenODB database. We made extensive use of external functions in our integration.

OSQL supports a query language whose semantics is based on domain calculus. The OSQL select function provides the basic query facilities of OSQL and closely resembles the select statement of SQL. For example, suppose we have a type Physician with an attribute specialty in our database. Then the query

```
select p for each Physician p where specialty(p) = 'respiratory';
```

returns the OIDs of the physicians whose specialty is "respiratory."

# SIFT – A Tool for Wide-Area Information Dissemination *

Tak W. Yan Hector Garcia-Molina
*Department of Computer Science*
*Stanford University*
*Stanford, CA 94305*
{tyan, hector}@cs.stanford.edu

## Abstract

The dissemination model is becoming increasingly important in wide-area information system. In this model, the user subscribes to an information dissemination service by submitting profiles that describe his interests. He then passively receives new, tered information. The Stanford Information Filtering Tool (SIFT) is a tool to help provide such service. It supports full-text filtering using well known information retrieval models. The SIFT filtering engine implements novel indexing techniques, capable of processing large volumes of information against a large number of profiles. It runs on several major Unix platforms and is freely available to the public. In this paper we present SIFT's approach to user interest modeling and user-server communication. We demonstrate the processing capability of SIFT by describing a running server that disseminates USENET News. We present an empirical study of SIFT's performance, examining its main memory requirement and ability to scale with information volume and user population.

## 1 Introduction

Technological advances have made wide-area information sharing commonplace. A suite of tools have emerged for network information finding and discovery; e.g., Wide-Area Information Servers (WAIS) [KM91], archie [ED92], World-Wide Web (WWW) [BLCGP92], and gopher [McC92] However, these new tools have one important missing element They provide a means to search for existing information, but lack a mechanism for continuously informing the user of new information. The exploding volume of digital information makes it difficult for the user, equipped with only search capability, to keep up with the fast pace of information generation. Instead of making the user go after the information, it is desirable to have information selectively flow to the user. In an *information dissemination* (aka. *alert, information filtering, selective dissemination of information*) service, the user expresses his interests in a number of long-term, continuously evaluated queries, called *profiles*. He will then passively receive documents filtered according to the profiles. Such a service will become increasingly important and form an indispensable tool for the dynamic environment of wide-area information systems.

A very simple kind of information dissemination service is already available on the Internet: mailing lists (see e.g., [Kro92]). Hundreds of mailing lists exist, covering a wide variety of topics. The user subscribes to lists of interest to him and receives messages on the topic via email. He may also send messages to the lists to reach other subscribers. LISTSERV is a software system for maintaining mailing lists. A problem with the mailing list mechanism as a tool for information dissemination is that it provides a crude granularity of interest matching A user whose information need does not exactly match certain lists will either receive too many irrelevant or too few relevant messages. The USENET News (or Netnews) system [Kro92], an electronic bulletin board system on the Internet, is similar in nature to mailing lists. While Netnews is extremely successful with millions of users and megabytes of daily traffic, at the same time it often creates an information overload. Like mailing lists, the coarse classification of topics into newsgroups means that a user subscribing to certain newsgroups may not find all articles interesting, and also he will miss relevant articles posted in newsgroups that he does not subscribe to.

Recent research efforts on information filtering focus on the filtering *effectiveness*, attempting to provide more fine-grained filtering using relational, rule based, information retrieval (IR), and artificial intelligence approaches. With few exceptions, they are often small scale (i.e., involving a small number of users or profiles) and thus the need to provide *efficient* filtering is not apparent However, in a large scale wide area system where the number of information providers and seekers are large, efficiency in the dissemination process is an important issue and must be addressed.

The Stanford Information Filtering Tool (SIFT) is a tool for information providers to perform large-scale information dissemination It can be used to set up a clearinghouse service that gathers large amount of information and selectively disseminates the information to a large population of users. It supports full-text filtering, using well known and well-studied IR models. The SIFT filtering engine implements novel indexing techniques, capable of scaling to large number of documents and profiles. It runs on several major Unix platforms and is freely available to the public by anonymous ftp at URL:

```
ftp://db.stanford.edu/pub/sift/sift-1.0.tar.Z
```

In this paper we describe SIFT We present its approach to user interest modeling and user-server communication We demonstrate the processing capability of SIFT by describing a running server that disseminates tens of thousands of Netnews articles daily to some 13,000 subscriptions We describe the implementation of SIFT, focusing on the filtering engine Finally we present an empirical study of SIFT's performance, examining its main memory requirement and ability to scale with information volume and user population

## 2 Other Previous Work

Boston Community Information System [GBBL85] is an experimental information dissemination system. Like SIFT, it allows a finer granularity of interest matching than mailing lists of Netnews. Users can express their interests with IR-style, keyword based profiles. The system broadcasts new information via radio channel to all users, who then apply their own filters locally. While the radio communication channel makes broadcast inexpensive, local processing of mostly irrelevant information is very expensive. (For every user, a personal computer is dedicated for this purpose – this is cited as a major source of complaints from the users.)

Information Lens [MGT+87] provides categorization and filtering of semi-structured messages such as email The user defines rules for filtering, and the processing is done at the user site It provides effective filtering, but local processing is expensive for large-scale information dissemination. Similarly, the "kill file" mechanism in certain news reader programs allow the user to locally screen out irrelevant articles. A kill file only removes specified articles from newsgroups that a user subscribes to, but it does not discover relevant articles in other newsgroups. To provide the same kind of filtering power as SIFT would require much local processing. It is more cost-effective to pool profiles together to share the processing overhead.

The Tapestry system [GNOT92] is a research prototype that uses the relational model for matching user interests and documents, filtering computation is done not on the properties of individual documents, but rather on the entire append only document database. Efficient query processing techniques are proposed for handling this kind of queries Tapestry is built on top of a commercial relational database system. The Pasadena system [WF91] investigates the effectiveness of different IR techniques in filtering. It collects new documents from several Internet information sources, and periodically run profiles against them Similar to SIFT, it uses a clearinghouse approach, but efficient filtering is not addressed.

In [YGM94b, YGM94c] we proposed a variety of indexing techniques for speeding up information filtering under IR models. There we evaluated these techniques using analysis and simulation. The SIFT filtering engine is a real implementation of one class of index structures that we found to be efficient

*This research was sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of ARPA, the U.S. Government, or CNRI.

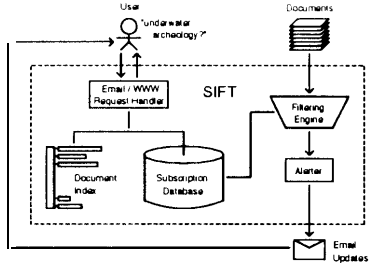



Figure 1: An overview of SIFT

# 3 SIFT

We begin our presentation of SIFT with an example. Suppose a user is interested in underwater archeology (Figure 1). He sends an email subscribe request to a SIFT server specifying the profile "underwater archeology," and optionally parameters that control, for example, how often he wants to be updated or how long his subscription is for. He may alternatively access the SIFT server via WWW, using a graphical WWW client interface to fill out a form with the subscription information. (Before he subscribes, he may test run his profile against an index of a sample document collection.) His subscription is stored in the subscription database. As the SIFT server receives new documents, the filtering engine will process them against the stored subscriptions, and notifications will be sent out based on the user-specified parameters.

In the following we detail the SIFT system from the perspectives of user interest modeling and communication protocol. We then illustrate SIFT using the Netnews SIFT server.

## 3.1 User Interest Modeling

A user subscribes to a SIFT server with one or more subscriptions, one for each topic of interest. A subscription includes an IR-style profile, and as mentioned additional parameters to control the frequency of updates, the amount of information to receive, and the length of the subscription. A subscription is identified by the email address of the user and a subscription identifier.

### 3.1.1 Filtering Model

The Interest profile can be expressed in one of two IR models: *boolean* and *vector space* [Sal89]. We first focus on the vector space model.

In vector space model, queries and documents are identified by terms, usually words. If there are $m$ terms for content identification, then a document $D$ is conceptually represented as an $m$-dimensional vector $D = \langle w_1, w_2, \ldots, w_m \rangle$, where weight $w_i$ for term $t_i$ signifies its statistical importance, such as its frequency in the document. We may also write $D$ as $\langle (t_{i_1}, w_{i_1}), \ldots, (t_{i_k}, w_{i_k}) \rangle$ where $w_i \neq 0$; e.g., ( (underwater, 60), (archeology, 60) ). A query is similarly represented. For a document-query pair, a similarity measure (such as the dot product) can be computed to determine how "similar" the two are. In an IR environment, the top ranked documents are retrieved for a query.

In an information filtering setting:, a key question is how many documents to return to the user. We could have allowed the user to receive a fixed number of top ranked documents per update period, e.g., as done in [FD92]. However, this is not very desirable: in a period when there are many relevant documents, he may miss some (low *recall*[1]); in a period when there are few interesting documents, he may receive irrelevant ones (low *precision*[1]). We instead allow the user to specify a *relevance* threshold, which is the minimum similarity score that a document must have against the profile for it to be delivered. A default value is supplied to the user for convenience.

Instead of using the vector space model, the user may use boolean profiles to specify words that he wants in documents received, and words to be excluded. For example, the boolean profile "fly fishing not underwater" is for documents that contain both words "fly" and "fishing" but not the word "underwater. The reader may note that the SIFT boolean model only allows conjunction and negation of words. However, the user may approximate disjunction semantics by submitting multiple subscriptions (though a document may match more than one subscriptions).

### 3.1.2 Profile Construction and Modification

To assist the user with the construction of a profile, a SIFT server provides a test run facility. The user may run his initial profile against an existing, representative collection of documents to test the effectiveness of his profile. He may interactively change the profile and (for weighted profiles) adjust the threshold to the desired level of precision and recall. When he is satisfied with the performance of the filtering, he may then subscribe with the selected settings.

After the user receives some periodic updates, he may decide to modify his profile or change the threshold for vector space profiles. He may do this by accessing the SIFT server. Furthermore, for vector space profiles, *relevance feedback* [Sal89], a well-known technique in IR to improve retrieval effectiveness, can be used. The user simply gives SIFT the documents that he finds interesting; after examining them, the server adjusts the weights of the words in the user's profile accordingly.

[1] Recall is the proportion of relevant documents returned, and precision is the proportion of documents returned that are relevant.

## 3.2 Communication Protocol

There are two modes of communication between the user and a SIFT server. In the interactive mode, the user subscribes, test-runs a profile, views, updates, or cancels his subscriptions. In the passive mode, the user periodically receives information updates. Instead of developing a new communication protocol, we make use of current technologies, email [Cro82] and World-Wide Web HTTP [BLCGP92].

First we discuss the interactive communication mode. Email communication is the lowest common denominator of network connectivity. By having an email interface, a SIFT server is accessible from users with less powerful machines, with limited network capability, or behind Internet-access firewalls. We adopt the LISTSERV mailing list syntax wherever possible, to ensure that minimal learning is required. We also use default settings to reduce the complexity of email requests for novice users.

As the appeal of hypermedia navigation and the development of sophisticated client interfaces are making WWW the preferred tool for wide area information sharing, we also developed a WWW access interface for SIFT. Using a WWW client program, the user interacts with the SIFT server through a user-friendly graphical interface. We believe the dual email and WWW access covers the tradeoff between wide availability and ease of use.

In the passive mode of user notification, a SIFT server sends out email messages that contain excerpts of new, potentially relevant documents (certain number of lines from the beginning, as specified by the user). After the user reads the excerpts, he may access the SIFT server to retrieve the entire documents. Currently the excerpts are formatted to be read from regular mail readers. We plan to offer the option of formatting them in HTML, so that the user may view the notifications from sophisticated WWW viewers, and then interactively retrieve interesting documents from SIFT or provide feedback via HTTP.

## 3.3 SIFTing Netnews

Using SIFT, we have set up a server for selectively disseminating Netnews articles (text articles only, binary ones are first screened out). Like any other SIFT server, the user accesses the Netnews SIFT server via email or WWW. The reader is encouraged to try it out, for email access, please send an electronic message to **netnews@db.stanford.edu**, with the word "help" in the body; for WWW access, please connect to **http://sift.stanford.edu**. In February 1994, we publicized the Netnews server in two newsgroups; within ten days of the announcement, we received well over a thousand profiles. The number of profiles keeps increasing and now (November 1994) exceeds 13,000, submitted by users from almost all continents. Table 1 shows some interesting statistics obtained from the server on the day of November 10, 1994. The average number of articles is over the week of November 4–10, 1994.

| | |
|---|---|
| Number of subscriptions | 13,381 |
| Number of users | 5,146 |
| Daily average number of articles | 45,127 |
| Average notification period (in days) | 1.1 |

Table 1: Netnews SIFT server statistics

Apparent from these numbers is that the load on the SIFT server is very high. It is necessary to match an average of over 45,000 articles against some 13,000 profiles and deliver most updates within a day. The efficient implementation of the SIFT filtering engine enables the job to be done on regular hardware, a DEC'station 5000/240. Even though we believe SIFT is efficient, there is a limit to the load that it can handle. It is necessary to replicate the server: in fact, we have been in contact with several sites (in the US and Europe) that expressed interests in providing the same service.

The Netnews SIFT server is not meant to be a replacement of the Netnews system, which is an ill

| | |
|---|---|
| FILE: | /pub/zhuge/1994/anomaly-short.ps |
| FILE: | /pub/zhuge/1994/anomaly-full.ps |
| COMMENT: | Short version to appear in SIGMOD '95; proofs and additional technical details can be found in the full version |

Paper 30

# View Maintenance in a Warehousing Environment*

Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, Jennifer Widom

*Computer Science Department*
*Stanford* University
*Stanford, CA 94305-2140, USA*
{zhuge,hector,joachim,widom}@cs.stanford.edu

Abstract

A warehouse is a repository of integrated information drawn from remote data sources. Since a warehouse effectively implements material&d views, we must maintain the views as the data sources are updated. This view maintenance problem differs from the traditional one in that the view definition and the base data are now decoupled. We show that this decoupling can result in anomalies if traditional algorithms are applied. We introduce a new algorithm, ECA (for "Eager Compensating Algorithm"), that eliminates the anomalies. ECA is based on previous incremental view maintenance algorithms, but extra "compensating" queries are used to eliminate anomalies. We also introduce two streamlined versions of ECA for special cases of views and updates, and we present au initial performance study that compares ECA to a view recomputation algorithm in terms of messages transmitted, data transferred, and I/O costs.

## 1 Introduction

Warehousing is an emerging technique for retrieval and integration of data from distributed, autonomous, possibly heterogeneous, information sources. A data *warehouse* is a repository of integrated information, available for queries and analysis (e.g., decision support, or data mining) [IK93]. As relevant information becomes available from a source, or when relevant information is modified, the information is extracted from the source, translated into a common model (e.g., the relational model), and integrated with existing data at the warehouse. Queries can be answered and analysis can be performed quickly and efficiently since the integrated information is directly available at the warehouse, with differences already resolved.

### 1.1 The Problem

One can think of a data warehouse as defining and storing integrated materialized views over the data from multiple, autonomous information sources. An important issue is the prompt and correct propagation of updates at the sources to the views at the warehouse. Numerous methods have been developed for materialized view maintenance in conventional database systems; these methods are discussed in Section 2.

This work was supported by ARPA Contract F33615-93-1-1339, by the Anderson Faculty Scholar Fund, by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation. The US Government is authorieed to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the US Government.



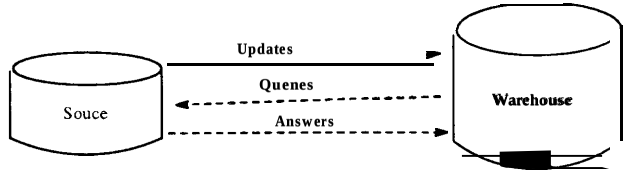



**Figure 1.1: Processing of updates in a single source model**

Unfortunately, existing materialized view maintenance algorithms fail in a warehousing environment. Existing approaches assume that each source understands view management and knows the relevant view definitions. Thus, when an update occurs at a source, the source knows exactly what data is needed for updating the view.

However, in a warehousing environment, the sources can be legacy or unsophisticated systems that do not understand views. Sources can inform the warehouse when an update occurs, e.g., a new employee has been hired, or a patient has paid her bill. However, they cannot determine what additional data may or may not be necessary for incorporating the update into the warehouse views. When the simple update information arrives at the warehouse, we may discover that some additional source data is necessary to update the views. Thus, the warehouse may have to issue queries to some of the sources, as illustrated in Figure 1.1. The queries are evaluated at the sources later than the corresponding updates, so the source states may have changed. This decoupling between the base data on the one hand (at the sources), and the view definition and view maintenance machinery on the other (at the warehouse), can lead the warehouse to compute incorrect views.

We illustrate the problems with three examples For these examples, and for the rest of this paper, we use the relational model for data and relational algebra for views. Although we are using relational algebra, we assume that duplicates are retained in the materialized views. Duplicate retention (or at least a replication count) is essential if deletions are to be handled incrementally [BLT86, GMS93]. Note that the type of solution we propose here can be extended to other data models and view specification languages.

Also, in the examples and in this paper we focus on a single source, and a single view over several base relations. Our methods extend to multiple views directly. Handling a view that spans several sources requires the same type of solution, but introduces additional issues; see Section 7 for a brief discussion.

### Example 1: Correct View Maintenance

Suppose our source contains two base relations $r_1$ and $r_2$ as follows:

$$r_1: \begin{array}{|c|c|}\hline W & X \\ \hline 1 & 2 \\ \hline \end{array} \qquad r_2: \begin{array}{|c|c|}\hline X & Y \\ \hline 2 & 4 \\ \hline \end{array}$$

Suppose the view at the warehouse is defined by the relational algebra expression:

$$v = \Pi_W(r_1 \bowtie r_2)$$

Initially the materialized view at the warehouse, MV, contains the single tuple [1]. Now suppose tuple [2,3] is inserted intb $r_2$ at the source. For notation, we use $insert(r, t)$ to denote the

insertion of tuple $t$ into relation $r$ (similarly for $delete(r,t)$), and we use $([t_1],[t_2],\ldots,[t_n])$ to denote a relation with tuples $t_1$, $t_2$, , t,. The following events occur:

1. Update $U_1 = insert(r_2, [2,3])$ occurs at the source. Since the source does not know the details or contents of the view managed by the warehouse, it simply sends a notification to the warehouse that update $U_1$ occurred.
2. The warehouse receives $U_1$ Applying an incremental view maintenance algorithm, it sends query $Q_1 = \Pi_W(r_1 \bowtie [2,3])$ to the source. (That is, the warehouse asks the source which $r_1$ tuples match with the new [2,3] tuple in $r_2$.)
3. The source receives $Q_1$ and evaluates it using the current base relations. It returns the answer relation $A_1 = ([1])$ to the warehouse.
4. The warehouse receives answer $A_1$ and adds ([1]) to the materializedview, obtaining $([1],[1])$.

In this example the final view at the warehouse is correct, i.e., it is equivalent to what one would obtain using a conventional view maintenance algorithm directly at the source.' The next two examples show how the final view can be incorrect.

### Example 2: A View Maintenance Anomaly

Assume we have the same relations as in Example 1, but initially $r_2$ is empty:

$r_1$ :

| W | X |
|---|---|
| 1 | 2 |

$r_2$ :

| X | Y |
|---|---|

Consider the same view definition as in Example 1: $V = \Pi_W(r_1 \bowtie r_2)$, and now suppose there are two consecutive updates: $U_1 = insert(r_2, [2,3])$ and $U_2 = insert(r_1, [4,2])$. The following events occur. Note that initially $MV = \emptyset$.

1. The source executes $U_1 = insert(r_2, [2,3])$ and sends $U_1$ to the warehouse.
2. The warehouse receives $U_1$ and sends $Q_1 = \Pi_W(r_1 \bowtie [2,3])$ to the source.
3. The source executes $U_2 = insert(r_1, [4,2])$ and sends $U_2$ to the warehouse.
4. The warehouse receives $U_2$ and sends $Q_2 = \Pi_W([4,2] \bowtie r_2)$ to the source.
5. The source receives $Q_1$ and evaluates it on the current base relations: $r_1 = ([1,2],[4,2])$ and $r_2 = ([2,3])$. The resulting answer relation is $A_1 = ([1],[4])$, which is sent to the warehouse.
6. The warehouse receives $A_1$ and updates the view to MVU $A_1 = ([1],[4])$.
7. The source receives $Q_2$ and evaluates it on the current base relations $r_1$ and $r_2$. The resulting answer relation is $A_2 = ([4])$, which is sent to the warehouse.
8. The warehouse receives answer $A_2$ and updates the view to $MV \cup A_2 = ([1],[4],[4])$.

[1] As stated earlier, for incremental handling of deletions we need to keep both [1] tuples in the view. For instance, if [2,4] is later deleted from $r_2$, one (but not both) of the [1] tuples should be deleted from the view.

If the view had been maintained using a conventional algorithm directly at the source, then it would be ([1]) after $U_1$ and ([1],[4]) after $U_2$. However, the warehouse fist changes the view to ([1],[4]) (in step 6) and to ([1],[4],[4]) (in step 8), which is an incorrect final state. The problem is that the query $Q_1$ issued in step 4 is evaluated using a state of the base relations that differs from the state at the time of the update ($U_1$) that caused $Q_1$ to be issued. □

We call the behavior exhibited by this example a *distributed incremental view* maintenance *anomaly*, or anomaly for short. Anomalies occur when the warehouse attempts to update a view while base data at the source is changing. Anomalies arise in a warehousing environment because of the decoupling between the information sources, which are updating the base data, and the warehouse, which is performing the view update.

### Example 3: Deletion Anomaly

Our third example shows that deletions can also cause anomalies. Consider source relations.

$r_1$ :

| W | X |
|---|---|
| 1 | 2 |

$r_2$ :

| X | Y |
|---|---|
| 2 | 3 |

Suppose that the view definition is $V = \Pi_{W,Y}(r_1 \bowtie r_2)$. The following events occur. Note that initially $MV = ([1,3])$.

1. The source executes $U_1 = delete(r_1, [1,2])$ and notifies the warehouse
2. The warehouse receives $U_1$ and emits $Q_1 = \Pi_{W,Y}([1,2] \bowtie r_2)$.
3. The source executes $U_2 = delete(r_2, [2,3])$ and notifies the warehouse
4. The warehouse receives $U_2$ and emits $Q_2 = \Pi_{W,Y}(r_1 \bowtie [2,3])$.
5. The source receives $Q_1$. The answer it returns is $A_1 = \emptyset$ since both relations are now empty.
6. The warehouse receives $A_1$ and replaces the view by $MV - A_1 = ([1,3])$. (Difference is used here since the update to the base relation was a deletion [BLT86].)
7. Similarly, the source evaluates $Q_2$, returns $A_2 = \emptyset$, and the warehouse replaces the view by $MV - A_2 = ([1,3])$.

This final view is incorrect: since $r_1$ and $r_2$ are empty, the view should be empty too. □

## 1.2 Possible Solutions

There are a number of mechanisms for avoiding anomalies. As argued above, we are interested only in mechanisms where the source, which may be a legacy or unsophisticated system, does not perform any "view management." The source will only notify the warehouse of relevant updates, and answer queries asked by the warehouse We also are not interested in, for example, solutions where the source must lock data while the warehouse updates its views, or in solutions where the source must maintain timestamps for its data in order to detect "stale" queries from the warehouse. In the following potential solutions, view maintenance is autonomous from source updating:

INDEX