Scalable and Efficient Active Service Integration

Paul Lewis Benninghoff

A dissertation presented to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science

June 2003

The dissertation "Scalable and Efficient Active Service Integration" by Paul Lewis Benninghoff has been examined and approved by the following Examination Committtee:

Dr. David Maier
Professor
Thesis Research Advisor

Dr. Phil Cohen
Professor

Dr. Lois Delcambre
Professor

Dr. Len Shapiro
Professor
Portland State University

Acknowledgements

This dissertation would not exist without the help and encouragement of many people. Dave Maier, my thesis advisor, deserves special mention for guiding me through this process with his characteristic combination of perception, patience and humor, and for offering to meet with me regularly at his home to push me through the final bits of writing and revising, often over an obscure microbrew. The rest of my committee deserves special mention as well. Phil Cohen provided financial support as well as performing committee duties. Lois Delcambre provided moral support and social diversion at many stages along the way, and many helpful comments on the content of the dissertation. Len Shapiro provided the Columbia optimizer as the basis for many of my experiments as well as many enlightening conversations on its workings and on optimization in general. Len was also gracious enough to accept the role of external committee member at the 11^{th} hour.

Numerous members of the OGI community also contributed meaningfully to this process. Ling Liu provided much energy, enthusiasm and encouragement in serving as a substitute advisor, of sorts, while Dave Maier was on sabbatical (and beyond). Kavita Hatwal modified the Columbia optimizer to perform "multiplex optimization" and helped me with a number of my experiments. Jo Ann Binkerd provided an outlet for my need to talk sports every once in a while, not to mention a seat at a Blazer game on more than one occasion. Others too numerous to mention provided ideas, comments, competition on the athletic fields, humor, friendship and otherwise made my experience at OGI richer and more enjoyable.

Last, but far from least, my wife, Kathy, and my two daughters, Grace and Emma, endured a great deal in allowing me to pursue this degree. I am grateful for their patience and support.

Financial support for this work was provided by the National Science Foundation under grants NSF IRI-9509955, NSF IRI-9619977, and NSF IIS 0086002NSF, and by the Office of Naval Research, under grant ONR N00014-95-1-1164. I was also the recipient

of a Wilson Clark fellowship, which supported me in my first year of graduate work. The support of all of these funding sources is gratefully acknowledged.

Contents

List of Tables

List of Figures

Abstract

Scalable and Efficient Active Service Integration

by Paul Benninghoff

We investigate issues in the construction of an *Active Service Integration System* (ASIS). An ASIS is a mediation system that provides event-based monitoring and integration over data-intensive, networked services. We describe the design and implementation of the Paradox Active Service Integration System, which we have built as part of our dissertation work. Paradox addresses many of the fundamental issues of ASIS construction. Paradox extends database technology and previous work in data integration to handle event-based processing over autonomous, heterogeneous network services in an efficient manner. We suggest capabilities and metadata that services may provide to aid in the active integration process.

Efficiency and scalability problems remain in Paradox that must be addressed in a practical system. We describe and evaluate methods for addressing several such problems that are unified by two central themes: inter-task sharing, and the specification and exploitation of increasingly rich service characteristics. Data caching is essential to a scalable and efficient ASIS. We describe how the long-lived nature of ASIS requests can be leveraged to make effective caching decisions. We present a detailed model and framework for effective, cost-based selection of a view cache in an ASIS. ASIS cache selection involves multiple complex tasks: the selection of the view cache, the optimization sub-problems that involve multiple, simultaneously-executing queries (MQO subproblems), and the generation of efficient plans that incorporate and maintain the chosen view cache. The resultant optimization problem is doubly-exponential in complexity. We describe a multi-pronged approach to handling this problem in a tractable manner.

We present a description and implementation of Multiplex Query Optimization (MuxQO), a novel method for efficiently handling problems that can be cast as a group of overlapping query optimization problems. We characterize the applicability of MuxQO, and we describe a performance evaluation that demonstrates the effectiveness of MuxQO in handling ASIS view selection. MuxQO handles cache selection, MQO subproblems, and optimal plan generation in an integrated fashion. We describe how a top-down optimizer can be modified to support MuxQO. MuxQO is applicable to a range of problems, including physical database design, multiple query optimization, evaluation of recursive queries, and the physical representation of new data formats and models such as XML.

Finally, we describe and evaluate a novel approach to exploiting rich application semantics to improve the efficiency and scalability of an ASIS. In particular, we describe a method for exploiting constraints on information change over time. Our approach can greatly improve the scalability of an ASIS with respect to the frequency of change events at component sources. We argue that application-level semantics are a rich vein to mine in improving the scalability and efficiency of active service integration.

Chapter 1

Introduction

This dissertation investigates issues in the construction of an *Active Service Integration System* (ASIS). An ASIS is a mediation system that provides uniform access and event-based monitoring over compositions of network-accessible, data-intensive services. Several trends in society, business and technology are converging to make the ASIS an interesting and important type of mediation system. But the challenges in building such a system are many and not well understood.

We describe the design and implementation of a basic ASIS that addresses many of the fundamental concerns of construction. We then describe efficiency and scalability problems that must be addressed to make such a system practical. We describe and evaluate methods for addressing these problems that are unified by two central themes: improved scalability and efficiency through inter-task sharing, and the specification and

exploitation of increasingly rich service characteristics including computational capabilities, data statistics and application semantics.

1.1 A New Kind of Mediator

The computing world is undergoing a transition to a "second stage" in the widespread adoption of Internet technology. The first stage was about a web of interconnected "pages" designed for human consumption. In contrast, the second stage is about a web of interconnected information sources and computational functionality presented in the form of "services" designed for machine consumption. Models of service invocation and coordination have differed in competing versions of this vision, with some subscribing to a document exchange model (or DEM), while others subscribe to a network object model (or NOM). But the essence of the vision is consistent: If you are connected to the Internet, you have access to myriad computational services that may be invoked and manipulated programmatically.

We are in the early developmental stages of this transition. As Dave Winer writes in his "Davenet" web letter, "We're at the 'Hello World' stage again" (Winer 2002). But real web services are beginning to emerge, and the momentum behind the transition is enormous. XMethods, the "virtual laboratory" for web services (www.xmethods.com), houses thousands of services submitted by independent developers and entrepreneurs (a sampling of recent submissions: "PopulatedPlaces", a service that provides Lat/Lon geocodes for 5+ million populated places around the globe; "FreshScore.com Live Score Service", that provides live sports scores world wide; "GlobalStockQuote", that provides updated London, New York or Nasdaq stock quotes for companies listed on these exchanges). Popular web sites such as Google (www.google.com) and Amazon (www.amazon.com) are beginning to wrap pieces of their web functionality in service APIs. And major software infrastructure vendors such as Microsoft, IBM and Oracle

have made major commitments to building tools and platforms for developing and deploying such services.

While "disintermediation" has been one of the clarion calls amidst the hype of the World Wide Web, a world of networked services creates a great need for new forms of mediation. Mediation systems are needed that can perform aggregation, integration and composition of web services conveniently, efficiently and scalably. We have seen clues to the demand for this kind of mediator functionality in the page-based web, where many popular web sites aggregate multiple information sources based on a unifying theme. Investment web sites such as the Motley Fool (www.fool.com), for example, bring together stock research, SEC data, financial news, personal finance service journalism, investment bulletin boards, and stock price feeds in a single location. But such sites represent a primitive form of mediation. Networked services provide building blocks for more dynamic and flexible forms of aggregation, integration and composition.

Another trend rising in the networked world is the demand for "real-time" information; for data associated with events as they occur. The business side of this demand can be seen in best-selling books such as "The Power of Now" (Ranadive and McNealy 1999) and "Sense and Respond" (Bradley and Nolan 1998), which describe how businesses can gain competitive advantage by leveraging the power of the Internet to provide real-time data on events of interest to them. Ranadive coins the term "event-driven business" for businesses that are able to use this technology to its fullest extent (Ranadive and McNealy 1999). Demand for real-time information is being seen at the personal and consumer level as well. Many Web sites now provide individuals with simple forms of customized event notification. For example, Amazon.com will send you email when a new book on a specified topic or a new CD by a specified artist arrives in stock. ESPN.com will send you the box scores of your home team's latest game as soon as it ends. Networked services provide the basis for more sophisticated forms of notification and monitoring of events and data. Such capabilities will be an indispensable feature of the service infrastructure.

In this dissertation we describe a new kind of mediation system that we call an Active Service Integration System (ASIS), which combines the capabilities described above. An ASIS provides monitoring and event-driven aggregation, integration, and composition of declarative requests over networked, data-intensive information sources and services. An ASIS provides the functionality typically associated with information integration together with a continuous monitoring capability akin to that of an Active Database Management System (ADBMS) (Widom and Ceri 1996). There are many challenges in constructing an ASIS. For example:

- **Service Description and Discovery**: How are services of interest to a client discovered and engaged? How are the services and capabilities of a service provider described to potential clients?
- **Heterogeneity and Autonomy**: Network service providers will display heterogeneity along many dimensions, from hardware and operating system platform, to development language, to the computational capabilities of the service itself. Furthermore, services will often span administrative boundaries, and will simultaneously serve clients from many different administrative domains. How can clients easily adapt to the consequent subtleties and limitations of individual services?
- **Security**: What is the proper model of security in a service mediator? How can potentially heterogeneous security models of individual providers be pieced together into a coherent whole?
- **Efficiency and Scalability**: How can requests involving multitudes of networked services and large volumes of data be executed efficiently? How can an ASIS handle large request loads and large numbers of services without a severely degraded performance?

Many of these issues are similar to those of other mediation systems that have received significant attention and research. But the active functionality and network service

context of an ASIS adds new challenges across all of these dimensions that are not well understood.

1.2 Motivation

Our research is motivated by the fact that many web services will be data-intensive, and so database technology will have much to offer the world of web services integration. But while most work on information integration has dealt with query processing over distributed, autonomous databases having heterogeneous *query* capabilities, there has been little research exploring the monitoring of complex conditions over distributed, autonomous data sources having heterogeneous *monitoring* capabilities, which is the central role of an ASIS. We describe the basic construction of an ASIS, and we describe techniques that extend database technology and previous work in information integration to perform cost-based, capability-driven request processing that is simpler, less error prone, and more efficient than manual approaches to service integration, and that adapts to the heterogeneous capabilities, including the heterogeneous monitoring capabilities, of Internet services.

Caching is among the fundamental techniques to scaling computing systems, and it is an indispensable technique in building scalable and efficient network-based information systems. The majority of computing systems that cache data rely on the *principle of locality* in deciding what to cache, with mixed results. But in the context of an ASIS we can do better. ASIS requests are long-lived, and so the set of active requests provides strong information on the future information demands of the system. This information provides the basis for good decision making on which data to retain and maintain in the mediator. We describe a framework for cost-based caching for active service integration that is motivated by this observation. But our framework results in a highly complex optimization problem. We describe a multi-pronged attack on this problem that makes it solvable in practice. We observe that the problem involves a great

deal of redundant planning work, and so by sharing the planning work via a technique we call *Multiplex Query Optimization* (MuxQO), we can solve the problem far more efficiently.

A final motivating observation we make in this dissertation is that much monitoring effort in an ASIS may be expended in evaluating conditions that cannot possibly contribute to the overriding condition of interest. In particular, if we can show that a monitored object is temporally distant from satisfying a condition of interest, we can ignore it for a while. We describe a technique to exploit this observation based on rich application semantics. In particular, we exploit constraints on information change to reduce system load, improve scalability and shorten response time in an ASIS.

1.3 Thesis Statement and Contributions

My thesis is that inter-task sharing together with the specification and exploitation of service characteristics and capabilities are key elements in building a scalable and efficient Active Service Integration System, and in building other systems where similar problems arise. In particular, techniques for sharing of result processing, data movement and plan optimization are important. Additional gains can be made in such systems by exploiting rich application semantics, in particular, the semantics of information change.

The primary contributions of this thesis are:

- A description and implementation of a basic ASIS that recognizes and addresses fundamental issues of construction. In particular, we describe a metadata model and an extension of database query processing techniques to handle event-based processing over heterogeneous, distributed, data-intensive services.
- A detailed model and framework for cost-based view (or cache) selection in an ASIS, and a multi-pronged approach to handling the resultant optimization problem in a tractable manner.

- A description and implementation of Multiplex Query Optimization (MuxQO), a method for efficiently handling groups of similar query optimization problems. We characterize the applicability of MuxQO, and we describe a performance evaluation that demonstrates the effectiveness of MuxQO in handling cost-based view selection in an ASIS.
- A description and an analytic evaluation of a novel approach to exploiting rich application semantics to improve the efficiency and scalability of an ASIS. In particular, we describe and evaluate a method for exploiting constraints on information change over time in order to reduce the workload and response time of monitoring complex, integrated conditions.

1.4 Thesis Outline

The first half of this dissertation describes the Paradox Active Service Integration System. Paradox is a prototype system that we developed that deals with many of the fundamental problems in ASIS construction and in the efficient processing of ASIS requests.

Chapter 2 provides a high-level overview of the Paradox system. We describe the design goals and philosophy behind the system. We present the system's architecture, its basic functionality and its major components. We illustrate the flow of control and data through the system in response to a request.

A central theme in the design of Paradox is that we can exploit knowledge of the characteristics of services and service providers to support requests in an efficient and scalable manner. We refer to this knowledge as metadata. In Chapter 3 we describe the metadata used by Paradox and how it is managed. We present the mechanisms for the provision and retrieval of service metadata. We also describe the metadata vocabulary that supports the Paradox system.

Chapter 4 describes the guts of the Paradox system. In particular, we describe how requests are converted into programs against the Paradox execution engine, and the operations that are supported by that engine. Skeletally, the process resembles that of query processing in database systems. But we add new logical and physical operators to support monitoring, integration, and heterogeneous source capabilities. We also provide a form of operator merging across requests to improve system scalability.

We have used the Paradox system to drive scenarios from the DARPA Command Post of the Future research project. These scenarios involve the integration of a variety of heterogeneous services developed by separate, geographically dispersed research groups. Chapter 5 presents one such scenario as an extended example of the application of the Paradox mediator.

In the second half of this dissertation, we describe some of the problems of scalability and efficiency that are not thoroughly addressed in the Paradox system. We go on to present advanced techniques for coping with these problems.

Chapter 6 presents a background discussion of query optimization basics that is needed to understand the techniques that will be described in the subsequent chapters. In particular, we discuss important data structures used in state-of-the-art approaches to query optimization.

Chapter 7 describes how an ASIS can share the work of processing, data movement and planning across multiple tasks, and how such sharing can greatly improve the performance and scalability of the system. One important opportunity for sharing arises where supplemental views can be materialized and maintained (i.e., cached) at the mediator and shared among multiple executions of a long-lived request. Another opportunity occurs where multiple tasks must be executed in response to a single event, and these tasks can share common partial results. We describe a cost-based framework for implementing these forms of sharing, and we describe a multi-pronged approach to making the resultant optimization problem tractable.

A third opportunity for sharing presents itself in the context of the framework described in Chapter 7. This time the opportunity is for sharing of planning and

optimization effort, as opposed to evaluation effort. In Chapter 8, we describe *Multiplex Query Optimization* (MuxQO) as a technique for exploiting this form of sharing. We describe the implementation of the MuxQO, and how current database query optimizers can be extended to incorporate MuxQO capabilities. We discuss the range of applicability of MuxQO. Finally, we describe a detailed performance evaluation that demonstrates the effectiveness of the technique in the context of the framework of Chapter 7.

Chapter 9 presents another advanced method for improving the scalability and performance of an ASIS. This method exploits rich application semantics to decrease the load on a system. In particular, we exploit *constraints on information change over time* to avoid expensive and frequent processing of conditions that cannot yet be satisfied. We present an analytic study of the effectiveness of our method.

Finally, Chapter 10 summarizes the contributions of the dissertation and discusses important related work in a systematic manner. We close by with lessons for others building a similar system.

Chapter 2

Paradox Overview

The work reported in this thesis revolves around the Paradox Active Service Integration System, which we have developed at OGI. Paradox provides composition, integration, aggregation and monitoring of collections of network-based services and data sources. We refer to Paradox as an *Active Service Integration System* (ASIS) because it provides functionality typically associated with information integration oriented toward a general form of network-accessible services, along with a continuous monitoring capability akin to that of an Active Database Management System (ADBMS, Widom 1993, Widom 1994, [Widom, 1996 #648]). In this chapter we present a high-level overview of the Paradox system, describing both the philosophy and architecture behind it.

2.1 About the name

Paradox is named after Paradox Lake, a small lake located in the Adirondack Mountains of northern New York State. During the springtime, snowmelt overflows the Schroon River, the primary artery leading away from Paradox (eventually) to the Atlantic Ocean, causing the river to reverse direction and flow back into the lake. This flow reversal is the "paradox" that inspired the lake's name. In analogous fashion, processing in the Paradox system is often initiated by information flowing from network-accessible sources and services (the "ocean") back towards the Paradox mediator (the "lake"), as opposed to the normal initiation of client-server flow from mediator to source. This same distinction has been described elsewhere in terms of source "push" versus mediator "pull." Paradox can rightly be thought of as a hybrid system that employs both push and pull. More specifically, in terms of the taxonomy of Zdonik and Franklin (Franklin and Zdonik 1998), Paradox employs aperiodic push combined with aperiodic and (less frequently) periodic pull. To torture the analogy further, the Paradox mediator is a resource-intensive server that maintains a large cache ("lake") of information extracted from sources to improve response time and scalability of the services it offers.

2.2 Areas of emphasis in Paradox

Paradox fits the basic definition of a mediator-based system as described in the introduction (Wiederhold 1992). There are at least five issues that distinguish Paradox from most other mediator-based systems:

1. Paradox maintains a strong adherence to a network-of-services model of computing.
2. Paradox supports a declarative model for specifying compositions of services, and a data-flow model for executing such compositions.

3. Paradox explicitly models and adapts to service providers that exhibit a wide range of computational capabilities.
4. Paradox provides a continuous monitoring capability over complex distributed conditions and service compositions.
5. Paradox models and exploits source semantics, particularly semantics involving information update and change, to improve efficiency and scalability.

These issues arise in the context of a distributed collection of autonomous, heterogeneous service providers. Service providers are autonomous in that each is expected to be within its own administrative domain and to be sensitive to the needs of a variety of clients, not just the Paradox mediator. They may be heterogeneous in a number of dimensions, including hardware and software platform, and computational capability.

Paradox does not attempt to solve all of the problems of active service integration. For example, Paradox ignores the very difficult and important problem of semantic data and service heterogeneity, and the related issue of global schema definition. Paradox pays limited attention to matters of wrapper construction, in particular to the issue of handling unstructured or semi-structured data sources. Nor does Paradox deal with challenging distributed systems issues such as fault tolerance or high availability.

Our choice of focus, in part, reflects a certain vision of the future of service integration:

- **Service delivery mechanisms will grow in sophistication**: The proliferation and growing sophistication of available online services and data sources requires service delivery mechanisms that go beyond what on-demand querying alone can support. Continuous condition monitoring and event-based processing is essential to tying services into complex processes, workflows and notifications, and will become an indispensable aspect of service delivery.
- **Service integration and composition will be data-intensive**: Compositions of network services will often involve one or more high-volume data sources. (Recall, for example, the 5 million cities in the Populated Places service.) As a consequence, efficient compositions can be obtained by employing a dataflow

architecture in which data is pipelined through a series of services. This model also favors the declarative specification of service compositions, which allows choices to be made at to how compositions are executed.

- **Service providers will provide varying levels of computational support**: Scalable and efficient integration over large numbers of services, particularly in the wide area, can be greatly aided where service providers are willing to assume significant computational burden in support of a more sophisticated level of service. At the same time, sources with limited capability will not go away. This trend means that an increasing range of computational capabilities must be accommodated and exploited by integration systems.
- **Unstructured data, and wrapping of unstructured data, is of decreasing import**: Much integration work in the past has focused on efficient construction of source wrappers, often for handling unstructured or semi-structured data formats like HTML (Raggett, Hors et al. 1998). Our view is that while the future of integration systems may not quite be wrapper-free, much of the difficulty of wrapper construction will go away as sources use technologies such as XML (Bray, Paoli et al. 2000) together with powerful APIs to expose the structure and semantics of their services and data. Important information in the future world of network services will be explicitly structured.

Semantic heterogeneity will continue to be a vital concern to integration systems of all stripes. If a *lingua franca* such as XML continues to gain enthusiastic support, resolving semantic heterogeneity will largely be a matter of translating between standardized *document type definitions* (DTDs) or some richer schema definition language such as XML Schema (Fallside 2001). Since the beauty of standardized DTDs (or schema), as other standards before them, is sure to be that there are so many of them, we see no shortage of work to be done in this area. But this is not our emphasis in this work. We will, however, describe how such translation systems could fit into the Paradox architecture.

We will also remain agnostic concerning global schema definition. Recent research has produced two primary approaches to global schema definition, *local as view* (LAV) and *global as view* (GAV). In a GAV system the global schema is specified as a view over the (local) information sources being integrated. A query over this schema can easily be unfolded into a query over the participating sources. This approach has been used by many research integration systems, including TSIMMIS (Chawathe, 1994), Garlic (Roth, 1996), and Disco (Tomasic, Raschid et al. 1998), among others. In a LAV system, in contrast, a global schema is defined as a group of predicates, and the (local) sources are described as a series of views over this schema. With the LAV approach, a query must be rewritten in terms of the set of source views before it can be processed. The Information Manifold is the primary exemplar of this approach (Levy, Rajaraman et al. 1996). The LAV approach offers the important advantage that information sources can be added to the system without modifying the global schema definition. Query reformulation, however, which is equivalent to the problem of rewriting queries using views in this context, is more difficult; many of the reformulation algorithms do not scale well. Recent work, however, has produced more efficient and scalable algorithms for accomplishing this task (Pottinger and Halevy 2000). GAV systems also offer more fine-grained control over global schema definition (Ullman 1997). The Paradox system, in contrast, takes the approach that the global schema is the straight syntactic union of the schema of the participating service providers. Table 2-1 summarizes the areas of emphasis and de-emphasis in the Paradox system.

Note that in later sections of this thesis we will describe advanced techniques for exploiting the semantics of information to build more efficient and scalable active service integration systems. These advanced methods are not currently implemented within Paradox, but they are very much aligned with the goals and philosophy of the system.

Paradox Concern	Paradox Apathy
1. Easy integration and composition of data-intensive networked services. **2.** Scalability and efficiency. **3.** Modeling and exploitation of heterogeneous computational capabilities. **4.** Continuous condition monitoring **5.** Modeling and exploitation of application semantics.	**1.** Semantic data and service heterogeneity. **2.** Defining global schema. **3.** Unstructured and semistructured data, and wrapper construction. **4.** Fault tolerance.

Table 2-1: Paradox concern and apathy

2.3 Paradox Architecture

The basic architecture of Paradox is depicted in Figure 2-1.

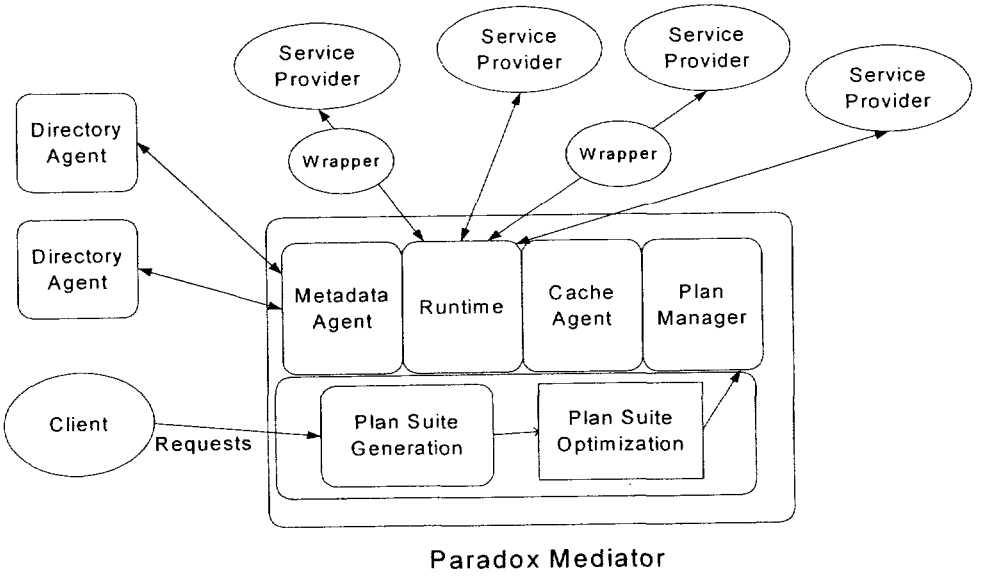

Figure 2-1: The Basic Paradox Architecture

Paradox is written entirely in the Java programming language (Joy, Steele et al. 2000). All remote communication occurs via Java Remote Method Invocation (RMI), an object-oriented Remote Procedure Call (RPC) extension to the Java language (Pitt, McNiff et al. 2001). The distributed object paradigm was chosen for its conceptual elegance in modeling a variety of network-based services. Java was chosen for its relatively clean instantiation of this paradigm.

The major components of the architecture include: *service providers* (also called *agents* or *sources*), which are network-resident providers of the basic services being integrated; *wrappers*, which homogenize the interface to a specific service or set of services; *directory agents*, a distinguished type of service provider, which track available services and maintain various metadata pertaining to these services; and the *Paradox mediator*, which provides request processing and includes components that perform plan generation and optimization, plan execution, management of the workload of plans, cache management, and management of metadata on sources and services. We describe each of these components in greater detail.

2.3.1 Service Providers

A service provider, also called an *agent* or a *source*, provides one or more digital services to the Paradox mediator. A digital service can be anything that can be encapsulated in a simple programmatic interface and provided over a network. In the domain of book sales useful services might include a query capability over a collection of books for sale or over a collection of book reviews. In investments useful services may provide stock quotes, portfolio tracking, corporate research, earnings projections, or stock and bond analysis. A service can be an encapsulated set of data or knowledge, a business process or best practice, or a complex function or computation. An example of the latter is a geo-coding capability that translates a street address to a global location based on latitude and

longitude, or a function that computes the distance or driving time between two addresses.

In Paradox, the content of a service is described to the mediator as a series of one or more predicates. For example, the content of a service that provides information on books for sale as well as a series of book reviews might be described by the two predicates:

$$Book(ISBN, Title, Author, Subject, Price).$$
$$Review(ISBN, Publication, URL).$$

Note that arguments are untyped and distinguished by their ordering within the predicate. The predicates that make up a single service represent the interface to that service. Multiple predicates within a single service may be called in combination, and such combined calls are interpreted declaratively. Note that Paradox does not support complex workflows having, for example, ordering constraints through multiple predicate calls.

Service providers encapsulate their service offerings and expose them via the Internet to the Paradox mediator, possibly with the help of a "wrapper" program, via the Java Remote Method Invocation (RMI) protocol. In general, providers are assumed to be autonomous, to operate within their own distinct administrative domain, and to be heterogeneous along several dimensions including hardware platform, operating system, development language, and computational capability.

2.3.2 Wrappers

Wrappers provide a degree of uniformity to the interactions between the mediator and a heterogeneous array of service providers. The primary role of a wrapper is to create a dataflow access model around a service that does not already conform to the required model. Such a model flows input data into the service call and flows results out to the receiver of the call's output. The model is similar to the iterator model used by many

database systems (Graefe 1993). As such, it supports three primary calls, shown in Figure 2-2.

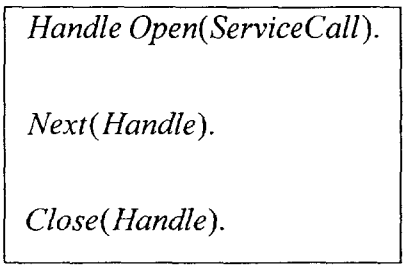

Figure 2-2: Paradox Wrapper Interface

Open() performs whatever initializations are required to prepare the service call for input and returns a unique handle to a stream used to process the remainder of the service call instance. The *ServiceCall* argument specifies the call (or combination of calls) being made, predicates to apply to the call, the list of arguments to return in the result, and an optional list of properties, such as a sort order for the result. We sometimes refer to this initialization process as *service instantiation.* The stream handle is then used in subsequent calls to the service. *Next()* synchronously returns the next "chunk" of results from the handle, and will return *null* when there are no more results to process; *Close()* closes down the service instance, reclaiming state and releasing resources associated with it.

In general, a single service call instance can accept multiple input or output streams coming from, or going to, arbitrary destinations. Paradox does not exploit this full generality. Paradox limits each service call to a single input stream and a single output stream. In addition, all inputs come from the mediator in Paradox, and all outputs flow back to the mediator. Extending our model to more of a peer-to-peer approach, in which sources can pass information directly to each other without mediator intervention, is left for future work.

Note, further, that a wrapper must handle buffering of input and output data. It must specify the maximum chunk of data (in records) that it can receive from its input at a time. By default it will be sent a single record at a time, and all wrappers must be able

to handle a single-record chunk. But for remote, wide-area network-based service providers it is usually more efficient to accept input and provide output in larger chunks. Wrappers must also deal with threading issues. Generally it is beneficial for service providers to enable concurrent access by multiple clients.

The dataflow model of services allows an arbitrary number of services to be strung together into a pipeline, it minimizes the space and delay associated with temporary storage of intermediate results in such pipelines, and it greatly simplifies the scheduling and synchronization of groups of services. For groups of services that process large volumes of data, significant savings in resources and time can accrue to this technique (Graefe 1993).

Note that, on the surface, a group of services pipelined together in this manner creates a *demand-driven* dataflow. That is, data flows through the pipeline when the top-level service calls *next()*, which triggers a *next()* call from its child, which triggers a *next()* call from its child, and so on down the pipeline. A pure demand-driven flow can be inefficient in a wide-area network setting such as the Internet, however, because it can exacerbate the effects of high-latency and burstiness found in such environments. The Paradox mediator hides some of this cumulative effect by dedicating a thread to pre-fetching and buffering as much upstream data as possible (to a configurable limit), rather than waiting for a *next()* call from the upstream service bring over the next granule of data. This method converts the execution model to one of data-driven dataflow with pushback. This method is similar to that employed to encapsulate parallelism in the Volcano system (Graefe 1990), and to that of *query scrambling* (Amsaleg 1998). Wrappers and service providers can cooperate to employ similar methods.

2.3.3 Directory Agents

The Paradox mediator finds the service providers that it needs to satisfy a request by consulting one or more directory agents. A directory agent is a distinguished type of service provider that offers a "yellow pages" style listing of service providers. This

service, like the telephone yellow pages, allows service providers to be located based on what it is they provide (as opposed to a "white pages" service that finds objects by name). In addition, directory agents provide metadata on service providers and the services they provide, and support searching for providers based on conditions over this metadata.

In particular, a directory agent must support the *Trader* service (named after the CORBA-based service of the same name (Orfali, Harkey et al. 1995)) with the call:

$$Query(+ServiceType, ?Constraint, -ResultList).$$

The *ServiceType* argument must be bound. It includes the service name, the call name, and the call arity. *Constraint* is an optional `boolean` condition over the metadata associated with the service. For example, a directory agent can be queried for agents that provide the *Book/5* call within the *BookSales* service (with no additional constraints) with the call: $Query((BookSales, Book, 5), [], ResultList)$. The *Query* call returns a single tuple with *ResultList* instantiated to a list containing a RMI reference and related metadata for each service provider (that the directory agent is aware of) that supports the given service and call.

Metadata provided by a directory agent must conform to the *Paradox Metadata Specification*, which includes a rich set of information on service and agent characteristics that is useful in processing requests. Characteristics include data-oriented statistics on service content, computational capabilities of the service provider itself. We describe this specification in detail in the chapter on metadata management.

In general, a directory agent need only provide the prescribed *Trader::Query/5* call. The agent can use any means it chooses to gather the information it provides. The Paradox system includes an implemented directory agent called the *Paradox Trader*, which is based partly on the CORBA Trader service. Service providers wishing to use the Paradox Trader must explicitly register their service offerings with it.

2.3.4 The Paradox Mediator

The Paradox Mediator is the central component of the Paradox system. The mediator performs a series of tasks associated with efficient and scalable processing of queries and monitoring of conditions over multitudes of heterogeneous, distributed service providers. The mediator is based on a modular design with major components that perform plan generation and optimization, management of plans and source monitors, cache creation and management, aggregation and management of metadata, and plan execution. We describe each component, now, in greater detail.

2.3.4.1 Metadata Agent

The Paradox Metadata Agent (PMA) collects and aggregates metadata on services and sources from various directory agents, as well as from local sources of metadata. The PMA monitors directories actively for information on services related to the mediator workload, and maintains a local cache of this information. The PMA acts as a liaison between the directory agents available to the Paradox system, local sources of metadata, which may include a system administrator or a local system monitoring function, and the planning, optimization and monitoring functions that use this data.

2.3.4.2 Plan Generator and Optimizer

The plan generator and optimizer takes a request in a Datalog-like form and produces an optimal group (or suite) of plans for computing the current result of the request and monitoring future changes to the request. In its more advanced form (not currently implemented) this component will also calculate an optimal cache configuration. Conceptually this process consists of plan generation, where feasible plans are produced, plan characterization, where an extensible set of characteristics is computed for each

plan, and plan selection, where a priority function over the set of characteristics is evaluated in choosing the most desirable plan.

2.3.4.3 Plan Manager

The plan manager stores and maintains long-lived requests and their associated plan suites. This process involves initiating and de-commissioning plan suites, tracking changes to service availability and other service characteristics, and triggering plan extensions or plan re-optimization where necessary.

2.3.4.4 Cache Agent:

The cache agent maintains a local semantic cache of data replicated from remote service providers. The cache agent provides an interface to this data that makes it appear to the rest of the system as much like any other service provider.

2.3.4.5 Runtime

The Paradox Runtime is the plan execution engine. The runtime provides operations needed to integrate and monitor remote services. These operations include a variety of standard database algorithms for performing joins, selections and projections, as well as mechanisms for combining such algorithms with remote services of varying capability in a pipelined execution. Also included are algorithms for monitoring remote sources in the face of heterogeneous source capabilities, and for combining and merging multiple monitoring requests.

2.4 Tracing A Request

The Paradox mediator provides a simple Java API for entering requests and receiving results. A request is entered as a string, and results are returned as a list of tuples of objects. The mediator can be run within a local Java client process, or it can be run in a remote Java process and communicate with clients via Java RMI. The system includes a simple GUI interface that allows manual entry of requests and viewing of results.

Paradox accepts requests as conjunctive datalog queries (Ullman 1988), which is equivalent to the set of select-project-join (SPJ) queries in the relational world. For example, the following request asks for the *Title*, *Price*, and Review *URL* of all books for which a review is available and the *Price* is less than $20:

BookRevPrice(Title, Price, URL) $\rightarrow$
BookSales::Book(Title, Auth, Price) & *Review(Title, URL)* & *Price* $<$ *20.*

In general, Paradox accepts requests of the form:

$$Head \rightarrow Conjunct_1 \,\&\, \ldots \,\&\, Conjunct_n \,\&\, Pred_1 \,\&\, \ldots \,\&\, Pred_m .$$

Here the $Conjunct_i$ are literals of the form $Service::Call(Var_1,\ldots,Var_n)$ where *Service* is optional and all arguments to *Call* are variables. The $Pred_j$ are predicates over one or two of the variables contained in the $Conjunct_i$, and *Head* is a literal whose arguments are all variables that appear in some $Conjunct_i$.

Operationally, a request is interpreted by Paradox in the following way: Compute the current result of executing the request, and continuously monitor the request for changes induced by changes to component services encompassed by the request. Note that this is a dynamic process. As new books or reviews are added to their respective sources, for example, we may expect additional answers. Continuous monitoring halts when the client explicitly revokes the request.

The flow of a request through the Paradox mediator is shown in Figure 2-3 below. When a request is received it is passed to the *request parser*, which confirms that the request is syntactically correct, parses it, and passes the resultant parse tree to the *plan verifier*. The verifier contacts the *metadata agent* to verify that the service providers necessary to process the request are available and to obtain metadata on these providers and their services. The metadata agent will contact *directory agents* as needed to obtain this information. The verifier passes a verified logical plan, adorned with service metadata, to the *plan optimizer*. The optimizer then generates an optimal suite of plans for executing and monitoring the request, including plans for creating and maintaining a cache to support the request. This plan suite is handed off to the *plan manager*, which passes the cache component of the suite to the *cache agent*. The plan manager and cache agent cooperate to initialize their respective plans with the *runtime engine*, which merges new monitoring conditions with existing *source monitors*, launches new source monitors where appropriate, attaches the specified plans to their assigned monitor conditions, and evaluates the current result of the request. On a continuous basis, then, until the request

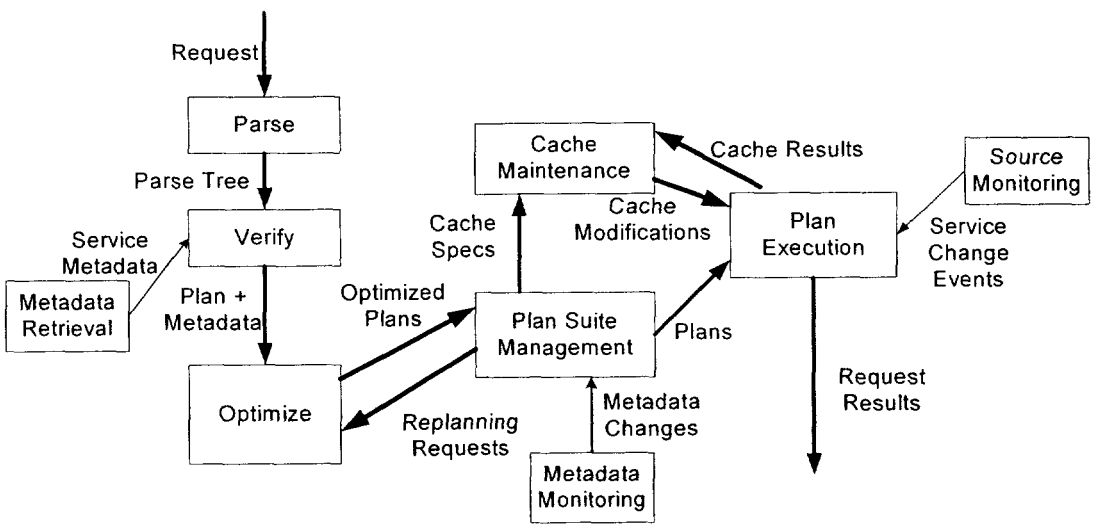

Figure 2-3: Request Flow in Paradox Mediator

is retracted, two kinds of event-based processing occur: Source monitors track changes to service content, and pass such changes to the runtime to evaluate corresponding changes (if any) to the overall request; and the plan manager tracks changes to metadata on services relevant to the request, with the assistance of the metadata agent, and maintains the plan suite for the request as appropriate.

2.5 Chapter Summary

In this chapter we provided a high-level introduction and overview of the Paradox system. We discussed the design goals and philosophy of the system, its functionality, its major components, and the flow of requests through the system. Paradox does not try to solve all of the problems of service integration. Notably, we ignore issues of semantic heterogeneity. We focus, instead, on efficient and scalable processing and monitoring of requests in the face of heterogeneous source capabilities. The chapters that follow will delve more deeply into the novel features of the Paradox system.

Chapter 3

Metadata in Paradox

Network accessible services and service providers can be expected to exhibit a variety of characteristics. A central theme of the Paradox system is that we can exploit knowledge of particular characteristics to provide scalable and efficient service integration, aggregation and monitoring. This knowledge is captured in the form of service metadata. In our overview of the Paradox system, in Table 2-1, we listed five areas of emphasis. Metadata plays a central role in each of these areas. We list them again below, and describe the role of metadata in each:

1. Easy integration and composition of data-intensive networked services: Wrappers, for sources that need them, go part of the way toward providing a uniform interface over network-based services. Metadata is used to go the rest of the way. Metadata is used to describe service characteristics. The Paradox mediator then adapts to these characteristics during request execution.

2. Scalability and efficiency: Paradox employs metadata to describe the computational capabilities of service providers and various aspects of the services they provide. All of these factors are weighed in generating efficient plans for request execution. They are also used to make decisions on data caching, and to employ specialized techniques for efficient distributed condition monitoring.
3. Modeling and exploitation of heterogeneous computational capabilities: As mentioned above, metadata is used to document heterogeneous source capabilities. Paradox automatically adapts to these capabilities, exploiting advanced capabilities for greater efficiency whenever possible.
4. Continuous condition monitoring: Several of the techniques mentioned above are used for efficient condition monitoring. In particular, we exploit metadata on information change for efficient caching and condition monitoring.
5. Modeling and exploitation of application semantics: Again, metadata that pertains to information change, including the semantics of information change, can be exploited for efficient request execution.

Metadata is central to everything that Paradox does. In this chapter we describe the mechanisms Paradox uses to obtain and manage metadata, the service and source characteristics we are interested in, and the way they are represented as metadata. Later chapters will focus on how and where this information is applied.

3.1 Metadata Management

A user of the Paradox system submits a request in a simple declarative language that specifies the content of the services needed. The mediator takes this request and generates effective plans for executing and monitoring the request. To do so, the mediator must have information about the services available to it and the capabilities of the providers of those services. In Paradox, this information is provided as metadata. Source and service metadata play a role in the Paradox system that is analogous to that of

catalog information in a typical database management system. Figure 3-1 essentially replicates our earlier picture of the general Paradox architecture (Figure 2-1), highlighting the components involved in the metadata management process in **bold**.

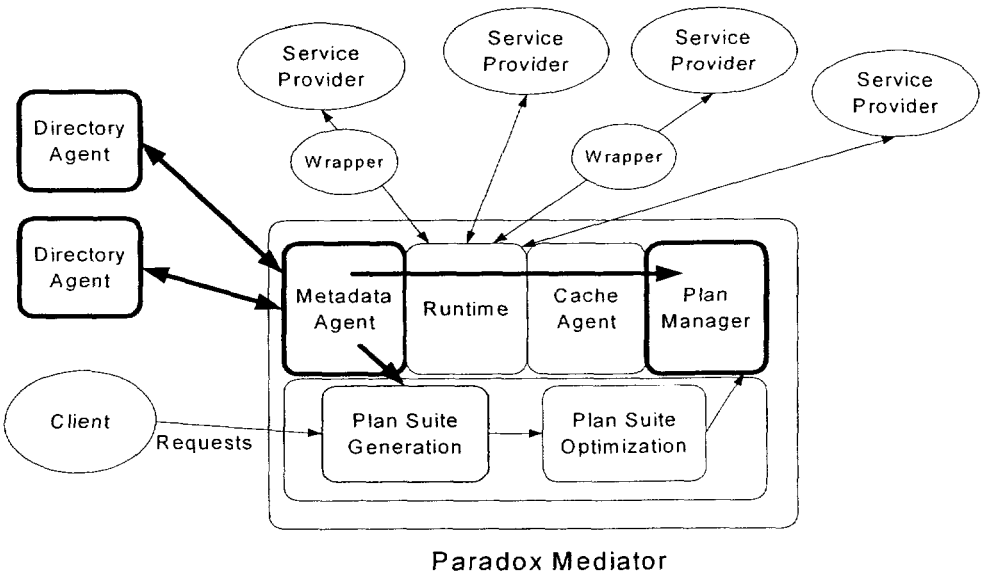

Figure 3-1: Metadata Management Components of Paradox

Paradox assumes the existence of one or more network-based services that we call *directory agents*. A directory agent gathers metadata about other network-based service providers and provides a "yellow pages" style access to it via a distinguished interface. Paradox relies on the *Paradox Metadata Agent* (PMA), resident at the mediator, to consolidate information from all of the directory services available to it and present this information to internal components of the system. In particular, the plan suite generator issues an initial query to the PMA on the availability of particular service content, based on a user request. The PMA issues the query to various directory agents and returns metadata on services that provide this content. The generator and the plan suite optimizer use this information to create plans for the execution and monitoring of the request, which are passed to the *Plan Manager*. In the meantime, the PMA monitors directory information for future changes that pertain to this content. Any changes that occur are

passed to the Plan Manager, which determines if and when any action will be taken in response to these changes (e.g., re-planning). The following example provides more concrete details for this process.

Example 3-1 (Basic Metadata Mechanisms): At startup time, the Paradox system is "bootstrapped" with a single, known directory agent. The PMA queries this directory agent for the addresses of other directory agents (a process that may continue recursively, if it is configured to do so). A set of directory agents is arrived at. Now suppose a user issues the simple request that we saw in Chapter 2, which asks for the *Title*, *Price*, and Review *URL* of all books for which a review is available and the *Price* is less than $20:

$$\begin{aligned} &\textit{BookRevPrice(Title, Price, URL)} \rightarrow \\ &\qquad \textit{BookSales(Title, Auth, Price)} \,\&\, \\ &\qquad \textit{Review(Title, URL)} \,\&\, \textit{Price} < 20. \end{aligned} \tag{3-1}$$

The plan suite generator receives this request, which is parsed and verified. Verification includes querying the PMA for a service provider that provides the *BookSales*/3 service call, and for a service provider that provides the *Review*/2 service call. If such service providers are available, the PMA returns a reference to a proxy object for each service provider, along with metadata on the service providers and their respective services. The Optimizer generates plans to execute and monitor the request based on this metadata and passes these plans to the Plan Manager. The PMA continues to monitor the directory agents for changes pertaining to the *BookSales*/3 and *Review*/2 calls. Suppose the previously returned service provider for *BookSales*/3 goes away. Then the PMA notifies the Plan Manager of this event. The Plan Manager may respond, for example, by modifying the monitoring plans for this request to utilize an alternative service provider. □

3.1.1 Directory Agents

Paradox assumes the presence of network resident directory agents to provide metadata on services and service providers. A directory agent gathers metadata about other network-based service providers. The gathering of metadata may be implemented in a variety of ways: Crawlers may go out and proactively discover service information; service providers may explicitly register information with directory agents; etc. But regardless of the metadata gathering method, Paradox expects this information to be made available via a specific "yellow pages" style interface. A "yellow pages" style interface is one in which relevant information can be searched for based on its function or content. (As opposed to a "white pages" service where service providers are located by name.) The relevant interface consists of a single service call as follows:

Query(Service, Arity, Result) (3-2)

The `Query/3` operation takes a service name and arity. It returns a `CallDescriptor` object as a result that represents a service call that matches the given service. There may be multiple `CallDescriptor` objects for a given service name and arity. Each `CallDescriptor` object includes a Java RMI reference to the relevant service provider, a call name, a *group* name, a *Call Combination Flag* (CCF), and a list of associated metadata. (We describe the attributes that may appear in this metadata list shortly.) Groups allow a provider to partition service calls in various ways. The group name indicates which group the given service is in. The CCF indicates how multiple service calls are combined. It is chosen from the set $\{P, PU, C\}$, which is interpreted as follows:

- P = "Partial": Means the service call provides a partial result. Identical service calls from other providers that are also partial should be combined with a normal set union operation.
- PU = "Partial Unique": Means the service call provides a partial result that cannot be replicated in any part by another service provider. Identical service

calls from other providers that are also partial can be combined with a disjoint set union operation.

- C = "Complete": Means the service call provides a complete result. The service provided by this call and this provider subsumes that of any other provider.

The `Query()` call is generally made with only the service name and arity specified. In a Paradox request, content is specified by Horn clause literals that are assumed to map directly to service names. More generally, this mapping could involve a far more complex and rich process. For instance, we might have a knowledge base mapping a call to a number of syntactically different but semantically equivalent calls. Or, in the XML world, we might map a call to a group of standardized DTDs that equate to the requested call. The current Paradox implementation punts on these issues. The service name and arity of each request literal is assumed to be identical to that advertised by the directory services used by Paradox. If a user request includes the literal `Book(Auth,Title,Price)`, for example, then we look for services that provide the service `Book` with arity 3, which we often write as `Book/3`.

Note, finally, that the `Query()` call provides a very simple, generic "yellow pages" style interface. This interface can easily be laid over a variety of standard directory services, such as LDAP (Howes, Smith et al. 1999), Microsoft Active Directory (Lowe-Norris 2000), or the Corba Trader service (Orfali, Harkey et al. 1995).

3.1.2 The Paradox Trader

While the Paradox mediator can interact with a variety of directory services, the current implementation provides one specific directory service that we call the *Paradox Trader*. This service is similar to the standard CORBA Trader service (Orfali, Harkey et al. 1995). The Paradox Trader provides the `Query()` call discussed above for retrieving metadata based on the service call name and arity. The service is *active* in that it supports the installation of triggers on the query call, and will notify trigger clients whenever specified directory information changes. Notification is done via a Java RMI callback

using a well-known interface. We describe the notification interfaces used by Paradox in Chapter 4.

The Paradox trader relies on explicit registration on the part of service providers to collect directory information. Three calls are provided to support service registration, modification and de-registration:

String Export(CallDescriptor Offer)
boolean Export(String OfferId, CallDescriptor Offer) (3-3)
boolean Withdraw(String OfferId)

A service provider that wishes to expose a service to the network issues an `Export/1` call. `Export/1` registers a new service call described by the argument `Offer`, which is a `CallDescriptor` record, as described previously. `Export/1` returns a string identifier that is unique within the scope of the directory service. If a service provider wishes to modify information about a service call that it has already exported, it issues an `Export/2` call. `Export/2` is used to modify an already registered service call, by replacing the previous call descriptor with a new one. Finally, if a service provider wishes to no longer provide a service call, it issues a `Withdraw/1` call. `Withdraw/1` removes a registered offer by its identifier. Both `Export/2` and `Withdraw/1` return `true` if they succeed, or `false` if there is no service associated with the specified identifier. Note that there is no notion of service expiration in this model: Exported services are assumed to be available until they are explicitly withdrawn.

Note that the Paradox Trader is meant to provide the basic functionality to allow the system to run. Paradox subscribes to an architectural model that includes the use of directory services, but we have not done directory service research here. Building a scalable, secure, robust service discovery service for the Internet is a complex research topic in its own right, and a key technology for transforming the Internet into a web of composable services (Czerwinski, Zhao et al. 1999).

3.1.3 The Paradox Metadata Agent

The Paradox Metadata Agent (PMA) provides a consolidated metadata interface for the internal mediator components that need it. These components interact only with the PMA, not with the various original sources of metadata. The plan suite generator and optimizer asks for the initial metadata pertaining to a user request from the PMA, which it uses to generate plans for executing and monitoring the request. All metadata updates pertaining to the request are forwarded from the PMA to the plan manager, which decides whether further action is required.

The PMA gathers initial metadata from network-resident directory agents. It then issues supplemental requests to specialized sources of metadata, the result of which can be used to override, modify or extend the original metadata. Paradox assumes that there are two sources of supplemental metadata: a system administrator source, and an internal monitor source. The system administrator source is designed for metadata entered by hand by a system administrator. The monitor source is designed for metadata that comes from a monitoring function within the mediator that measures characteristics of services and service providers during execution. The final, effective metadata that applies to a given <provider, service, arity> triple is given by a customizable function over the metadata returned by these three sources. This process should be clear as we look at it more closely below.

The PMA answers to a single service call, `GetMetadata/3`, which is defined as a regular, compound Paradox request. As a regular request, it is executed for its current value, which is returned to the optimizer, and then it is monitored for changes to its value, which are forwarded to the plan manager. The monitoring process continues until the original user-issued request is withdrawn. `GetMetadata/3` is defined as follows:

$$
\begin{aligned}
&\textit{GetMetadata(Service,Arity,Result)} \leftarrow \\
&\qquad \textit{Query(Service,Arity,Metadata1)\&} \\
&\qquad \textit{GetProvider(Metadata1,Provider)\&} \\
&\qquad \textit{MonitorMetaData(Provider,Service,Arity,Metadata2)\&} \\
&\qquad \textit{AdminMetadata(Provider,Service,Arity,Metadata3)\&} \\
&\qquad \textit{ResolveMetadata(Meta1,Meta2,Meta3,Result).}
\end{aligned}
\tag{3-4}
$$

`GetMetadata/3` takes a service name and arity as arguments. It goes out to all of the services that provide the `Query/3` call (all directory agents) to retrieve metadata for each available service of interest. A simple internal function, `GetProvider/2`, pulls the provider-identifier from the initial metadata. The <provider, service, arity> triple is then used as a key to access the two supplemental metadata services mentioned above: the monitor source (via the `MonitorMetadata/4` call), and the system administrator source (via the `AdminMetadata/4` call). Finally, the internal function `ResolveMetadata/4` is invoked to produce the final result. This function can be overridden to customize the way in which the three varieties of metadata are combined.

The PMA is basically just another service provider. The only difference between it and other service providers is that it is local to the mediator and it does not register its capabilities with any directories. It is called in a hard-wired fashion from the optimizer. For example, if Request 3-1 above were submitted, the optimizer would issue two requests to the PMA:

(1) GetMetadata("BookSales", 3, Result)

(2) GetMetadata("Review", 2, Result)

As regular requests, each of these calls is executed for its current value, which is returned to the optimizer. Then each call is monitored for changes to its value, which are forwarded to the plan manager. The monitoring process continues until the original user-issued request (Request 3-1) is withdrawn.

The design of the troika of metadata sources is based on a simple rationale. In general, we would like a service provider to tell us directly about its services, via a directory agent. But often an autonomous source may not provide all of the metadata that we want, or the metadata may not be reliable or up to date. Another possibility is that relevant metadata depends on factors beyond the service provider. For instance, latency and bandwidth are important in evaluating the cost of a plan involving an agent, but these factors depend on both the mediator and the agent (and for that matter, the entire route between the two). Under any of these scenarios, we have the option of extending or overriding values based on administrator knowledge of the source, or based on observations of the agent's past behavior and performance as recorded by a system

monitor. One possible scenario, for example, would be having the administrator source provide default values for bandwidth and latency between an agent and the mediator. When a plan executes using this agent, the monitor source records the actual observed latency and bandwidth; this value can be used to modify the default values for future planning. An example of methods for incorporating monitoring information can be found in the Hermes system (Adali, Candan et al. 1996).

Finally, note that in this architecture metadata information propagates from service provider to directory service to the PMA asynchronously and, possibly, after considerable delay. Paradox is designed to be tolerant of asynchrony and temporary inaccuracy in information gleaned from its directory services. This design manifests itself in two ways: First, information provided via directory services is not expected to change frequently. If some information does change frequently, an alternative channel can be specified for its provision (which might, for example, require direct access to the service that it pertains to); second, the Paradox runtime is prepared to handle exceptions and failures caused by incorrect service calls. Note that an architecture that supports synchronous propagation of directory information does not scale to a large number of services or to a large Paradox workload, particularly in a WAN environment. In general, WAN scalability requires tolerance for data that diverge from the ACID properties of traditional database systems. Paradox relies on and provides network-based information that conforms to BASE (Basically-Available, Soft-state, Eventual consistency) semantics (Fox, Gribble et al. 1997). Practically speaking, conforming to BASE semantics in this context means that the state of agents used by Paradox, and Paradox's own state, may at times be inconsistent. As a practical matter any Paradox client must be prepared to accept some temporary inconsistency. We will have more to say about possible inconsistencies in Paradox processing in Chapter 4.

3.2 The Paradox Specification for Metadata

Metadata in Paradox takes the form of a set of key-value pairs. The Paradox Specification for Metadata (PSM) is a listing of keys and value domains that are

recognized and exploited by the system. The specification represents an attempt to capture the characteristics of a wide variety of heterogeneous sources, and to provide guidance in how useful information can be provided with minimal burden to a service provider. Elsewhere we discuss how the Paradox system exploits metadata. Here we simply discuss what the metadata is that Paradox exploits.

The metadata in the PSM can be divided into four basic categories:

1. Service content and invocation: Describes the basic content, functionality and location of the service. Implicitly, this category of metadata also defines how the service is invoked, at a basic level.
2. Computational capabilities and offerings of the source and service: Describes the scope of query and monitoring operations that can be invoked over the basic service content.
3. Characteristics and statistics of the data content of a service: Describes properties, distributions and semantic information that can be useful in evaluating the costs of using a service, and in optimizing the use of a service.
4. Systemic characteristics: Describes characteristics of the overall Paradox system in using a service that affect the costs of its use.

We will motivate and discuss each category in turn, and piece together the specification in the process.

3.2.1 Service Content Metadata

Service content information describes the basic content, functionality and location of the service, which in turn defines how the service is invoked on a basic level. This information is so fundamental that it is not really "metadata," per se. In fact, all of this information is encapsulated in a single `CallDescriptor` type, which was described earlier (in Section 3.1.1). But it is a vital part of the description of a service, so we describe it here as well.

The first key attribute in this category is the object reference, under the key name "ObjectRef". The object reference is a Java RMI proxy object that encapsulates the network address and protocol for invoking the service. The object reference is of type

"RemoteIterator". It supports the basic `Open()`, `Next()` and `Close()` calls of Paradox agents as described in Chapter 2.

The attributes under key names "Service" and "Arity" describe the semantic content (by convention) of the service call itself and how it is referenced syntactically within the content language of the basic iterator calls. "Service" is a Java String object that is the functor of the call. "Arity" is an integer that gives the number of arguments for the functor. As noted earlier, Paradox assumes that these two attributes uniquely define the content and functionality of the service call. A service call is interpreted as set-oriented relation. It returns all the tuples satisfying the invoked call. A call that is a simple function or predicate returns a single tuple. Arguments are ordered. The <ObjectRef, CallName, Arity> triple is assumed to be unique. One obvious extension to this description is to add argument types, to provide the complete call signature. We leave this extension for future work.

The group name attribute, a Java String under key name "Group", provides a level of indirection between a service provider and its call names. It allows a provider to group a series of service calls into a declarative interface. Such groups of service calls can share other metadata properties. For example, two calls may be joined together at a service provider if each call supports the join capability and they are in the same group.

KEYNAME	VALUE TYPE	SAMPLE VALUE
"ObjectRef"	RemoteIterator	IterRef1234
"Service"	String	"QuoteService"
"Arity"	Int	5
"Group"	String	"StockInfoGroup"
"CCF"	in {P,PU,C}	C

Table 3-1: Service Content Metadata

The final attribute in this category is the call combination flag, which we described in Section 3.1.1. The triple <ObjectRef, Service, Arity> is unique, but different service

providers can provide the same <Service, Arity> pair. The call combination flag describes how calls provided by different agents with the same name and arity should properly be combined. It comes under the key name "CCF", an enumeration type (String in Java) taken from the set {P, PU, C}, which stand for "Partial", "Partial Unique", or "Complete", respectively.

Table 3-1, above, summarizes the service content attributes by key name and type, along with sample values for these attributes.

3.2.2 Source Capability Metadata

Heterogeneity is a fact of life in the networked world, and the Paradox system must interoperate with a variety of heterogeneous agents. Several dimensions of heterogeneity, such as computer language and platform, can be easily papered over with a combination of wrappers and standard network protocols. But one important dimension, not so easily "wrapped away," involves the computational capabilities provided by an agent for invoking, querying and monitoring a service. A service may be provided by a standard relational database system, by an object database, by a collection of flat files, by a sensor, or by a source on the World Wide Web that provides only a HTML form-based interface. But homogenizing the computational capabilities of such sources would involve, for example, recreating relational query processing in a wrapper over the flat-file source. We think this approach is unreasonable. An autonomous source may also limit or control the interface exported to a network for reasons of security or performance. For example, a bookseller may not allow arbitrary clients to ask for every book available, since such a query would require a massive data transfer, and a small number of such queries could swamp the server. Such policy matters are reflected in a source's interface as a limited capability. There is typically no reasonable way to homogenize varying capabilities of this sort in a wrapper.

A goal of the Paradox system is to handle a collection of capabilities that encompass a wide range of realistic sources with as little wrapper support as possible. This goal implies that the mediator will see a variety of capabilities, and it must adjust to them.

Capabilities affect both the set of feasible plans and the optimal plan for a query. Paradox tries to take advantage of advanced capabilities where they exist, and to utilize processing capabilities at the mediator, where possible, to enable expressive requests in the presence of more primitive capabilities. Broadly speaking, Paradox considers two categories of capability: capabilities related to the queries or service calls accepted by a source, and capabilities related to monitoring and notification of changes at a source.

Query Capabilities

In the query category, capabilities manifest themselves in one of two ways: as limited binding patterns in a call; or as an extended query capability over one or more calls.

Binding patterns are simple constraints over arguments in a call, indicating that an argument must be bound or unbound under certain circumstances, and therefore limiting the set of queries accepted at a source. Consider the example of a bookseller. Suppose the bookseller provides the service call:

Book(Title, Subject, Author, Price)

Without considering binding patterns, we may assume that we can ask the bookseller for information on all of its books. But to prevent large data transfers, the source may require that at least one of the `Title`, `Subject` or `Author` attributes be specified (bound). In this case, we list the capability as follows:

Book(+Title, ?Subject, ?Author, ?Price)
Book(?Title, +Subject, ?Author, ?Price)
Book(?Title, ?Subject, +Author, ?Price)

The "+" indicates that the corresponding attribute must be bound. The "?" indicates that the corresponding attribute may be either bound or unbound (i.e., there is no constraint on that attribute). Note that the acceptance of a bound attribute can be viewed as a filtering

capability, equivalent to a (limited) selection operation in relational algebra. We may also require that an attribute be unbound, which we mark with a "-". A flat file source, for example, may only be capable of performing a complete "dump" of its contents, with no filtering. Such a dump is equivalent to "all attributes unbound".

Binding patterns limit the kinds of queries that we can ask of a source. In addition, they can limit the feasible plans that can be used to execute larger requests involving the source as a component. Consider a source of book review information, providing the following collection of data:

Review(?Title, ?Author, ?ReviewURL)

Now suppose we want to ask for the Title, Price and ReviewURL for all reviewed books. We can specify this request as:

$$\begin{aligned} &\textit{ReviewedBook(Title, Price, ReviewURL)} \leftarrow \\ &\quad \textit{Book(Title, \_, \_, Price),} \\ &\quad \textit{Review(Title, \_, ReviewURL).} \end{aligned} \tag{3-5}$$

To compute this request, we must execute a *dependent join*. A dependent join is a join in which bindings from the inner relation are required by the outer relation. We must first invoke the `Review` call with no arguments bound. That is, we retrieve all of the reviews. For each `Review` record we must take the `Title` binding and invoke the `Book` call at the bookseller with the specified `Title`. Neither retrieving the entire Book collection first nor retrieving the `Books` and `Reviews` in parallel is feasible since both are in violation of the required binding patterns.

Binding pattern limitations are expressed in metadata as a list of acceptable patterns under the key name "`Modes`". Each pattern is a list of argument binding specifications, taken from the set {+,?,-}, where the list order corresponds to the argument order for the call. The `Book` service, for example, would have a `Modes` value of `[[+,?,?,?],[?,+,?,?],[?,?,+,?]]`.

Binding pattern limitations can be viewed as a problem and an opportunity. They are a problem in that they limit requests that we can make. They are an opportunity in that they may decrease the space of possible plans that we need to explore in implementing a request, and so offer the opportunity for early pruning and more efficient request optimization. Both these factors manifest themselves in a more complex execution planning process.

In contrast, extended processing capabilities are pure opportunity. Aggregated and integrated data-intensive services can be provided in a more scalable manner when the sources can absorb some of the computational burden from the mediator. The Paradox system will generate plans that exploit such capabilities where appropriate. In particular, the system recognizes three forms of extended capability: *tuple restructuring* (or *projection*), *compound calls* (or *joins*), and (degree of) predicate support (e.g., in a selection operation). These capabilities are recorded in metadata under the key name "Query". The value of "Query" is a set of supported capabilities.

Tuple restructuring refers to the ability to pull attributes out of the body of a service call and order them arbitrarily into a new tuple structure. This simple operation can save significant data transfer costs, since it allows unneeded attributes to be filtered away at their source. A service that supports this capability will have the attribute "project" in its "Query" set.

Compound call support refers to the capability of a source to support the declarative composition of multiple service calls. That is, multiple service calls can be pushed to the source as a unit (within a single iterator), and the source can perform the equivalent of a relational join over the calls. Two services are joinable at a service provider if they are within the same service group and they both provide the join capability. Returning to Request 3-5, suppose a single agent provides both the `Book` and `Review` services within a single "BookShop" group, and that each service supports joins. Then the Paradox mediator has the option of passing the entire *ReviewedBook*/3 request to the agent. A service that supports compound calls will have the attribute "join" in its "Query" set.

Predicate support refers to the level of expressiveness supported in unary predicates and binary predicates (if joins are also supported). Note that a binding pattern of "+" or

"?" indicates that unary value equality (for a given value) is supported for the given attribute. Also, binary value equality is assumed to be supported as part of join support. But there are two extensions to this that may be specified in metadata under the key name "Query". The first extension is support for comparison predicates, meaning the basic comparisons: {>,<,>=,<=,!=} over numbers and strings. The second extension is support for arbitrary, user-defined predicates. This means that the service supports the local execution of arbitrary, user-defined Java predicates. Note that to support this mechanism Paradox exploits the mobile code capabilities of the Java platform. Indeed, Java RMI provides this capability almost for free. A service that supports (just) comparison predicates will have the attribute "comp" in its "Query" set. A service that supports user-defined predicates is also assumed to support comparison predicates, and it will have the attribute "udp" in its "Query" set.

Monitoring and Notification

The Paradox system must be able monitor conditions on individual services in order to monitor complex distributed conditions over multiple services. But network services provide varying degrees of monitoring support. The Paradox approach to this problem has three elements: a model of primitive monitoring capabilities that can be mapped to a wide variety of services; metadata elements that allow the capabilities of individual services to be described in terms of this model; and a plan generator that can automatically compose the primitive capabilities available to provide the needed aggregate capability. Here we present the first two elements: the monitoring model and its specification as part of the PSM.

The mediator must do three things in monitoring an individual service: Describe the monitoring condition or event of interest; detect the occurrence of this event or condition; and access or compute the data associated with the event or condition. A service may provide varying degrees of support for each of these functions. And so our model of monitoring primitives breaks out along these three near-orthogonal dimensions (we say "near-orthogonal" because a small number of value combinations are not possible): The *monitoring language* dimension describes how a monitoring condition is

specified; the *external notification* dimension describes the degree to which external notification is provided; and the *information delivery* dimension describes how information associated with a condition is retrieved. Monitoring metadata consists of a triple of values, one for each dimension, under the key name "Monitor". We now expand on each of these dimensions.

Dimension 1: Monitoring language. This characteristic is very similar to the `Query` characteristic described earlier. The values "project", "join", "comp" and "udp" are all supported, and have the same meaning as in the context of the Query capability. In addition, we add support for what we call "parajoin" monitors. A parajoin monitor can be viewed as a special join in which one of the joining calls is a user-defined relation. Another way to view it is as a (very large) disjunctive condition. By convention, agents that support parajoin monitors also support a protocol that allows the monitor client to manipulate the user-defined relation in the monitor (e.g., insert into it, delete from it, etc.). The parajoin monitor becomes particularly important in the semantic optimization techniques we present in Chapter 9.

Dimension 2: External Notification. There are two components to the external notification capability: whether the agent supports proactive notification, and how the form of notification (if any) is interpreted. The external notification value is a pair.

The first element in the pair indicates whether a notification is actively pushed to a monitor client. This element has two possible values, "push" and "nopush". "Push" means the agent explicitly sends a notification to the monitor client. In Paradox, this notification is done via a uniform RMI interface (callback). "Nopush" indicates that the agent does not push notifications. The monitor client must be proactive about detecting a condition firing, in this case.

The second notification parameter indicates whether any form of change notification exists, and if so, how it should be interpreted. This parameter can have three possible values. Full condition notification, represented by the value "Full," means that no additional processing is needed to determine if the condition of interest was satisfied, i.e., the condition was (definitely) satisfied. Possible condition notification, represented by

the value "Possible," indicates that a change occurred that may have satisfied the specified condition, but additional processing may be needed to determine if the condition was satisfied. A source that can tell you only that "some change has occurred," e.g., a file with a "last modified" flag, would fit this category. The final value this parameter can have is no notification, given by the value "none."

Note that the pair, ("push", "none"), is the only external notification combination that cannot occur. Clearly, if an agent provides no notification support, there is nothing that can be pushed to the client. In contrast, ("Nopush", "Full") and ("Nopush", "Possible") are perfectly reasonable combinations. In these cases, full or possible notification requires an explicit "pull" from the client. The agent (wrapper) interface supports this pull via the standard call, `ConditionOccurred(MId, TS)`, where `MId` is a monitor identifier, and `TS` is a timestamp. We describe this process in detail in Chapter 4.

Dimension 3: Information Delivery. A service may be more or less helpful in designating what has changed. The final monitoring dimension describes the level of support provided by the agent in delivering the data associated with a condition or event occurrence. There are three possible values that can be offered in combination. The information delivery attribute consists of a set derived from the three possibilities, or a fourth, default, value.

Active information delivery, represented by the value "active," indicates that the data associated with a condition firing are actively pushed to the client together with the change notification. Note that this capability must be accompanied by a ("push", "full") value for external notification. (The converse is not the case, however.) In conjunction with ("push", "full"), this capability allows data delivery to piggyback on notification.

Event-specific pull, represented by the value "EPull", indicates that an event identifier is provided with an event notification that allows data associated with the event to be retrieved at a later time. An agent can provide the basis for this support in a variety of ways: By materializing timestamp-annotated changes to the full condition (defined as a view) or to the relevant base relations (Liu, Pu et al. 1996); by querying the log (Salem, Beyer et al. 2000); etc.

Timestamp-based pull, represented by the value "TPull," indicates that an agent supports querying via its monitoring language over changes since a given time. Like event-specific pull, data are retrieved by invoking a standard call. Unlike event-specific pull, however, the agent is not tracking a specific monitoring condition. The condition must be fully specified in the call. Techniques for supporting this capability at an agent are similar to the "EPull" case, but there is less potential for optimizing the process.

Finally, no information retrieval support, represented by the value "none," means that none of the three forms of support above are provided. The only method that a client can use to retrieve condition data, in this case, is to perform explicit "query and diff" operations using the agent's "Query" capabilities.

The source capability component of the PSM describes a large variety of realistic network-based services and sources. The plan generation, execution and monitoring modules of Paradox adapt automatically to sources that specify their capabilities using this model. We describe this process in detail in Chapter 4. Table 3-2 summarizes the source capability attributes of the PSM by key name and type:

KEYNAME	VALUE TYPE
"Modes"	Set of Binding Patterns
"Query"	Subset of {project, join, comp, udp}
"Monitor"	Monitor Language-External Notification-Information Retrieval Triple

Table 3-2 : Service Capability Metadata

3.2.3 Data Characteristics

Data characteristic metadata describes statistical and semantic properties of the data content of a service, or of events associated with a service. These properties can have a strong influence on the efficiency of executing and monitoring complex requests. The Paradox optimizer uses these properties to evaluate request execution and monitoring plans.

The first two characteristics provide information on the volume of data associated with a service. The expected size of a tuple is given under the key name "TupleSize". This property measures the average bytes per tuple for the service (tuples need not be of fixed size). The *cardinality* property, under the key name "card", is the size, in tuples, of the data associated with the service call. These two properties provide some of the information needed to estimate the amount of data transfer that will be involved in a given set of service calls.

The next three characteristics provide information on the attribute (or column) values associated with a service. The *unique values* property, under the key name "ColVals", measures the number of unique values in each column of the service call. For example, for the *Book*/3 call described earlier, a "card" value of [10000, 5000, 0] indicates 10000 unique book titles, 5000 unique authors, and an unbounded (unknown) number of prices. The *column range* property, under the key name "range", measures the range of values found in a given column, where the column type is numeric. For example, for *Book*/3, a "range" value of [0, 0, [5, 200]] indicates that there is no range binding on either *Title* or *Author*, but the *Price* of a book is between $5 and $200. Key values can be expressed in the PSM via the key name "KeyInfo". The "KeyInfo" value is a set of sets of integers. Each set defines the columns that make up a minimal key. For the *Book*/3 service, if *Title* and *Author* uniquely define a book, then the service might have a "KeyInfo" value of [[1,2]]. These three properties can be used to estimate the selectivity of predicates applied to data intensive services, which, again, are part of the puzzle in estimating the data transfer involved in a set of service calls.

The *stability* property is a property of possible change events associated with a service. This property appears under the key name "Stability", and it indicates whether a service change can occur. It can take one of two values: "fixed" means no change can occur to this service; "dynamic" means that a change can occur. If any event is associated with a service call, including data updates or modifications, the call must be "dynamic". A functional service, such as one that computes the distance between two addresses, would be "fixed", as would a static data source. Stability information is important to distributed condition monitoring, as we will see in Chapter 4.

A more advanced measure of the properties of information change is the *change type* property, under key name "ChangeType". The value of "ChangeType" is a list of triples that give a change identifier, the frequency of the change, and the cardinality of the change. The change identifier is a string that is unique for the given service. The frequency is an <Integer, TimeUnit> pair that specifies the number of times in the given time unit that this change event is expected to occur. Cardinality is the expected size, in tuples, of the data associated with the change type. For example, if a bookseller adds 10 new book titles (on average) to its inventory once a week, but then has an additional shipment once a month of 100 new titles, then its *Book* service might have a "ChangeType" value of {(ship1, (1, week), 10), (ship2, (1, month), 100)}. Change type information is vital for computing an optimal semantic cache to support condition monitoring. We describe its use for this purpose in Chapters 7 and 8.

Finally, integrity constraints may be provided, under the key name "IC". Integrity contraints are expressed as regular Horn clauses, just as a Paradox request is, except that often they are headless. The current implementation of Paradox does not make use of integrity constraints. In Chapter 9, however, we will discuss how a class of integrity constraints may be used to optimize complex condition monitoring. In particular, we discuss the use of a special kind of integrity constraint that constrains the way the information associated with a service can change over time.

Note that many of the metadata attributes in this category are similar to the statistical measures found in the catalogs of database management systems. They are used for a similar purpose as well. In fact, many of the measures employed by Paradox are cruder than those provided by a typical dbms. For example, where Paradox uses the "Range", "Card" and "KeyInfo" attributes to estimate the selectivity of predicates, many dbms's might use a more elaborate statistical structure such as a histogram, which can provide a basis more accurate estimates. But these estimates can be crude and error prone as well. Selectivity estimation, in particular, is known to be fraught with peril, even within the controlled setting of an individual dbms.

The simplicity of many of the metadata measures in Paradox is by design. More elaborate statistical information that supports better estimation, such as histograms, could also be provided in the Paradox architecture. But, in general, there is a trade-off between

the level of detail of such statistical data and the accuracy of the estimates they enable, and the burden imposed on an autonomous agent to maintain the data reliably. In a WAN environment, where integration is over autonomous service providers, we favor cruder but simpler methods. Cruder measures can be combined with adaptive, runtime measurements, and measures provided by a system administrator to yield reasonable estimation in support of optimization. As one example of evidence to support our view, adaptive selectivity estimation based on monitoring has been shown to be a good alternative to detailed histograms (Chen and Roussopoulos 1994). Approaches that use sampling are also promising (Lipton, Naughton et al. 1990). Another technique that we endorse, in combination with those provided by Paradox, is adaptive query processing techniques (Kabra and DeWitt 1998) (Ives, Florescu et al. 1999) (Levy and Lomet 2000). In these methods, plans are monitored to see that their costs are within a bound of what was expected during optimization. If they are not, the execution plan may be changed midstream. We believe that near-perfect statistical information is unattainable. It's better to adapt to inaccuracies than to pretend they don't exist.

Table 3-3 summarizes the data characteristic attributes of the PSM by key name and type:

KEYNAME	VALUE TYPE
"TupleSize"	Size in bytes
"Card"	Number of tuples
"ColVals"	List of unique column values
"Range"	List of value ranges per numeric column
"KeyInfo"	List of lists of column numbers that make up keys
"Stability"	In {"fixed", "dynamic"}
"ChangeType"	List of TypeId-Frequency-Cardinality triples
"IC"	List of integrity contraints defined by (headless) Horn clauses

Table 3-3 : Data Characteristic Metadata

3.2.4 Function and Predicate Characteristics

Services are not always data-intensive. Some services provide complex functions or predicates, and they can be compute-intensive. This category of service includes services that support user-defined predicates that are registered with the mediator and that can be expensive to execute. The costs of executing such functions or predicates should be considered in the overall cost of a complex request. There are two properties in the PSM that cover this requirement.

The first property in this category measures the cost of executing a service call. The "CPU" property measures the expected computational time, in seconds, associated with a single service call. This property should be provided for expensive functions and predicates.

The other property in this category applies to predicates. "Selectivity" is a real number between 0 and 1 that measures the fraction of instances that the predicate is expected to return `true`. In general, it is difficult to provide this measure for a given predicate in a vacuum. So if a service provides this measure directly, it is likely to be used as a default that is best adjusted or overridden as the predicate's behavior is observed. The monitor metadata source may supplement the default information with observed values in which the predicate is run against specific data streams.

Table 3-4 summarizes the function and predicate attributes of the PSM by key name and type:

KEYNAME	VALUE TYPE
"CPU"	Real (in seconds)
"Selectivity"	Real (between 0 and 1)

Table 3-4 : Function and Predicate Metadata

3.2.5 System Characteristics

The final category of metadata describes system conditions that are relevant to evaluating the cost of execution plans. Currently, Paradox recognizes two such properties, which are specifically relevant to network interactions between the mediator and remote agents. These are "Bandwidth" and "Latency." Bandwidth is measured in bits/second, and is meant to measure the effective bandwidth achieved via our RMI-based protocol between the mediator and the service provider. Latency measures the delay associated with an initialization of an RMI connection during an `Open()` call. In general, the cost of a remote service call will be the sum of the associated latency, the amount of data transferred multiplied by the bandwidth, and the number of discrete service calls multiplied by the time per call. Note that latency and bandwidth do not apply to an agent in a vacuum, and so we do not expect them to be provided by an individual agent via a directory service. Instead, we expect either the monitor or the administrator metadata source to provide these properties.

Table 3-5 summarizes the system property attributes of the PSM by key name and type:

KEYNAME	VALUE TYPE
"Bandwidth"	Integer (bits/second)
"Latency"	Real (seconds/message)

Table 3-5 : System Property Metadata

3.3 Chapter Summary

Metadata is integral to everything the Paradox system does. In this chapter we have described the architectural mechanisms in Paradox for providing and gathering metadata, and the Paradox Specification for Metadata, which is the language used to describe service and system characteristics. The chapters that follow will describe how this

information can be exploited to provide active integration of a variety of distributed services in a scalable and efficient manner.

Chapter 4

Plan Generation and Execution in Paradox

Paradox provides coordinated, "active" access to groups of data-intensive distributed services. Service providers register the characteristics of their offerings in a group of distributed directory services. Paradox accepts requests in the form of conjunctive queries with comparison predicate (Ullman 1988) (equivalent to select-project-join queries) over the universe of all directory-registered service offerings. A request is interpreted as a long-lived query that returns all of the current information satisfying the request, and that continuously updates the state of the request in response to changes in component services.

In the previous chapter we described the mechanisms for incorporating metadata into the Paradox system and the content of that metadata. In this chapter we continue to flesh out the architecture outlined in Chapter 2 by describing how requests are converted to

groups of programs against the Paradox execution engine, and the operations that are supported by that engine. Figure 4-1 replicates our earlier picture of the general Paradox architecture (Figure 2-1), highlighting the components that we emphasize in this chapter in **bold**:

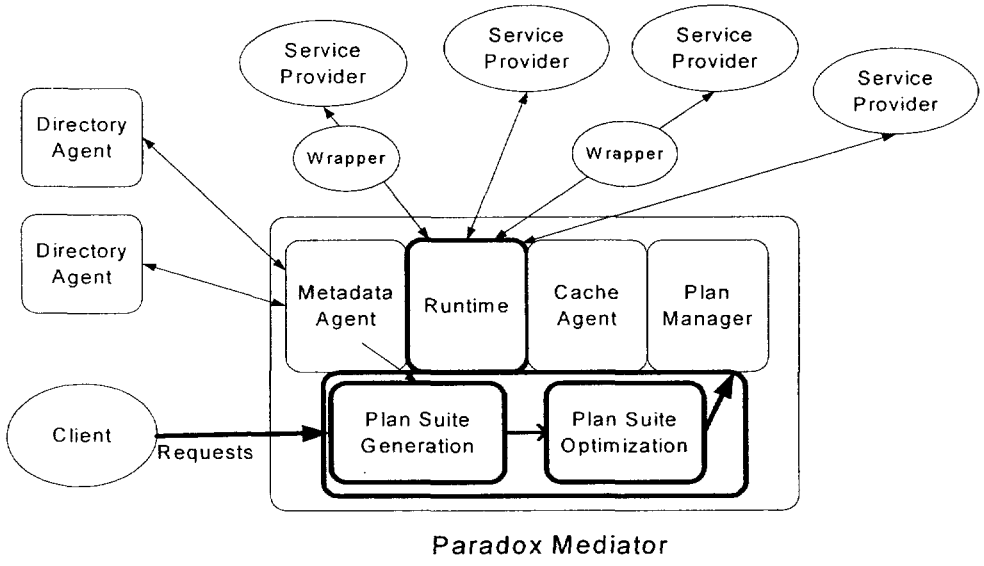

Figure 4-1: Request Processing Components of Paradox

A request to the Paradox system is parsed to a syntax tree which is passed to the plan generation and optimization component of the system. The job of this component is to produce an optimal *physical plan suite*. A physical plan suite consists of a program that runs on the Paradox execution engine for the initial computation of the request result, and additional physical plans for computing changes to the requested condition over time in response to change events occurring at any component service.

The process of plan generation and execution in Paradox has strong similarities to that of query processing in relational database systems. Key differences enable support for request monitoring, for handling network-based services and data sources, and for coping with the heterogeneous capabilities of these sources in an efficient manner. Figure 4-2 shows a more detailed break-down of the process. A request is parsed into a syntax tree, which is mapped to an initial logical plan expressed as a tree of operators

from the system's logical algebra. This algebraic expression is then used as a basis for generating a logical plan suite for the request. Each plan in the suite then goes through an optimization process that includes a series of logical transformations, including the specification of a join ordering for the plan, and then a mapping to a physical plan, expressed in the physical algebra of the system, that can be run against the Paradox execution engine. The capabilities of the component service providers guide plan transformations along the way, and only feasible plans are generated. We will describe the major steps shown in Figure 4-2 in detail, and we will also describe the workings of the Paradox execution engine.

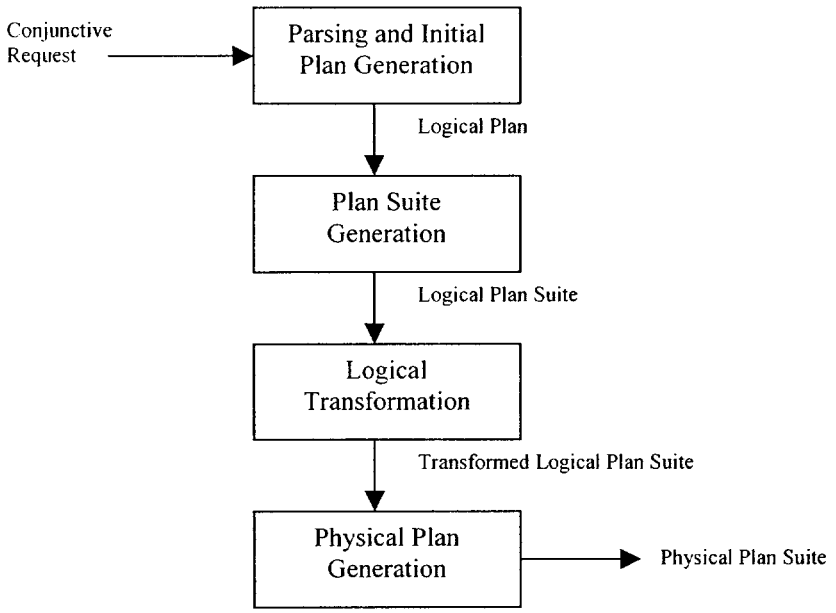

Figure 4-2: Plan Generation in Paradox

4.1 A Simple Example

The presentation in this chapter is operational in nature and driven by a simple example. Consider a computer reseller who deals in laptop computers. He makes use of

several network-accessible services to buy and sell products at a profit. One such service provides information on laptop computers. Specifically, this service provides consumer ratings between 1 (best) and 5 (worst), and other descriptive information keyed on the make and model of the product via the service call, *Laptop*/4:

$$Laptop(Make, Model, Rating, Description)$$

The *Laptop* service call provides no monitoring capability and it changes relatively infrequently.

A second service provides access to computer classified ads from newspapers and magazines throughout the country in the form of a *Classified*/5 call:

$$Classified(CId, Make, Model, Seller, Price)$$

This call lists a unique classified ad identifier, the make and model of the product being offered for sale, an identifier for the seller of the item, and its price. The *Classified* service call provides an update notification capability that will notify customers of the listing of new items based on a unary predicate applied to the update. The update itself can be retrieved in a separate operation.

Finally, an auction service provides information on items available for bid, and the status of these bids, via the calls Item/5 and CurrentBid/2:

$$Item(ItemNo, Make, Model, Begins, Ends)$$
$$CurrentBid(ItemNo, Amount)$$

The auction service supports declarative joins across all of its service calls. The *CurrentBid* call requires that the *ItemNo* attribute be bound. A notification service will monitor a condition over the service calls and indicate when updates meeting the condition occur, and can optionally include the updates themselves together with the notification.

We assume in this example that the make and model attributes of all three services are drawn from a common domain.

Our reseller wants to find laptop buying opportunities in the classified ads on items that can be sold at auction for a profit. When a laptop computer that is rated higher than a '3' by the consumer rating service is offered by the auction service, the same make and model is offered in the classifieds, and the price of the item in the classifieds is lower than the current bid for the item at auction, the reseller wants to be notified so that he can consider buying the item from the classifieds and then reselling it via the auction site. This integrated service can be expressed as the `BuyingOpp/4` request (4-1) below:

$$
\begin{aligned}
&\textit{BuyingOpp(CId, ItemNo, Make, Model)} \leftarrow \\
&\qquad \textit{LapTop(Make, Model, Rating, Desc)} \,\&\, \\
&\qquad \textit{Rating} >= 3 \,\&\, \\
&\qquad \textit{Classified(CId, Make, Model, Seller, Price)} \,\&\, \\
&\qquad \textit{Item(ItemNo, Make, Model, Begin, End)} \,\&\, \\
&\qquad \textit{CurrentBid(ItemNo, Bid)} \,\&\, \\
&\qquad \textit{Bid} > \textit{Price}.
\end{aligned}
\tag{4-1}
$$

Table 4-1 shows some of the basic capability-related metadata by key name for the service calls used in this request. We will introduce other metadata objects for these service calls as needed.

Notice, in particular, that all service calls in our example request are dynamic. `Item/5` and `CurrentBid/2` are supported by the auction service within a single service group that supports declarative, composite calls (joins), but change event monitoring over joined conditions is not supported by the service. Both individual calls in the auction service, however, support both active pushing of change data, and active notification followed by event-based retrieval of change data (epull). The `Classified/5` call in the classified ad service does not support active pushing of change data, but it does support event-based retrieval. The `LapTop/4` call provides a very limited service, with no explicit event notification and only a limited querying

	LapTop/4	Classified/5	Item/5	CurrentBid/2
"Group"	G1	G2	G3	G3
"CCF"	C	C	C	C
"Mode"	[(?,?,?,?)]	[(?,+,+,?,?), (+,?,?,?,?)]	[(?,?,?,?,?)]	[(+,?)]
"Stability"	dynamic	dynamic	dynamic	dynamic
"KeyInfo"	[[1,2]]	[[1]]	[[1]]	[[1]]
"Query"	{}	{proj,comp}	{proj,comp,join}	{proj,comp,join}
"Monitor"	({}, (nopush,possible), {})	({comp}, {epull})	({proj, comp}, (push,full), {active,epull})	({proj, comp}, (push,full), {active,epull})

Table 4-1: Service Call Metadata for BuyingOpp/5 Request

capability. Variable binding constraints exist for the `Classified/5` and `CurrentBid/2` calls. Finally, notice that keys are defined for each service call, and the `BuyingOpp/4` request subsumes the key values for the service calls in its body. Details on the meaning of this metadata were presented in Chapter 3.

4.2 Plan Suite Generation

The first major step in turning a Paradox request into an optimal physical plan suite is to generate a *logical plan suite* (also called a *query suite*). A logical plan suite consists of a logical plan for the initial computation of the request result, and one or more additional logical plans for computing changes to the requested condition over time in response to change events occurring at any component service. A request is first parsed and transformed to an initial logical plan. Relevant service metadata is retrieved and the logical plan is constrained based on service capabilities. The resultant logical plan is then transformed into a series of *differential requests*, one for each dynamic service call.

As with many relational query optimizers, Paradox employs a logical algebra to represent logical execution plans (Graefe 1993). The basic logical algebra used in

Paradox contains five operators: `LSelect`, `LProject`, `LJoin`, `LJoinD`, and `LGet`. `LSelect` corresponds to relational selection, `LProject` corresponds to relational projection, `LJoin` represents a general, n-ary relational join operation, `LJoinD` is a binary dependent join operator, in which the binary join predicates must be converted into unary predicates by applying variable bindings from the left side of the join before passing the result to the right side, and `LGet` represents a leaf operation that makes a set-oriented call to a remote service. These operators are sufficient for handling the basic conjunctive requests that Paradox accepts. In addition, however, Paradox must monitor and compute changes to a request. To support this capability at the logical level, we add the two related logical operators, `LAlert` and `LAlertNGet`. `LAlert` captures the notion of detecting a change event in a dynamic service call, alerting the mediator, and providing some method for retrieving the associated data. `LAlertNGet` is a restriction on `LAlert` that requires event detection and data retrieval to be combined in a single operation. As we will see, `LAlert` and `LAlertNGet` are always accompanied by a `LGet` that is tied to the data associated with the event reported by the alert. Some sources do not support the separation of notification of a change event from the retrieval of data associated with the event. Under these circumstances, `LAlertNGet` must be chosen. On the other hand, where such separation is supported, a more efficient implementation may be possible. A cost-based choice between alternative implementations of monitoring, where a choice is available, is made in mapping logical to physical plans.

The original logical plan tree is generated by separating each individual service call in the request and all related selection predicates into leaf `LGet` nodes, and inserting a single n-ary `LJoin` at the root, as the parent of the `LGet` operators. Relevant service references and metadata are retrieved for each component request by issuing a request to the Metadata Agent. In the current implementation, we assume there is only one service provider associated with each service call. If we find more than one, we pick the first one returned. If there are no services for a service call in the request, we return failure[1]. This

[1] There are a number of important issues associated with multiple providers of a single service call, in handling unions, etc., that any deployed system must cope with (see the work of Florescu, et al., for example (Florescu, et al. 1997), but these are not the focus of our work.

behavior is roughly what one would expect if the CCF value for every service call were equal to C (Complete). At this time, the Paradox implementation does not make full use of the range of possible CCF values.

We then traverse the set of LGet nodes and merge nodes that meet the following three criteria:

1) They contain calls against a common service provider.
2) The service provider supports joins over the service calls.
3) There is a common variable or join predicate between the calls.

This merging heuristic assumes that pushing as much as possible into a single remote call is desirable, unless such a combination results in a Cartesian product, in which case remote calls may be better made separately (in which case either the Cartesian product will be computed locally, or it can be avoided altogether). In the process of merging LGet nodes, we also identify variables in the call that either appear in another leaf node or appear in the head of the request. All other variables may be projected away. A similar process (without the merging) is applied to the parent LJoin node.

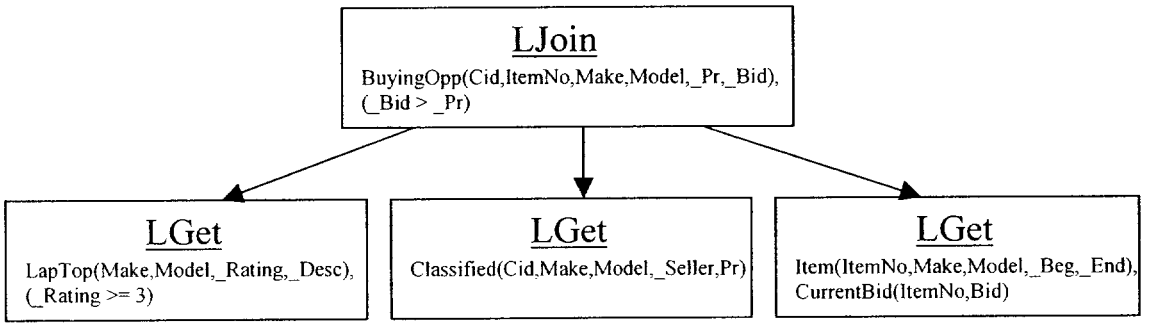

Figure 4-3: Logical Plan Tree

The logical plan tree produced by this process for our example is depicted in Figure 4-3. Variables that are not needed above the current node, and thus may be projected away, are prefixed with an underscore.

After the basic logical plan tree has been derived, we generate logical differential plans for computing the change in the overall request induced by individual service change events. Differential plans are generated via a series of capability-driven transformations of the initial logical plan tree. The basic process is to traverse the set of `LGet` nodes, and, for each node that contains at least one dynamic call, create a new, differential plan associated with that node by transforming the original `LGet` into one of the alert operators (either `LAlert` or `LAlertNGet`) followed by a modified `LGet`. We call the algorithm `SuiteGen`. The complete algorithm is shown in Figure 4-4.

A couple of notes of explanation are needed for the `SuiteGen` algorithm. The condition in line (2), that the leaf node call is dynamic, is true if any individual call within the leaf node is dynamic. Line (5) clones the nodes up the tree from the target leaf node. In this case the only cloned nodes are the target `LGet` node and its parent `LJoin` node. All of the cloned nodes will be modified in the process of transforming the original plan tree into a differential plan, but none of the non-target leaf nodes need to be modified or cloned. We are able to share the non-target nodes among numerous plans in this manner. By doing so, and avoiding cloning the entire tree, we achieve a compact representation for the plan suite as a whole.

Line (6) in Figure 4-4 handles the case where a service supports the joining of multiple service calls within a single leaf node, but it does not support an alert or differential capability over the same composite call. This capability matches that of many modern-day relational database management systems, for example. In this case, the calls contained in the leaf node must be separated into sets in which alerts and differential computation are supported (line (7)). The resulting sets can then be processed one by one in the normal manner, via a recursive call to `SuiteGen()` (line (8)).

Beginning with line (11), we iterate over each change event type associated with the call within the target `LGet` node and generate the incremental plan that corresponds to that event type. Lines (12)-(15) perform the core task in this process, which is a capability-driven transformation. If the source for the `LGet` target supports separate alerting, the target node is split into an `LAlert` and `LGet` pair that operates on the change event and its associated data. Note that the `LAlert` node represents the

```
   SuiteGen(LogicalPlanTree T, LeafNodeSet LNodes) {
(0)  Suite = {};
(1)   for each L in Lnodes
(2)     if L.Call() is dynamic
(3)        Suite += DiffTransform(L, T);
(4)        return Suite;
   }

   DiffTransform(LGet L, LogicalPlanTree T) {
     ReturnSet = {};
(5)  T'= Clone nodes of T on path from L up to T.root();
(6)   if (mergedNode(L) && mergedDeltaNotSupported(L)) {
(7)     LeafNodeSets NSets = breakOutSingletonCalls(L);
(8)     for each NSet in NSets
(9)       ReturnSet += SuiteGen(T', NSet);
      }
(10)  else {// singleton node or mergedDeltaSupported
(11)    for each change event type, E, in L {
(12)      if SeparateAlertSupported(L)
(13)        L -> LAlert(Delta(L, E, I)) + LGet(Delta(L, I));
(14)      else
(15)        L -> LAlertNGet(Delta(L, E, I)) + LGet(Delta(L, I));
(16)      Adjust Ancestor Nodes;
(17)      ReturnSet += {T'};
        }
      }
   }
```

Figure 4-4: Algorithm for Generating a Plan Suite

possibility of separating the monitoring of a change event from retrieving the data associated with the event, but it does not represent a physical commitment to do so. If the source for the `LGet` target does not support separate alerting, the `LGet` is transformed into an `LAlertNGet` and `LGet` pair. The `LAlertNGet` represents a physical commitment to perform alerting and the retrieval of associated data in a single step. Just as the `LJoinD` operator represents a restricted form of the `LJoin` operator, the `LAlertNGet` operator represents a restriction of the `LAlert` operator to a subset of its potential physical implementations. Any conditions (i.e., selections or projections) over the service call are passed to the alert operator at this stage. If the alert operator is an `LAlert`, however, some conditions may have to be handled elsewhere in the plan. We cover this issue in detail when we discuss the creation of physical plans.

As a notational convention, we prepend a delta, Δ, to the original call name when referring to a change event associated with the call. An `LAlert` operator monitors the change event of a given type, and returns a variable that is either an identifier used to retrieve the associated data in a separate process, or a local iterator that holds the associated data. We express this functionality syntactically with the delta form of the call and arguments for the event type, `E`, and the variable, `I`, within the `LAlert`. Additional conditions are specified separately. An `LAlertNGet` operator, in contrast, necessarily passes a local iterator to its corresponding `LGet`. Either way, the resulting data flows through an `LGet` operator. This data takes the form of a multiset of tuples, and each tuple has an associated multiplicity count. A positive multiplicity count value indicates one or more added tuples, a negative value indicates one or more deleted tuples. One benefit of using multiplicity counts is that it allows additions and deletions to be handled in a uniform manner. We express this functionality with the delta form of the call and all of its normal attributes, plus two additional arguments, one for an identifier or iterator, and one for the multiplicity count. The transformation of line (12) as applied to the *`Classified/4`* call is illustrated in Figure 4-5. This figure shows the original `LGet`, with a generic parent node, and its transformed form. Note that there is no additional condition to pass to the `LAlert` in this example.

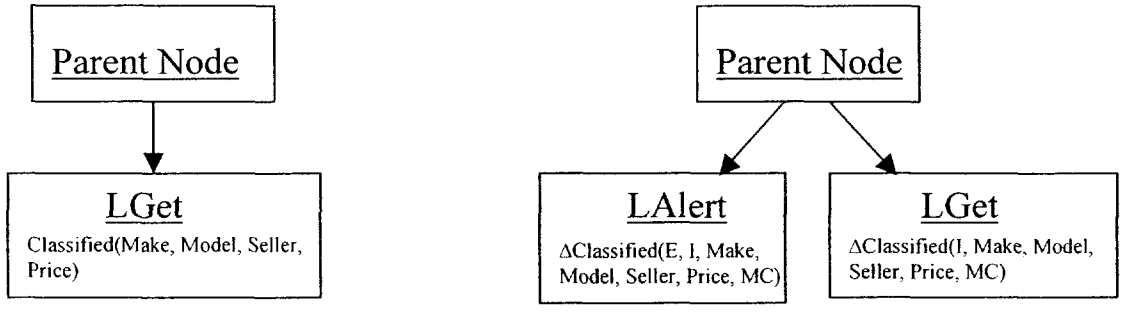

Figure 4-5: Transforming the Target LGet

Note, further, that if the original call is a compound call, a new Horn clause is formed with the compound call as the body. The head of this derived clause becomes the target of the transformation we have just described.

Finally, in lines (16)-(17), the ancestor nodes of the `LGet` in delta form, or of the `LAlertNGet`, are adjusted to reflect passing the multiplicity count argument up the tree, and the new logical plan tree is returned.

As noted earlier, we could perform plan suite generation at the logical plan level or at the query level. By performing it at the logical plan level we avoid repeated parsing and plan generation. But as a notational convenience we will often refer to an incremental plan by its logical query form. For example, Request (4-2) shows the differential query for *BuyingOpp* induced by changes to *LapTop*.

$$\begin{aligned}\Delta BuyingOpp(CId, ItemNo, Make, Model, MC) \leftarrow\\ &\Delta LapTop(Make, Model, Rating, Desc, MC)\ \&\\ &Rating >= 3\ \&\\ &Classified(CId, Make, Model, Seller, Price)\ \&\\ &Item(ItemNo, Make, Model, Begin, End)\ \&\\ &CurrentBid(ItemNo, Bid)\ \&\\ &Bid > Price.\end{aligned} \tag{4-2}$$

Note that in a query form, we stick with the convention of specifying the data associated with a change event by prepending a Δ to the corresponding call. We also include the multiplicity count argument, `MC`, since it percolates up to the head of the query, and we will occasionally need an event type argument as well, when multiple event types are supported. But there is no need, in this notation, to be explicit about the separation of event monitoring from the retrieval of the data associated with the event, as there is with a complete logical plan.

As a further shorthand, we will sometimes omit the variable names and predicates of a clause, when they are clear from context. For example, we can express Request (4-2) as:

$$\Delta BuyingOpp \leftarrow \Delta LapTop \,\&\, Classified \,\&\, Item \,\&\, CurrentBid.$$

We will also refer to this request as $\Delta BuyingOpp/\Delta Laptop$, read as "delta BuyingOpp with respect to delta Laptop." Figure 4-6 shows the complete, equivalent logical plan for $\Delta BuyingOpp/\Delta Laptop$. Note that the *Laptop* call does not support the separation of alerting from data retrieval, so an `LAlertNGet` is required for this plan. Figure 4-7 shows the logical plan for $\Delta BuyingOpp/\Delta Classified$. Note that the auction service does not support a monitoring capability over compound (join) calls, so $\Delta BuyingOpp/\Delta Item$ and $\Delta BuyingOpp/\Delta CurrentBid$ must be handled with two separate plans. These plans are shown in Figures 4-8 and 4-9, respectively. The complete query suite, in abbreviated form, is shown as Requests (4-3).

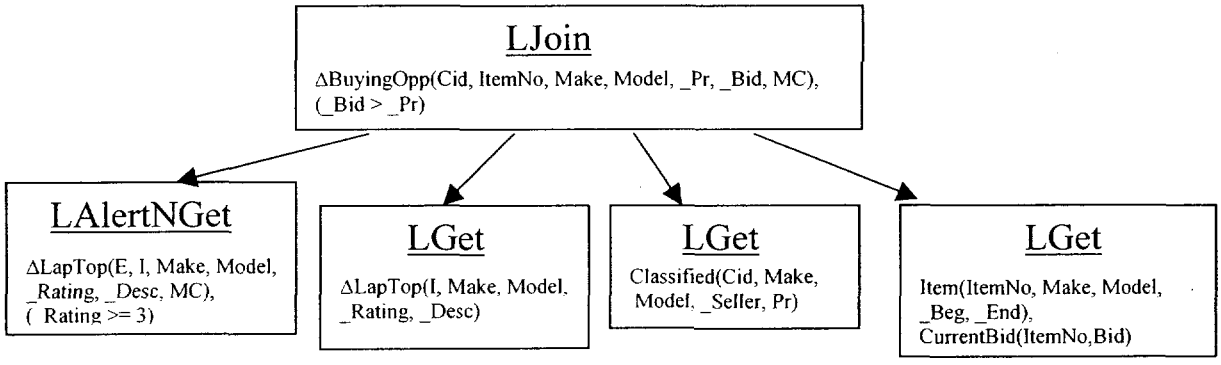

Figure 4-6: Logical Plan for ΔBuyingOpp/ΔLapTop

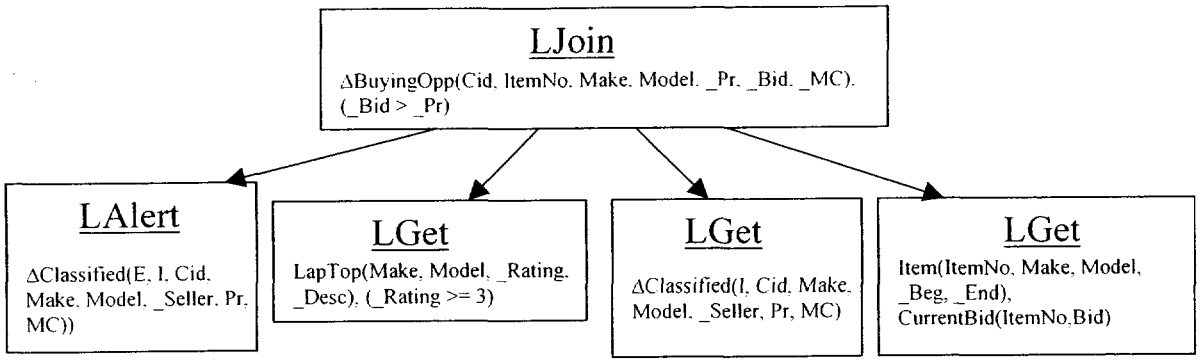

Figure 4-7: Logical Plan for ΔBuyingOpp/ΔClassified

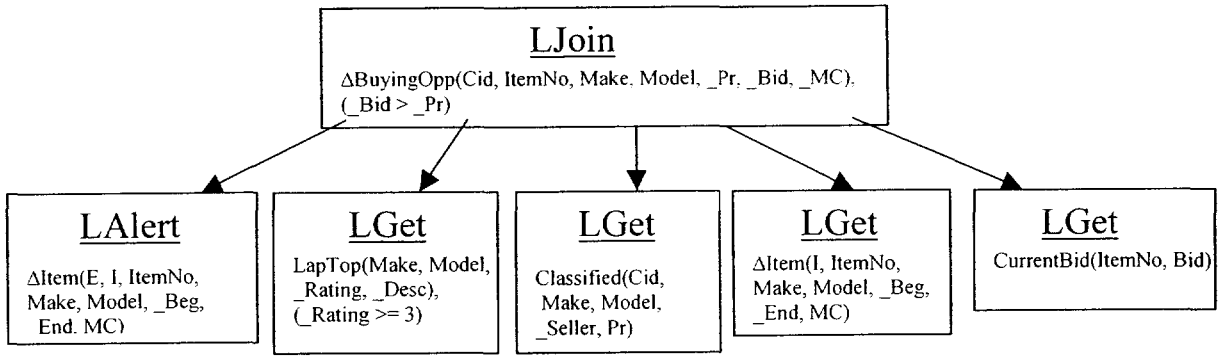

Figure 4-8: Logical Plan for ΔBuyingOpp/ΔItem

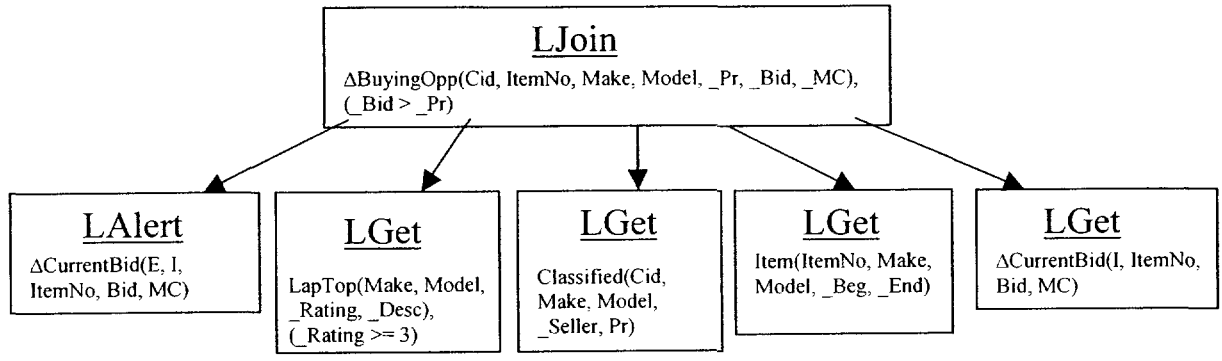

Figure 4-9: Logical Plan for ΔBuyingOpp/ΔCurrentBid

$$\begin{aligned}
&\Delta BuyingOpp \leftarrow \Delta LapTop \,\&\, Classified \,\&\, Item \,\&\, CurrentBid.\\
&\Delta BuyingOpp \leftarrow LapTop \,\&\, \Delta Classified \,\&\, Item \,\&\, CurrentBid.\\
&\Delta BuyingOpp \leftarrow LapTop \,\&\, Classified \,\&\, \Delta Item \,\&\, CurrentBid.\\
&\Delta BuyingOpp \leftarrow LapTop \,\&\, Classified \,\&\, Item \,\&\, \Delta CurrentBid.
\end{aligned} \tag{4-3}$$

Generating differential plans for a Paradox request is similar to the process of generating plans for the incremental maintenance of materialized views in the setting of a non-distributed database management system. Extensive research has been done in this area (Gupta and Mumick 1995). In particular, our use of multiplicity counts is similar to that presented by Gupta, et al. (Gupta, Mumick et al. 1993). Our plans consider change events and related data at only one source at a time. If change events at multiple sources occur simultaneously, the mediator serializes them, handling them one at a time in the order that the mediator is notified about them. Due to network delays, the order of change event notification may not be the true chronological order of the events. Furthermore, information changes may be batched into a single change event at a service provider (or by a wrapper implementing a monitoring capability). Consequently, change events at different sources may be the result of multiple global information states that are interleaved in time, and these states may not be recoverable by the mediator. The end result is that the Paradox system cannot guarantee that it is tracking the global state of a request perfectly over time. It will, however, converge to a consistent and correct request state. This sort of imperfection in monitoring requests over multiple services is simply a fact of life in distributed systems when the service providers are autonomous. It is impractical to expect autonomous agents to participate in a two-phase commit with an untrusted source, or in some other heavy-weight protocol necessary to ensure transactional semantics, particularly in a WAN environment (Fox, Gribble et al. 1997). Fortunately, however, a large class of useful applications can tolerate a degree of temporary inconsistency.

A related issue that must be handled in an ASIS is that of avoiding or compensating for update anomalies. An update anomaly occurs when updates and incremental request processing are interleaved in such a way that we compute an incorrect result. Zhuge, et al. show that update anomalies can be handled by requiring

that the attributes in a request subsume a primary key for each component service call (Zhuge, Garcia-Molina et al. 1995; Zhuge, Garcia-Molina et al. 1998); we adopt this approach to handling the issue. Where this restriction is too limiting, an alternative is to use timestamps, if they are provided by the relevant services, along with bounds on "clock drift" between distributed sources (Liu, Pu et al. 1996). In either approach, however, applications must be able to cope with the potential of temporary inconsistencies.

4.3 Computing a Feasible Join Ordering

Once the suite of logical plans has been generated from a request, each plan is optimized via a two-phase process of logical and physical transformation. The logical transformation phase selects a join ordering in a capability-driven, heuristic manner. The physical transformation phase then performs a cost-based selection of physical algorithms followed by heuristic plan manipulation to produce the final plan. Here we describe how a join ordering is obtained.

In the last section we saw that plans in a logical plan suite are always rooted in a single *n*-ary join operator. As part of the join-ordering process, Paradox changes the *n*-ary join at the root into a series of binary `LJoin` and (where source capabilities require it) `LJoinD` (dependent join) nodes. Note that if an alert node (either an `LAlert` or an `LAlertNGet`) is part of the plan, it is not considered at all at this stage. It is simply added as a third, specially-handled child operator, to the first join of the chosen join ordering. Paradox considers only left-deep plan trees and it generates only feasible plans. A left-deep tree is one in which the right child of every join node is a leaf. A feasible plan is one in which all binding constraints are satisfied. When the right hand side of a join requires a binding from the left hand side in order for binding constraints to be met, the join must be dependent. Our algorithm also attempts to delay Cartesian products as long as possible. It should be noted, however, that in the presence of limited binding patterns, the left deep limitation may make Cartesian products necessary where they could be avoided in an equivalent bushy tree (Florescu, Levy et al. 1999).

Our algorithm proceeds in a bottom-up, greedy manner, first building a 2-way join, then a 3-way join, etc., until we have completed the join ordering. At each step, we take the feasible join ordering that minimizes the estimated cardinality of the intermediate join result. We derive these cardinalities based on data characteristics described in the metadata associated with service calls in our plan. At step one, we order the GET nodes that can feasibly start the join from smallest to largest estimated size, and this ordering becomes our preferred ordering of "1-way joins". At step N, we start with the preferred (N-1)-way join, and we estimate the size and test the feasibility of the result of adding each remaining service call to this preferred join. We order each resulting N-way join by estimated cardinality, take the smallest as our preferred N-way join, and continue the process. At each step, we add a `LJoin` node if possible. If acceptable binding patterns of the right child of the join demand a dependent join, however, we add a `LJoinD` node instead[2]. If we are unable to find a feasible N-way join in this manner, we backtrack to our next-choice (N-1)-way join, and proceed with the search for a feasible N-way join again. We are done when a complete, feasible join ordering is found, or when we have completed an exhaustive search without finding any feasible ordering.

Returning to our example, consider the plan for ΔBuyingOpp/ΔClassified, shown in Figure 4-5, and consider additional metadata pertaining to data characteristics shown in Table 4-2. Underscores in this table indicate that a value is either not applicable or not provided. Blank values indicate that the item is not relevant to this example.

	LapTop/4	Classified/5	Item/5	CurrentBid/2
"Card"	500	100,000	50,000	50,000
"ColVals"	[100,500,5,_]	[_,500,5000,_,_]	[_,200,1800,_,_]	[_,_]
"Range"	[_,_,[1-5],_]	[_,_,_,_,_]	[_,_,_,_,_]	[_,_]
"ChangeType"		{1-1:Hr-1000}		

Table 4-2: Data Characteristic Metadata for BuyingOpp Service Calls

[2] `LJoinD` is really a "quasi-logical" operator. It restricts the set of algorithms that can be chosen to implement it to the set of physical dependent joins. Any physical join algorithm, dependent or non-dependent, can be chosen to implement a `LJoin`.

In this example, the `LGet` node containing the *LapTop* service call is expected to have the lowest cardinality. Based on the selection predicate applied to *LapTop*, *Rating >= 3*, and the "Range" metadata that specifies the range of values for the third attribute of `LapTop` as between 1 and 5 (inclusive), the Paradox system estimates (under the assumption of a uniform value distribution) that the predicate will have selectivity of 0.60. The estimated cardinality of the *LapTop* leaf node, therefore, is 300. The estimated cardinality of Δ*Classified* is significantly larger, at 1000. Since binding pattern constraints exist for change events over the *Classified* call, however, Δ*Classified* is not a feasible starting point in any case. In fact, the only other feasible starting point is the compound call of (*Item/5*, *CurrentBid/2*), which has an expected cardinality of 50,000. Therefore, *Laptop/4* is chosen as first in the join ordering.

Moving to the selection of a two-way join, we will choose to join *LapTop* with Δ*Classified*. The join predicate in this case is a conjunction of two equalities on the Make and Model attributes. In computing the selectivity of a conjunctive join predicate, Paradox considers only the more restrictive conjunct (i.e., correlations are ignored). In this case, the Model attribute is chosen. Based on the ratio of column values for this attribute, we estimate join selectivity of 500/5000, or 0.1. Based on a cardinality of Δ*Classified* of 1000, we then estimate the cardinality of the join at $0.1 \cdot 300 \cdot (1000/5000) = 30$. The Model attribute is also chosen in estimating the cardinality of a two-way join between *LapTop* and the auction service. But in this case the larger cardinality of the *Item* call results in a larger join size estimate. Note, further, that the join of *LapTop* with Δ*Classified* is feasible, since if bindings of the *Make* and *Model* attribute are made from *LapTop* to Δ*Classified*, then all binding constraints are satisfied. This requirement also means that the join must be dependent, so the `LJoin` node is turned into a `LJoinD` node. Finally, the 3-way join is completed by joining against the compound call of the Auction service. The resulting logical plan tree is shown if Figure 4-10.

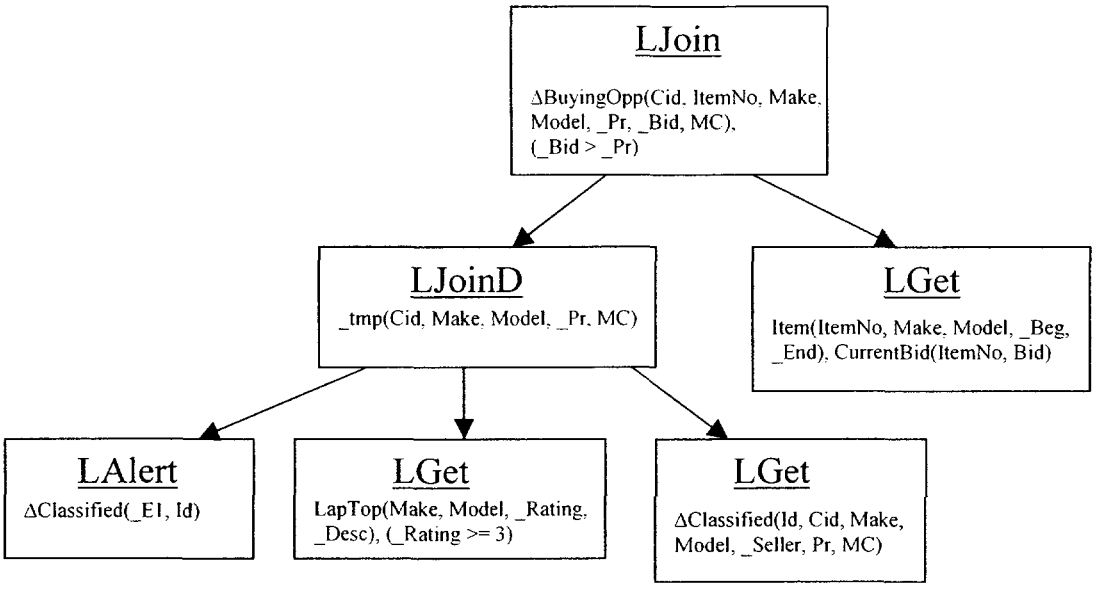

Figure 4-10: Logical Plan for ΔBuyingOpp/ΔClassified after Join Ordering

4.4 Logical to Physical Plan Mapping

After all logical transformations have been completed and a join ordering has been determined, we must map the resulting logical plan to a program (or physical plan) over the Paradox execution engine. In this section we first describe the physical operators provided by the execution engine, and then we describe the process of mapping a logical plan to a physical one.

4.4.1 The Paradox Execution Engine

The Paradox execution engine provides a variety of algorithms and operations that support the integration of information from multiple network-based services. The engine includes a number of basic algorithms that you would expect to find in any relational database engine, as well as some operations specifically oriented towards

network-based data integration. Each physical operation conforms to the *iterator* model (Graefe 1993), and takes either one or two input arguments. Each input argument is either an iterator or a service call. The current implementation is main-memory only. A main memory executor is sufficient for our research purposes, but any enterprise-strength system would require out of core operation. We enumerate the physical operators with brief descriptions:

1) **PGet**: The `PGet` operation invokes a call to a remote service, wrapping the call in an iterator. `PGet` allocates a local buffer and pre-fetches data into that buffer in an attempt to lessen the effects of network latency and burstiness on data flow during plan execution.
2) **Sel_Proj**: `Sel_Proj` combines the standard relational selection and projection operations into a single operator. It takes a unary predicate and an ordered list of columns as arguments. It applies the predicate to an input tuple stream and produces an output stream of tuples that satisfy the predicate and that contain only the specified columns in the specified order. Pure relational selection and projection are special cases of this operator.
3) **Local_NLJ**: `Local_NLJ` is a local nested-loops join. It takes two input tuple streams and a binary join predicate. It materializes its inner (right) input. Then, in the general case, for each tuple of the outer (left) input stream, the materialized inner set is scanned completely. A special form of this operator exploits the sorting order of the inner input, avoiding a complete scan for every outer tuple. This optimization is particularly important when the inner input is large. All pairs of tuples that satisfy the join predicate are combined and put to the output stream. This operator can optionally accept a unary selection predicate and a projection list as well, allowing selection and projection operations to be merged into the join.
4) **Remote_NLJ**: `Remote_NLJ` is a remote nested-loops join operator. `Remote_NLJ` is similar to a local nested-loops join, except that it does not materialize its inner input and evaluate the join condition locally. Rather, the join condition is partially evaluated to a `PGet` and (possibly) a `Sel_Proj`

for each outer tuple. Thus this operation requires a remote invocation for each tuple in the outer input. If the inner input is a high-latency service call, this operation can be very expensive. As with `Local_NLJ`, `Remote_NLJ` can optionally accept a unary selection predicate and a projection list, allowing selection and projection operations to be merged into the join.

5) **`Merge_Join`**: `Merge_Join` is a standard relation sort-merge join operation (Ullman 1988). It takes two iterator inputs and a binary equi-join predicate (i.e., the predicate must include at least one equality constraint between attributes of both input iterators). The inputs to `Merge_Join` must be sorted on the attributes in the equality constraint. As with the other join operations, `Merge_Join` can also handle unary selection predicates and arbitrary projection lists, to allow combination with projection and selection operations.
6) **`Sort`**: The `Sort` algorithm takes a single iterator argument and one or more attributes and produces the result of sorting its input stream on the given attributes. `Sort` materializes its input, performs a standard Quicksort on the materialized set, and outputs the result as an iterator stream. The requirement of materialization means that this operation can be expensive in terms of both latency and resource consumption. In Paradox, `Sort` is used purely as a "glue" operator or "enforcer" (Graefe 1993) to ensure that input arguments to a `Merge_Join` are sorted as required.
7) **`Para_Join`**: The `Para_Join` operator essentially pushes a full join operation to a remote service provider. The operator takes an iterator as its left input, a service call as its right input, and a binary join predicate over its two inputs. It pushes the service call specification and the join predicate to the remote service provider. It then pushes the entire left input to the service provider as an iterator stream. As with our other join algorithms, projection and selection can be merged into this operator. This operator can only work when supported by the `ParaJoin` capability of the remote service. If the service provider does not directly accept an iterator stream, but, rather, it materializes the left hand side before performing the join, this requirement is masked by a source wrapper. `Para_Join` can be used to implement a

complete join or a semi-join operation in which the join must be completed locally at the Paradox mediator.

8) **PAlert**: The PAlert operation is responsible for monitoring change events, optionally applying conditions to these events, and returning an identifier that allows a PGet to retrieve the data associated with an event ocurrence.
9) **PAlertNGet**: The PAlertNGet operation is responsible for monitoring change events, optionally applying conditions to these events, and retrieving associated data in iterator form.

Both the PAlert and the PAlertNGet operators must cope with heterogeneous source capabilities in the process of providing their functionality. We explain the workings of these operators in detail in Section 4.4.

Table 4-3 show the alternative physical operators that can be chosen to implement operators in the logical algebra of the Paradox system. Note that LJoinD limits the possible join algorithms that can be chosen to a subset of those that can implement a LJoin. Note, also, that Sort does not appear in this table, since it is a glue operator, with no direct logical analog. Sort operations appear in physical plans only where they are needed to enforce a sort order for the Merge_Join operation.

Logical	Physical
LGet	PGet
LSelect	Sel_Proj
LProject	Sel_Proj
LJoin	Local_NLJ, Remote_NLJ, Merge_Join, Para_Join
LJoinD	Remote_NLJ, Para_Join
LAlert	PAlert, PAlertNGet
LAlertNGet	PAlertNGet

Table 4-3: Logical Operator to Physical Operators Mapping

4.4.2 Logical to Physical Plan Mapping

The final stage of physical plan creation in Paradox is the mapping of the logical plan to a physical plan that can be run against the execution engine. The primary task in this process involves selecting physical join algorithms for each logical join in the plan. Decisions are also made on the physical operators used to implement monitoring, on merging selections and projections into joins where appropriate, and on inserting "glue" operators if necessary. Paradox employs a capability-driven approach together with a heuristic cost model to accomplish this task. More sophisticated cost models and more exhaustive enumerations of physical plans could be incorporated into our system. Our goal in this module of the Paradox system is not to break new ground or to demonstrate the best specific cost-based approach to this task, but rather to demonstrate that a cost-based approach is feasible and desirable.

```
SelectJoinsAndAlert(Logical_Plan_Tree T) {
(0)   for each Join Node, J, in T {
(1)     LC is LeftChild, RC is RightChild
(2)     if (isDelta(LC)) then getAlert(LC) <- PAlertNGet;
(3)     if (J is LJoinD) then
(4)       if (isDelta(RC)) then
(5)         adjustConditions(getAlert(RC),RC);
(6)         getAlert(RC) <- PAlert or fail;
(7)       J <- ChooseDJoin(J, LC, RC);
(8)     else // J is LJoin
(9)       if (isDelta(RC) and isLAlertNGet(getAlert(RC)) then
(10)        J <- ChooseNonDJoin(J, LC, RC);
(11)      else if (|CJ|+|LC| < |RC|) ) then
(12)          J <- ChooseDJoin(J, LC, RC);
(13)          if (isDelta(RC)) then
(14)            adjustConditions(getAlert(RC),RC);
(15)            getAlert(RC) <- PAlert;
(16)      else
(17)          J <- ChooseNonDJoin;
(18)          if (isDelta(RC)) then
(19)            getAlert(RC) <- PAlertNGet;
}
```

Figure 4-11: Physical Join and Alert Selection

The physical plan creation process first traverses the left-deep logical plan tree bottom-up, transforming each logical join node into a physical join algorithm. If the plan contains an `LAlert` node, then the corresponding `LGet` node and its parent join node require special handling. The transformation algorithm follows a set of cost-based heuristics. The algorithm is shown in Figure 4-11.

The algorithm begins by checking the special case where the leftmost child is an `LGet` containing a delta form (Line (2)). In this case, we make the corresponding alert operator into a physical `PAlertNGet` operator. Note that a `PAlert` in this case would simply require an additional, unnecessary message exchange. Line (3) checks if the current join is dependent. If so, we first test for the special case that the right child is an `LGet` containing a delta form. If it is, we make the corresponding alert operator into a `PAlert` and we move any conditions not supported by the `PAlert` to the `LGet`, via the call `adjustConditions()` (Lines (5)-(6)). Note that the attempt to make the alert operator a `PAlert` will fail if its logical form was a `LAlertNGet`. A logical `LAlertNGet` cannot pass an iterator into the right hand side of a dependent join. We complete the handling of a `LJoinD` node by choosing a dependent join algorithm (Line (7)).

If the current join node is an `LJoin`, we check for the special case that the right child is a delta `LGet` and the corresponding alert is an `LAlertNGet` (Line (9)). In this case, we must choose a non-dependent join algorithm (Line (10)). Otherwise, we make a cost-based choice between a dependent join or a non-dependent join based on network data transfer (Line (11)). If a dependent join is chosen, we choose a dependent join, and if the right child is a delta `LGet`, we make the corresponding alert a `PAlert` and adjust conditions (Lines (14)-(15)). Otherwise, we choose a non-dependent join, and if the right child is a delta `LGet` node, we make the corresponding alert a `PAlertNGet` (Lines (17)-(19)).

The algorithm shown in Figure 4-11 depends on two ancillary functions: `ChooseDJoin()`, shown in Figure 4-12, and `ChooseNonDJoin()`, shown in Figure 4-13. Choosing a dependent join via `ChooseDJoin()` is very simple; we simply prefer a `Para_Join` to a `Remote_NLJ` if it is supported by source capabilities. Choosing a

```
ChooseDJoin(Join J, Child LC, Child RC) {
   (1)   if (RC supports Para_Join) then
   (2)     return Para_Join(J, LC, RC);
   (3)   else
   (4)     return Remote_NLJ(J, LC, RC);
}
```

Figure 4-12: ChooseDJoin() Function

```
ChooseNonDJoin(Join J, Child LC, Child RC) {
   (1)   let Merge_Cost =
   (2)     if (NotSorted(LC))?(|LC|*Log|LC|):0 +
   (3)     if (NotSorted(RC))?(|RC|*Log|RC|):0
   (4)   let Local_NLJ_Sorted_Cost =
   (5)     if (NotSorted(LC))?(|RC|*Log|RC|):0 +
   (6)     |LC|*Log(|RC|)
   (7)   let Local_NLJ_Cost = |LC|*|RC|
   (8)   if Merge_Cost is least
   (9)     return Merge_Join(J, LC, RC);
   (10)  else if (Local_NLJ_Sorted_Cost is least)
   (11)    return Local_NLJ_Sorted(J, LC, RC);
   (12)  else if (Local_NLJ_Cost least)
   (13)    return (Local_NLJ(J, LC, RC);
}
```

Figure 4-13: ChooseNonDJoin() Function

non-dependent physical join algorithm via `ChooseNonDJoin()` is a bit more complicated. We consider a `Merge_Join` and two versions of the `Local_NLJ` algorithm, one in which the right argument is sorted, and one in which it is not. For the sake of comparison, the cost of `Merge_Join` is assumed to be dominated by sorting costs (lines (1)-(3)), while the cost of sorting (if necessary) plus the costs of repeatedly scanning the inner table are considered for the `Local_NLJ` algorithms (lines (4)-(7)). Note that we consider the "interesting property" of sort order, only adding the sort cost if the input is not sorted already. Data transmission will be the same regardless of the non-dependent join algorithm chosen, so it is disregarded as a cost factor. `Sort` operators are inserted as needed, at this stage, though this step is not shown explicitly in Figure 4-13.

Note that by defining separate operations for alerting and computing change data, we are able to generate a more flexible range of plans, and we will often be able to choose a more cost-effective plan. Yet we are still able to expose relevant costs when source capabilities dictate that these two operations are inextricably intertwined. We implement this process with relatively simple cost-based rules in Paradox. But the same approach could be used effectively in the context of a state-of-the-art cost-based optimization engine.

Once the join and alert algorithms have been chosen, one final pass is made over the leaf nodes of the plan tree to finalize the handling of sorting, selection and projection, and monitoring. For each leaf `PGet` node, if sorting is supported by the source and if a sort node exists between it and its nearest ancestral join node, the sort is merged into the leaf node. Then, if any selection or projection is in the leaf is not supported by the source, it is pulled out in the form of a `Sel_Proj` node. The `Sel_Proj` node is pushed into the join node above it if it does not jump past a materialization in the process. This strategy is based on the (conventional) heuristic that reducing the amount of materialized data reduces plan cost. That is, a `Sel_Proj` node does not leap over a `Sort` node, nor is a `Sel_Proj` merged into the right hand side of a `Local_NLJ` node. Keeping selection and projection below materialization saves on materialization cost and memory footprint. Returning to our example, working from the logical plan of Figure 4-10, we have two join algorithms to choose. Proceeding bottom-up, the first join must be dependent, and the classifieds service does not support the `Para_Join` operator, so we choose a `Remote_NLJ`. Moreover, since the Δ*Classified* `LGet` node is the right child of a dependent join, the corresponding alert must become a `PAlert`. Note, also, that since the monitoring capability of the Classified service does not support projection, the projection becomes part of the Δ*Classified* `LGet` node. The second join does not have to be dependent, but the estimated cardinality of the current join plus the estimated cardinality of the left child (the previous join, in this case) is less than the cardinality of the right child. Therefore, by line (11) of Figure 4-11, we choose to make the join dependent. In this case, we choose the `Para_Join` operator because the auction service supports it. Finally, traversing the leaf nodes finds that selections and projections are not supported by the Laptop service, so selection condition `Rating >= 3` and the

projection of the `Desc` attribute must be handled in the `Remote_NLJ` note. Other projection operations in the leaves of the plan are supported by the corresponding services. The resultant physical plan is shown in Figure 4-14.

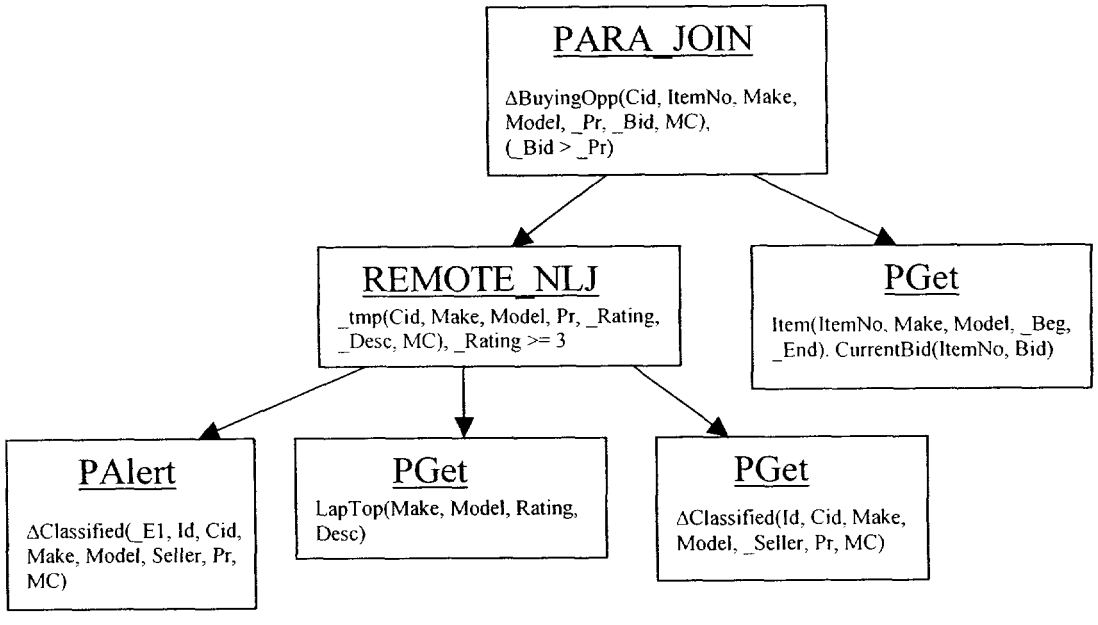

Figure 4-14: Complete Physical Plan for ΔBuyingOpp/ΔClassified

4.5 Monitoring Change Events

Following the generation of a suite of physical plans, the complete, "from scratch" plan can be executed immediately, but each additional plan in the suite must be deployed into the Paradox runtime environment. This process involves instantiating the physical monitoring operation (either a `PAlert` or a `PAlertNGet`) that tracks the appropriate service change event and condition that triggers the plan, and attaching the plan to this operation.

The two alerting operators share a great deal of functionality. In fact, they can be thought of as variations of a single operator, which we will refer to, generically, as the `Alert` operator. Each `Alert` operator is instantiated with a service change event at a

single service and (optionally) a condition on this event, which represents the possibility that the overall Paradox request may have fired. The job of the operator is to continuously monitor a specified service change event and condition in the face of heterogeneous service capabilities and, whenever such an event occurs, to pass needed variables or data to the associated plan that evaluates the overall condition and to initiate the evaluation of this plan. If possible, the operator must perform this service in a manner that minimizes response time and resource utilization, and that scales to a large number of requests. In this section we describe the general mechanism involved, how heterogeneous capabilities are handled, and how multiple events and conditions over the same service are handled in a scalable manner.

4.5.1 Basic Mechanism

In Paradox, both the `PAlert` operator and the `PAlertNGet` operator are implemented in the Java programming language using the (single) `PhysAlert` class. A single `PhysAlert` instance handles all of the monitoring for a given service provider, which is likely to encompass many `Alert` operators. The class provides an `Alerting` interface that includes two methods:

```
Id SetAlert(EventType, Condition, AlertClient);
Void RemoveAlert(Id);
```

The `SetAlert()` call begins a new monitoring process. It takes an optional `EventType` argument (which may be null if only one event type is supported), a conjunct of literals and (possibly) comparison predicates, `Condition`, that specifies the service change event and condition to be monitored, and an object that implements the `AlertClient` interface, which will receive event notifications and associated parameters when the specified event and condition are satisfied. `SetAlert()` returns an alphanumeric identifier, `Id`, which must be unique with respect to the mediator. A

call to `RemoveAlert()` ends the instance of the monitoring process that corresponds to its `Id` argument.

Notification occurs via the `AlertClient` interface. This interface contains a single asynchronous callback from the `PAlert` operator:

```
Void Alert(AlertId, EventId, ResultHandle);
```

In this call the identifier, `AlertId`, refers to the alert that this notification pertains to (the same identifier returned by the `SetAlert()` call that created it). The identifier, `EventId`, if it is non-null, uniquely identifies the change event at the source, and can be used by a `PGet` operator to retrieve the data associated with the event. The final parameter, `ResultHandle`, if it is non-null, is a handle to a non-empty iterator over the data associated with the change event. If the data associated with a monitored change event is the right child of a dependent join, the corresponding `Alert()` call will return a non-null `EventId`, and a null-valued `ResultHandle`. The class implements a `PAlert` operator. Otherwise, the `EventId` will be null, and a non-empty, non-null `ResultHandle` is passed along. The class implements a `PAlertNGet` operator. In the former case, the instantiated `EventId` becomes a selection condition that allows the right hand side of a dependent join to grab the correct data associated with the change event that triggered plan execution. In the latter case, the non-empty `ResultHandle` is inserted directly as a leaf node into the triggered plan execution.

The final piece to this puzzle is the object that receives the notification. In Paradox, the `PlanManager` object (an object of the `PlanManager` class) implements the `AlertClient` interface, and it is passed to each `PAlert` operator in every `SetAlert()` call. The `PlanManager` object tracks all of the active plans by their associated `AlertId`'s by storing them in a hash table. When an `Alert()` call is made on the `PlanManager`, the associated plan is retrieved, the `EventID` and `ResultHandle` parameters are passed to the plan, and a thread is forked to execute the plan. Note that while Paradox is configured with a single `PlanManager`, a load

balancing scheme that operates over multiple `PlanManager` objects could be deployed to help system performance and scalability.

4.5.2 Handling Heterogeneity

The `PhysAlert` class handles a collection of heterogeneous source capabilities in providing the uniform `Alerting` interface described above. Heterogeneity in monitoring at a source can be due to policy considerations (e.g., for security or efficiency reasons), to the semantic nature of the service being provided (e.g., access to a database, a complex computational function or predicate, a sensor or positioning system), or to the capacities of the computer systems involved (e.g., a data store may be a flat file, a legacy network or hierarchical database, or a modern relational system with advanced active functionality). In Chapter 3 we described how we model heterogeneous monitoring capabilities in metadata. Here we describe how the `PhysAlert` class adapts to the attributes of this model in supporting the `Alerting` interface. We describe important calls between the source of the change event being monitored and the `Alert` operator that does the monitoring. Note that the calls supported by the source (or the source wrapper) vary depending on source capabilities, as does the monitoring and retrieval protocol. To some extent adapting to heterogeneity is a responsibility shared between the plan generation process and the `PhysAlert` class. But the `PhysAlert` class bears the brunt of the burden.

Recall that our model breaks out along three nearly orthogonal dimensions: monitoring language, external notification, and information delivery. The `PhysAlert` class adapts its processing to each of these dimensions.

Dimension 1: Monitoring language. The monitoring language dimension describes the operations supported by a service in monitoring change events. All sources must provide access to the basic service call of any service it provides (possibly with binding pattern restrictions), but more advanced services may support declarative composition with other service calls, or additional predicates or operations on top of the basic call.

If a PhysAlert object receives a condition on a service change event that is completely supported by the service's monitoring language, it simply registers the condition with the service and processes alerts in the normal manner based on the other dimensions of the monitoring capability. If, instead, the PhysAlert object receives a condition that contains operations not supported by the monitoring language at the source, the unsupported operations are stripped from the original condition, the remaining condition is registered at the source, and the unsupported operations are applied locally in a post-processing step.

Recall that our plan optimization process considers source capabilities when it creates an Alert operator. This process guarantees that any condition passed to a PAlert is completely supported by the monitoring language at the source, and that the only operations that may be passed to a PAlertNGet operator that may not be supported at the source are select or project conditions. When such unsupported operations occur, normal alert processing will produce an iterator over change data that may be a superset of the data that satisfy the overall condition. The Alert operator creates a Sel_Proj operator that takes this iterator as its input, and that implements the unsupported operators. When a notification from the source occurs, the Sel_Proj is invoked as the root of an "alert plan", and when the first data item (if any) is produced by this operator, the Sel_Proj is passed as the ResultHandle in the corresponding Alert() call to the plan manager. Notice that, under this scenario, it is possible for the mediator to receive notification of a change event but for the associated plan to never be initiated. This situation can occur if all of the associated change event data is filtered out downstream, in the local processing stage of the Alert operator.

Table 4-4 summarizes the service (or service wrapper) calls invoked by the Alert operator relevant to this monitoring dimension.

Dimension 2: External Notification. Recall that there are two components to the external notification capability: whether the agent supports proactive notification, and how the form of notification (if any) is interpreted.

Service Wrapper Call	Comment
`Id SetAlert(T, Cond, PAlert)`	Alerting process returns an event id to the `PAlert` operator.
`Id SetAlert(T, Cond, PAlertNGet)`	Alerting process returns an iterator to the `PAlertNGet` operator. Post processing may be needed.

Table 4-4: Service Calls for Monitoring Language Dimension

The first element in the pair indicates whether a notification is actively pushed to a monitoring client. An increasing number of new and advanced services provide active notification. But services that wrap legacy systems, and other less advanced services, are unlikely to provide this capability.

If notification is actively pushed, then, when a `SetAlert()` call is made, the `Alert` operator merely registers a callback with the service (possible via a source wrapper) and waits for a notification. If notification is not actively pushed, then the `Alert` operator must actively poll the source for changes. The polling interval is an administrative parameter that may be overridden on a service-by-service basis. Precisely what is involved in this polling process depends on the second notification parameter.

The second parameter indicates whether any form of change notification exists, and if so, how it should be interpreted. Possible values are "Full", "Possible", and "None". A value of "Full" means that once notification occurs at the `Alert` operator (either actively or passively), we are sure that a change event meeting the specified condition has occurred. Advanced services, including services built on top of recent relational database offerings, can be expected to support "Full" notification. Furthermore, a value of "Full" implies that at least one information delivery capability is provided (i.e., either "Active", "EPull", or "TPull"; "Full" is incompatible with an information delivery capability of "None"), which allows us to retrieve the change event data either via an identifier, a time interval, or an iterator passed to the operator with the notification.

A value of "Possible" or "None" indicates that additional processing is needed to determine if the specified condition has occurred. "Possible" would characterize systems that can indicate that some change has occurred without knowing the exact nature of the change (e.g., a file with a "last modified" attribute). "None" represents a baseline default for low-capability services and legacy systems. The nature of this additional processing

is dependent on the information delivery capabilities of the source. If the value of the parameter is "Possible", such processing occurs only when the `Alert` has been notified of a possible change event, and any information delivery capability except "Active" is supported. If the parameter value is "None", the processing occurs at every polling interval, and the information delivery capability must be either "TPull" or "None".

Dimension 3: Information Delivery. The final monitoring dimension describes the level of support provided by the source in delivering the data associated with a condition or event occurrence. We have mentioned dependencies between this dimension and Dimension 2. Indeed, the dimensions we have outlined are not entirely orthogonal. There are three possible capabilities in this category, which may be offered in combination.

Active information delivery, represented by the value "Active," indicates that the data associated with a condition firing are actively pushed to the client together with change notification. This capability would be typical of recent RDBMS's designed to support web services, publish-subscribe systems, and other "push" technologies. Notification can be "push" or "nopush" when this capability is supported, but no separation of messages for change event notification and data delivery is required or supported. Active information delivery implies "Full" notification. That is, the data in the iterator is guaranteed to meet the conditions specified in the `Alert` operator. If "Active" delivery is an option and the `Alert` operator is a `PAlertNGet`, then this option will be chosen for data delivery.

Event-specific pull, represented by the value "EPull", indicates that an event identifier is provided with an event notification that allows data associated with the event to be retrieved at a later time via the query capability of the service. For instance, in our example, if event-specific pull is supported and we are alerted to a change event on the *Classified* service with identifier *EId*, a `PGet` operator with service call, Δ*Classified(EId, Cid, Make, Model, Seller, Pr, MC)*, can be used to retrieve the associated data. Note that "Full" notification usually accompanies the "EPull" capability, but it is not strictly required. "Possible" notification is conceivable in combination with "EPull", which

would indicate that the service in question partitions change data in a course-grained manner by id. Notification can be active ("push") or inactive ("nopush"). If the "EPull" capability is offered it will be chosen under two conditions: if the `Alert` operator is a `PAlert`, or if it is a `PAlertNGet` but the "Active" capability is not supported. In the latter case querying for change data by *EId* occurs within the `PAlertNGet` operator.

Timestamp-based pull, represented by the value "TPull", indicates that a service supports querying (and notification) of change data over a time interval. Information sources that provide historical information, or access to log files, or that track their evolving state as a series of update records, can offer this capability in a natural way. Notification can be active or passive, but it must include a service timestamp associated with the change event being reported. (A globally synchronized clock would allow us to remove this requirement.) An `Alert` operator uses "TPull" by tracking the timestamp of any previously-retrieved change data and querying for future change data from that time to the latest notification time. The `PAlert` implementation can use "TPull" with "Full" notification by maintaining a mapping from unique event identifiers to the time interval encompassing the corresponding change event. From the point of view of the `Alert` operator, the "TPull" capability is a less preferred alternative to "EPull". "TPull" is only chosen when it is the sole alternative. This preference for "EPull" is somewhat arbitrary, and is based on the assumption that if an "EPull" capability exists it is more likely to be highly optimized.

Finally, no information retrieval support, represented by the value "None", means that the `Alert` operator must rely on the query capabilities of the source to perform "Query plus Diff" processing to retrieve the data associated with the monitored condition. "Query plus Diff" processing means that the source is queried for all of the data that meets the monitored condition at the current time, and this result is compared with a previous result of the same query that has been materialized within the `Alert` operator. If the result has changed, then the alert fires and an iterator over the changed data is passed in an `Alert()` call to the `PlanManager`. Note that no information retrieval support is highly unlikely to occur together with "Full" notification, but it is not strictly forbidden. "Possible" notification (which is much more likely) or "Full" notification implies that "Query plus Diff" processing occurs only when a notification occurs. If no

notification is supported, "Query plus Diff" processing must be executed at a regular interval.

Note that "Query plus Diff" processing is typically very expensive. A system such as Paradox has poor scaling properties over services that require this kind of processing for change detection and computation. For reasonable scalability, such sources should be employed sparingly, and even then only where significant notification latency can be tolerated.

Table 4-5 summarizes the active notification callbacks from Service (or wrapper) to `Alert` operator that apply to the external notification and information delivery dimensions of a service monitoring capability. Table 4-6 summarizes the equivalent passive notification calls from Alert operator to Service (or wrapper).

Service Wrapper Callback	Comment
`Alert.alert(Aid, Eid)`	Active ("push") notification callback from service wrapper to (either kind of) `Alert` operator. Passes an event identifier. Implies "Full" or "Possible" notification. Implies "EPull" as an information delivery option.
`PAlertNGet.alert(Aid, It)`	Active ("push") notification callback from service wrapper to `PAlertNGet` operator. Passes an iterator over change event data. "Full" notification is required, as is "Active" information delivery.
`Alert.alertTS(Aid, Ts)`	Active ("push") notification callback from service wrapper to `Alert` operator with timestamp. Implies "Full" or "Possible" notification, and "TPull" as an information delivery option.
`Alert.alert(Aid)`	Active ("push") notification callback from service wrapper to `Alert` operator. Implies "Full" or "Possible" notification, and "None" as an information delivery option.

Table 4-5: Alert callbacks related to External Notification and Information Delivery Capabilities

Service Wrapper Call	Comment
`Bool eventFired(Aid, Eid)`	Passive notification call from `Alert` operator to Service. If true, event occurred and event identifier is passed. If false, event did not occur. Implies "Full" or "Possible" notification. Implies "EPull" as an information delivery option.
`Bool eventFired(Aid, It)`	Passive notification call from `PAlertNGet` operator to Service. If true, event occurred and iterator is passed. If false, event did not occur. Implies "Full" notification, and "Active" information delivery.
`Bool eventFired(Aid, Ts)`	Passive notification callback from `Alert` operator to Service with timestamp. If true, event occurred and timestamp of event is passed. Implies "Full" or "Possible" notification, and "TPull" information delivery option.
`Bool eventFired(Aid)`	Passive ("push") notification callback from service wrapper to `Alert` operator. Implies "Full" or "Possible" notification, and "None" as the information delivery option.

Table 4-6: Passive Alert Calls Related to External Notification and Information Delivery Capabilities

4.4.3 Handling Multiple Alerts on a Source

Along with providing the necessary wrapping to produce a uniform alert interface over a wide range of sources, the `PhysAlert` class also optimizes the handling of multiple alerts over a service by merging them where possible. Most modern databases and other information sources that support event and condition-based alerts or triggers make no effort to merge processing or result delivery over multiple triggers. Two alerts on such a source with identical conditions will evaluate the condition twice and return the result twice. This behavior is especially costly in a distributed environment where bandwidth conservation is important. Moreover, the result will be processed redundantly through two distinct iterators at the mediator, and it may be materialized redundantly

there as well. In network-based services such as those integrated by the Paradox system, monitored conditions can be similar in nature, displaying a high degree of overlap. Thus the opportunity for optimization is significant. Merging is essential where sources do not provide an advanced monitoring capability, for example, where "Query plus Diff" processing can be merged.

A generic alert-merging scenario is shown in Figure 4-15 below. Suppose two alerts are set that pertain to the same service change event, but have different conditions, A and B, respectively. Without merging, two leaf iterators will be produced, one for each event-condition pair. With merging, a single leaf iterator will be produced that applies the disjunct of the conditions being merged, A or B. This iterator will be shared between two filters, which apply each of the original conditions. Data that meets both conditions is sent and processed redundantly in the non-merged case, but only once in the merged case. Note that additional materialization may occur in the merged case, since the two client iterators of the shared iterator may consume input at different rates.

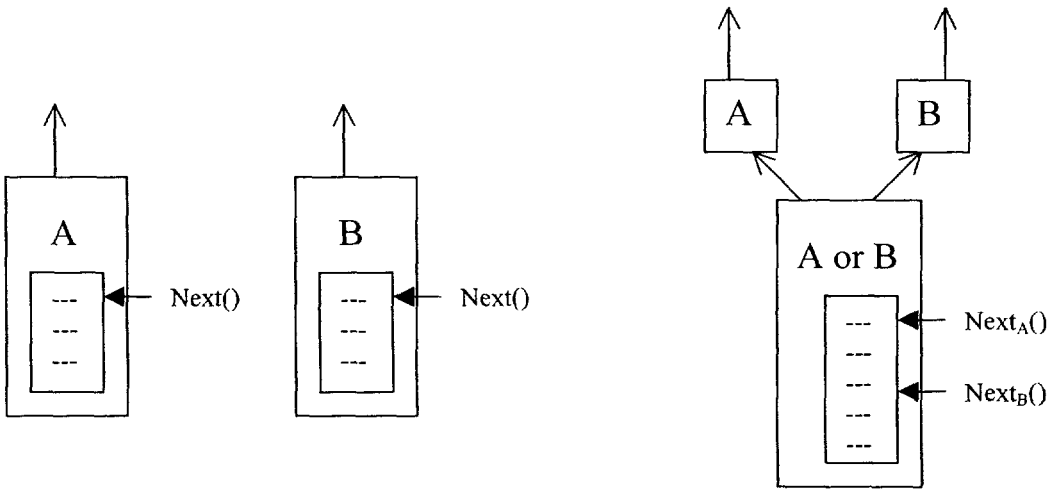

Figure 4-15: Unmerged v. Merged Alert Conditions

Paradox allocates at most one `PhysAlert` object per service provider. The `PhysAlert` object manages all of the alert processing for its provider, including making and implementing all merging decisions. Each time a `PhysAlert` object receives a `SetAlert()` call, it evaluates whether the new alert may be merged with one or more

existing alerts. An alert, A1, may be merged with an alert, A2, if A1 and A2 are *mergeable*.

Definition 4-1 (Mergeable Alerts): Two alerts with conditions defined as conjunctive queries with comparison predicates over a set of service calls, are *mergeable* if

1. They contain exactly the same set of service calls and change types.
2. They contain exactly the same binary predicates between service calls.
3. They share one or more projection attributes.
4. The corresponding `PGet` operation is not the right hand side of a dependent join.
5. The estimated data size of the intersection of the two conditions is above a configurable threshold for the given service provider, Θ.

□

Because merging is not without risk or cost, Paradox takes a conservative approach to determining mergeability. The policy implied by Definition 4-1, in essence, is to consider merging conditions that differ only in their unary (selection) predicates (and possibly in their projection attributes). Furthermore, we do not attempt to merge an alert if its associated leaf operator is the right hand side of a dependent join. Such a merger would require that the dependent join be changed to a non-dependent join. Even where binding-pattern constraints do not prevent such a change, Paradox deems changing an optimized plan to be too risky. Note that a decision of whether or not to change a plan to support alert merging could be made in a cost-based manner. But, again, Paradox currently relies on a simple heuristic. Finally, the effectiveness of merging depends critically on the degree of data overlap between the conditions being merged, and thus (estimated) condition overlap is an element of the mergeability criterion. The overlap must exceed a configurable threshold, Θ. Paradox uses a default threshold of Θ = 500K bytes, but this threshold may be overridden (by a system administrator) on a per-service-provider basis.

Note that a new alert may be mergeable with more than one target alert. In this case, all of the targets and the new alert are merged together. If an alert that is a

component of a merged alert is removed, because its corresponding request has been deleted, the remaining component alerts must be reevaluated for mergeability.

The goal in merging a set of alert conditions is to produce a minimal merged alert condition that allows easy derivation of the original alerts. Since mergeable alerts may differ only in their list of projected attributes and in their selection conditions, we simply take the union of the projected attributes and the disjunction of the selection conditions to create the merged alert condition. To derive the iterator handle for each original alert, then, we create a `Sel_Proj` operator for each original condition that applies its respective projection and selection conditions. We then place each of these operators on top of the shared `PGet` operator for the merged alert condition.

We return to a modified version of our example. Assume that there are no binding pattern restrictions on the Δ*Classified* service call, and that a Local_NLJ algorithm had been chosen for joining Δ*Classified* with `LapTop`. Call our original alert A1.

```
A1: (ΔClassified(T,Cid,Make,Model,_Seller,Pr,MC))
```

When A1 is first encountered, there is no `PhysAlert` object associated with the classifieds service, so one is created, and the `SetAlert` method is called to initiate `A1`. The `PhysAlert` object registers the alert condition with the classifieds service. At this stage, any time a Δ*Classified* change event occurs, the `PhysAlert` object initializes an iterator that contains the data associated with the event, and passes a handle to that iterator in an `Alert` callback to the plan manager, which initiates execution of the physical plan that evaluates the overall request.

Now suppose we encounter a new alert, A2, having the following condition:

```
A2: (ΔClassified(Id,Cid,Make,Model,Seller,Pr,MC),Pr >= $4000)
```

Assume the leaf plan node corresponding to this alert is the child of a non-dependent join. When A2 is encountered we make a `SetAlert` call to the existing `PhysAlert`

operator. The operator finds that A2 is mergeable with A1. A new alert is registered with the classifieds service. This new alert is different from the original because an additional projection attribute, `Seller`, must be returned in the data. Now when a `ΔClassified` change event occurs, a *shared* iterator is initialized and passed as the child of two additional new iterators: The first projects the `Seller` attribute away, the second applies the selection condition `(Pr >= $4000)`. Each of these iterators is run to its first output (if any), and then passed in an `Alert()` callback to the plan manager.

4.6 Chapter Summary

In this chapter we explained the major components of the Paradox system, describing the process of plan generation and optimization, and describing the execution engine that is targeted by the resulting plans, including a detailed discussion of the implementation of monitoring functionality using the `PhysAlert` class. While the process is broadly similar to that of database query processing, there are important differences. We introduce new logical operators to handle monitoring, and specialized physical operators to handle integration, heterogeneity and monitoring. We maintain a separation of monitoring a change event and retrieving the data associated with a change event, which provides flexibility in monitoring complex conditions that allows for more efficient execution. We also support the merging of monitoring conditions, which provides for greater scalability and efficient resource utilization. This merging process is our first look at "scaling via sharing" in this thesis, a theme that we will return to in depth in future chapters.

Chapter 5

An Example from the Command Post of the Future

The Command Post of the Future research project (CPOF), initiated and funded through DARPA, has provided a good testing ground for the Paradox system. We have worked with researchers at OGI in the Distributed Systems Research Group and at the Center for Human Computer Communication (CHCC) to develop application scenarios and system components that could be used in conjunction with Paradox's integration capabilities to support the requirements of CPOF. We describe one such scenario here to show how the pieces of the Paradox system come together in a real application.

One CPOF scenario involves the evacuation of personnel from a city under rebel attack. Military vehicles have been dispatched to the city to provide resistance and to help with evacuation. A sensor agent provides information on the changing positions of these vehicles. A fleet of *HumVees* is systematically combing the area in search of U.S.

Nationals to be evacuated. The existence and locations of evacuees are being actively discovered by various means in real time. As each evacuee is discovered, he is *designated* for evacuation and assigned to a HumVee that is in a position to evacuate him quickly. A commander, in charge of making such designations, wants to be notified whenever one of the HumVees comes within a short distance of any undesignated evacuee. Figure 5-1 shows the software components involved.

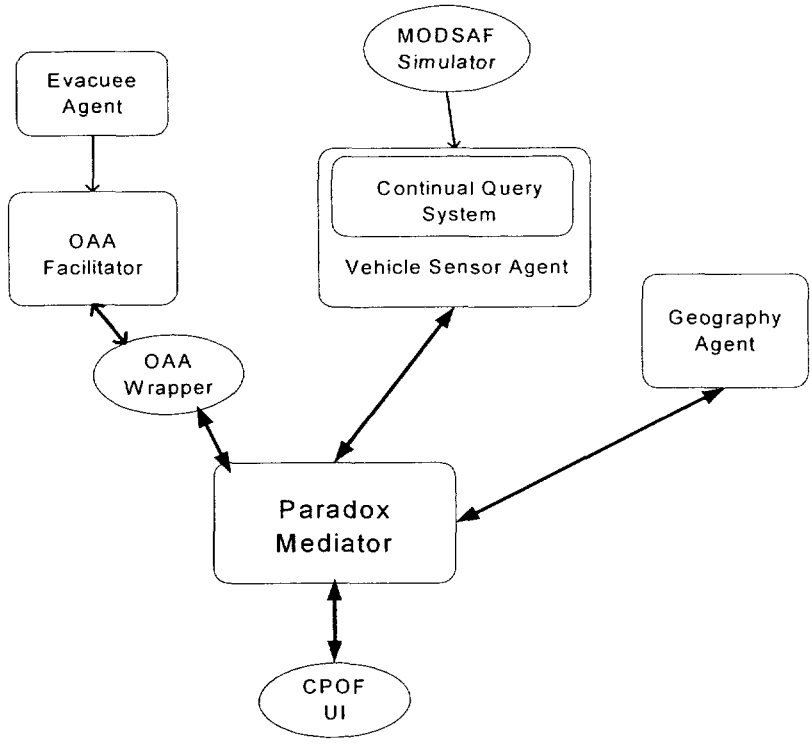

Figure 5-1: CPOF Architecture

5.1 Agents Involved

As shown in Figure 5-1, this application scenario requires the integration of several disparate, networked service providers, and two integration components. Three agents provide relevant services: The Vehicle Sensor Agent, the Geography Agent, and the

Evacuee Agent. In addition, the Open Agent Architecture, a multi-agent integration system developed by CHCC (Cohen 1994; Kumar 2000), is employed as a conduit to the evacuee agent.

5.1.1 The Vehicle Sensor Agent

The *Vehicle Sensor Agent* (VSA) provides information on military vehicles, their types, and their movements. The VSA provides movement data for hundreds of military vehicles within a given geographic boundary via the service call:

Position(VehicleID, Location)

Here *VehicleID* is a unique string-valued identifier, while *Location* is a string representation of a *LatLon* type, which provides the latitudinal and longitudinal coordinates of a vehicle's location. The *Position*/2 call is dynamic, as vehicles being tracked can change their positions, and vehicles can move in and out of the sensor agent's geographic range.

In addition to vehicle position data, the VSA provides information on the type of each vehicle in the fleet. This type information is provided via the service call:

Vehicle(VehicleID, VehicleType)

Again, *VehicleID* is a string-valued, unique identifier for a vehicle. *VehicleType* is the variety of vehicle, such as "Tank" or "HumVee", which is also string-valued. The *Vehicle*/2 call is considered dynamic since vehicles can be added to or removed from the fleet. However, a given vehicle's type can never change. Note that Paradox, as implemented, does not distinguish between a data-intensive service in which tuples can be added or deleted and one in which tuples can be modified. Later in this thesis, however, we will discuss a potential benefit to making this distinction.

5.1.2 The Geography Agent

A *Geography Agent* provides functions related to geographic calculations and conversions. For this scenario, two relevant service calls are provided:

AddressAt(Address, Location)
Distance(Location1, Location2, Distance)

AddressAt/2 converts a string address, *Address*, to corresponding *LatLon* coordinates in the variable, *Location*, or vice-versa. This type conversion is needed since some services can only operate with a normal street address, while others can only operate with the special *LatLon* type. The call must be made with either *Address* or *Location* bound. The Distance/3 service takes two *LatLon* types, *Location1* and *Location2*, and calculates the distance, in meters, between them. *Distance*/3 requires that the two location arguments be specified. Both of these functions are static.

5.1.3 The Evacuee Agent

An *Evacuee Agent* provides information that tracks the status and location of personnel of interest. Two service calls are relevant here:

Current(Person, Address)
Undesignated(Person)

Undesignated/1 tracks the status of known evacuees that are not yet designated for evacuation. *Current*/2 tracks the address of the last known location for the given person. Address here is a street address or building location within the city, which may require conversion to *LatLon* form for further processing. Person is a unique person identification number. Both of these calls are dynamic, as newly located personnel can

be undesignated, and the location of personnel can change. *Current*/2 requires that the *Person* field be specified.

5.2 Agent Implementation and Metadata

We have implemented the three agents described above in order to test the Paradox system in this context.

The Vehicle Sensor Agent was implemented using the Continual Query (CQ) system developed at OGI (Liu, Pu et al. 1998; Liu, Pu et al. 1999). As a Paradox service provider, the key capability provided by the CQ system is multi-relational triggering over an Oracle 8 relational database management system (Koch and Loney 1997). The VSA uses CQ to store and update vehicle type and position information in the database in the corresponding relations, Vehicle/2 and Position/2. Initial Vehicle/2 data comes from a file, and updated vehicle type information can be provided via the Oracle DBMS. Position data is more complex. The VSA receives a stream of simulated sensor data on vehicle positions and movements from the MODSAF system, a widely-used military simulation system (Cohen 2000). In general, the MODSAF stream can simulate the tracking of hundreds or thousands of vehicles. For our test scenario, we assume 300 vehicles are being tracked. MODSAF refreshes the positions of these vehicles roughly once every 12 seconds. The VSA filters the incoming MODSAF stream and propagates position changes to the Position/2 relation. The capability of CQ allows these two service calls to be declaratively composed as part of a single group, providing support for full querying and monitoring of select-project-join queries with comparison predicates over the two relations. A monitor on the position service, for example, reports a batch of position changes once every 12 seconds. A given batch of changes contains deletions of position data for vehicles that can no longer be tracked because they have moved out of range, additions of position data for vehicles that have recently come into range, and modifications of position data, modeled as deletions plus additions, for vehicles that have changed position within sensor range. These changes can be actively pushed to the monitor's client. Changes to Vehicle/2 are far less frequent. The characteristics of the

VSA are reflected in the Paradox metadata provided by this agent, which are listed in Table 5-1.

Key	Vehicle/2 Value	Position/2 Value
"Service"	"Vehicle"	"Position"
"Arity"	2	2
"Group"	"VSN1"	"VSN1"
"CCF"	Partial	Partial
"Modes"	{[?, ?]}	{[+, -]}
"Query"	{project, join, comp}	{project, join, comp}
"Monitor"	({project, join, comp}, (push, full), {active})	({project, join, comp}, (push, full), {active})
"TupleSize"	128 bytes	192 bytes
"Card"	300	300
"ColVals"	[300, 10]	[300, 300]
"KeyInfo"	[[1]]	Null
"Stability"	dynamic	dynamic
"Change Type"	[("veh_ct1", 1 per day, 2)]	[("pos_ct1", 1 per 12 secs, 100)]

Table 5-1: Metadata for the Vehicle Sensor Agent

Note that a number of metadata key values are intentionally not provided by the VSA. These include "CPU", "Selectivity", "BandWidth", "Latency", and "IC".

We implemented the Geography Agent as a simple Java RMI object that provides the AddressAt/2 and Distance/3 functions. Our agent does not support declarative composition of these calls, so they are provided within separate service groups. The Paradox metadata for the Geography Agent is shown in Table 5-2.

Key	AddressAt/2 Value	Distance/3 Value
"Service"	"AddressAt"	"Distance"
"Arity"	2	3
"Group"	"Geo1"	"Geo2"
"CCF"	Complete	Complete
"Modes"	{[+,?],[?,+]}	{[+,+,?]}
"Query"	{}	{}
"Monitor"	Null	Null
"TupleSize"	128 bytes	192 bytes
"KeyInfo"	{[1],[2]}	{[1,2]}
"Stability"	static	static

Table 5-2: Metadata for the Geography Agent

Finally, the Evacuee agent is implemented as an Adaptive Agent Architecture agent that holds data in a simple flat file. As with the Geography agent, the Evacuee agent provides its two calls in separate service groups. Active monitoring is supported for either call. The Paradox metadata for this agent is listed in Table 5-3.

Key	Current/2 Value	UnDesignated/1 Value
"Service"	"Current"	"UnDesignated"
"Arity"	2	1
"Group"	"Evac1"	"Evac2"
"CCF"	Complete	Complete
"Modes"	{[+, ?]}	{[?]}
"Query"	{}	{}
"Monitor"	({project, join, comp}, (push, full), {EPull})	({project, join, comp}, (push, full), {EPull})
"TupleSize"	128 bytes	64 bytes
"KeyInfo"	{[1]}	{}
"Stability"	dynamic	dynamic

Table 5-3: Metadata for the Evacuee Agent

Access to the capabilities of the Evacuee agent is mediated through the AAA Facilitator. The AAA Facilitator receives a Prolog-style predicate and passes it through to a distributed agent that can handle it, provided such an agent has registered its capability with the Facilitator's built-in directory service. The AAA Facilitator also provides blackboard-style memory access that can be used by connected agents to exchange information in a pre-arranged manner (Kumar 2000). We implemented a Paradox wrapper for AAA that allows Paradox to access the capabilities of any number of agents that are connected to a AAA Facilitator. Because the metadata employed by AAA is limited only to predicate names and their arity, we utilize the AAA blackboard to pass additional metadata from AAA agents to the wrapper. Such metadata is read from the blackboard by the wrapper, which then registers the equivalent metadata in PSM format with a network-resident directory service. Note that providing Paradox metadata in this manner involves a minor modification to a AAA agent, but it is not strictly required. But, naturally, the absence of such metadata may result in less efficient request execution. Furthermore, calls from the Paradox execution engine to a AAA agent involves the additional overhead of being routed through the AAA Facilitator. This overhead can be reflected in metadata for such agents, either provided by the wrapper, or observed by the monitoring process at the Paradox mediator.

5.3 Request Processing

A commander in this scenario wants to issue the following standing order: "Notify me whenever a HumVee comes within 2 kilometers of an undesignated person, and give me the relevant vehicle ID, person ID, the location of the person and the vehicle, and the distance between them." Given the set of network services described above, this standing order can take the form of the following Paradox request:

```
NotifyMe(VehID, Person, Loc1, Loc2, D) :-
     Vehicle(VehID, Type), Position(VehID, Loc1),
     Undesignated(Person), Current(Person, Addr),
     AddressAt(Addr, Loc2), Distance(Loc1, Loc2,D),
     Type = "HumVee", D < 2000.
```

When this request is submitted to the Paradox system, the request compiler produces the full execution plan shown in Figure 5-2. Note the existence of a Cartesian product in this plan (the LOCAL_NLJ operator). The Cartesian product is necessary due to the binding restrictions on the `Distance/3` predicate.

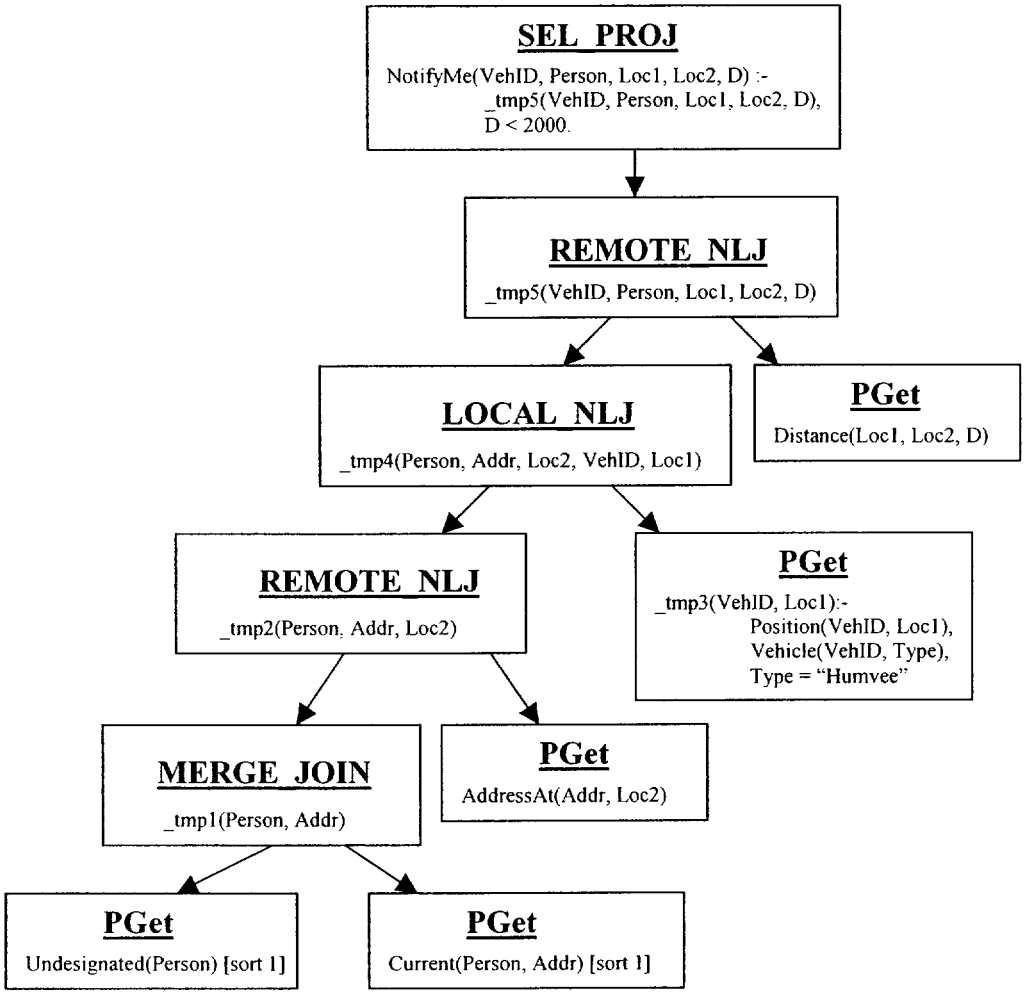

Figure 5-2: Execution Plan for Full Request

Three additional monitoring plans are generated to round out the plan suite, one for each leaf node that includes a dynamic service call. The first such plan computes ΔNotifyMe/ΔUndesignated, and is shown in Figure 5-3. This plan is triggered executed whenever an Alert operator on Undesignated fires. This operator retrieves

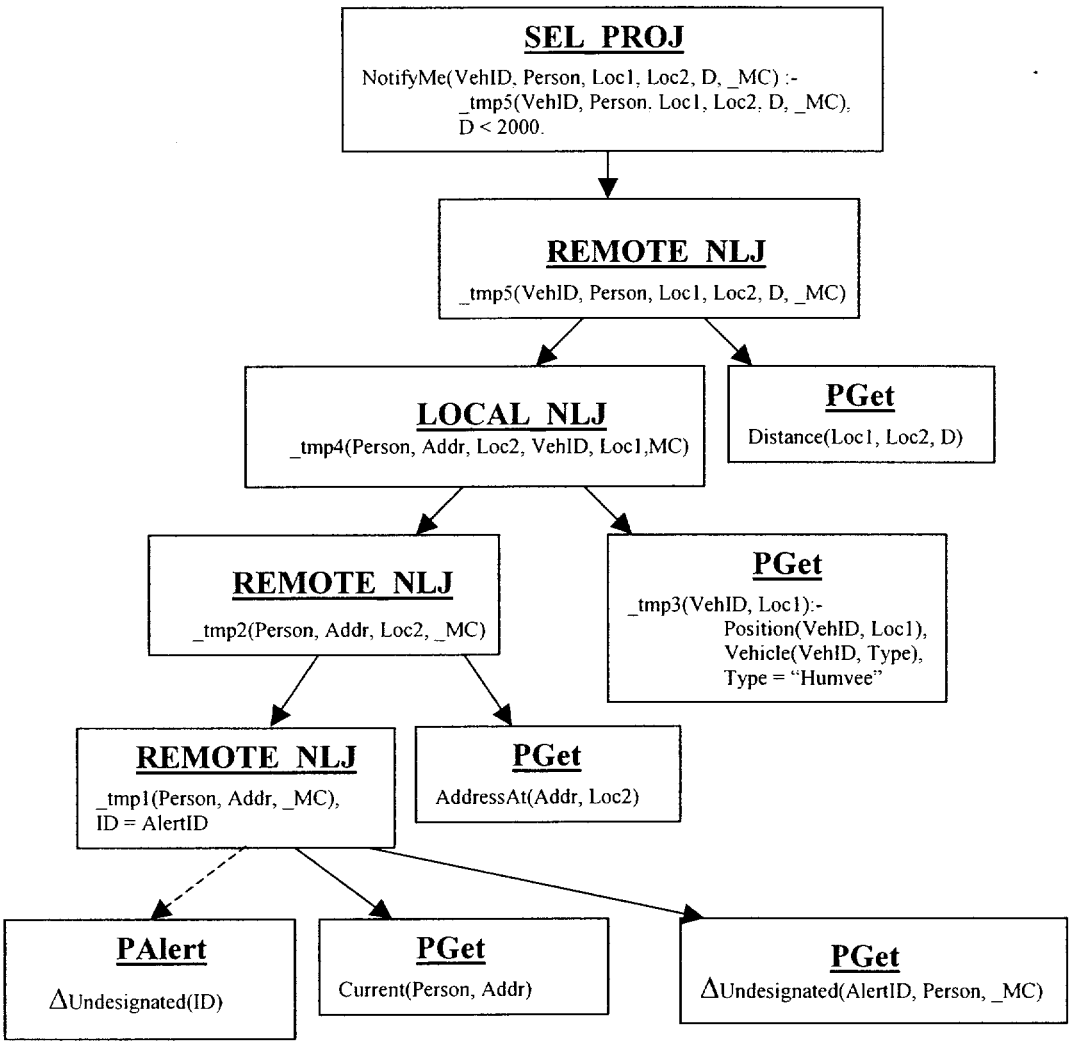

Figure 5-3: Execution Plan for ΔNotifyMe/ΔUndesignated

an AlertID associated with the change event, which is bound to a dependent join condition on its parent REMOTE_NLJ operator. Note the choice of a REMOTE_NLJ

operator to join ΔUndesignated/3 with Current/2. This incremental plan differs from the full query plan, which used a MERGE_JOIN to join Undesignated/1 and Current/2. Along with the event identifier, ΔUndesignated adds the multiplicity count argument, _MC, which is carried up the tree to the result.

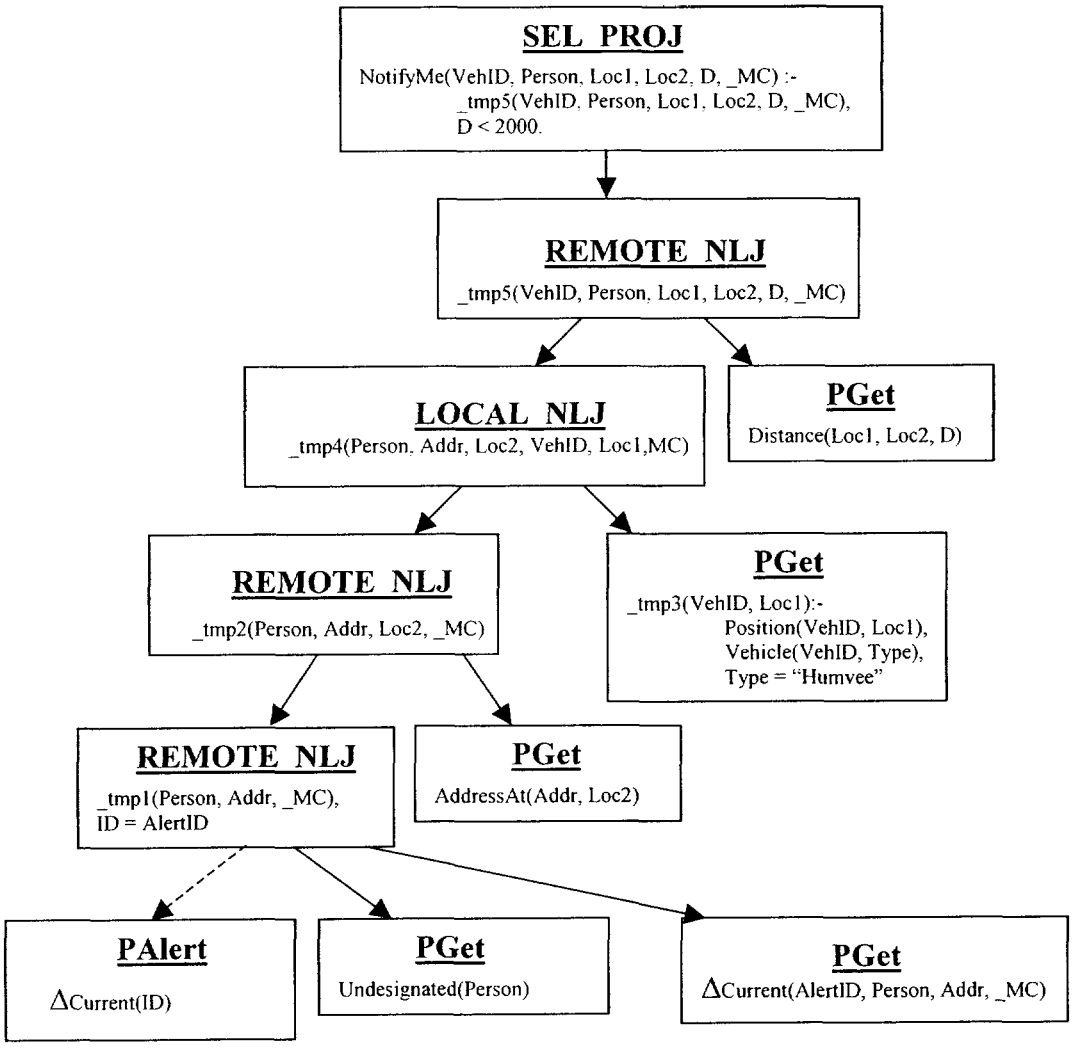

Figure 5-4: Execution Plan for ΔNotifyMe/ΔCurrent

A second monitoring plan is shown in Figure 5-4. This plan is executed whenever a PAlert operator on Current/2 fires. Again, this plan differs from the

full query plan of Figure 5-2 in that the `MergeJoin` of `Undesignated/1` and `Current/2` becomes a remote nested loops join, but this time `ΔCurrent/3` becomes the inner argument of the join.

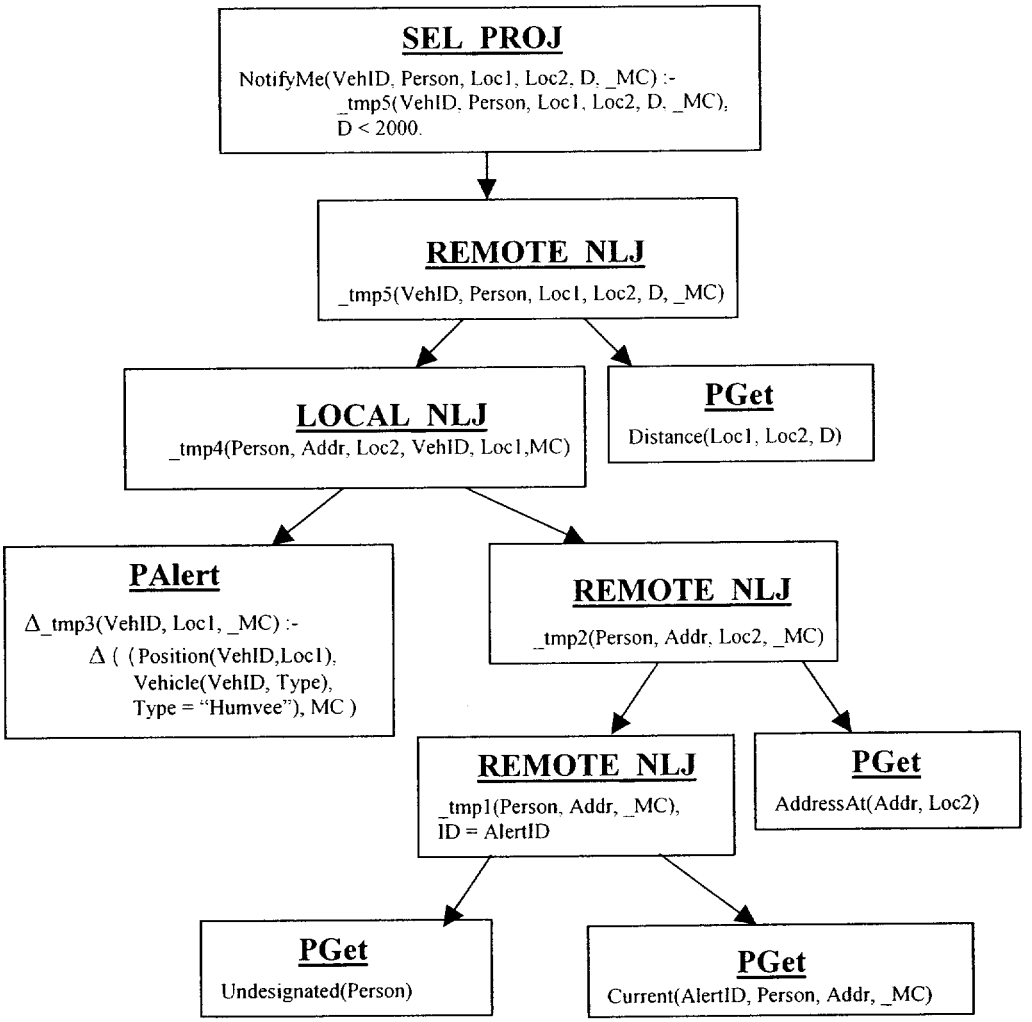

Figure 5-5: Execution Plan for `ΔNotifyMe/Δ_tmp3`

The final monitoring plan in the suite is shown in Figure 5-5. Here we take advantage of the CQ agent's ability to monitor full join conditions. We place a compound condition monitor on the CQ agent for the derived predicate, `_tmp3/3`. There is not separate `PGet` operator here, since the CQ agent will push the data

associated with a change event when the event occurs. Other than the insertion of the compound monitor, this plan resembles the original complete plan.

After generating these plans, the Paradox mediator installs `PAlert` monitors for tracking each relevant change event. A monitor on `Undesignated` is placed on the Evacuee Agent, with the corresponding action to execute the plan of Figure 5-3 at the mediator. Similarly, a monitor is placed on `Current` with a corresponding action to execute of the plan of Figure 5-4, and a monitor is placed on the compound goal `_tmp3/3` at the CQ agent, with the execution of the plan of Figure 5-5 as its corresponding action. Meanwhile, the complete current result is computed via the execution of the plan of Figure 5-2.

We deployed the application described in this chapter using the Paradox system and the agents described. We simulated a number of update patterns, which resulted in the scenarios and plans described in this chapter, and the system behaved correctly. The architecture and adaptability of Paradox made for a very convenient integration process for the agents involved.

The astute reader will note that much processing can be avoided in the execution of this plan suite if intermediate results are cached (and maintained) at the mediator. In particular, the Cartesian product offers an opportunity for big savings, since one side of the Cartesian product is completely recomputed in each monitoring plan, even though it has not changed as a result of the fired monitor. In the following chapters we consider caching and other techniques for improving the performance and scaling of active service integration.

Chapter 6

Query Optimization Basics

The chapters that follow this one require knowledge of several topics in query optimization that we cover here. We start with the basic process of query optimization in database management systems. We then touch upon the issue of optimizing queries in the presence of materialized views. We then discuss the issue of multiple query optimization, which deals with optimizing a group of queries for efficient simultaneous execution.

Of particular interest in this discussion is the key data structure involved in the query optimization process. This data structure is called either the *dynamic programming table* or the *memo structure*, depending on whether the query optimizer operates *bottom-up* or *top-down*. We also emphasize the role in traditional query optimizer technology of the *principle of optimality*, which states that an optimal execution plan is made up of optimal sub-plans. This principle is deeply embedded in current optimizer technology, but it breaks down in the optimization of multiple queries that execute simultaneously.

6.1 Relational Query-Optimization Basics

Declarative programming refers to a programming model in which the programmer specifies "what, not how". That is, the programmer specifies what he wants the behavior of the program to be, not the specifics of how that behavior is implemented. The declarative model requires a separation between a specification language and a virtual machine that implements the language. Implicit in this model is the notion that a specification may map to many different combinations of virtual machine operations. The process of interpreting a declarative program includes choosing a "good" mapping. The "goodness" of a mapping may depend on dynamic variables in the computing environment, and what is good under one set of conditions may not be good under another one. Declarative programming has been associated with lower rates of programmer error, a high degree of robustness and adaptivity, and amenability to proving program properties.

Relational query processing technology is perhaps the most successful instance of the declarative programming paradigm, in terms of market acceptance and ubiquity. A declarative *query specification language*, often SQL, is used to describe the characteristics of the data that is wanted[1]. The *query execution engine* provides the virtual machine, in the form of a series of physical operators, which implement the language. The operations of the specification language are sometimes referred to as the *logical algebra* of the system, while the operations of the virtual machine are the *physical algebra* (Graefe 1993). A given expression in the logical algebra may map to a large number of possible expressions in the physical algebra. The job of the *query compiler and optimizer* is to convert a query specification to an equivalent logical algebraic expression, and find an optimal mapping from that expression to an expression in the physical algebra. The efficiency of such a mapping depends heavily on dynamic properties of the data in the database (e.g., the size and statistical properties of relations),

[1] Strictly speaking, relational calculus is declarative, but relational algebra and SQL are not. But the latter are declarative in the sense that a given relation algebra (or SQL) expression can be mapped to many implementations. The operations wanted are expressed, but their implementation and composition is not.

on the existence of specialized data structures that support fast data access (e.g., indexes, materialized views), and on the characteristics of the computing environment (e.g., system load, resource contention).

6.2 Steps in Optimization

Broadly speaking, as shown in Figure 6-1, the query compiler and optimizer performs three major steps:

1. Query Parsing and Validation: A valid SQL string is turned into an initial logical algebraic expression.
2. Logical Plan Transformation: The logical algebraic expression is manipulated to produce an alternative logical expression.
3. Execution Plan Production and Costing: The logical expression serves as the basis for enumerating and evaluating a series of physical execution plans, from which a single, "best" plan is chosen.

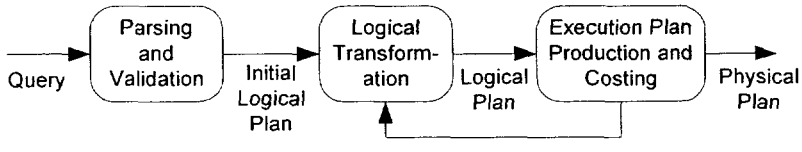

Figure 6-1: Query Parsing and Optimization

Step 1 also involves such issues as type checking and view expansion, and the creation of an initial relational algebra tree. In the Paradox system, as we have seen, validation includes the process of checking that distributed services exist that can handle the atomic components of the query. Step 2 involves performing meaning-preserving algebraic transformations based on the properties of the logical algebra. Some transformations may represent heuristic improvements to the logical plan, such as removing sub-queries from conditions or pushing selections and joins down the expression tree. Other

transformations involve traversing the search space, for example, regrouping commutative and associative operators such as joins, unions or intersections. There are many sources of further detail on the basic steps of query compilation and optimization (Ullman 1988; Korth and Silbershatz 1991; Garcia-Molina, Ullman et al. 2000).

6.3 Cost-based Enumeration

Steps 2 and 3 dominate the complexity of query compilation and optimization. These steps encompass a complex search problem, in which the search space is the set of all possible physical plans for implementing the initial logical plan, and the goal is to find the "best" such plan. State-of-the-art query optimizers employ a cost-based method for evaluating the efficiency of a plan. A cost function takes a physical plan as input and provides an estimate of the relative expense of executing the plan based on dynamic characteristics of the database and the computing environment. In general, a cost function must traverse an entire plan tree to find its cost. The plan of least cost is the "best" plan. This process is sometimes referred to as *cost-based enumeration.*

A baseline, naive approach to cost-based enumeration is to enumerate each plan one by one, compute its cost, and save the best plan so far until all plans have been considered. But such an approach is prohibitively expensive because the space of possible plans is massive for even a moderately complex query, and repeated costing is expensive. Modern optimizers employ two primary techniques to control this complexity:

1. Use heuristic methods to avoid enumerating the entire space of physical plans.
2. Make use of the *principle of optimality* to avoid optimizing any sub-expression more than once.

The first technique may take many forms. For example, a greedy heuristic method for finding a *n*-way join ordering might start by pairing off the two relations that have the smallest estimated result size, and then choose the next pairing from that first pair and the remaining single relations, and so on until the ordering is complete. The original System-R optimizer (and many optimizers today) considers only left-deep join

orderings (especially for large joins). Another common join-ordering heuristic is to ignore Cartesian products. Other methods include heuristic-based "branch and bound" enumeration, random sampling of plans in the search space, and hill climbing techniques (Garcia-Molina, Ullman et al. 2000). These methods can be employed in combination as well. In general, all methods that apply heuristics to limit the search space suffer from the same weakness: They are not guaranteed to produce the optimal result (based on the cost function). The price of producing the occasional poor plan should be weighed against the increased efficiency in the optimization process when using these methods.

The second technique depends on the "principle of optimality", which states that the optimal plan for a query cannot contain a sub-optimal sub-plan. This principle implies that optimal plans for larger queries can be built incrementally from the optimal plans of smaller queries. It allows us to optimize a logical sub-plan only once, store its optimal physical plan and associated cost, and plug this physical sub-plan and cost into any larger plans that we consider.

Example 6-1. Suppose we have two physical join operators, a nested-loops join operator, `NLJ`, and a hash-join operator, `HJ`, and we exhaustively optimize the query

$$Answer(X) \leftarrow A(X,W) \,\&\, B(X,Y) \,\&\, C(X,Z). \tag{6-1}$$

As part of this process we enumerate physical plans for this query that involve joining $C(X,Z)$ with $(A(X,W) \,\&\, B(X,Y))$. First we consider using `NLJ` at the top level. If we have not yet optimized $(A(X,W) \,\&\, B(X,Y))$, then we must do so as part of optimizing the full nested-loops plan. Next we consider using `HJ` at the top level. This time, due to the principle of optimality, when we come to the sub-expression $(A(X,W) \,\&\, B(X,Y))$, we can simply plug in the optimal sub-plan that we have already computed, along with its associated cost. We need not consider all the plans that combine a top-level hash-join with sub-optimal plans for the join of A and B.

Like heuristic pruning techniques, applying the principle of optimality greatly shrinks the plan space that we need to search and increases the efficiency of the

optimization process. But unlike heuristic techniques, these methods are *safe* in that they do not diminish the quality of the resulting plan. We think of the principle of optimality as a *law*, not as a heuristic.

Unfortunately, however, the principle of optimality is not a valid law as stated. A sub-optimal sub-plan may be part of a larger optimal plan if it produces a result that can be manipulated more efficiently by downstream operators than that of an optimal sub-plan. The prototypical example of this phenomenon is related to sort order.

Example 6-2. Consider Query (6-1) of Example 6-1 once again. Suppose that instead of the hash-join operator we have a sort-merge join operator, SMJ. Suppose, further, that the cost of joining A and B using NLJ is 30 units, while the cost of joining A and B using SMJ is 40 units. Note that although the result set of $(A \& B)$ is logically the same whether NLJ or SMJ is used, there is a *physical* difference between them that can affect downstream processing. The SMJ result is (necessarily) sorted on X, while the NLJ result is not. Suppose that the most efficient way to join $(A \& B)$ with C is by using SMJ, at a cost of 20. If NLJ is used to compute $(A \& B)$, then we must perform a SORT operation on the result of this computation before we can perform the sort-merge join with C. In contrast, if SMJ is used to compute $(A \& B)$, then we do not need to perform the sort. Suppose the SORT operation has cost 15. Then the optimal plan includes the sub-optimal sub-plan of using SMJ to compute $(A \& B)$. This scenario is depicted in Figure 6-2.

Selinger discovered a simple solution to this problem, which was to save not only the optimal physical plan for each logical sub-plan, but also the best physical plan that produced a result having an *interesting order*, for all interesting orders relevant to the query (Selinger, Astrahan et al. 1979). With this modification, we can safely employ the second technique above without affecting the optimality of the resulting plan. Selinger's

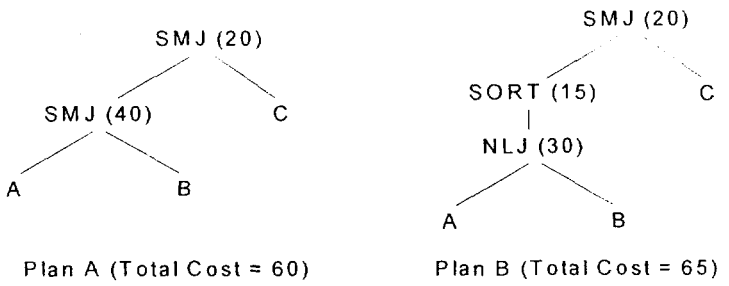

Figure 6-2: An optimal plan with a sub-optimal sub-plan

notion of interesting orders was later generalized to that of *physical properties* (Graefe and Dewitt 1987). The technique is widely applied in commercial optimizers. The application of this technique manifests itself in an important data structure that represents the state of the optimization process at any point in time. This structure stores the logical sub-plans being optimized, and the optimal physical plans and their associated costs for all sub-plans whose optimization is complete. The structure can grow very large for complex queries. It is referred to as either the dynamic programming table or the memo structure, depending on whether the search-space is explored in bottom-up or top-down fashion.

6.3 Top-Down and Bottom-Up Optimization

There are two broad approaches to exploring the space of possible plans in query optimization. In *bottom-up* plan exploration, the process begins with the sub-expressions at the leaves of the logical plan tree, and works its way up to the root. In *top-down* plan exploration, the process begins at the root of the tree, and works down to the leaves.

Note that much of the combinatoric complexity of the physical plan space in a relational query derives from the `join` operator. Since the `join` operator is both commutative and associative, a combination of `joins` results in an exponential explosion of possible plans. The `union` and `intersection` operations can have a

similar effect, but they are less common, particularly in large combinations (although unions can be quite common in a data integration setting). For the remainder of this section, we will focus on the optimization of multi-way join queries.

In the bottom-up optimization of a complex join, we start with each single relation that participates in the join. For each, we find the lowest-cost plans for accessing that relation that cover all groups of interesting physical properties. These plans are saved in a table, together with their costs, and the estimated size of the plan output. Next, we consider joins of pairs of relations. We consider both possible orderings of the pair, and all possible physical join algorithms. We do a table look-up to find the cost and size of the single relation sub-plans. Next, we consider all 3-way joins. For each, if we are searching exhaustively, we consider all (three) possible binary groupings, and both orderings of each grouping together with each physical join algorithm. This process continues until we have built a complete plan. At each stage, all sub-expressions of the expression we are considering will already exist in the (in-memory) table, together with its best-plan (for each set of interesting physical properties), cost, and result-size estimate. Figure 6-3 shows how the table might look at the completion of the optimization of a 3-way join of A, B and C. For simplicity we assume there are no interesting physical properties here. This bottom-up process is an example of *dynamic programming* (Aho, Hopcroft et al. 1983). By using dynamic programming, rather than raw exhaustive search, we reduce the complexity of query optimization from O(N!) to $O(3^N)$ (Ono and Lohman 1990).

Expression	Size	Property	Cost	Best Plan
A	500	None	100	SCAN(A)
B	1000	None	200	SCAN(B)
C	2000	None	400	SCAN(C)
AB	1200	None	500	HJ(B,A)
BC	5000	None	600	NLJ(C,B)
AC	3000	None	570	NLJ(C,A)
ABC	2300	None	650	HJ(BC,A)

Figure 6-3: Dynamic Programming Table for Bottom-up Optimization

Top-down optimization of complex joins proceeds in a different fashion. We start at the root and consider a single logical binary grouping at a time. For each binary grouping we consider a single physical join operator and move down the tree, searching as needed for the best plan for each subgroup and set of physical properties relevant to the parent operator. When the best plan for a given grouping and physical operator is found, we move to the next physical operator. When all physical operators have been considered, we move to the next possible binary grouping by applying a logical transformation to the previous binary grouping. We continue until all groupings are exhausted. This process fills out a data structure that is conceptually similar to the one created in bottom-up optimization, but the structure is filled out in a different order. Figure 6-4 shows what the structure might look like when we are in the midst of considering the first (nested-loops) of two physical algorithms for the third possible logical grouping of the 3-way join of A, B and C. Note that we are currently considering the fifth complete plan for the query, but we have not considered any plans for the sub-expressions AC. This process is an instance of *memoization* (Michies 1968; Russel and Norvig 1995), the top-down dual of dynamic programming. We refer to the data structure as the *memo structure*. Top-down memoization also reduces the complexity of exhaustive join-order enumeration from O(N!) to $O(3^N)$ (Shapiro, Maier et al. 2001).

Logical Group	Optimization State for Group
ABC	Logical Expr: (AB)C Physical Expr: NLJ((AB),C), HJ((AB),C) Logical Expr: C(AB) Physical Expr: NLJ(C,(AB)), HJ(C,(AB)) Logical Expr: A(BC) Physical Expr: NLJ(A,(BC)) Best Plan (So Far): HJ((AB),C) Cost: 750 Size: 2300
AB	Logical Expr: AB Physical Expr: NLJ(A,B), HJ(A,B) Logical Expr: BA Physical Expr: NLJ(B,A), HJ(B,A)
A	Logical Expr: A Physical Expr: Scan(A) Best Plan: Scan(A) Cost: 100 Size: 500 Property: None
B	Logical Expr: B Physical Expr: Scan(B), IndexScan(B) Best Plan: Scan(B) Cost: 200 Size: 1000 Property None
C	Logical Expr: C Physical Expr: Scan(C) Best Plan: Scan(C) Cost: 400 Size: 2000 Property None

Figure 6-4: Memo Structure for Top-down Optimization

The first generation of commercial query optimizers, based on the pioneering work of the system R group at IBM, employed the method of bottom-up plan exploration with dynamic programming (Selinger, Astrahan et al. 1979). Work motivated by the desire to make optimizers more easily extensible produced the *top-down* method (Graefe and Dewitt 1987). Lohmann introduced a rule-based approach within the context of bottom-up optimization that improved extensibility in that context and was employed in the Starburst system (Lohman 1988). Note the Lohmann's rules are in essence the bottom-up analogue to top-down transforms. The issue of whether one method is inherently more extensible remains a controversial one.

The debate is also open as to whether top-down or bottom-up optimization provides superior performance. Proponents of the top-down method note that the memoization generates complete plans more quickly. Once a complete plan is obtained, it provides an upper bound on total query cost that can be used to prune away parts of the top-down search space without complete expansion. This process is known as *group pruning* (Shapiro, Maier et al. 2001). Group pruning can be especially effective when combined with heuristics in such a manner that inexpensive groupings and cheaper physical algorithms can be considered quickly, and so good upper bounds can be produced quickly.

But regardless of whether a top-down or bottom-up strategy is employed, the effective optimization of complex queries is costly. If guaranteed optimality is a goal, the expense increases exponentially in the number of joins in the query ($O(3^N)$), and the expense is dominated by the expansion and maintenance of a large data structure of partial results: the dynamic programming table in the bottom-up case, the memo-structure in the top-down case.

In the coming chapters we build on the ideas presented here in describing optimization tasks that arise in improving the scalability and efficiency of an ASIS. We show that the principle of optimality does not hold for some of the problems that arise in this setting. We discuss how sharing of the optimization effort embodied in the memo structure can be shared to make the problems that arise more tractable. And we describe an implementation based on a state of the art top-down query optimizer that demonstrates the effectiveness of sharing in this context.

Chapter 7

Sharing in an Active Service Integration System

Everything you really need to know about scaling distributed information systems you learned in kindergarten. The first commandment of Robert Fulghum's best selling book by (nearly) the same name is this: "share everything" (Fulghum 1993); and sharing is a key principle in managing growth in a distributed information system. As workload grows and as the number of distributed system components grows, it becomes essential that common planning, processing and data movement tasks be performed once, and shared over a range of higher-level tasks. In Chapter 3 we encountered one form of sharing in an ASIS, where multiple monitoring requests that map to a single source could share a single source monitor. In this chapter we discuss the issue of sharing in active service integration in greater detail.

In active service integration, an important opportunity for large-scale sharing arises where supplemental views can be materialized and maintained at the mediator, and shared among multiple executions of a long-lived request. Another sharing opportunity

occurs where multiple tasks must be executed in response to a single event, and these tasks can share common partial results. We describe how these forms of sharing can be exploited in detail. Exploiting these two forms of sharing presents a massive optimization problem. We describe a multi-pronged approach to making this problem more tractable, and show how a third opportunity for sharing of planning and optimization presents itself in this context. We lead into a new technique for exploiting this opportunity that we call *Multiplex Query Optimization*, which is discussed in detail in Chapter 8.

7.1 Associative Caching and Materialized Views

Caching is perhaps the fundamental tool for handling scale in distributed systems. As Van Jacobson writes, "With 25 years of Internet experience, we've learned exactly one way to deal with exponential growth: caching" (Rabinovich 1998). What is true on a macro scale, for the Internet as a whole, is also true at the micro scale, for a single-mediator-based system. By moving data from distributed sources to the mediator, a system can achieve better response time, more efficient resource utilization, scale-up in terms of number of sources and number of clients supported, and greater robustness in the face of source failures.

Caching, of course, is fundamental and ubiquitous in computing systems. Anywhere that a data storage hierarchy exists in computing, a cache is sure to follow: Operating systems maintain a file system cache in main memory; databases cache data from disk in a memory-resident buffer pool; DNS caches domain name to IP address mappings; distributed systems routinely replicate remote data and store it closer to where it is needed, etc. In many information systems, including local and client-server databases, page and tuple caching is prevalent. But page caching cannot be applied to a system that integrates autonomous sources, since the mediator has no knowledge of the physical layout of source data. Nor is tuple caching attractive in this setting, since it is too fine-grained to be efficient over a wide area. In predicate caching (Keller and Basu 1996),

and closely related notions including semantic caching (Dar, Franklin et al. 1996) and view caching (Roussopoulos, Chen et al. 1995), caches are stored and manipulated based on their logical content, represented as a query or view. Such *associative caching* schemes provide the necessary abstraction between cache data and source data layout for dealing with autonomous sources. Variants of associative caching have been studied by a number of researchers in the context of data integration (Roussopoulos, Chen et al. 1995; Adali, Candan et al. 1996; Scheuerman, Shim et al. 1996; Godfrey and Gryz 1997; Luo, Naughton et al. 2000). Our approach to caching for active service integration can be viewed as a variant of associative caching as well.

The vast majority of caching schemes, including those referenced above, rely on *locality of reference* to be effective. That is, they rely on the familiar heuristic that data that have been requested in the past will be requested again in the (near) future with high probability. Locality of reference is an effective heuristic in many settings, but it is also often used as a general default, in the absence of good knowledge of system or application behavior, with mixed results. An ASIS, however, has access to important information that describes its future behavior, which allows us to design a more specific and effective caching strategy. A monitoring request sent to an ASIS, such as Paradox, induces a suite of long-lived queries that represents (part of) the future workload of the system with high certainty. Consequently, a caching strategy can be designed around a (partially) knowable future, and thus can be more accurate than one based on heuristic "guesses" about the nature of future requests. Further, a query suite involves a (possibly large) number of very similar queries that can often share intermediate results. Caching such shared intermediate results can yield dramatic savings.

Of course, it is not quite accurate to say that the future is knowable in this context. A particular query in a query suite is only executed in response to specific events at a relevant service. The future workload depends on when and how the information content of relevant services changes. In particular, the frequency of change, the amount of data involved in the change, and the statistical properties of the data (value distribution, etc.) are important in determining the future workload of the system, and in determining the

best cache for supporting this workload. Fortunately, for real applications, we often know quite a bit about these properties of information change. For example:

- An online bookstore adds new titles and restocks old titles once weekly. The number of new titles is known, on average, and does not fluctuate greatly. Depletion of inventory, on the other hand, is continuous.
- A web magazine that publishes book reviews comes out once a month and contains about 5 new reviews in each issue.
- A legacy insurance claims information system is updated in batch mode once nightly.
- A military information system that traces vehicle movements in a battlefield updates vehicle positions every 5 seconds, with 50% of the traced vehicles, on average, changing position at each interval.
- A source of stock market quotes updates its quotes once every 5 minutes, with 80% of the quotes being modified, on average, at each interval. Stock symbols are rarely deleted or added.
- A portfolio-tracking application that uses the stock market source tolerates up to one hour of staleness in its stock quote data, and thus brings updated quote data in once an hour.
- An auto-parts supply chain operates such that small requests are met by groups of regional dealers based on parts on hand on a daily basis, while large-scale inventory changes occur based on need projections and are filled by a central parts-manufacturing facility on a monthly basis.

We would like to exploit knowledge of this kind in our caching strategies where possible. Note that we may not have information of this sort for all sources or services. But even where we do not, we can employ techniques to glean this kind of knowledge (for example, by monitoring the frequency and size of updates from a service over time). Few information sources change in a truly random fashion.

Armed with service change information, and the monitoring queries that run when changes occur, an ASIS has the leverage to apply an aggressive form of caching in which

a cache is defined semantically and its coherence is maintained continuously. This form of caching is equivalent to the creation and maintenance of supplemental materialized views in support of a workload of requests. We will refer to this technique as supplemental view maintenance (SVM) to distinguish it from other caching schemes.

Example 7-1: Suppose that we use our mediator-based system as the platform for "MyCitySearch.com", a service that provides customized integration of myriad information services related to entertainment and other events happening in various cities. A concertgoer wants to be notified anytime a musical "R&B" act that is big enough to have been reviewed by a source of concert reviews is coming to a local club. In addition to the artist, club and review information, the customer wants a URL to information on purchasing CDs by the artist. This request might be expressed by the following query involving three network-based information services:

$$\begin{aligned} &\textit{ComingSoon(Artist, Club, Date, ReviewURL, PuchaseURL)} \leftarrow \\ &\qquad \textit{ClubListing(Artist, "MyCity", Club, Date)} \ \& \\ &\qquad \textit{ConcertReview(Artist, ReviewURL)} \ \& \\ &\qquad \textit{MusicForSale(Artist, "R\&B", PurchaseURL).} \end{aligned} \tag{7-1}$$

Suppose that *MusicForSale*/3 information changes infrequently and by small amounts, with about 10 new artists added once every two weeks, with two, on average, being "R&B" artists. Suppose, further, that concert review information is higher volume, with batch updates averaging 500 reviews in size occurring once per week. Club listings, on the other hand, change daily, with 200 listings being deleted daily (since they are now in the past) and 200 new listings being added daily (to replace the deleted ones), on average.

Using incremental techniques, as we do in Paradox, whenever *ClubListing* is updated, we compute changes to Query (7-1) with respect to this update, $\Delta ComingSoon / \Delta ClubListing$, by executing the incremental query:

$$\begin{aligned} &\Delta ComingSoon(Artist, Club, Date, ReviewURL, PuchaseURL, \_MC) \leftarrow \\ &\qquad \Delta ClubListing(Artist, \text{"MyCity"}, Club, Date, \_MC) \ \& \\ &\qquad ConcertReview(Artist, ReviewURL) \ \& \\ &\qquad MusicForSale(Artist, \text{"R\&B"}, PurchaseURL). \end{aligned} \tag{7-2}$$

Even if $\Delta ClubListing$ is small, Query (7-2) may be expensive, involving remote service calls to sources of concert review information and to sources of music-for-sale information. The efficiency of computing $\Delta ComingSoon / \Delta ClubListing$, however, can be greatly improved if we compute and store the following supplemental view at the mediator:

$$\begin{aligned} &ReviewAndSaleInfoByArtist(Artist, ReviewURL, PuchaseURL) \leftarrow \\ &\qquad ConcertReview(Artist, ReviewURL) \ \& \\ &\qquad MusicForSale(Artist, \text{"R\&B"}, PurchaseURL). \end{aligned} \tag{7-3}$$

With an up-to-date copy of *ReviewAndSaleInfoByArtist* stored at the mediator, we can compute $\Delta ComingSoon / \Delta ClubListing$ by executing the following query:

$$\begin{aligned} &\Delta ComingSoon(Artist, Club, Date, ReviewURL, PuchaseURL, \_MC) \leftarrow \\ &\qquad \Delta ClubListing(Artist, \text{"MyCity"}, Club, Date, \_MC) \ \& \\ &\qquad ReviewAndSaleInfoByArtist(Artist, ReviewURL, PurchaseURL). \end{aligned} \tag{7-4}$$

Where Query (7-2) requires multiple remote calls to possibly large collections of data, Query (7-4) requires single call to a smaller, locally stored view, which is likely to be much less costly.

Naturally these savings do not come for nothing. Storage space must be allocated for the supplemental view, and the view must be created and maintained. In our example, however, savings occur daily, each time *ClubListing* is updated and Query (7-4) must be executed. In contrast, we incur the cost of maintaining *ReviewAndSaleInfoByArtist* when either *ConcertReview* or *MusicForSale* is updated, which happens only three times every two weeks. If we assume, for the sake of illustration, that the savings per execution of Query (7-4) is equivalent to the cost of each incremental maintenance query for

ReviewAndSaleInfoByArtist, then, disregarding storage space costs and the one-time cost of creating the supplemental view, we save $11/14 \approx 78\%$ of the total cost of monitoring Query (7-1) over time. □

7.2 Computing the Value of Materialized Views

In general, given a workload of queries, W, and a set of views, V, we can compute the value of materializing and maintaining V by computing the difference between the cost of executing W without V and the costs of executing W with V and of maintaining V. More specifically, we give the following definition:

Definition 7-1 (Expected Value of Materialized Views): Given

1) A database with schema and related metadata, S.
2) A workload, $W = \{(q_0, f_0), \ldots, (q_n, f_n)\}$ over S where the q_i are queries, and f_i gives the frequency of execution of q_i over time.
3) A set of materialized views, V, with an associated view maintenance workload $X = \{(m_0, g_0), \ldots, (m_n, g_n)\}$ over S where the m_i are maintenance queries for V, and where g_i gives the frequency of execution of m_i over time. Note that X is a function of V and S (since S is assumed to include update frequency information).
4) A query optimizer *Opt* with an associated cost function, C.

The *expected value of materializing and maintaining V* with respect to W, S and *Opt* is given by:

$$VoM(V,W,S,Opt) = \sum_{i=0}^{n} f_i \times C(q_i,\{\}) - \left(\sum_{i=0}^{n} f_i \times C(q_i,V) + \sum_{i=0}^{m} g_i \times C(m_i,V) \right) \qquad (7\text{-}2)$$

where $C(q, MV)$ is the estimated cost of the optimal plan for query q in the presence of materialized views MV, as given by *Opt* using S. □

For notational brevity, we will often leave off explicit references to any or all of *W*, *S*, or *Opt* when their values are clear from context. For example, we may refer to *VoM(V,W)* or *VoM(V)*.

Notice that the first term on the right-hand side of Equation (7-5) is the cost of the workload with no views materialized. The second term is the cost with materialized views, which is broken down into the workload cost and the view-maintenance cost.

Our definition of value assumes that the workload of queries repeats at some interval, as do the maintenance queries for *V*. Thus the measure computed for *VoM* is cost savings over time. The definition assumes that the database and related metadata, *S*, and the optimizer, *Opt*, do not change. Note, further, that the term "maintenance query" presumes that the views being maintained already exist. We ignore the initial cost of computing *V*, since it is a one-time cost.

Often we may want to consider the space costs of maintaining a set of materialized views. In this case, a useful measure will be the value per storage unit for the set of views. We refer to this measure as the *expected cache efficiency* of the view set:

Definition 7-2 (Expected Cache Efficiency): Given Definition 7-1 above, we say that the *expected cache efficiency* of *V* with respect to *W*, *S*, and *Opt* is given by:

$$VoM(V,W,S,Opt)/Size(V)$$

where $Size(V)$ is the estimated storage size of *V* as given by *Opt*. □

The units for cache efficiency are cost savings per storage unit (often bytes or blocks) over time. This measure is useful when we have limited space for view materialization. In particular, this comes into play when we want to keep all materialized views in main memory.

7.3 The Generic View Selection Process

Given Equation (7-5) for computing the value of a set of materialized views, a method for computing the optimal view set to materialize immediately suggests itself: Begin by estimating the workload the system will process, generate all possible useful view sets for this workload, compute the value or cache efficiency for each view set, and choose the best view set based on these values. A simple refinement of these steps involves computing possible view sets by first generating individual candidate views, and then combining these to create candidate view sets. A conceptual sketch of this process is shown in Figure 7-1 below.

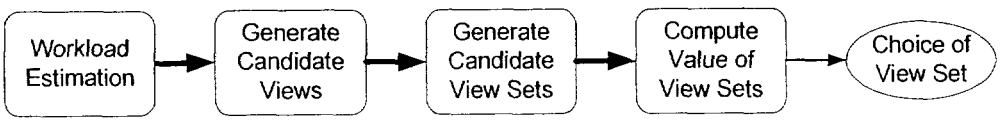

Figure 7-1: View Selection Process

We make several remarks about this process, which we will elaborate on in the following sections:

- The quality of view selection depends on the degree to which the estimated workload accurately reflects the real workload of the system.
- View selection is a massive and expensive optimization problem. In particular, value computation involves performing a combinatorial process, query optimization, over each query in the workload, and it must be performed over a potentially combinatorial number of possible view sets.

- For a large workload, the space of potentially useful view sets is large. Since the value computation is repeated for each candidate view set, one important way to increase the efficiency of view selection is to use heuristic methods to limit the space of view sets considered. Care must be taken, however, since the use of heuristics may produce a sub-optimal result.
- The process of value computation involves repeatedly running the same set of queries over a varying physical data layout. This process may involve a large amount of redundant work. Greater efficiency can be achieved if this redundant work is avoided.
- The processes shown in Figure 7-1 above are strongly interrelated and should be tightly coupled. The view selection process requires an end-to-end, systemic solution.

To this point we have discussed view selection in a generic sense. Note that this process extends to the notion of index selection as well. In fact, an index can be seen as a limited, projection-only view over a base table or materialized view. Thus, to a large extent, our discussion and methods can be applied to automatic computation of physical database design given a workload and a layout of base data or services. But our specific interest is in the materialization of views in an ASIS. We now discuss the view selection process in detail in this context.

7.4 Selecting Materialized Views in an ASIS

In this section we outline elements of our approach to selecting supplemental materialized views in an ASIS. In so doing, we describe in detail how the process outlined above maps to the active service integration environment, and we elaborate on the points raised previously about the process.

The architecture of our approach is shown in Figure 7-2 below. The figure shows a similar group of steps to those listed in the generic sketch of view selection from Figure 7-1, but there are some additional steps and several noteworthy details:

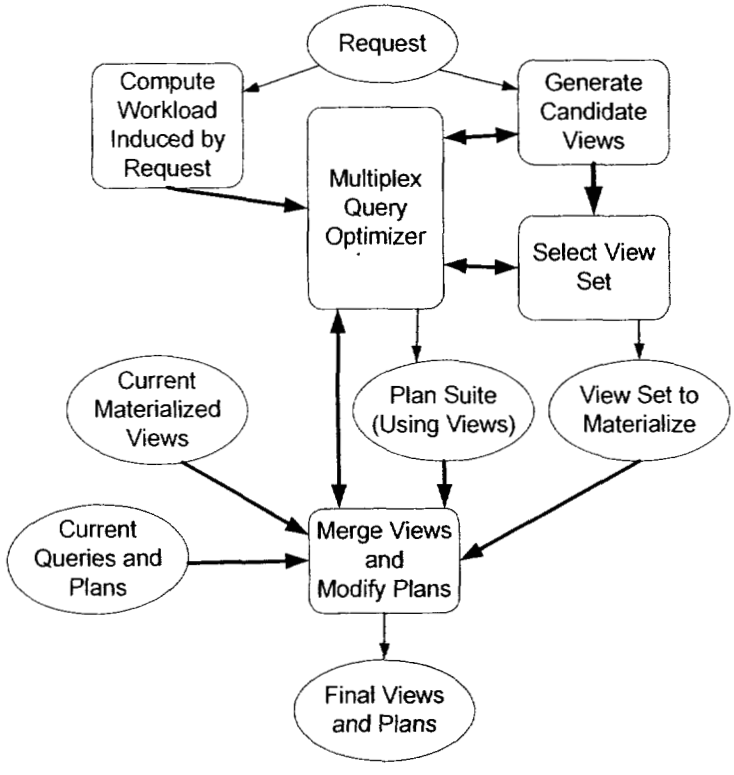

Figure 7-2: View Selection in an ASIS

1. In an ASIS, we want to adapt the selected view set on a continuous basis, as requests are submitted. But to reconsider the current set of views in light of the entire workload each time a new request is made is prohibitively expensive. Instead, we consider the workload induced by each individual request in isolation, and compute a set of views to (incrementally) add to the current view set.
2. The view selection process should be tightly integrated with the query-processing component of the ASIS. We achieve this integration by interacting closely with

the query optimizer during candidate view generation, view set selection and view merging.

3. Instead of enumerating candidate view sets before computing their values, we build a good view set with the help of a feedback loop from the query optimizer on the value of partial view sets. This method allows us to apply greedy heuristics to complete view set selection more efficiently.
4. The continuous nature of the process means that views and execution plans that exploit those views are both needed. These are best generated in concert.
5. A specialized query optimization component is used in the process that reuses common optimization work over a large group of smaller optimization problems. We call this process *Multiplex Query Optimization* (or MuxQO). Note that MuxQO differs from Multiple Query Optimization in that the queries being optimized are not expected to run at the same time.
6. Once we have computed the views and plans needed to support the current request, the new views are merged with existing views and the query plans are adjusted to accommodate any resulting view modification. Note that we include this step for completeness. We will give a rough outline of how this step may be accomplished, but it is not our primary focus.

In the remainder of this section we describe this process in detail, and expound on the issues we have raised.

7.4.1 Estimating the ASIS Workload

The generic view selection process, as illustrated in Figure 7-1, takes the future workload of the system as a given. Thus the process must begin with an estimate of the workload, and the effectiveness of the process depends on the accuracy of this estimate. For traditional database systems, data warehouses, OLAP systems or data mining systems, administrators typically employ an iterative approach that combines the anticipated future

needs with the assessment of past system usage to arrive at an estimate. This method is more an art than a science, and it is error prone.

In an ASIS it is equally difficult to apply such methods to approximate all of the requests that a system will see. But as each request is submitted, it provides strong information on the future workload of the system. A request induces a suite of long-lived queries, each of which is executed in response to a specific class of event. Recall that in Paradox we model events as changes at the <agent, service, call> level. Furthermore, service changes only occur within dynamic service calls, and such changes are assumed to be independent of each other.

Assume that for each dynamic service call we define a set of *change types*, $\{t_1, \ldots, t_n\}$. Change types capture that a single service call may support a variety of change events. For example, in an auto-supply-chain application, small inventory changes may occur daily at a dealer's local auto-parts center based on exchanges with regional dealers, while large-scale inventory changes occur monthly, when parts are shipped from a central manufacturer based on projected need. In general, a data-intensive service may encounter different sorts of transactions that are applied to the underlying data, perhaps at differing intervals, and involving different volumes of data; each might be modeled as a different change type. For each t_i, assume we have the following metadata:

1. f_i, the expected *frequency* of t_i.
2. C_i, the expected *cardinality* of the data associated with t_i.
3. d_i, the expected *value distribution* of the data associated with t_i.

Figure 7-3: Change Type Metadata

Note that, as with other metadata in the Paradox system, this information may be provided directly by the agent providing the service, by a directory service, from data gathered locally at the mediator via system monitoring, or from data entered by an ASIS administrator. We assume we can identify any given change by its type when it occurs. Commonly, we expect to see only a small number of change types (often only one) for a

given <agent, service, call> triple. Further, distinct types that have cardinalities and distributions that are very similar can safely be treated as a single type, with their frequencies merged. Also note that, in the absence of distribution information for a change type, we will assume that data associated with the change follows the distribution of data associated with the overall service call. We assume that distribution information is as described in the *Paradox metadata specification* of Chapter 3.

Given the background above, the following definition is central to the estimation of workload in an ASIS.

Definition 7-3 (Expected Workload Induced by a Request): Suppose we are given a request R involving dynamic (agent, service, call) triples $\{(a_1, s_1, c_1), \ldots, (a_n, s_n, c_n)\}$ and, possibly, some group of non-dynamic <agent, service, call> triples, together with metadata, M, modeled as a set of *key-value* pairs, pertaining to all such calls. If for each (a_i, s_i, c_i) there is associated change metadata in the form of change type-frequency-cardinality-distribution 4-tuples, $\{(t_{i,1}, f_{i,1}, C_{i,1}, d_{i,1}), \ldots, (t_{i,m(i)}, f_{i,m(i)}, C_{i,m(i)}, d_{i,m(i)})\}$, as defined in Figure 7-3, then the *expected workload induced by R* is given by

$$\bigcup_{i=1}^{n}\left(\bigcup_{j=1}^{m(i)}\left(\{\Delta R/\Delta c_i\}, f_{i,j}, M \cup \{Card(\Delta c_i) = C_{i,j}, Dist(\Delta c_i) = d_{i,j}\}\right)\right) \qquad (7\text{-}3)$$

where workload is defined as a set of triples, (Q, f, M), where Q is a query, f is the frequency with which Q is executed, and M is associated metadata, modeled as a set of *key-value* pairs, and the $\cup$ operator is a special merge-union operation that combines queries that are executed based on the same change event (details follow). □

Example 7-2 Recall that Example 7-1 involved the request, *ComingSoon*/6, defined by Query (7-1). Each component service call of *ComingSoon*/6 is offered by a single service provider, and each such offering is dynamic. The dynamic calls are:

ClubListing / 4 , *ConcertReview* / 3 , and *MusicForSale* / 4 . Each call has exactly one change type, which we will call *default*. Assume that no specific distribution information is provided for these change types, so this data defaults to existing metadata on its corresponding call. Therefore, with the frequency and cardinality information as given in Example 7-1, and the frequency factor normalized to a two-week time period, we have the following association of change metadata with dynamic calls:

$$\begin{aligned}&(ClubListing/4, \{(default, 14, 400, null)\})\\&(ConcertReview/3, \{(default, 2, 500, null)\})\\&(MusicForSale/4, \{(default, 1, 10, null)\})\end{aligned} \tag{7-4}$$

Further, the suite of queries that defines $\Delta ComingSoon$ with respect to the component service calls are as follows:

$$\begin{aligned}&\Delta ComingSoon(Artist, Club, Date, Rev, CDPurch, \_MC)/\Delta ClubListing \leftarrow\\&\quad \Delta ClubListing(Artist, "MyCity", Club, Date, \_MC)\ \&\\&\quad ConcertReview(Artist, Rev)\ \&\\&\quad MusicForSale(Artist, "R\&B", CDPurch).\end{aligned} \tag{7-5}$$

$$\begin{aligned}&\Delta ComingSoon(Artist, Club, Date, Rev, CDPurch, \_MC)/\Delta ConcertReview \leftarrow\\&\quad ClubListing(Artist, "MyCity", Club, Date)\ \&\\&\quad \Delta ConcertReview(Artist, Rev, \_MC)\ \&\\&\quad MusicForSale(Artist, "R\&B", CDPurch).\end{aligned} \tag{7-6}$$

$$\begin{aligned}&\Delta ComingSoon(Artist, Club, Date, Rev, CDPurch, \_MC)/\Delta MusicForSale \leftarrow\\&\quad ClubListing(Artist, "MyCity", Club, Date)\ \&\\&\quad ConcertReview(Artist, Rev)\ \&\\&\quad \Delta MusicForSale(Artist, "R\&B", CDPurch, \_MC).\end{aligned} \tag{7-7}$$

By Equation (7-6), the workload induced by *ComingSoon* / 6 is given by the set:

$$\left\{\begin{aligned}&(\{\Delta ComingSoon/\Delta ClubListing\}, 14, M \cup \{Card(\Delta ClubListing) = 400\}),\\&(\{\Delta ComingSoon/\Delta ConcertReview\}, 2, M \cup \{Card(\Delta ConcertReview) = 500\}),\\&(\{\Delta ComingSoon/\Delta MusicForSale\}, 1, M \cup \{Card(\Delta MusicForSale) = 10\})\end{aligned}\right\}$$

□

Note that the workload computed by Equation (7-6) is interpreted as a continuously repeating workload. In fact, we assume that it repeats indefinitely. With a request syntax that includes a termination condition as in, for example, the Continual Queries project (Liu, Pu et al. 1998), we might do better than this. But we do not address this issue here. Also note that we do not consider the one-time cost of the initial computation of $ComingSoon/6$. In general, we assume that one-time costs are dominated by long-lived, repeating costs. Furthermore, each change event is assumed to be independent, and so each query group, Q, in the $\langle Q, F, M \rangle$ triples in the workload, is singleton. Deriving the complete workload of an ASIS based on the induced workload for each individual request is simply a matter of taking the union of the individual workloads.

While we can compute the full workload of an ASIS based on the current set of active requests, we reiterate that we do not, in general, want to consider the workload as a whole when performing view selection. Even recent work on automated physical database design, in which indexes and views are selected in a traditional, centralized database context in batch (or off-line) mode, considers each query of an estimated workload in isolation (Chaudhuri and Narasayya 1997; Agrawal, Chaudhuri et al. 2000). Our goal is more demanding in that we seek to update the set of materialized views in a fluid fashion, as requests come to the system. To attempt to reconsider the entire workload each time a new request is made, under these circumstances, would be untenable. We live with something short of global optimality as a result, but we believe the tradeoff is reasonable.

7.4.2 Generating Candidate Views

Much previous work on choosing materialized views in a database system uses *syntactic relevance* as the criterion for candidate view selection, including recent work on view selection in a data-warehousing context (Harinarayan, Rajaraman et al. 1996; Gupta

1997; Gupta, Harinarayan et al. 1997; Shukla, Deshpande et al. 1998; Gupta and Mumick 1999). That is, a syntactic analysis is performed on the queries in the workload to produce a set of candidates. For a select-join query, for instance, each subjoin having every possible subset of the selection conditions might be considered.

Example 7-2 Consider the simple 3-way select-project-join query defined as follows:

$$Ans(X,Y,Z) \leftarrow A("AValue", X) \,\&\, B(X,Y) \,\&\, C(Y,Z). \tag{7-8}$$

Then the set of all views that are syntactically relevant to *Ans*/3 might include the following:

$$\begin{aligned}
&V1(W,X) \leftarrow A(W,X).\\
&V2(X) \leftarrow A("AValue", X).\\
&V3(X,Y) \leftarrow B(X,Y).\\
&V4(Y,Z) \leftarrow C(Y,Z).\\
&V5(X,Y) \leftarrow A("AValue", X), B(X,Y).\\
&V6(X,Y,Z) \leftarrow A("AValue", X), C(Y,Z).\\
&V7(W,X,Y) \leftarrow A(W,X), B(X,Y).\\
&V8(W,X,Y,Z) \leftarrow A(W,X), C(Y,Z).\\
&V9(X,Y,Z) \leftarrow B(X,Y), C(Y,Z).\\
&V10(W,X,Y,Z) \leftarrow A(W,X), B(X,Y), C(Y,Z).\\
&V11(X,Y,Z) \leftarrow A("AValue", X), B(X,Y), C(Y,Z).
\end{aligned} \tag{7-9}$$

□

We endorse syntactic relevance as a baseline for generating candidate views in an ASIS, but the notion of syntactic relevance needs some clarification. We adopt a pragmatic, optimizer-dependent definition. Informally, the set of views syntactically relevant to a query, Q, is the set of logical intermediate results (or *groups*) in the plan space considered in an exhaustive optimization of Q. This informal, "operational" definition has intuitive appeal. A view, V, is only useful in executing a query, Q, if it can be used to execute Q more efficiently in the query evaluator under consideration. Clearly, a pre-materialized intermediate result of any execution plan for Q can be used to

reduce the cost of that execution plan. To the extent that query optimization involves searching a space of potentially optimal plans, the process will produce a good set of candidate materialized views as a by-product.

Our definition also suggests a way to implement candidate view generation: Let the optimizer do it. We propose two variations on this theme. In one variation the optimizer outputs the list of logical groups as a by-product of the normal optimization process. In the second variation the optimizer executes in "logical-only" mode, synthesizing all of the logical groups involved in a complete optimization, but without considering physical plans or costs at all. Note that a cleanly designed optimizer should be easily modified to include either of these functions. The first method has the advantage of overlapping plan generation and candidate view generation. In general, however, we believe that the second method is superior. Techniques that are appropriate and important for pruning the plan search space in a complete optimization may be inappropriate for pruning the space of candidate views. For example, the group pruning process of top-down optimization (described in Chapter 5) can significantly decrease optimization time by pruning away expensive groups, but expensive groups might well correspond to effective materialized views. Further, the expense of enumerating logical groups will generally be small compared to the enumeration and costing of complete physical plans.

An important benefit in using the optimizer to perform candidate view generation is that it integrates candidate view selection with the query-processing component of the system. This integration assures that selected views are computable by, and that they fit naturally with the operators of the query execution engine. The Paradox optimizer accepts a "logical only" flag that invokes candidate view generation as described above. The views produced conform to the capabilities, limitations, and peculiarities of the Paradox execution engine. For example:

- Since source data are distributed, candidate views include "single call" or straight "base-table" views, which are not considered in most view selection work that assumes a local database environment. In Example 7-5, *V1*, *V3* and *V4* are examples of such views.

- Candidate views are computable with respect to source capabilities and binding patterns. In Example 7-5, for instance, if the only acceptable mode for *A/2* is $A(+,?)$, then views *V1*, *V7*, *V8* and *V10* could not be considered.
- Candidate views fit naturally with the operators supported by the *logical algebra* of the query optimizer. For example, if source capabilities support it, semijoin views will be considered. In Example 7-5, for instance, a view such as $V12(X,Y) \leftarrow B(X,Y),C(Y,Z)$ would be a candidate.
- Each candidate view can be produced together with its logical properties, such as cardinality, which is useful if memory is limited or if cache efficiency is a view selection criterion. Interesting physical properties can also be enumerated.

These are all important characteristics of candidate views in an ASIS. Generating candidates with these characteristics outside of the optimizer amounts to reproducing significant optimizer logic, with attendant software maintenance difficulties.

Note that in generating candidate views in an ASIS, we need not consider the entire query suite induced by a request, but only the request itself. This is because each view that is syntactically relevant to some query in the suite is either relevant to the original request, or it involves data associated with a service change event -- data that are not available for view materialization. For example, one query in the suite induced by Query (7-11) is Query (7-13):

$$\Delta Ans(X,Y,Z,\_MC) \leftarrow A("AValue",X) \,\&\, \Delta B(X,Y,\_MC) \,\&\, C(Y,Z). \qquad (7\text{-}10)$$

But only views that involve $\Delta B(X,Y,\_MC)$ are syntactically relevant to this query and not in View Set (7-12). Clearly, we cannot pre-compute views that depend on future service change data. Therefore, we feed the single, original request into the candidate view generator to produce all candidate views for the entire query suite.

Another simple optimization applies where some of the component services in a request are not dynamic. No long-lived query in a query suite involves a call to every dynamic service. Thus any view produced in candidate generation that contains all such calls can be eliminated. For example, if *B*/2 were the only dynamic service call in Query

(7-11), then the only long-lived query in the query suite would be Query (7-13). The set of syntactically relevant views is as follows:

$$\begin{aligned} &V1(W,X) \leftarrow A(W,X). \\ &V2(X) \leftarrow A("AValue",X). \\ &V4(Y,Z) \leftarrow C(Y,Z). \\ &V6(X,Y,Z) \leftarrow A("AValue",X),C(Y,Z). \\ &V8(W,X,Y,Z) \leftarrow A(W,X),C(Y,Z).. \end{aligned} \tag{7-11}$$

This set is equivalent to removing every view from View Set (7-12) that contains a call to *B*/2.

But even given these optimizations, an exhaustive enumeration of syntactically relevant views can produce a large set of candidates. An *n*-way natural-join request in which all service calls are dynamic, for example, results in $O(2^n)$ candidate views. Often we will want to employ additional heuristics to reduce the size of the set, particularly for large joins. Many of the same heuristics routinely applied to query optimization, such as ignoring Cartesian products or applying selection conditions as early as possible, are appropriate. The Paradox optimizer accepts a series of flags that signal the use of various heuristic pruning techniques. If used in conjunction with the "logical groups" flag, the appropriate heuristics are applied to the process of generating candidate views. This approach provides us with knobs to adjust in the view selection process where appropriate.

7.4.3 Building A View Set

Given a set of candidate views, one possible approach to finding a set of views to materialize is to enumerate every possible subset of the candidate set, compute the value of each, and choose the set with the greatest value (possibly within storage bounds or other criteria). The cost of this process is dominated by the cost of value computation, which occurs once for each view set considered and involves running query optimization over a workload of queries. If this exhaustive approach is applied to an *n*-way natural join request where all *n* sub-calls are dynamic and the set of candidate views is itself exhaustively generated, then $n \times 2^{2^n}$ runs of the query optimizer would be required. Such

a doubly exponential algorithm may be acceptable for very small values of n, but clearly it does not scale well.

We propose building a view set in a partially exhaustive, partially greedy manner based on feedback from the query optimizer. The algorithm is shown in Figure 7-4.

```
     ViewSetBuild(Int n, Boolean terminate) {
(1)     leastCost = WorkloadCost(W,{});
(2)     bestSet = {};
(3)     For each subset Sub of CandidateViews
            of size <= n {      // exhaustive to n views
(4)         cost = WorkloadCost(W,Sub);
(5)         if (cost < leastCost) {
(6)             bestSet = Sub; leastCost = cost;
            }
(7)         if (terminate) return bestSet;
        }
(8)     Loop Forever {          // greedy until done
(9)         baseLine = bestSet;
(10)        For each view V in CandidateViews - baseLine {
(11)            Sub = baseLine + V;
(12)            cost = WorkloadCost(W,Sub);
(13)            if (cost < leastCost) {
(14)                bestSet = Sub; leastCost = cost;
                }
(15)            if (terminate) return bestSet;
            }
        }
     }
```

Figure 7-4: Algorithm for Building a View Set

The algorithm as written assumes that the workload, `W`, and the set of candidate views are available. It also assumes that the Boolean flag, `terminate`, becomes `true` when some termination condition is met (e.g., the number of views in the view set has reached a limit, the size of the view set exceeds available memory, etc.). The function `WorkloadCost(W, MVs)`, called on lines `(1)`, `(4)` and `(12)`, does most of the work, computing the cost of executing the workload, `W`, in the presence of materialized views, `MVs`. This function involves running the optimizer over each query in the workload. With an empty view set, as called on line (1), the function matches the first term in the

computation of *VoM* (Equation (7-3)). Subsequent calls to `WorkloadCost()` match the second term of *VoM* computation. The view set that produces the lowest `WorkloadCost()` value is the one with the highest *VoM*. Note that changing the selection criterion to cache efficiency, for example, is simple, provided the estimated sizes of the candidate views are available.

A number of modifications and optimizations can be applied to `ViewSetBuild()`. One class of modifications is based on heuristics that limit how candidate views should be combined into sets. For example, based on the heuristic that network data transfer is dominant in processing a plan suite, we can limit a view set such that it cannot contain any view that can be computed solely from other views in the set. For instance, if Query (7-11) is the base request, no set that includes the subset $\{A, B, (A \,\&\, B)\}$ could be considered.

An important class of optimization involves exploiting upper bounds on the cost of a workload to prune optimization effort. The value of the current best view set, converted as necessary to execution cost, provides an upper bound on the cost of subsequently considered view sets. In computing the value of a new view set, optimization over a workload can terminate if the upper bound is exceeded. For the optimization of each query, the bound can be adjusted based on the frequency of execution for the current query, and thus traversing the workload of queries in order of execution frequency should help to maximize pruning. The bound can be checked after each individual run of the optimizer, or, better yet, it can be integrated into each individual optimizer run. An optimizer with an aggressive pruning strategy has a natural synergy, and a magnified effect, with this process. A top-down optimizer that supports group-pruning can be especially effective (Shapiro, Maier et al. 2001).

Consider how this optimization would work. First, we add an upper-bound argument to the function `WorkloadCost()`. The new call is `WorkloadCost(W,MVs,UB)`, which returns the estimated cost of executing workload `W` with materialized views `MVs`, provided this cost is within the upper bound `UB`. If `UB` is exceeded, the function returns `INFINITY`. In line (1) of Figure 7-4, we call `WorkloadCost(W,{},INFINITY)`. In lines (4) and (12) the call becomes

WorkloadCost(W,Sub,LeastCost). Suppose W = $\{(q_1, f_1), \ldots, (q_n, f_n)\}$, which we keep sorted by decreasing frequency. Then we implement WorkloadCost() as follows:

```
   WorkloadCost(Workload W, ViewSet MVs, Int UpperBound) {
(1)    {(q_{n+1},f_{n+1}),...,(m_{n+m},f_{n+m})} = MaintWorkload(MVs);
(2)    Int totcost = 0;
(3)    for (i=1; i<(n+m); i++) {
(4)       Int cost = Opt(q_i,MVs,(UpperBound-totcost)/f_i);
(5)       totcost += (cost × f_i);
(6)       if (totcost ≥ UpperBound)
(7)          return INFINITY;
       }
(8)    return totcost;
   }
```

Figure 7-5: Algorithm for Computing Workload Cost

Here the maintenance workload, computed on line (1), is mapped to all of the same service change events that are part of the base workload, W (some of the m_i may be empty). The important call here is Opt/3, line (4). Opt(Q,MVs,UB) calls the query optimizer on query set Q, in the presence of materialized views MVs, with cost upper bound UB. If the optimizer cannot find a plan for Q within cost UB it returns INFINITY. An optimizers that supports pruning can seamlessly integrate the upper bound argument to increase its pruning level.

7.4.4 View Merging

Once a view set has been built, the final step is to merge the set with existing materialized views that have been chosen to support other requests. The motivation for this step is straightforward: If two selected materialized views are nearly the same, we do not want to redundantly store and maintain them. This step is where we make allowance for the global sub-optimality of choosing views based on each request in isolation. Of course,

this "allowance" only gets us part of the way toward global optimality, but it does so tractably.

First we have to describe what we mean by "nearly the same". A newly selected view can be merged with an existing one if it references all of the same service calls with identical binary predicates (join conditions) between them.

Definition 7-4 (Mergeable Views): Two views, defined as conjunctive queries with comparison predicates over a set of service calls, are *mergeable* if

1. They contain exactly the same set of service calls.
2. They contain exactly the same binary predicates between service calls. □

Example 7-3 Consider the following three views:

$$V1(X,Y) \leftarrow A("AValue",X) \,\&\, B(X,Y).$$

$$V2(Y) \leftarrow A(X,Y) \,\&\, B(Y,Z).$$

$$V3(X,Y) \leftarrow A("AValue",Y) \,\&\, B(X,Y).$$

Then $V1$ is mergeable with $V2$, but neither $V1$ nor $V2$ is mergeable with $V3$. □

Note that the *mergeable* relation is reflexive, commutative, and transitive.

The goal in merging two views is to produce a merged view that requires minimal space, and from which either original view can be derived at minimal cost. SPJ views that are mergeable may differ in their list of projected attributes or in their selection conditions. Thus if we take the union of the projected attributes and the disjunction of the selection conditions we create a new view that meets the criteria. Note that this is essentially the same process that we applied to merging multiple monitoring conditions in Chapter 3. The algorithm is shown in Figure 7-6.

We assume the input views are mergeable here, and thus the service calls and the join condition are the same for both views. The notion of "canonical form" needs some explanation. The idea is to produce a uniform variable naming scheme by pulling selection and join conditions out of the service calls, ordering the calls and ordering the variables within the calls from left to right based on the service call ordering. Note,

further, that the two original views can easily be obtained from the merged view. Indeed, given the notation of Figure 7-6:

$$V1 = (\{Merge\}, (true), SC1, PAs1)$$
$$V2 = (\{Merge\}, (true), SC2, PAs2)$$

That is, a simple project-select filter over the merged view can reconstitute either input view.

```
Merge(View V1, View V2) {
   Put V1 and V2 in Canonical Form;
   Let V1 = (Calls, JC, SC1, PAs1);
   // Calls, Join Condition, Select Condition, Project Attrs
   Let SAs1 = Selection Attributes of SC1
   Let V2 = (Calls, JC, SC2, PAs2);
   Let SAs2 = Selection Attributes of SC2
   return Merge = (Calls, JC, (SC1 OR SC2), (SAs1 ∪ SAs2 ∪
                   PAs1 ∪ PAs2));
}
```

Figure 7-6: Algorithm for merging two mergeable views

Example 7-4 Suppose we are given the mergeable views, *V1* and *V2*, from Example 7-3. These views have the following canonical forms:

$$V1(X2, X4) \leftarrow A(X1, X2) \,\&\, B(X3, X4) \,\&\, (X2 = X3) \,\&\, (X1 = "AValue").$$

$$V2(X2) \leftarrow A(X1, X2) \,\&\, B(X3, X4) \,\&\, (X2 = X3).$$

Now we have

$$Calls = \{A, B\}$$
$$JC = (X2 = X3)$$
$$SC1 = (X1 = "AValue")$$
$$SC2 = true$$
$$PAs1 = \{X2, X4\}$$
$$PAs2 = \{X2\}$$
$$SAs1 = \{X1\}$$
$$SAs2 = \{\}$$

So we return

$$Merge = \big(\{A, B\}, (X2 = X3), ((X1 = "AValue") \vee (true)), \{X1, X2, X4\}\big)$$

Or, equivalently

$$Merge(X1, X2, X4) \leftarrow A(X1, X2) \,\&\, B(X3, X4) \,\&\, (X2 = X3).$$

Furthermore, to compute *V1* or *V2* from Merge, we have

$$V1(X2, X4) \leftarrow Merge(X1, X2, X4) \,\&\, (X1 = "AValue").$$
$$V2(X2) \leftarrow Merge(X1, X2, X4).$$

□

Note that *Merge*() produces the minimal view that contains the two mergeable input views. We can do so easily because we have limited ourselves to SPJ queries and we have chosen a simple, syntactic basis for view mergeability. Note, also, that the *Merge*() operation is commutative, associative and transitive.

Our approach for merging a set of newly selected views with existing views is as follows: For each new view, V, we search the current set of materialized views for a view, V', that can be merged with V. Since the *mergeable* relation is transitive, we can find at most one such view. If we find one, then we replace it with $Merge(V, V')$. If we do not find one, we simply add V to the set of materialized views.

But one additional task remains. If we add the view $Merge(V, V')$ and use it as a basis for obtaining V and V', then we must modify all queries that use or update V or V' to use $Merge(V, V')$ instead. Let $NMMV = \{V_1, \ldots, V_n\}$ be the set of new mergeable materialized views, and let $OMMV = \{V_1', \ldots, V_n'\}$ be the corresponding set of old

mergeable materialized views. Assume that we have simple view definitions for all V_i and V_i' in terms of $Merge(V, V')$. Then the algorithm of Figure 7-7 makes the proper query and plan alterations.

Our approach to view merging and plan modification is not without drawbacks. The process may involve significant re-optimization costs, and it may compromise the quality of the view selection process. But we believe the benefits of the process outpace its pitfalls. A view can be obtained from a merged view via a local (at the mediator) select-project operation, which is generally cheap. Thus, the effect of our approach on the quality of view selection will be minimal. Moreover, re-optimization can be postponed if necessary until system load is light, or until a query must execute, which can help negate its effect.

```
AdjustForMerges(NMMV, OMMV) {
   for each existing query Q {
      Vs = set of views referenced in Q;
      if ((Vs ∩ OMMV) = {})
         view expand and reoptimize Q;
   }
   for each new query Q {
      Vs = set of views referenced in Q;
      if ((V ∩ NMMV) = {})
         view expand and reoptimize Q;
   }
   for (i = 1 to n)
      replace maintenance queries for V'i with
      new maintenance queries for Merge(Vi,V'i)
}
```

Figure 7-7: Merge Adjustment Algorithm

A number of alternative approaches exist, but most are not attractive. The alternative of performing global optimization for every new request is prohibitively expensive. The alternative of not merging at all is unattractive, since it can result in inefficient space utilization and redundant maintenance work.

Another approach is to begin view selection by rewriting the new request using the current set of materialized views (Halevy, Mendelzon et al. 1995). This approach suffers from the fact that a multitude of rewrites may be possible. View selection would proceed as usual for each possible rewrite separately, which is very expensive.

There is at least one alternative, however, that is viable and deserving of consideration. We can perform view merging on the set of candidate views before building our view set. Then, when computing the costs of the requested workload for a view set that includes one or more merged views, we can subtract the cost of the maintenance queries that will be removed if the view set is chosen. We refer to this approach as *candidate pre-merging*. Candidate pre-merging avoids re-optimization for the queries associated with the new request that access mergeable views. It also provides a more accurate assessment of the true (global) value of views that are added based on a new request. One drawback of the approach, however, is the cost of searching for and performing merges for every candidate view. Another potential drawback is that the revocation of requests can cause trouble. For example, suppose view V1 is optimal for request R1 in isolation, view V2 is optimal for R2 in isolation, but Merge(V1,V3) is chosen for the combined workload of R1 and R2. Now suppose request R1 is revoked. We can either "unmerge" Merge(V1,V3) to be left with V3, or else we must redo view selection for R2 in order to obtain V2. In the former case, we believe that candidate pre-merging will result in a worse materialized view configuration over time when requests are frequently revoked. In the latter case, the costs of request revocation can be very significant, potentially causing a cascade of view re-selection operations.

We are aware that our analysis of merging alternatives is based largely on intuition, and may not be terribly convincing; intuition should be used to ask questions, not to answer them (Patterson 1997). We do believe a detailed analysis of merging alternatives, including rigorous performance tests, is called for. We leave such an analysis for future work.

7.5 Multiple Query Optimization in An ASIS

Another major opportunity for sharing in an ASIS is in sharing intermediate results amongst simultaneously executing queries. Generating plans that exploit this form of sharing is known as Multiple-Query Optimization (MQO). MQO applies to an ASIS in two major areas: within the view selection process for individual requests, and over the entire ASIS workload across individual requests.

7.5.1 MQO Within View Selection

In our previous discussion of the value of materialized views, we assumed that the costs of all queries in a workload are independent. This assumption is embodied in Equation (7-5), which computes the cost of each query separately. For some applications, however, we may know in advance that certain groups of queries always (or often) execute simultaneously. If such query groups share common data, then we may be able to employ multiple query optimization techniques to execute them more efficiently in tandem.

In fact, multiple query optimization opportunities are inherent to the process of view selection in an ASIS. A view, *V*, becomes a candidate for materialization in support of a request, *R*, because it can be used as an intermediate result in computing a query in the query suite induced by *R*. But this property implies that *V* must share one or more service calls with *R*. For each shared service call that is dynamic, a change event for that call will result in the simultaneous execution of a maintenance query on V and a monitoring query on R.

Table 8-1 illustrates this point by showing a generic optimization table for view set selection for an ASIS request. Assume the request is a conjunct of service calls, $S_1 \ldots S_n$. The empty set, for the basic plan suite, followed by *VS1 ... VSn* represent the view sets being considered. The initial, from scratch, computation of the request includes a simultaneous requirement to compute the initial value of the candidate view set, which is

a multiple query optimization problem. For a given candidate view set, *VSi*, and a given change event, ΔS_j, the incremental request computation *Qj,VSi* must be executed, but any maintenance query on the view set induced by ΔS_j, VSi_j, if any, must be executed as well. Again, if VSi_j is non-empty, we have another multiple query optimization problem. Sub-problems of cost-based view selection that are potential multiple query optimization problems are shown in Table 7-1 in parentheses.

	{}	VS1	...	VSn
INIT	$\{Q\}$	$\begin{Bmatrix} Q_{\{VS1\}} \\ VS1 \end{Bmatrix}$	...	$\begin{Bmatrix} Q_{\{VS1\}} \\ VSn \end{Bmatrix}$
ΔS_1	$\{Q_1\}$	$\begin{Bmatrix} Q_{1,VS1} \\ VS1_1 \end{Bmatrix}$	...	$\begin{Bmatrix} Q_{1,VSn} \\ VSn_1 \end{Bmatrix}$
⋮	⋮	⋮	⋮	⋮
ΔS_n	$\{Q_n\}$	$\begin{Bmatrix} Q_{n,VS1} \\ VSn_n \end{Bmatrix}$	...	$\begin{Bmatrix} Q_{n,VSn} \\ VSn_n \end{Bmatrix}$

Table 7-1: Optimization Table for View Selection for an ASIS Request

As further illustration, returning to Example (7-1), if we are maintaining the materialized view, *ReviewAndSaleInfoByArtist*, as defined in Query (7-3), then changes to *ConcertReview* information trigger the execution of two queries simultaneously: one to update the condition, *ComingSoon* , and one to maintain the materialized view, *ReviewAndSaleInfoByArtist*. Using incremental techniques the queries are as follows:

$$\Delta ComingSoon(Artist, Club, Date, ReviewURL, PurchaseURL, \_MC) \leftarrow ClubListing(Artist, "MyCity", Club, Date, \_MC) \,\&\, \Delta ConcertReview(Artist, ReviewURL) \,\&\, MusicForSale(Artist, "R\&B", PurchaseURL). \quad (7\text{-}12)$$

$$\Delta ReviewAndSaleInfoByArtist(Artist, ReviewURL, PurchaseURL) \leftarrow \Delta ConcertReview(Artist, ReviewURL) \,\&\, MusicForSale(Artist, "R\&B", PurchaseURL). \quad (7\text{-}13)$$

Clearly the result of Query (7-16) can be used as an intermediate result in the execution of Query (7-15). One way to execute these two queries in tandem, then, is to first compute Query (7-16), and then join the result with *ClubListing* to obtain $\Delta ComingSoon$. By taking advantage of simultaneous execution to reuse the work of Query (7-16), this multi-plan may be (though it is not necessarily) more efficient than executing the two queries separately.

It is a simple matter to adjust Definition 7-1 to account for knowledge of simultaneous query execution.

Definition 7-5 (Expected Value of Materialized Views with Simultaneous Query Execution): Given

1) A database with schema and related metadata, *S*.
2) A workload, $W = \{(q_0, f_0), \ldots, (q_n, f_n)\}$ over S where the q_i are query sets that execute simultaneously and f_i gives the frequency of execution of q_i over time.
3) A set of materialized views, *V*, with an associated view maintenance workload $X = \{(m_0, g_0), \ldots, (m_n, g_n)\}$ over S where the m_i are maintenance query sets for V, and where f_i gives the frequency of execution of q_i over time. X is a function of V and S.
4) Let $R = \{r_0, \ldots, r_l\}$ be the set of query sets induced by merging elements of W and X that execute simultaneously with associated frequencies $H = \{h_1, \ldots, h_l\}$;

5) A multiple query optimizer *Opt* with an associated cost function, *C*.

The *expected value of materializing and maintaining V* with respect to *W*, *S* and *Opt* is given by:

$$VoM(V,W,S,Opt) = \sum_{i=0}^{n} f_i \times C(q_i,\{\}) - \sum_{i=0}^{l} h_i \times C(r_i,V) \qquad (7\text{-}14)$$

where $C(q,MV)$ is the estimated cost of the optimal multi-plan for query q in the presence of materialized views *MV*, as given by *Opt* over *S*. □

Deriving the complete workload of an ASIS based on the induced workload for a single request is straightforward. Essentially, the complete workload is the union of the workloads associated with each active request. In combining these workloads, however, we must take care that queries associated with the same change event are combined into query sets. We associate a pair, $\langle (agent, service, call), changeType \rangle$ with each $\langle Q, F, M \rangle$ triple that describes a workload. When combining workloads, we first combine elements associated with the same $\langle (agent, service, call), changeType \rangle$ pair by taking the union of the query groups associated with each such element. We then perform a union operation as usual.

Example 7-3 Let *W1*, the workload induced by request, *R*1, and *W2*, the workload induced by request, *R2*, be defined as follows:

$$W1 = \{\langle\langle (a_1, s_1, c_1), t_1\rangle, \langle Q_{11}, F_{11}, M_{11}\rangle\rangle, \langle\langle (a_2, s_2, c_2), t_2\rangle, \langle Q_{12}, F_{12}, M_{12}\rangle\rangle\}$$
$$W2 = \{\langle\langle (a_1, s_1, c_1), t_1\rangle, \langle Q_{21}, F_{21}, M_{21}\rangle\rangle, \langle\langle (a_3, s_3, c_3), t_3\rangle, \langle Q_{22}, F_{22}, M_{22}\rangle\rangle\}$$

Query Q_{11} of $W1$ and query Q_{21} of $W2$ are both triggered by the same change event. Therefore, if the set of current active requests in an ASIS is $\{R1, R2\}$, then the corresponding workload is $W1 \cup W2$, where:

$$W1 \cup W2 = \left\{ \begin{array}{l} \langle\langle (a_1, s_1, c_1), t_1\rangle, \langle Q_{11} \cup Q_{21}, F_{11}, M_{11}\rangle\rangle, \langle\langle (a_2, s_2, c_2), t_2\rangle, \langle Q_{12}, F_{12}, M_{12}\rangle\rangle, \\ \langle\langle (a_3, s_3, c_3), t_3\rangle, \langle Q_{22}, F_{22}, M_{22}\rangle\rangle \end{array} \right\} \text{Q}$$

ueries Q_{11} and Q_{21} have been merged into a single query set. Also note that $F_{11} = F_{21}$ and $M_{11} = M_{21}$, since they are change metadata parameters associated with the same $\langle < agent, service, call >, changeType \rangle$ event. □

Suppose `W = {(q1, f1), . . . , (qn, fn)}`, which we keep sorted by decreasing frequency. Then we implement `WorkloadCost()` as shown in Figure 7-8:

```
   WorkloadCost(Workload W, ViewSet MVs, Int UpperBound) {
(1)    {(m1, f1), . . . , (mn, fn)} = MaintWorkload(MVs);
(2)    Int totcost = 0;
(3)    for (i=1; i<n; i++) {
(4)       Int cost = Opt((qi ∪ mi),MVs,(UpperBound-totcost)/fi);
(5)       totcost += (cost × fi);
(6)       if (totcost ≥ UpperBound)
(7)          return INFINITY;
       }
(8)    return totcost;
   }
```

Figure 7-8: Algorithm for Computing Workload Cost

Here the maintenance workload, computed on line `(1)`, is mapped to all of the same service change events that are part of the base workload `W`. (Some of the m_i may be empty). The important call here is `Opt/3`, line `(4)`. `Opt(Q,MVs,UB)` calls the query optimizer on query set `Q`, in the presence of materialized views `MVs`, with cost upper bound `UB`. If the optimizer cannot find a plan for `Q` within cost `UB` it returns `INFINITY`. An optimizers that supports pruning can seamlessly integrate the upper bound argument to increase its pruning level.

7.5.2 Additional MQO Opportunities in an ASIS

As monitoring requests accumulate over a world of network-based services, disparate requests will tend to overlap. That is, they will share common service calls. In general, we expect requests to grow at a faster pace than the number of accessible services. Indeed, it is only natural that a service will not exist unless it meets the needs of many clients. A significant degree of request overlap is to be expected.

Whenever two requests involve a common dynamic service call, a service change event on that common call will result in the execution of multiple simultaneous monitoring queries. Moreover, such monitoring queries share common data. At the least, they share the data associated with the service change event. Therefore, they present an opportunity for partial result sharing, and for the application of MQO techniques.

Note that the technique of merging monitoring operations in the the Paradox system, as described in Chapter 4, provides the beginnings of this sort of multiple query optimization. But such merging extends only as far as the data associated with the change event. Where multiple requests share more than one service call, opportunities for greater optimization are possible. Note, however, that since we handle cost-based view selection on a request-by-request basis, this form of MQO problem cannot readily be integrated into the view selection process. The fact that we lose out on such MQO problems, however, does not justify attempting global view selection. Instead, a reasonable compromise is to track inter-request MQO problems, prioritize them by service call overlap, and perform re-optimizations in MQO style when system cycles are available to do so.

7.6 Chapter Summary

We have covered a lot of ground in this chapter. We have shown how data and processing effort can be shared across multiple ASIS requests via a view cache. We have described a framework for cost-based view-cache selection in an ASIS that exploits the knowledge of the ASIS workload that comes from the long-lived nature of ASIS requests. But cost-based view selection induces a massive optimization problem. We described a multi-pronged approach to attacking this problem and making it tractable. Finally, we described how many simultaneously-executing queries arise in an ASIS, in particular when caching and the associated coherency demands are considered. These simultaneously-executing queries provide another opportunity for sharing, as they will often share intermediate results. We can exploit this opportunity by using multiple query optimization techniques to optimize and execute them in groups.

But another opportunity for sharing has been left untapped in the techniques we have described in this chapter. As we will see in Chapter 8, a great opportunity for sharing of optimization effort is present in the ASIS view selection problem. We can exploit this opportunity through a novel technique we call Multiplex Query Optimization (MuxQO). Moreover, MuxQO can be applied to a wide range of problems, and can be useful for performing multiple query optimization as well.

Chapter 8

Multiplex Query Optimization

In the previous chapter we described various ways in which sharing could be exploited to improve the performance and scalability of an active service integration system. We discussed an associative caching scheme in which views over service calls are materialized and maintained in support of request processing, and methods for sharing partial results among simultaneously executing queries. Exploiting these forms of sharing presents a massive optimization problem, which, in turn, presents another important opportunity for sharing. This new opportunity involves the sharing of planning effort. In this chapter we discuss a technique for exploiting this opportunity that we call *Multiplex Query Optimization* (*MuxQO*).

The basic idea behind MuxQO is to share optimization work over a number of similar query optimization tasks. In particular, we share the work as embodied in the memo structure or dynamic programming table that is central to the optimization process

(as described in Chapter 6). Note that the problem addressed and the techniques employed in Multiplex Query Optimization are fundamentally different from those of *Multiple Query Optimization* (*MQO*) (Sellis 1988). MQO involves finding an optimal plan for multiple queries that execute simultaneously, with the goal of improving the performance of execution. MuxQO, in contrast, involves sharing optimization work among queries that do not, in general, execute simultaneously, with the goal of improving the performance of optimization. In MuxQO, we push the optimization of a group of queries through a single optimization space; in effect, we *multiplex* queries through the space. In fact, as we will show, MuxQO can be used to speed the optimization phase of MQO.

In its simplest form, a multiplex query optimizer takes a query, optimizes it, and retains the important data structures that store optimization results for the query and its sub-problems. As additional queries are fed to the system, these data structures are reused as much as possible for the new queries, reducing optimization effort. More complex MuxQO systems may perform explicit memory management on their data structures, which can grow very large, or may carefully control the ordering of queries fed to the system to maximize reusable work and minimize memory demand.

MuxQO is effective where a large optimization problem can be transformed to a set of query optimizations that share common sub-problems. Examples include problems that fall into the following three classes:

1) Problems in which the optimization of a group of highly similar queries arises naturally. One example of this scenario is the optimization of monitoring queries in an ASIS. A query suite must be optimized to a plan suite, and the queries involved often have a high degree of similarity.
2) Problems in which a single set of queries is repeatedly optimized in the presence of varying physical database designs, access paths, derived data sets, or data characteristics. The problem of optimizing a single query over two slightly varied access path configurations, for instance, can be transformed into the optimization of two separate but similar queries. This scenario occurs in automated physical database design; in materialized view selection, including view selection in an

ASIS; in the incremental reoptimization of queries based on changes in database statistics, access methods, or other derived data; and in the dynamic reoptimization of queries based on materialized intermediate results.

3) Problems in which a set of queries is optimized for simultaneous execution, and differing shared partial results are considered.

Note that each of these scenarios occurs prominently in an ASIS. In fact, the ASIS materialized view selection problem involves the simultaneous occurrence of all three scenarios: A group of highly similar queries is repeatedly optimized over varying sets of materialized views, the presence of which induces sub-optimization problems involving multiple simultaneously executed queries. We will describe this problem in detail, and in so doing illustrate how Multiplex Query Optimization applies to each of the scenarios listed above. We describe an analytic model of the savings that MuxQO provides in this context based on optimality groups and logical multi-expressions, and we present experimental results that confirm and augment our analysis.

8.1 Optimization Sharing in an ASIS

In Chapter 7 we described the cost-based selection of a set of views to materialize and maintain in an ASIS. Given a workload of requests, an exhaustive approach to this problem involves finding the cost-based value of all possible sets of syntactically relevant views, and choosing the most valuable set (subject, perhaps, to various constraints). The cost of this process is dominated by the cost of value computation, which occurs once for each view set considered, and involves running query optimization over every query in the workload. If this exhaustive approach is applied to a workload of k natural join queries having n dynamic sub-calls each, then $k \times (n+1) \times 2^{2^n}$ runs of the query optimizer are required. Clearly such a doubly exponential algorithm does not scale well.

Defeating this level of complexity demands an integrated, multi-pronged attack, many aspects of which were discussed in Chapter 7. The first prong is to consider each

request in isolation; that simplification eliminates the *k* factor. A second prong involves applying well-known query optimization heuristics to reduce the time of each optimizer run, such as those touched upon in Chapter 6. A third prong is to reduce the number of candidate views by a combination of heuristics and the imposition of limitations in the query processing capabilities of the system. A fourth prong is to further reduce the view sets considered, building a complete view set based on a combination of heuristics and optimizer feedback. We presented an algorithm for building a view set in this manner in Chapter 7. A fifth and final major prong, which we discuss now, is to use multiplex query optimization, in conjunction with the processes of view set building and view merging, to share common optimization work over optimizer invocations.

8.1.1 Optimization Sharing in a Query Suite

As we have noted previously, a request in an ASIS induces a suite of monitoring queries that should be optimized together for repeated execution over time. The queries in a suite have a high degree of similarity, since each monitoring query shares all but one service call with the initial request. It follows that optimizing these queries involves a large degree of overlapping work, since the queries will share many optimality groups amongst them.

Example 8-1 (MuxQO in a Query Suite): Consider, again, Example 7-1, where the initial request is:

$$\begin{aligned} &ComingSoon(Artist, Club, Date, ReviewURL, CDPurchaseURL) \leftarrow \\ &\quad ClubListing(Artist, "MyCity", Club, Date) \,\&\, \\ &\quad ConcertReview(Artist, ReviewURL) \,\&\, \\ &\quad MusicForSale(Artist, "R\&B", PurchaseURL). \end{aligned} \tag{8-1}$$

where *ClubListing*, *ConcertReview* and *MusicForSale* are all dynamic. The query suite induced by this request includes the initial request plus the three monitoring queries:

$$\Delta ComingSoon(Artist, Club, Date, ReviewURL, CDPurchaseURL, \_MC) \leftarrow$$
$$\Delta ClubListing(Artist, \text{"MyCity"}, Club, Date, \_MC) \,\&$$
$$ConcertReview(Artist, ReviewURL) \,\&$$
$$MusicForSale(Artist, \text{"R\&B"}, PurchaseURL). \qquad (8\text{-}2)$$

$$\Delta ComingSoon(Artist, Club, Date, ReviewURL, CDPurchaseURL, \_MC) \leftarrow$$
$$ClubListing(Artist, \text{"MyCity"}, Club, Date) \,\&$$
$$\Delta ConcertReview(Artist, ReviewURL, \_MC) \,\&$$
$$MusicForSale(Artist, \text{"R\&B"}, PurchaseURL). \qquad (8\text{-}3)$$

$$\Delta ComingSoon(Artist, Club, Date, ReviewURL, CDPurchaseURL, \_MC) \leftarrow$$
$$ClubListing(Artist, \text{"MyCity"}, Club, Date) \,\&$$
$$ConcertReview(Artist, ReviewURL) \,\&$$
$$\Delta MusicForSale(Artist, \text{"R\&B"}, PurchaseURL, \_MC). \qquad (8\text{-}4)$$

Recall that in an ASIS such as Paradox, the current value of the request is first computed via the full request, then all future changes to that value are computed via the monitoring queries. Thus we must optimize all of (8-1)-(8-4). Assuming we optimize the queries in order, and that we optimize each query exhaustively, the optimization of (8-2) involves optimizing many of the same groups that were already optimized in (8-1). The optimality groups of query (8-2) are as follows:

$$\Delta ClubListing$$
$$ConcertReview \ \mathbf{x}$$
$$MusicForSale \ \mathbf{x}$$
$$(\Delta ClubListing \,\&\, ConcertReview)$$
$$(\Delta ClubListing \,\&\, MusicForSale)$$
$$(ConcertReview \,\&\, MusicForSale) \ \mathbf{x}$$
$$(\Delta ClubListing \,\&\, ConcertReview \,\&\, MusicForSale) \qquad (8\text{-}5)$$

All of the groups in (8-5) marked with an **x** are also optimality groups of Query (8-1). Note, further, that the set of interesting and relevant properties associated with these common groups is the same for both queries. In optimizing Query (8-2), then, three of the seven sub-problems involved were already completed during the optimization of (8-

1). By using MuxQO we can avoid redoing this work. An analogous situation exists between the optimality groups of (8-3) and (8-1), and those of (8-4) and (8-1).

How much savings does this process give us? Recall, from Chapter 6, that the number of logical expressions (and thus multi-expressions) is a good first-cut measure of the cost of optimization. In this example, four logical expressions are associated with the three groups of query (8-2) that are shared with query (8-1), while the remaining four groups account for 11 logical expressions. The savings should be roughly 4/15, or 27%, in this case. Similarly, the work of 4 of the 15 logical expressions in each of (8-3) and (8-4) can be avoided. Thus, if all 4 queries are processed together, 48 logical expressions must be fleshed out, while 60 logical expressions would be required if the queries are processed separately. This analysis gives us an estimated savings of 25%.

Note, further, that each optimality group shared between any pair of (8-2), (8-3) and (8-4) is also a group for query (8-1). That is, if we let *G(Q)* be the set of optimality groups associated with query Q, then:

$$(G(8-2) \cap G(8-3)) \cup (G(8-2) \cap G(8-4)) \cup (G(8-3) \cap G(8-4)) \subset G(8-1)$$

This containment has implications for memory management in using MuxQO to optimize this query suite. If we optimize query (8-1) and save all optimality group results, then it is not necessary to save any additional optimality group results generated by queries (8-2), (8-3) or (8-4). MuxQO can be a very memory intensive process, so memory management can become important. We will discuss the general issue of reclaiming memory from MuxQO data structures later in this chapter. □

In general, suppose we are given an n-way conjunctive request, Q, in which all service calls are dynamic:

$$Q \leftarrow X_1 \,\&\, X_2 \,\&\ldots\&\, X_n.$$

Assume, for simplicity, that no calls are repeating. That is, $(i \neq j) \rightarrow (X_i \neq X_j)$. We label the monitoring queries induced by Q such that $Q_i = \Delta Q / \Delta X_i$, as follows:

$$Q_1 \leftarrow \Delta X_1 \,\&\, X_2 \,\&\, \ldots \,\&\, X_n.$$
$$Q_2 \leftarrow X_1 \,\&\, \Delta X_2 \,\&\, \ldots \,\&\, X_n.$$
$$\vdots$$
$$Q_n \leftarrow X_1 \,\&\, X_2 \,\&\, \ldots \,\&\, \Delta X_n.$$

Then, since query Q_i $(i = 1 \ldots n)$ shares $(n-1)$ dynamic calls with query Q, it will also share all optimality groups that involve only calls in that shared set. This means that $(2^{(n-1)} - 1)$ of the $(2^n - 1)$ optimality groups for Q_i will be shared with Q, or about 50%. Similarly, Q_i will share all logical multi-expressions with Q that stem strictly from the shared set of calls. In this case, based on our discussion in Chapter 6, $3^{(n-1)} - 2^n + n$ of the $3^n - 2^{(n+1)} + n + 1$ logical multi-expressions associated with Q_i will be shared with Q, which approaches 33% as n increases.

Note, finally, that any optimality group (or multi-expression) in Q_i $(i = 1 \ldots n)$ that is not shared with Q must involve ΔX_i. But since ΔX_i appears only in Q_i, such groups are not shared by any Q_j $(j \neq i)$. Therefore all optimality groups shared by any pair of queries in the query suite are contained in the set of groups associated with Q. That is:

$$\bigcup_{i=1}^{n-1} \bigcup_{j=i+1}^{n} \left(G(Q_i) \cap G(Q_j) \right) \subset G(Q)$$

This containment implies that in applying MuxQO to a query suite, if we optimize query Q first and save all optimality group results associated with Q, then it is not necessary to save any additional optimality group results generated by queries $Q_1 \ldots Q_n$ over multiple optimizations. As we noted in our discussion of Example 8-1, this observation has useful implications for memory management in MuxQO.

8.1.2 Optimization Sharing in View Selection

As we described in Chapter 7, the view selection problem involves repeated optimization of a group of queries over varying view sets proposed for materialization. Our method

for view selection in an ASIS involves computing a good set of views for each request in isolation, and then merging this set with the current set of existing materialized views. In this process, the group of queries that is repeatedly optimized is exactly the query suite induced by the request. For each proposed set of materialized views, then, we find similar potential for optimization sharing as was described in the previous section. But additional sharing across view sets and in the queries for evaluation and maintenance of views can also take place, yielding larger savings than in a single optimization of a query suite.

Recall from Chapter 7 that view selection in an ASIS involves filling in a table of plans and costs for the query (or group of queries) associated with each $(DynamicCall, ViewSet)$ pair. We label the columns of the table with each successive proposed view set (including the empty set). Each column corresponds to the complete query suite evaluated in the presence of the proposed materialized view set, plus queries for creating and maintaining the view set.

Example 8-2 (MuxQO in View Selection): Returning to Example 8-1, assume, for ease of exposition, that we consider a set of three possible candidate views:

$$\begin{aligned} &V1 : ClubListing \\ &V2 : ConcertReview \\ &V3 : (ClubListing \,\&\, ConcertReview) \end{aligned} \tag{8-6}$$

Suppose, further, that we greedily choose a view set of size no greater than 2, and that our first-choice view turns out to be *V1*. Note that in our earlier notation we defined Q_i to be the monitoring query, $\Delta Q/\Delta X_i$. Here we augment this notation. We define $Q_{i,VS}$ to be the monitoring query, $\Delta Q/\Delta X_i$, in the presence of the view set *VS*. For all X_i in the view, *V*, we let $V_{i,VS}$ be the view maintenance query, $\Delta V/\Delta X_i$, in the presence of view set *VS*. Note that in the view set notation we need only list views that are relevant to the query in question. All candidate views must be relevant to the queries in the query suite, but not all candidate views are relevant to each other. For example, *V1* is not relevant to

V2 since the former cannot be used to compute the latter. Both *V1* and *V2*, in contrast, are relevant to *V3*. Table 8-1 below gives the optimization table for this example.

The presence of multiple queries in a table entry indicates that these queries are executed together. We will assume in this section that such query groups are handled in the following manner:

1. Queries in the group are placed in a partial order such that sub-queries are executed before the queries that subsume them.
2. Each query that subsumes other queries in the group is rewritten to use the result of each query that it immediately subsumes. In general, this process may result in multiple rewrites.

Once the rewrites have occurred, the queries in a group can be optimized separately. Where there are multiple rewrites for a single query, the optimal plan over all rewrites is chosen. For example, the cell $(\Delta ClubListing, \{V1\})$ in Table 8-1 contains the query set $\{Q_{1,\{V1\}}, V1_1\}$. If we write V1 as $[ClubListing]$, then the two queries are defined as follows:

	{}	{V1}	{V2}	{V3}	{V1,V2}	{V1,V3}
INIT	$\{Q\}$	$\begin{Bmatrix} Q_{\{V1\}} \\ V1 \end{Bmatrix}$	$\begin{Bmatrix} Q_{\{V2\}} \\ V2 \end{Bmatrix}$	$\begin{Bmatrix} Q_{\{V3\}} \\ V3 \end{Bmatrix}$	$\begin{Bmatrix} Q_{\{V1,V2\}} \\ V1,V2 \end{Bmatrix}$	$\begin{Bmatrix} Q_{\{V1,V3\}} \\ V1,V3 \end{Bmatrix}$
$\Delta ClubListing$	$\{Q_1\}$	$\begin{Bmatrix} Q_{1,\{V1\}} \\ V1_1 \end{Bmatrix}$	$\{Q_{1,\{V2\}}\}$	$\begin{Bmatrix} Q_{1,\{V3\}} \\ V3_1 \end{Bmatrix}$	$\begin{Bmatrix} Q_{1,\{V1,V2\}} \\ V1_1 \end{Bmatrix}$	$\begin{Bmatrix} Q_{1,\{V1,V3\}} \\ V1_1, V3_{1,\{V1\}} \end{Bmatrix}$
$\Delta ConcertReview$	$\{Q_2\}$	$\{Q_{2,\{V1\}}\}$	$\begin{Bmatrix} Q_{2,\{V2\}} \\ V2_2 \end{Bmatrix}$	$\begin{Bmatrix} Q_{2,\{V3\}} \\ V3_2 \end{Bmatrix}$	$\begin{Bmatrix} Q_{2,\{V1,V2\}} \\ V2_2 \end{Bmatrix}$	$\begin{Bmatrix} Q_{1,\{V1,V3\}} \\ V3_{1,\{V1\}} \end{Bmatrix}$
$\Delta MusicForSale$	$\{Q_3\}$	$\{Q_{3,\{V1\}}\}$	$\{Q_{3,\{V2\}}\}$	$\{Q_{3,\{V3\}}\}$	$\{Q_{3,\{V1,V2\}}\}$	$\{Q_{3,\{V1,V3\}}\}$

Table 8-1: Optimization Table for Example 8-2

$$V1_1 = \Delta[ClubListing] \leftarrow \Delta ClubListing.$$
$$Q_{1,\{V1\}} = \Delta Q \leftarrow \Delta[ClubListing] \,\&\, ConcertReview \,\&\, MusicForSale.$$

We assume that $V1_1$ will execute immediately before $Q_{1,\{V1\}}$, and that we can optimize the two queries separately.

Note that this approach is reasonable, and often will be optimal in terms of resource consumption. But in some cases it may be better to employ multiple query optimization techniques for these query groups. We will extend our discussion to include MQO techniques in a later in this chapter.

Fleshing out the optimization table proceeds top-to-bottom, left-to-right. The first column of optimizations is the basic query suite, which exhibits the same properties of optimization sharing that were shown earlier. Each of the monitoring queries, Q1, Q2 and Q3 shares 3 of its 7 optimality groups with Q, encompassing 4 of its 15 logical multi-expressions. Over the entire suite, 9 of 21 optimality groups and 12 of 60 multi-expressions can be shared across optimizations.

Now consider the second column, which involves the following 6 queries:

$$\begin{aligned}
&(1)\ V1 = [ClubListing] \leftarrow ClubListing.\\
&(2)\ Q_{\{V1\}} = Q \leftarrow [ClubListing] \,\&\, ConcertReview \,\&\, MusicForSale.\\
&(3)\ V1_1 = \Delta[ClubListing] \leftarrow \Delta ClubListing.\\
&(4)\ Q_{1,\{V1\}} = \Delta Q \leftarrow \Delta[ClubListing] \,\&\, ConcertReview \,\&\, MusicForSale.\\
&(5)\ Q_{2,\{V1\}} = \Delta Q \leftarrow [ClubListing] \,\&\, \Delta ConcertReview \,\&\, MusicForSale.\\
&(6)\ Q_{3,\{V1\}} = \Delta Q \leftarrow [ClubListing] \,\&\, ConcertReview \,\&\, \Delta MusicForSale.
\end{aligned} \tag{8-7}$$

The queries $Q_{\{V1\}}$, $Q_{1,\{V1\}}$, $Q_{2,\{V1\}}$, and $Q_{3,\{V1\}}$ ((2), (4), (5) and (6) in (8-7), respectively) are precisely the original query suite rewritten to include the proposed view set $\{[ClubListing]\}$. As such, they exhibit the same optimization sharing properties as the query suite itself, which we discussed previously. Note that we need only optimize each of these queries in its rewritten form, since we have already optimized it in its "raw" form, the form in which it does not use the view. If the query is more expensive to

execute using the view than not using the view, then we use the plan generated in the earlier optimization.

In general, a complete optimization of a query using a view set must consider rewrites that employ each subset of the view set. Since our algorithm considers view sets in cardinal order, however, we will have already considered at least some of these rewrites. For example, if the current set size is N and we have exhaustively considered view sets to size N-1, then we will have already considered all rewrites involving proper subsets of the current view set. It remains only to consider a rewrite that involves the complete currently proposed view set. Of course, if we have considered view sets to size N-1 in a partially greedy manner, we may have more rewrites to consider. We may also want to prune certain rewrites heuristically. Regardless of the strategy chosen, this "rewrite sharing" is an important form of optimization sharing. But it is a product of the view sets considered and their order of consideration, not something that requires multiplex query optimization.

But in addition to this sharing amongst the queries in the rewritten suite in the second column, there is group and multi-expression sharing with the original query suite as well. Query $Q_{\{V1\}}$ differs in only one conjunct from the original request, Q, and thus shares three of seven groups and four of fifteen multi-expressions with Q. Query $Q_{2,\{V1\}}$, in addition to sharing all but one conjunct with $Q_{\{V1\}}$ (query (2)), differs by only one (other) conjunct from query Q_2 of the original query suite. The optimality groups of $Q_{2,\{V1\}}$ are listed in (8-7) below:

$$
\begin{array}{l}
[\mathit{ClubListing}] \ \mathbf{x} \\
\Delta\mathit{ConcertReview} \ * \\
\mathit{MusicForSale} \ \mathbf{x} * \\
([\mathit{ClubListing}] \,\&\, \Delta\mathit{ConcertReview}) \\
([\mathit{ClubListing}] \,\&\, \mathit{MusicForSale}) \ \mathbf{x} \\
(\Delta\mathit{ConcertReview} \,\&\, \mathit{MusicForSale}) \ * \\
([\mathit{ClubListing}] \,\&\, \Delta\mathit{ConcertReview} \,\&\, \mathit{MusicForSale})
\end{array}
\qquad (8\text{-}8)
$$

In (8-8), the groups shared with $Q_{\{V1\}}$ are marked with an **X**, and the groups shared with Q_2 are marked with a *. In total, five of the seven groups are shared with previously optimized queries, and seven of fifteen logical multi-expressions are shared. An analogous pattern of optimization sharing holds for query $Q_{3,\{V1\}}$.

In addition, the queries that apply to the creation and maintenance of the proposed view set overlap completely with previously optimized queries. Query V (Query (1) in (8-7)) is completely subsumed by Q, the query corresponding to the original request. Query $V1_1$ (Query (3) in (8-7)) is subsumed by Q_1. In total, disregarding the issue of rewrite sharing, 18 of the 30 optimality groups, accounting for 24 of the 63 logical multi-expressions, associated with this column of queries are shared with previous optimization work.

Note that the third column will exhibit an identical pattern of optimization sharing as the second column just described. Column 4 will follow a similar pattern, but will exhibit some differences, since the view being considered is a join of two dynamic service calls. Here we optimize the following list of queries:

(1) $V3 = [ClubListing \ \& \ ConcertReview] \leftarrow ClubListing \ \& \ ConcertReview.$
(2) $Q_{\{V3\}} = Q \leftarrow [ClubListing \ \& \ ConcertReview] \ \& \ MusicForSale.$
(3) $V3_1 = \Delta[ClubListing \ \& \ ConcertReview] \leftarrow \Delta ClubListing \ \& \ ConcertReview.$
(4) $Q_{1,\{V3\}} = \Delta Q \leftarrow \Delta[ClubListing \ \& \ ConcertReview] \ \& \ MusicForSale.$ (8-9)
(5) $V3_2 = \Delta[ClubListing \ \& \ ConcertReview] \leftarrow ClubListing \ \& \ \Delta ConcertReview.$
(6) $Q_{2,\{V3\}} = \Delta Q \leftarrow \Delta[ClubListing \ \& \ ConcertReview] \ \& \ MusicForSale.$
(7) $Q_{3,\{V3\}} = \Delta Q \leftarrow [ClubListing \ \& \ ConcertReview] \ \& \ \Delta MusicForSale.$

The queries $Q_{\{V3\}}$, $Q_{1,\{V3\}}$, $Q_{2,\{V3\}}$, and $Q_{3,\{V3\}}$ ((2), (4), (6) and (7) in (8-9), respectively) are the original query suite rewritten to include the complete proposed view set $\{[ClubListing \ \& \ ConcertReview]\}$. This query group exhibits similar optimization sharing properties as the query suite itself, except, since the view set includes a join view, the effective number of joined tables is decreased. The modified query suite consists of 2-way joins rather than 3-way joins, and so each monitoring query, $Q_{i,\{V3\}}$, shares one of

three groups and one of four logical multi-expressions with $Q_{\{V3\}}$. Additional optimization sharing exists with the original query suite, which, again, resembles the pattern seen with Columns 2 and 3, except that it applies to queries of decreased join arity. Query $Q_{\{V3\}}$ shares the single group and multi-expression, *MusicForSale*, with query Q. Query $Q_{3,\{V3\}}$ shares the single group and multi-expression, Δ*MusicForSale*, with query Q_3. Finally, as with Columns 2 and 3, the queries that apply to the creation and maintenance of the proposed view set involves optimality groups that overlap completely with previously optimized queries.

In this example, for Column 4, 14 of the 21 optimality groups and 17 of the 28 logical multi-expressions associated with this set of 7 queries have been handled by previous optimizations. In general, as larger joins are considered for materialization, optimization sharing remains significant and can increase as a percentage of total work, but the overall optimization work associated with the column of queries decreases, since the join size of the rewritten query suite is smaller.

Column 5 presents the first case, in our example, of optimization sharing where the proposed view set involves multiple views. Here we must optimize the following set of queries:

$$
\begin{aligned}
&(1)\ V1 = [ClubListing] \leftarrow ClubListing.\\
&(2)\ V2 = [ConcertReview] \leftarrow ConcertReview.\\
&(3)\ Q_{\{V1,V2\}} = Q \leftarrow [ClubListing] \,\&\, [ConcertReview] \,\&\, MusicForSale.\\
&(4)\ V1_1 = \Delta[ClubListing] \leftarrow \Delta ClubListing.\\
&(5)\ Q_{1,\{V1,V2\}} = \Delta Q \leftarrow \Delta[ClubListing] \,\&\, [ConcertReview] \,\&\, MusicForSale.\\
&(6)\ V2_2 = \Delta[ConcertReview] \leftarrow \Delta ConcertReview.\\
&(7)\ Q_{2,\{V1,V2\}} = \Delta Q \leftarrow [ClubListing] \,\&\, \Delta[ConcertReview] \,\&\, MusicForSale.\\
&(8)\ Q_{3,\{V1,V2\}} = \Delta Q \leftarrow [ClubListing] \,\&\, [ConcertReview] \,\&\, \Delta MusicForSale.
\end{aligned}
\qquad (8\text{-}10)
$$

Queries $Q_{\{V1,V2\}}$, $Q_{1,\{V1,V2\}}$, $Q_{2,\{V1,V2\}}$, and $Q_{3,\{V1,V2\}}$ ((3), (5), (7) and (8) in (8-10), respectively) correspond to the original query suite rewritten to include the complete proposed view set, $\{[ClubListing],[ConcertReview]\}$. These queries exhibit the familiar optimization-sharing pattern of the original query suite. But there is additional sharing,

this time from the queries for the proposed view sets $\{[ClubListing]\}$ and $\{[ConcertReview]\}$. Query $Q_{\{V1,V2\}}$ differs by one conjunct from $Q_{\{V1\}}$ and by one (different) conjunct from $Q_{\{V2\}}$, thus five of seven groups and seven of fifteen logical multi-expressions are shared with these two previous optimizations. Query $Q_{1,\{V1,V2\}}$ shares the same amount of optimization work with queries $Q_{\{V1,V2\}}$ and $Q_{1,\{V1\}}$, as does $Q_{2,\{V1,V2\}}$, with queries $Q_{\{V1,V2\}}$ and $Q_{2,\{V1\}}$. Still greater sharing applies to $Q_{3,\{V1,V2\}}$, which involves the optimality groups listed in (8-11).

In (8-11), groups shared with $Q_{\{V1,V2\}}$ are marked with **X**, groups shared with $Q_{3,\{V1\}}$ are marked with *, and groups shared with $Q_{3,\{V2\}}$ are marked with **$**. Here six of seven groups are shared with previously optimized queries, and nine of fifteen logical multi-expressions are shared.

$$
\begin{array}{l}
[ClubListing]\ \mathbf{X\$} \\
[ConcertReview]\ \mathbf{X}\ * \\
\Delta MusicForSale\ *\$ \\
([ClubListing] \,\&\, [ConcertReview])\ \mathbf{X} \\
([ClubListing] \,\&\, \Delta MusicForSale)\ \$ \\
([ConcertReview] \,\&\, \Delta MusicForSale)\ * \\
([ClubListing] \,\&\, [ConcertReview] \,\&\, \Delta MusicForSale)
\end{array}
\qquad (8\text{-}11)
$$

Finally, the queries that apply to the creation and maintenance of the proposed view set are identical to queries seen previously. In total, disregarding sharing of rewrites, 21 of 28 optimality groups and 30 of 60 logical multi-expressions associated with this column of queries are shared with previous optimization work.

The sixth and final column in this example involves a proposed view set with multiple views where one view in the set is a subview of another view in the set. But any reasonable rewrite of the query suite in the presence of this view set will only involve one of the two views in the set. Consequently, these queries will be repeats of queries we

have seen before. To illustrate, the following set of queries represent rewrites using the complete view set:

$$
\begin{aligned}
&(1)\ V1 = [ClubListing] \leftarrow ClubListing.\\
&(2)\ V3 = [ClubListing\ \&\ ConcertReview] \leftarrow [ClubListing]\ \&\ ConcertReview.\\
&(3)\ Q_{\{V1,V3\}} = Q \leftarrow [ClubListing\ \&\ ConcertReview]\ \&\ [ClubListing]\ \&\\
&\qquad\qquad MusicForSale.\\
&(4)\ V1_1 = \Delta[ClubListing] \leftarrow \Delta ClubListing.\\
&(5)\ V3_1 = \Delta[ClubListing\ \&\ ConcertReview] \leftarrow \Delta[ClubListing]\ \&\ ConcertReview.\\
&(6)\ Q_{1,\{V1,V3\}} = \Delta Q \leftarrow \Delta[ClubListing\ \&\ ConcertReview]\ \&\ \Delta[ClubListing]\ \& \qquad (8\text{-}12)\\
&\qquad\qquad MusicForSale.\\
&(7)\ V3_2 = \Delta[ClubListing\ \&\ ConcertReview] \leftarrow [ClubListing]\ \&\ \Delta ConcertReview.\\
&(8)\ Q_{2,\{V1,V3\}} = \Delta Q \leftarrow \Delta[ClubListing\ \&\ ConcertReview]\ \&\ [ClubListing]\ \&\\
&\qquad\qquad MusicForSale.\\
&(9)\ Q_{3,\{V1,V3\}} = \Delta Q \leftarrow [ClubListing\ \&\ ConcertReview]\ \&\ [ClubListing]\ \&\\
&\qquad\qquad \Delta MusicForSale.
\end{aligned}
$$

But clearly in queries $Q_{\{V1,V3\}}$ and $Q_{3,\{V1,V3\}}$ ((3) and (9) above) the sequence ([*ClubListing & ConcertReview*] & [*ClubListing*]) can be reduced to ([*ClubListing & ConcertReview*]), which reduces $Q_{\{V1,V3\}}$ to $Q_{\{V3\}}$ and $Q_{3,\{V1,V3\}}$ to $Q_{3,\{V3\}}$. Similarly, in query $Q_{1,\{V1,V3\}}$ ((6) above) the sequence (Δ[*ClubListing & ConcertReview*] & Δ[*ClubListing*]) can be reduced to (Δ[*ClubListing & ConcertReview*]), which reduces $Q_{1,\{V1,V3\}}$ to $Q_{1,\{V3\}}$. Finally, in query $Q_{2,\{V1,V3\}}$ ((8) above) the sequence (Δ[*ClubListing & ConcertReview*] & [*ClubListing*]) can be reduced to (Δ[*ClubListing & ConcertReview*]), which reduces $Q_{2,\{V1,V3\}}$ to $Q_{2,\{V3\}}$.

Indeed, the only queries listed in (8-12) that have not been optimized previously (or cannot be reduced to a form that has been optimized previously) are queries $V3_1$ and $V3_2$. But these two queries correspond to optimality groups that have been seen before, within queries $Q_{1,\{V1\}}$ and $Q_{2,\{V1\}}$, respectively. The end result is that the optimal plans for all of the queries listed in (8-12) can be found with simple lookups, and no additional optimization work, with the help of multiplex query optimization. Considering only the

new queries, we see sharing of all 6 of the optimality groups and all 8 logical multi-expressions for this proposed view set.

Table 8-2 summarizes the sharing of optimality groups and logical multi-expressions for each view set considered in this example. Rewrite sharing is not

	{}	{V1}	{V2}	{V3}	{V1,V2}	{V1,V3}	Totals
Optimality Groups	9/28	18/30	18/30	14/21	21/28	6/6	86/143
Logical Multi-Expressions	12/60	24/62	24/62	17/28	30/60	8/8	115/280

Table 8-2: Optimization Sharing in Example 8-1

considered. The optimality group savings of 86/143, or about 60%, translates to multiexpression savings of 115/280, or about 41%. These numbers represent an estimate of the savings that can be gained in this example from using multiplex query optimization. We demonstrate the savings experimentally later in this chapter. □

8.2 Optimization Sharing in Multiple Query Optimization

While we have made a point of emphasizing that Multiplex Query Optimization is a completely separate notion from Multiple Query Optimization, MuxQO can be used to speed the process of MQO in much the same way that it can be used for ASIS view selection. Moreover, many of the subproblems encountered while performing MuxQO for ASIS view selection can benefit greatly from MQO techniques, and these techniques can be folded seamlessly into the MuxQO process.

If standard query optimization techniques are used to optimize a group of individual queries, the resultant plans may be suboptimal if the queries are executed simultaneously. Simultaneous execution can cause a *locally suboptimal* plan for one query to be *globally optimal* for the query group if it produces an intermediate result that

can be shared with one or more other queries in the group. The principle of local optimality, which is central to standard query optimization (as we discussed in Chapter 6), breaks down in this event. Standard optimization techniques do not consider this possibility.

Example 8-3 (MQO): As a common illustration of MQO, consider two 3-way joins:

$$Q1 \leftarrow A \,\&\, B \,\&\, C.$$
$$Q2 \leftarrow B \,\&\, C \,\&\, D.$$

Suppose the optimal plan for *Q1* involves joining *A* with *B*, and then joining the result with *C*, while the optimal plan for *Q2* involves joining *B* with *C*, and then joining the result with *D*. If *Q1* is executed together with *Q2*, however, the least cost for executing both queries may be achieved if the join of *B* and *C* is computed once and reused to compute both *Q1* and *Q2*, in the former case by joining the result with *A*, and in the latter case by joining the result with *D*. By computing a common subexpression once we achieve a globally optimal plan, even though the plan used to compute *Q1* is locally suboptimal in this case. □

Multiple query optimization techniques expand standard query optimization methods to account for the possibility of shared intermediate results. This process can be divided into two steps:

1. Identify candidate sets of intermediate results for inter-query sharing.
2. Compute the cost of materializing these intermediate results, and then of computing the set of simultaneously-executing queries in the presence of these materialized intermediate results.

These steps should sound familiar from our discussions of ASIS view selection. Step 1 is similar to the selection of sets of candidate views, and step 2 is similar to computing the

cost of incremental request computation in the presence of candidate view materialization. Indeed, view selection and Multiple Query Optimization are very similar problems. In view selection, views that are subexpressions of one or more requests in a set are considered for long-term materialization and maintenance. Whereas, in MQO, subexpressions of more than one request in a simultaneously executing set are considered for transient materialization.

The phrases emphasized above suggest two simple modifications to our ASIS view selection algorithms in adapting them to Multiple Query Optimization. First, instead of generating candidate views from a single request, we generate them from every request in the set, and we only retain those that apply to at least two requests. Note that, in general, this process should consider not only straight subexpressions of a given query in the set, but also expanded subexpressions that subsume those of more than one query, in a manner similar to the view merging process we described in Chapter 7. Second, since the candidate views are transient and the query set is computed only once, the cost of materializing the views themselves becomes more significant and must be considered, but we need not consider maintenance costs. In this stage MQO becomes a process of optimizing a single group of queries repeatedly in the presence of a changing set of derived relations; a process that can benefit greatly from MuxQO. Because MQO is a doubly-exponential process, it has often been considered impractical. But MuxQO can help to make the process tractable. In fact, MQO can benefit from many of the other techniques we have discussed in the past two chapters as well.

In active service integration, batches of simultaneously executing queries arise naturally, and frequently, in two situations: for an individual request where a change event requires the computation of changes to the request and the maintenance of views materialized in support of the request; and over multiple requests that depend on the same service change event. Furthermore, the benefits of MQO are particularly pronounced in this setting. First, because the same batch of queries will be executed together repeatedly, and so the costs of generating the multiple-query plan can be amortized over multiple executions, and secondly, because an ASIS is typically a network-constrained environment, so the benefit of sharing intermediate results is likely to be large. Inter-

request MQO can be added as a secondary optimization to the techniques we have described, perhaps being applied in the background as computational resources become available. But for individual requests, MQO can be fully and seamlessly integrated with the view selection process.

Example 8-4 (MQO in View Selection): Returning to Example 8-2, recall that we considered a set of three possible candidate views:

$V1 : ClubListing$
$V2 : ConcertReview$
$V3 : (ClubListing \ \& \ ConcertReview)$

Which induced the optimization table shown as Table 8-1. In the original example, we assumed that supplemental view materialization or maintenance was always performed before the request computation itself. But that approach may be suboptimal in terms of overall resource utilization, and it may delay response time for the request update. If we remove this assumption, and instead turn each combination of view maintenance or materialization plus request computation into a Multiple Query Optimization problem, then our optimization table is as shown in Table 8-3.

	{}	{V1}	{V2}	{V3}	{V1,V2}	{V1,V3}
INIT	$\{Q\}$	$\begin{Bmatrix} Q \\ V1 \end{Bmatrix}$	$\begin{Bmatrix} Q \\ V2 \end{Bmatrix}$	$\begin{Bmatrix} Q \\ V3 \end{Bmatrix}$	$\begin{Bmatrix} Q \\ V1,V2 \end{Bmatrix}$	$\begin{Bmatrix} Q \\ V1,V3 \end{Bmatrix}$
$\Delta ClubListing$	$\{Q_1\}$	$\begin{Bmatrix} Q_1 \\ V1_1 \end{Bmatrix}$	$\{Q_{1,\{V2\}}\}$	$\begin{Bmatrix} Q_1 \\ V3_1 \end{Bmatrix}$	$\begin{Bmatrix} Q_{1,\{V2\}} \\ V1_1 \end{Bmatrix}$	$\begin{Bmatrix} Q_1 \\ V1_1,V3_1 \end{Bmatrix}$
$\Delta ConcertReview$	$\{Q_2\}$	$\{Q_{2,\{V1\}}\}$	$\begin{Bmatrix} Q_2 \\ V2_2 \end{Bmatrix}$	$\begin{Bmatrix} Q_2 \\ V3_2 \end{Bmatrix}$	$\begin{Bmatrix} Q_{2,\{V1\}} \\ V2_2 \end{Bmatrix}$	$\begin{Bmatrix} Q_{1,\{V1\}} \\ V3_{1,\{V1\}} \end{Bmatrix}$
$\Delta MusicForSale$	$\{Q_3\}$	$\{Q_{3,\{V1\}}\}$	$\{Q_{3,\{V2\}}\}$	$\{Q_{3,\{V3\}}\}$	$\{Q_{3,\{V1,V2\}}\}$	$\{Q_{3,\{V1,V3\}}\}$

Table 8-3: Optimization Table with MQO Subproblems

In this optimization table, query sets with more than one element are treated using MQO techniques.

Let us consider a single MQO problem in this optimization table, the case where the view set is {V3} and the triggering change event is ΔClubListing (fourth column, second row in the table). Our task is to optimize two simultaneously executing queries, Q_1 and $V3_1$, defined as:

$$Q_1 = \Delta Q \leftarrow \Delta ClubListing \ \& \ ConcertReview \ \& \ MusicForSale.$$
$$V3_1 = \Delta[ClubListing \ \& \ ConcertReview] \leftarrow \Delta ClubListing \ \& \ ConcertReview.$$

The first step is to produce a set of candidate transient materialized views. Such views must be potential subexpressions of at least two queries in the query set. It is easy to see that any subexpression of $V3_1$ is also a subexpression of Q_1, so we need only consider subexpressions of the view maintenance query for transient materialization. We use the same algorithm to generate these candidate views as we do for permanent view selection. Applied to $V3_1$ this procedure yields:

$$TV_1 = [\Delta ClubListing] \leftarrow \Delta ClubListing.$$
$$TV_2 = [ConcertReview] \leftarrow ConcertReview.$$

This set of candidates induces four possible transient view sets. For each set, we compute the optimal plan for materializing the set, and then for computing $V3_1$ and Q_1 in the presence of the materialized views. Again, this process is analogous to the permanent view selection process. It yields the table of "normal" query optimization tasks shown in Table 8-4.

Note that, under the assumption that both permanent and transient materialized views are kept in main memory (whenever possible), many of these optimization tasks are identical to ones that have been encountered before in the permanent view selection process. In particular, the set of tasks $\{V3_1, Q_{1,\{V3_1\}}, TV_2, Q_{1,\{TV_2\}}\}$ fits this category. Still

{}	{TV1}	{TV2}	{TV1, TV2}
$\begin{Bmatrix} V3_{1} \\ Q_{1,\{V3_1\}} \end{Bmatrix}$	$\begin{Bmatrix} TV_{1} \\ V3_{1,\{TV_1\}} \\ Q_{1,\{TV_1\}} \end{Bmatrix}$	$\begin{Bmatrix} TV_{2} \\ V3_{1,\{TV_2\}} \\ Q_{1,\{TV_2\}} \end{Bmatrix}$	$\begin{Bmatrix} TV_{1} \\ TV_{2} \\ V3_{1,\{TV_1,TV_2\}} \\ Q_{1,\{TV_1,TV_2\}} \end{Bmatrix}$

Table 8-4: MQO Optimization Table

other tasks are likely to appear in full as subplans in the MuxQO memo structure (if group pruning has not prevented them from showing up to this point). The tasks $\{TV_1, V3_{1,\{TV_2\}}, Q_{1,\{TV_2\}}\}$ fit this category. Finally, the remaining tasks, $\{V3_{1,\{TV_1\}}, V3_{1,\{TV_1,TV_2\}}, Q_{1,\{TV_1,TV_2\}}\}$ may share common subexpressions with previous optimizations in the view selection context or with each other. This Multiple Query Optimization subproblem folds seamlessly into the MuxQO process, and is reduced in large part to a series of simple table lookups. Other MQO subproblems can be handled in a similar manner, with similar benefit.

In the context of ASIS view selection, groups of maintenance queries and request updates that are executed together in response to a change event can be treated very effectively as MQO problems. Using MuxQO, MQO becomes much more efficient and practical, and the benefit of performing MQO is significant since the resultant plans are executed multiple times, and the network-contrained nature of active service integration systems means that such plans often produce solid cost savings. Note that we have presented an MQO approach that optimizes for total resource consumption. An alternative is to optimize for request response time, since request computation is the primary concern of an ASIS, and cache or view maintenance is not seen directly by the user. One way to attempt to optimize for response time is to optimize the request update query in isolation, then all remaining maintenance queries are optimized using MQO techniques where the set of candidate transient views includes all subexpressions that arise from the request update plan. View maintenance can be significantly delayed when

this approach it taken, however, which in turn may cause an inadvertent increase in response time where request update plans must wait on dependent view maintenance. This danger is particularly acute when system load is high.

8.3 Performance Experiments

In this section we describe performance experiments conducted to support and supplement the analysis presented in this chapter. We have discussed two forms of optimization sharing in the context of view selection in an ASIS: sharing of rewrites and previously optimized queries in their complete form, and sharing of subtasks of the optimization process as embodied by multiplex query optimization. Our experiments focus solely on the impact of MuxQO. Our goal is to measure and compare the performance of ASIS view selection in the presence and absence of MuxQO for a significant range of input requests, to explain the observed measurements, and to draw conclusions about where MuxQO can be effective, and how effective it can be.

This section is laid out as follows: We briefly describe the Columbia Query Optimizer, which served as the test bed for our experiments, and the modifications necessary to turn Columbia into a simple Multiplex Query Optimizer; we describe our experimental design; we present a summary of the measurements we observed using this design; and we analyze the results and attempt to draw conclusions.

8.3.1 The Columbia Query Optimizer

We chose to conduct our experiments using a single optimization framework called Columbia. Columbia is a platform for query optimization research developed by researchers at Portland State University and the Oregon Graduate Institute. Columbia is the successor to, and is based in large part on, the Cascades system developed by Goetz Graefe and colleagues at Portland State University (Graefe 1995). Cascades forms the

basis of the optimization component of commercial database offerings from Microsoft and Tandem Computers (now part of Compaq). The Cascades system, in turn, traces its roots to the Volcano system developed at OGI and the University of Colorado (Graefe and McKenna 1993) and the Exodus system developed at the University of Wisconsin (Graefe and Dewitt 1987).

Columbia is a top-down system. As such, it employs the search process associated with top-down optimization that we outlined in Chapter 6. We believe that the effects of MuxQO are largely independent of whether an optimizer employs top-down or bottom-up search, but we have not verified this hypothesis empirically.

It was a relatively simple matter to turn Columbia into a multiplex query optimizer. In normal operation, Columbia first asserts a root node, which is logically equivalent to the full query. The system then begins the process of generating logical and physical multi-expressions, expanding the memo structure on demand. Once optimization is completed, the memo structure is destroyed and its memory recovered. In multiplex operation, the memo structure is left intact after each optimization. A new root node is created for the next incoming query, assuming the query is not logically equivalent to an existing node in the memo structure that has been accumulated so far. As the new root is expanded, only those sub-expressions that have not previously been considered result in the creation of new memo nodes. Our implementation can change from multiplex mode to normal operation at the flip of a mode bit. A batch of queries can be optimized in normal mode, in multiplex mode, or in any combination.

8.3.2 Experimental Design

We focus on conjunctive requests with no selection predicates, that is, on multi-way equi-join requests. As we have discussed earlier, the join operator is responsible for much of the complexity of optimization in service integration. In general, studying the performance of join optimization is very challenging because of the large dimensionality of the space of possible queries. Despite an extensive literature on the subject, no single, standard benchmark has emerged, but rather a disparate collection of approaches, each

with its own strengths and weaknesses. We find guidance and inspiration in negotiating this territory in the careful analysis provided by Vance (Vance 1998).

One important dimension of variability is the number of joins in a query. For a given fixed join size, n, the cardinalities of each of the n input relations can vary independently. The presence and characteristics of predicates provides further complexity. We limit the set of predicates in our queries to equi-join predicates, but even so, the number and pattern of such predicates can vary enormously. Moreover, for any given pattern, the selectivity of each predicate can vary independently, or arbitrary subsets of the predicates may be correlated. The presence or absence of indexes is yet another dimension in the query space.

The ASIS view selection problem, as we have described it, brings additional dimensions of complexity over "plain" join query optimization. Given a request over n services, the monitoring and processing capabilities of the agents that provide the services become a factor. The number and cardinality of change types for each service call provides another degree of variability. The strategy employed to generate candidate views and combine these into view sets can also vary widely.

Finally, several basic elements of the optimization process can be varied as well. For instance, the choice of cost function can affect the rest of the query optimization process. The optimization search space can be explored bottom-up or top-down. As we have noted, we will look only at a top-down optimizer here. Several parameters of the Columbia optimizer itself can be manipulated. For example, the transformation rule set can be varied, which affects the search space of optimization, and pruning techniques can be enabled or disabled.

Our approach to handling this daunting degree of dimensionality comes in three parts. First, we identify dimensions that are independent of the phenomena we care to measure, and we confirm this independence empirically. Second, we choose not to investigate certain dimensions that would involve a huge amount of additional work, in the interest of expedience. For example, we apply our techniques in the context of a top-down optimizer, but not in the context of a bottom-up optimizer. Third, we parameterize the space of tests with important dimensions and take measurements over a range of

values for each parameter that is sufficient to uncover important trends. In general, we admit to the limitations of our results. Our tests are not exhaustive or definitive, but rather a starting point that illuminates important trends in the value and limitations of Multiplex Query Optimization, and that serves as a context for further exploration.

8.3.2.1 Independent Dimensions

We are focused on measuring the affect of Multiplex Query Optimization. We need not consider dimensions of ASIS view selection that are independent of this technique. Based on our previous analysis in this chapter, the affect of MuxQO is influenced primarily by the set of groups and associated multi-expressions that are expanded by the Columbia optimizer, and the degree to which there is redundancy within this set. Characteristics of the data that affect the cost of query plans should have little or no affect on these factors. The cardinality of services referenced by a request, for example, affects the cost of a plan and its subplans, but it does not affect the set of groups or multiexpressions that must be considered. The same can be said for the selectivity of predicates in a request, and the presence or absence of indexes. There is one caveat to this line of reasoning, however. Parameters that affect the cost of plans can impact the degree of pruning that takes place during optimization. A greater degree of pruning may translate to fewer groups and multiexpressions being created and explored, particularly in the case of top-down optimization (Shapiro, Maier et al. 2001). But we expect such pruning to have little affect on the degree of sharing in MuxQO. To the degree that groups and multiexpressions are pruned away, they are likely to contain shared and unshared sub-problems in equal proportion to that of the entire set of groups and multiexpressions associated with a request. Thus, the percentage of savings attributable to MuxQO should not change as such parameters are varied. By similar reasoning, varying the cost function within Columbia, or enabling or disabling various forms of pruning, should have no significant affect on our results.

We have confirmed that our reasoning is borne out in practice via a series of simple tests. We independently varied the range and distribution of cardinalities of

service calls, service changes, and the selectivity of join predicates for a range of query topologies, join sizes and view selection strategies (i.e., for a number of tests with otherwise identical characteristics to those that we report on below), and we observed near-identical results for percentage of MuxQO savings. Likewise, we did the same while varying the cost function used by Columbia, and independently enabling and disabling each of three varieties of pruning ("group pruning", "epsilon pruning" and "cucard pruning") supported by Columbia, with similar results. We do not report these numbers here.

8.3.2.2 Other Fixed Dimensions

There are a number of other dimensions that we leave fixed in our experiments below, but we are not so sanguine, in some cases, that the affects of MuxQO are independent of them. We consider only the problem of ASIS request optimization and supplemental view selection. As has been described in detail in this thesis, MuxQO is particularly well-suited to this problem. We have suggested that a number of other problems can benefit from MuxQO as well, we have not demonstrated that hypothesis empirically.

As we have noted, all of our tests use the Columbia optimizer, which employs a top-down search of the plan space. Given a common plan space, we expect that the effect of MuxQO is independent of the order in which the space is searched, and therefore our techniques would apply with equal efficacy to top-down or bottom-up optimization. It is more questionable whether MuxQO would be an appropriate technique in an optimizer that employs randomization techniques, hill-climbing, simulated annealing or the like.

We use a fixed set of transformation rules in the Columbia optimizer. These include only those logical transformation rules required to exhaustively search the space of bushy join trees, and implementation rules that assume two possible physical join algorithms: merge join, and dependent nested-loops join. The logical rules, in particular, affect the size of the search space dramatically. Varying the rule set to search only left-deep query plans, for example, would significantly decrease optimization time for a given request.

We assume that each service call named in a request is provided by exactly one agent. We assume that each such call is dynamic, and that it has a single change type associated with it that occurs with a frequency of 1. We assume that each service call can accept any binding pattern, and that all possible agent capabilities are available, so that such capabilities are not an issue in the optimization. We are essentially removing the issue of agent heterogeneity from our tests. None of these factors seem likely to influence the effects of MuxQO significantly.

Finally, we fix the algorithm for creating the set of candidate views in our experiments. We consider the set of candidates that contains all straight service calls and all 2-way joins between service calls where a join predicate exists. This algorithm may not be particularly desirable in terms of the optimal view set it produces. But in terms of optimization time, this set remains reasonably simple, yet it captures important properties of any more exhaustive set of candidates. Note that we would expect, and in fact we observe, that optimization subproblems in the view selection problem that involve queries of lower arity than the initial request contribute relatively little to overall optimization time. For example, rewriting a n-way join to use a view that is a join of 2 of the n tables yields a (n-1)-way join. Such (n-1)-way (and smaller) join queries tend to be dominated by larger queries in the overall set of optimizations in view set selection. If we considered more exhaustive sets of candidate views, for example, 3-way and 4-way joins, then the additional queries that we must optimize in view selection would not contribute greatly to overall optimization time. We claim, then, that the candidate generation algorithm we use produces MuxQO behavior that is representative of that produced in more exhaustive candidate sets.

8.3.2.3 Dimensions Chosen

Our tests examine three primary dimensions in the ASIS view selection problem space: the arity of the join request, the pattern of join predicates, and the strategy used to build a view set from a given set of candidates.

Previously we have shown that an exhaustive approach to ASIS view selection is doubly-exponential in request arity, so it follows that request arity is an important variable to look at. We start with 3-way join requests, and work our way up as high as 10-way joins, depending on other variables. This range of join sizes falls well within the range of realistic queries, and allows us to see clear trends as requests grow larger.

In exploring patterns of join predicates, we borrow from previous work in join optimization (e.g., (Vance 1998) and (Steinbrunn 1996)) and look at a variety of join topologies. Note that join topologies derive from the notion of *join graphs*, which provide a way of representing the pattern of predicates among relations in a query. A join graph for a query is created by representing each relation in the query as a node in the graph, and then, for each join predicate in the query between relations `R1` and `R2`, place an undirected edge in the graph from node `R1` to node `R2`. In our experiment we look at three join graph topologies: the *Star*, the *Chain*, and the *Clique*. In a Star query, a join predicate is present from a single "hub" relation to each of the other relations in the query, so that the resultant join graph takes a star shape. In a Chain query, the join graph is connected, all but two nodes have precisely two edges, and the two remaining end nodes have one edge each (the graph looks like a chain). In a Clique query, a predicate is present between each pair of relations, creating a maximally connected graph. Figure 8-1 shows the query graphs for a Star, a Chain and a Clique query, each having arity 5.

Note that join graphs are interesting from the point of view of optimization testing because optimization complexity often varies with join graph topology. A graph with a large diameter, such as a Chain, tends to be easier to optimize than a graph with a small diameter, such as a Star. A graph with a low *density* (i.e., a small number of predicates), such as a Chain or a Star, is generally easier to optimize than a high-density graph, such as a Clique (Vance 1998). By choosing Chains, Stars and Cliques, we succinctly capture

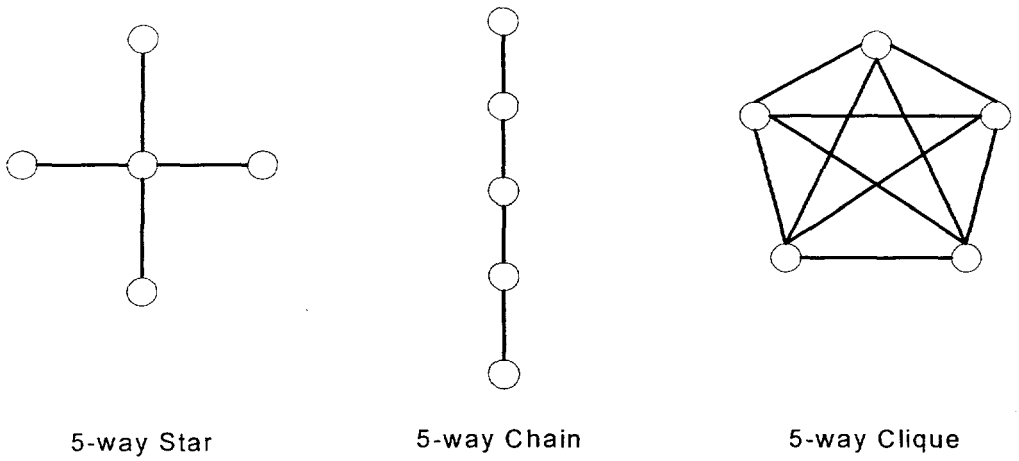

Figure 8-1: Query Topologies

a range over these dimensions (note that high density and high diameter are difficult to find in combination). Note, further, that given our fixed algorithm for generating candidate views, which includes all 2-way joins in which a join predicate exists, join graph topology will influence the size of the candidate set. Higher density corresponds to a larger number of candidate views.

High Diameter	Chain	
Low Diameter	Star	Clique
	Low Density	High Density

Table 8-5: Query Graph Density and Diameter

For each combination of join size and topology, we measure the value of MuxQO in optimizing the basic query suite, with no consideration of supplemental materialized views. Next, we measure the value of MuxQO where view sets are considered. We test a variety of strategies for building a set of views from a set of candidate views. All strategies use a heuristic for rejecting possible view set combinations that we call the *no-overlap* heuristic. The no-overlap heuristic says that a view, V, can be added to a set, S, if

V brings data to the mediator that may contribute to the current request result, but that is not already in *S*. For example, if the request is a three-way join `ABC` with candidate view set `{A, B, C, AB, BC}`, and the current view set is the singleton set `{AB}`, then we may add either the view `BC` or the view `C`, but we may not add the views `A` or `B`. This heuristic is reasonable, in particular, if we assume that network communications dominates the cost of data integration.

Given the no-overlap heuristic, we consider three view-set building strategies in our measurements:

1. **Greedy Singleton**: The *Greedy Singleton* method computes the value of each view set of size one. Based on these values, it heuristically combines singleton sets, adding the most valuable remaining singleton item that does not violate the no-overlap heuristic to the current set, until no such addition is possible.
2. **Greedy Incremental**: The *Greedy Incremental* method repeatedly generates the best view set of size n+1 by computing the value of adding each remaining view to the current best view set of size n, provided the new view set does not violate the no-overlap heuristic.
3. **Exhaustive**: The *Exhaustive* method computes the value of every combination of candidate views that does not violate the no-overlap heuristic.

Clearly, these three strategies are ordered by increasing thoroughness, and also by increasing complexity. We envision a strategy similar to our exhaustive approach being used for requests of small arity. But the exhaustive approach becomes infeasible as request size gets large. For larger requests, an ASIS would likely want to switch to something resembling one of our greedy strategies.

8.3.3 Experimental Environment and Results

Here we present lower-level details of our experiments, as well as our experimental results.

8.3.3.1 Computing Environment

Our performance experiments were run on a 233 MHz Pentium II processor with 128 Megabytes RAM running the Windows NT Workstation 4.0 operating system. Unlike the execution model that we described in Chapter 7, we used the Paradox optimization engine to generate batches of test queries based on an initial request, which we then translated to the form required by the Columbia optimizer. Our tests are based on using Columbia to optimize the test queries in "normal batch mode," and then running the same queries under "MuxQO mode," except as noted below. Query generation time is not considered in our measurements. In fact, it is very small compared with optimization time. Each experiment was run 5 times, and the average result for each measure is reported below. (In fact, there was no significant variance between runs for any of the experiments reported.)

8.3.3.2 Handling Memory Limitations

Note that MuxQO requires a tradeoff between the memory footprint of optimization and optimization time. If main memory is exceeded during MuxQO, the performance benefits of the technique will quickly deteriorate. Our tests assume the presence of sufficient main memory to handle MuxQO. For large requests, however, our test environment did not have sufficient memory. Yet we were determined to gather useful large-query numbers. To do so, we tweaked our test to greatly shrink its memory footprint and provide a simulation of the benefits of MuxQO for large queries, given sufficient main memory. Our modified test works as follows: For each query in the test suite, we execute a partial MuxQO batch that contains only those queries that have run previously and that can share partial results with the target query, followed by the target query itself. We record only the numbers for the target query. When using this approach for an arity 10 request, for example, instead of executing literally thousands of queries in a single MuxQO batch, we never need to execute a batch larger than 10 queries.

Note that while a naive implementation of MuxQO can be very memory-intensive, there is hope for reducing memory usage for cases in which main memory is limited. At the end of this chapter we briefly discuss an idea for reducing the memory footprint of MuxQO that we call *MuxQO Garbage Collection*. We have not implemented this technique, however.

8.3.3.3 Optimizer Settings and Data Characteristics

For all of our experiments, the Columbia optimizer was run in "Release" mode (optimized Microsoft Visual C++ code) with group pruning enabled. The corresponding catalog gives every basic service call (relation) a uniform cardinality of 2000, one change type for each service call (in the form of an auxiliary "delta relation") with frequency 1 and cardinality (also) 2000, and every join predicate a uniform selectivity of 0.00316. (Cardinality and selectivity were essentially chosen at random.) We use a simple cost model that considers both CPU and I/O costs, and a simple rule set that explores all bushy join trees, and assumes two possible physical join operators: merge join and nested-loops join. As noted in Section 8.2.2.1, experiments show that these variables are independent of the effects of MuxQO.

8.3.3.4 Results

We begin by presenting results for running basic query suites. That is, no view selection is considered. Since the query sets are relatively small in this case, we start at a relatively high join arity of 6, and then move to joins of arity 8 and arity 10. We consider Chain queries, Star queries and Clique queries for each of these arities. We present the number of queries in the suite, the total number of tasks (Tasks) processed, the number of optimality groups considered (Groups), the number of multiexpressions created (Mexprs), and the total elapsed time in the optimization of the entire suite. Note that we explained the term *multiexpression* in Chapter 6. Tasks are discrete chunks of

optimization work as seen internally by the Columbia optimizer. Tasks are a finer-grained measure of the work done by the optimizer than multiexpressions.

Raw numbers are presented in Table 8-5 as well as the percentage savings attributable to MuxQO. A graph summarizing the percentage savings for MuxQO in each category is shown in Figure 8-2.

Request	Num	Groups			Tasks (000's)			MExprs (000's)			Time (secs)		
		Raw	Mux	Save	Raw	Mux	Save	Raw	Mux	Save	Raw	Mux	Save
Chain-6	7	441	255	42%	51.2	35.2	31%	16.8	12.1	28%	1.41	1.07	24%
Star-6	7	441	255	42%	50.9	34.3	32%	16.5	11.6	29%	1.39	1.04	25%
Clique-6	7	441	255	42%	50.0	34.7	31%	16.7	12.1	28%	1.35	1.08	20%
Chain-8	9	2295	1279	44%	852	565	34%	214	149	30%	18.5	13.0	30%
Star-8	9	2295	1279	44%	862	552	36%	210	143	32%	21.1	14.0	34%
Clique-8	9	2295	1279	44%	823	551	33%	212	148	30%	18.4	12.9	30%
Chain-10	11	11,253	6143	45%	13,821	8,921	35%	2,450	1,678	31%	296	224	24%
Star-10	11	11,253	6143	45%	14,611	8,953	39%	2,411	1,601	34%	505	329	35%
Cliq-10	11	11,253	6143	45%	13,180	8,596	35%	2,414	1,661	31%	286	218	24%

Table 8-6: Basic Query Suites

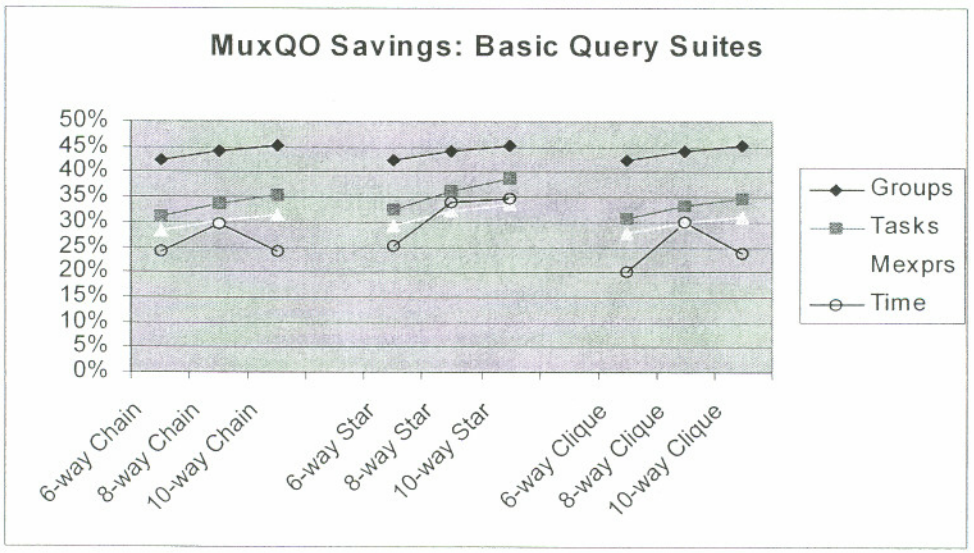

Figure 8-2: MuxQO Savings for Basic Query Suites

Next we present results for running view selection using the greedy singular heuristic for view set enumeration. We anticipate that this heuristic method will be effective for queries of relatively high arity. We start these results at arity 6, and move,

Query	Num	Groups			Tasks (000's)			MExprs (000's)			Time (secs)		
		Raw	Mux	Save	Raw	Mux	Save	Raw	Mux	Save	Raw	Mux	Save
Chain-6	105	4074	1439	65%	420	208	50%	140	76	46%	13.76	9.34	32%
Star-6	105	4074	1439	65%	417	206	50%	137	74	46%	13.67	9.09	34%
Clique-6	239	6024	2101	65%	530	268	49%	184	101	45%	20.1	14.5	28%
Chain-8	173	27,846	9151	67%	9,142	4181	54%	2362	1196	49%	203	106	48%
Star-8	173	27,846	9151	67%	9,235	4192	55%	2311	1161	50%	229	117	49%
Clique-8	404	49,371	15,934	68%	13,064	6153	53%	3612	1844	49%	306	161	47%
Chn-10	257	169,874	53,247	69%	182966	78724	57%	33593	16335	51%	3,933	2,254	43%
Star-10	257	169,874	53,247	69%	192635	81554	58%	32997	15835	52%	4,153	1,993	52%
Cliq-10	725	354,158	108,651	69%	291195	130272	55%	59112	28904	51%	6,340	3,706	42%

Table 8-6: Greedy-Singular View Selection

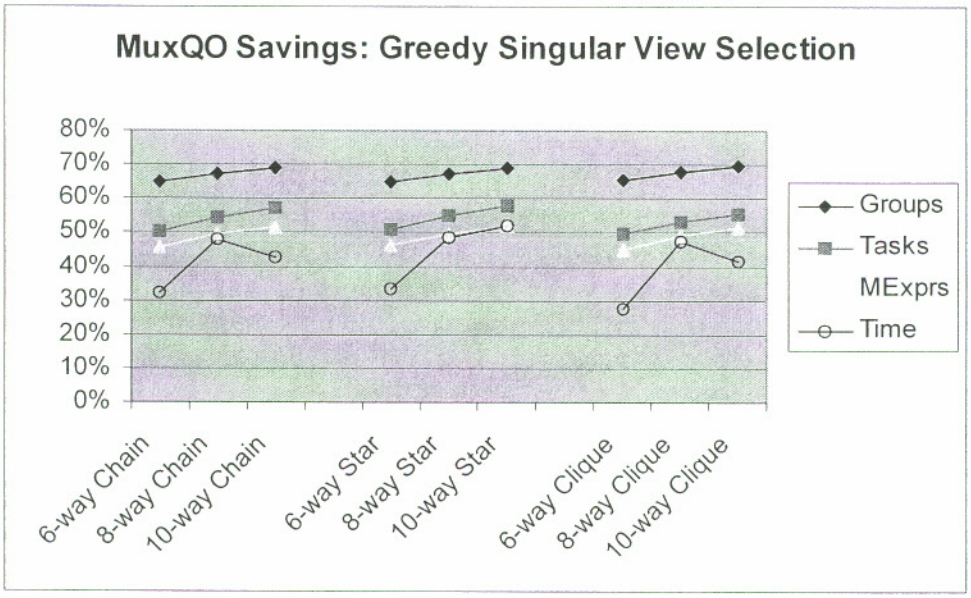

Figure 8-3: MuxQO Savings for Greedy Singular View Selection

once again, to joins of arity 8 and arity 10. We consider Chain queries, Star queries and Clique queries for each of these arities. Raw numbers are presented in Table 8-6, as well

as the percentage savings attributable to MuxQO. A graph summarizing the percentage savings for MuxQO in each category using greedy-singular set enumeration is shown in Figure 8-3.

Next we show view selection results for greedy-incremental set enumeration. Greedy-incremental set enumeration is significantly more complex then greedy-singular enumeration. We expect optimization times to be large for large join arities using this method. We show results ranging from arity 4 to arity 6 and arity 8. We consider Chains, Stars and Cliques for each of these arities. Raw numbers are presented in Table 8-6, as well as the percentage savings attributable to MuxQO. A graph summarizing the percentage savings for MuxQO in each category using greedy-incremental enumeration is shown in Figure 8-4.

Query	**Num**	**Groups**			**Tasks** (000's)			**MExprs** (000's)			**Time** (secs)		
		Raw	Mux	Save	Raw	Mux	Save	Raw	Mux	Save	Raw	Mux	Save
Chain-4	78	789	261	67%	26.2	12.5	52%	10.2	5.5	46%	3.67	3.42	6.8%
Star-4	83	944	291	69%	33.1	14.8	55%	12.8	6.6	49%	3.95	3.60	8.9%
Clique-4	124	1,195	378	68%	37.5	17.4	54%	15.4	8.0	48%	5.68	5.28	7.0%
Chain-6	252	9,415	2,465	74%	874.9	356.1	59%	295.9	135.7	54%	28.3	17.4	38%
Star-6	210	10,689	2,639	75%	1,181.6	434.8	63%	385.2	163.2	58%	34.5	18.0	48%
Clique-6	461	14,942	3,955	74%	1,211.8	518.8	57%	425.6	201.0	53%	41.9	26.7	36%
Chain-8	533	93,150	20,391	78%	29,344	9,935	66%	7,655	3,005	61%	637	249	61%
Star-8	425	92,106	19,903	78%	33,389	10,431	69%	8,196	3,063	63%	809	287	65%
Clique-8	1,304	155,643	37,512	76%	36,490	14,662	60%	10,321	4,523	56%	818	364	55%
Chain-10	4,905	330,042	29,757	91%	48,183	9,073	81%	12,653	3,491	77%	1109	263	76%
Star-10	6,369	464,898	51,576	89%	65,189	13,929	79%	20,148	5,256	74%	1595	401	75%

Table 8-7: Greedy-Incremental View Selection

The final results we present are for exhaustive view selection. Recall that even when considering views exhaustively, we still maintain the heuristic of non-overlap, and we consider only single tables and two-way join views where a join predicate exists (i.e., no Cartesian products). But despite these restrictions, greedy exhaustion can become

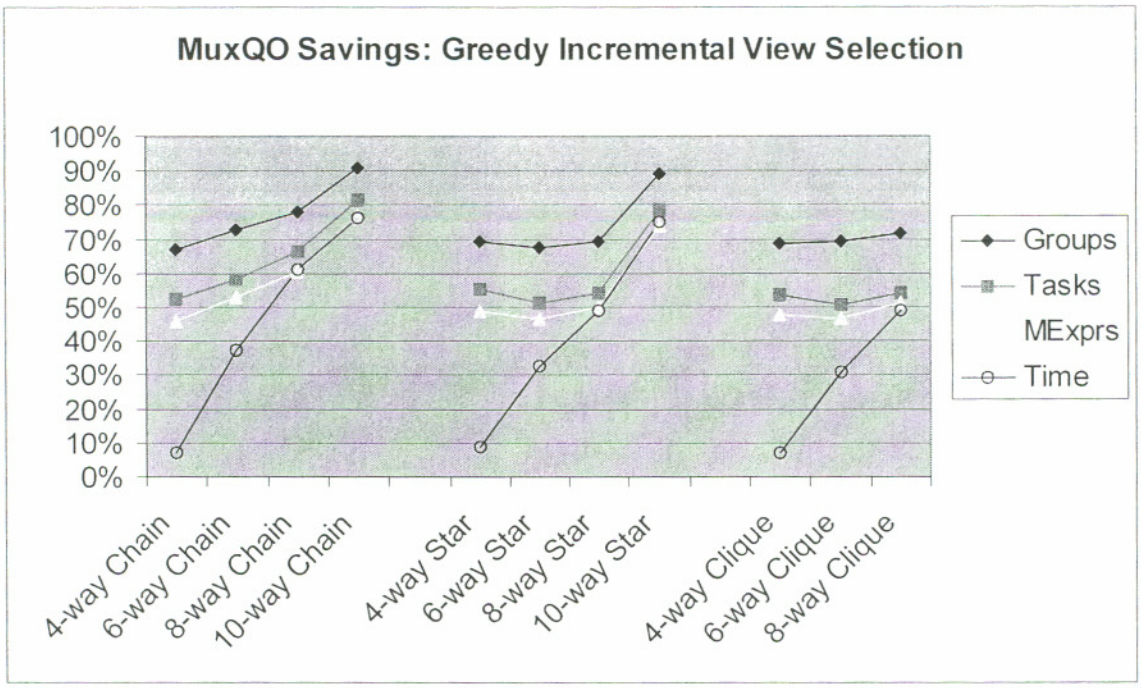

Figure 8-4: MuxQO Savings for Greedy-Incremental View Selection

Query	Num	Groups			Tasks (000's)			MExprs (000's)			Time (secs)		
		Raw	Mux	Save	Raw	Mux	Sav	Raw	Mux	Save	Raw	Mux	Save
Chain-4	183	1,824	444	76%	57.8	22.4	61%	22.7	10.6	54%	8.12	7.73	4.8%
Star-4	188	1,879	476	75%	59.2	23.2	61%	23.2	10.9	53%	8.35	7.77	6.9%
Clique-4	419	3,400	886	74%	90.7	38.7	57%	37.7	18.5	51%	17.2	16.5	3.9%
Chain-5	551	10,684	1,853	83%	562.6	172.8	69%	207.3	77.0	63%	23.3	15.2	35%
Star-5	599	11,788	2,244	81%	606.3	194.4	68%	223.2	85.8	62%	23.7	14.9	37%
Clique-5	3,011	44,872	8,726	81%	1,839	693	62%	706.4	302.9	57%	59.3	35.1	41%
Chain-6	1,652	60,263	7,508	88%	5,270	1,270	76%	1,802	529	71%	129.3	43.2	67%
Star-6	1,946	73,913	10,699	86%	6,233	1,631	74%	2,123	669	69%	157.8	54.1	66%
Clique-6	24,948	690,960	102,473	85%	46,121	15,260	67%	16,379	5,963	64%	1,114	430	61%
Chain-7	4,905	330,042	29,757	91%	48,183	9,073	81%	14,954	3,491	77%	1,109	263	76%
Star-7	6,369	464,898	51,576	89%	65,189	13,929	79%	20,148	5,256	74%	1,595	401	75%

Table 8-8: Exhaustive View Selection

very time consuming as joins get large, particularly where the query graph is dense, as in the case of clique queries. We show results beginning at arity 4, and increasing in single steps to arity 5, arity 6, and arity 7. We consider Chains and Stars for all of these arities, but we consider Cliques only for arities 4-6. Arity 7 Cliques became prohibitively expensive, both in terms of time consumption, and the number of queries and amount of data generated in the test. Raw numbers are presented in Tables 8-8, as well as the percentage savings attributable to MuxQO. A graph summarizing the percentage savings for MuxQO in each category using greedy set enumeration is shown in Figure 8-5.

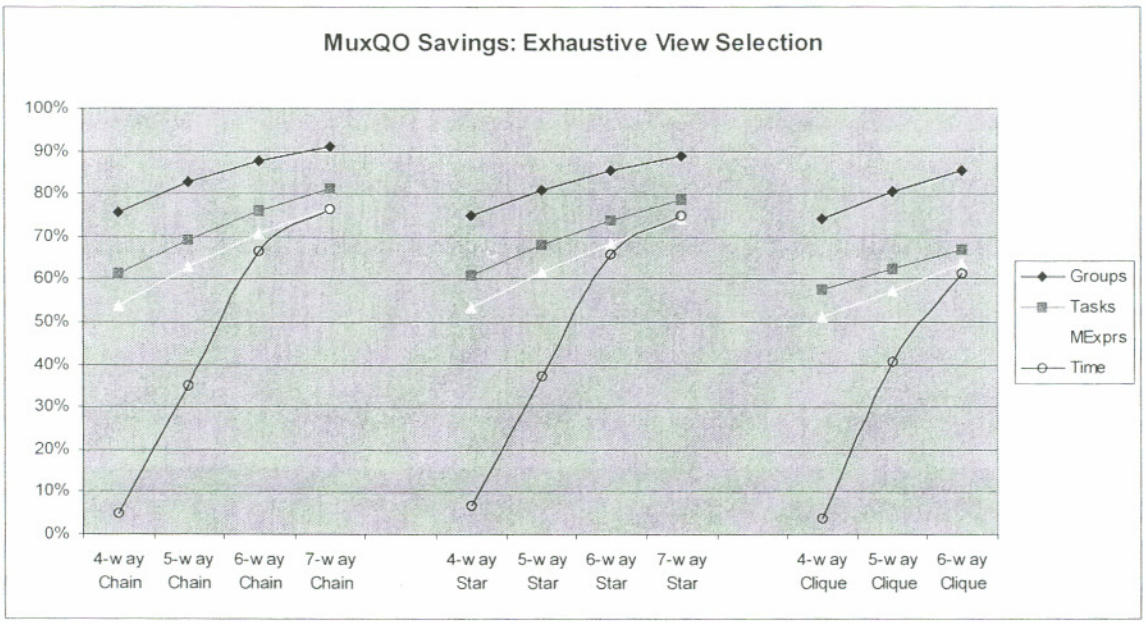

Figure 8-5: MuxQO Savings for Exhaustive View Selection

8.3.4 Discussion

To a large degree the performance numbers above speak for themselves; we conclude this chapter by highlighting some of what they say. First, the benefits of MuxQO for low-arity requests are relatively modest. In particular, start-up costs and

other overheads associated with MuxQO absorb much of its positive impact when the problem being tackled is small. For 4-way joins of varied topology, even where exhaustive view selection is employed, task savings in the 57%-62% range translate to time savings merely in the 4%-7% range. Based on these numbers, it is questionable whether MuxQO is worth the trouble for requests that join information from just a few sources. However, the resource costs of applying MuxQO are also small in such cases.

On the other hand, as joins become large, especially where a thorough search of the space of plans and candidate view sets is desired, the savings attributable to MuxQO is dramatic, and the technique appears to be indispensable. For exhaustive view selection, for example, the percentage time savings due to MuxQO increases by leaps and bounds as join arity increases. While savings are modest for arity 4, they increase to 35%-37% for arity 5, 61%-67% for arity 6, and 75%-77% for arity 7. MuxQO has a dampening effect on the doubly-exponential complexity of view selection. One way to view this benefit is as a time savings that can free a server to work on other tasks. For a 10-way star query using greedy-incremental view selection, for instance, the 42% savings due to MuxQO translates to roughly 45 minutes of compute time in our environment. Alternatively, MuxQO might be viewed as enabling a more thorough search, and more efficient execution, for the same amount of effort. For instance, our numbers show that using MuxQO to process arity 8 requests of varying topology allows us to do a Greedy-Incremental view selection at a relatively small premium (~15%) over the costs of performing a Greedy-Singular view selection without using MuxQO.

Another point to note in our numbers is the interaction of join topology and view selection heuristics. Increasing predicate density, in particular, can combine with other view selection heuristics to produce large fluctuations in the number of candidate view sets for consideration. For example, when applying the greedy-singular heuristic, optimizing a 6-way chain query (no MuxQO) involves 421K tasks, while optimizing a 6-way clique (no MuxQO) involves 530K tasks; an increase of 26%. The greedy-incremental heuristic, in contrast, results in 874K tasks for a 6-way chain query, and 1,212K tasks for a 6-way clique; an increase of 39%. More dramatically, exhaustive view set selection results in 5,270K tasks in the 6-way chain case, and 46,121K tasks for

a 6-way clique; an increase of 775%. These fluctuations imply that the view selection heuristic chosen should depend not only on the arity of the request, but on the topology and density of the request as well. It is worth mentioning, also, that while the benefits of MuxQO are not independent of query topology or view selection heuristic, they are not very sensitive to these factors either. MuxQO is effective over a range of query topologies and view selection heuristics.

The bottom line here is that our experiments underscore the theme we have emphasized over the past several chapters: that MuxQO is one important method in a multi-pronged approach to handling ASIS view selection; that sharing of optimization effort can work. Importantly, MuxQO has significant affect, it combines easily and, to a large degree, in an orthogonal manner with other methods we have presented, and current optimizers can be easily adapted to employ the technique. But a combination of methods is needed to dampen the doubly-exponential complexity of ASIS view selection. Each technique moves a larger space of problems from the realm of the intractable, to the realm of the practical.

Chapter 9

Exploiting the Semantics of Information Change in Active Service Integration

In previous chapters we have seen how various forms of sharing can help make active service integration more tractable. But even where such techniques are employed, active service integration can be an expensive process, and scalability remains a major challenge. Query processing systems have seen steadily increasing query complexity and volume over the years due to the increasing sophistication of users and the prevalence of program-generated queries; the same trends can be expected in ASIS workloads. Increased request complexity and load combined with high change-event frequency can overwhelm even a well-architected ASIS. For many desirable applications, incremental processing techniques and the sharing methods we have described do not take us far enough. We claim that it is crucial to develop general methods of exploiting application semantics to meet the scalability challenges inherent in active service. This chapter represents an attempt to do so.

A large class of ASI applications concern themselves with dynamic attributes of a relatively stable collection of objects: prices of stocks, bonds, currencies and other financial instruments; locations of moving objects; usage rates of high maintenance capital equipment; supply volumes for products or spare parts, etc. Furthermore, attributes of interest in such applications often do not or can not change arbitrarily; they obey a set of constraints. For instance, a vehicle cannot move faster than its maximum speed, an equipment item cannot be in use for more than 8 hours in an 8-hour shift, the production capacities of a manufacturing plant constrain the rate of depletion of supplies and material, etc. Applications in this class can consume heavy resources in computations that are not strictly necessary in that they do not produce results. But we can greatly decrease such unnecessary work by exploiting application semantics. In particular, we can exploit *constraints on information change over time* to avoid expensive and repetitive checking for conditions that cannot yet be satisfied. In this chapter we present a framework for exploiting the semantics of information change. We explore design points and describe issues in generalizing this framework, and we present an analytic model of the savings that accrue to our methods. Our model shows that our methods can significantly decrease the workload and increase the scalability of systems that require distributed condition monitoring. In addition, they can appreciably improve the response time between a condition occurrence and its recognition in such systems.

9.1 Partitioning Objects By "MinTTI"

Our approach applies to conditions over distributed services that involve tracking attributes of a set of objects that change in a time-constrained manner. We are given a monitoring request expressed as a distributed query over a collection of services, and one or more integrity constraints that apply to attributes in the request. We assume a baseline monitoring interval, M_0, for the request. We generate a companion query that, for each object, computes a lower-bound on the time before an attribute change can result in the

status of the object changing with respect to the request (i.e., in a currently non-satisfying object satisfying the condition of interest, or in a currently satisfying object no longer satisfying the condition). Call the original request Q, then we refer to the companion query as the *Minimum Time Till Interesting* with respect to Q, or $MinTTI_Q$. We partition the set of monitored objects into buckets based on their $MinTTI_Q$ values, and track changes to each bucket at the largest possible time interval that is less than the shortest $MinTTI_Q$ value for the bucket. A snapshot of this process is depicted in Figure 9-1. The black dots in this figure represent objects that the system is monitoring. These objects are plotted along the X-axis based on their *MinTTI* values. Mi represents the monitoring interval for the group of objects that have *MinTTI* values between M_i and M_{i+1}, which are said to be in the ith bucket.

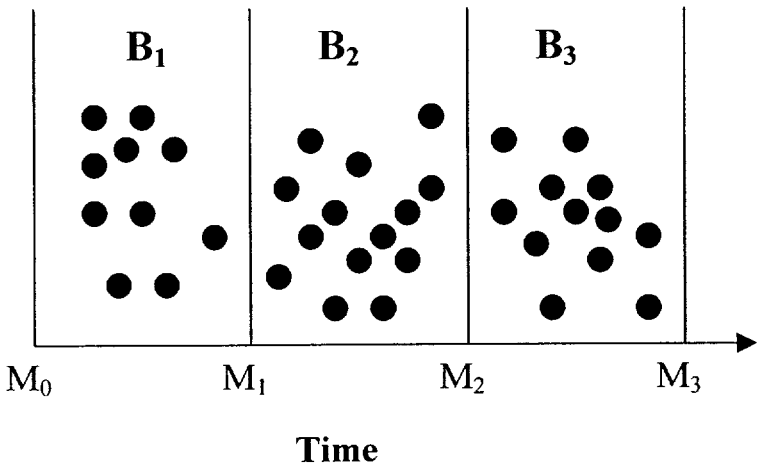

Figure 9-1: Partitioning Objects Based on MinTTI

The intuition behind our approach is very simple. Objects that are far from satisfying a condition of interest need not be monitored as vigilantly as those that could satisfy the condition at any moment. We need not expend resources computing a complex condition based on near-term changes an object with a large MinTTI value. We

save effort associated with such changes. On the other hand, we must expend effort to maintain the object partitions.

9.2 A Motivational Example

Consider an example distributed condition arising (once again) from DARPA's Command Post of the Future research project. We have an information agent that provides sensor data on the movements of enemy units over a far-reaching battle area. Another information source provides the location of landing zones where helicopters can land to transport personnel and material in and out of the area. Other agents provide information on types of enemy units and the ranges of weapons associated with the units by type. Finally, a geocoder agent is able to compute distances between entities based on the coordinates provided by the location sensors. The battleground scenario is depicted in Figure 9-2.

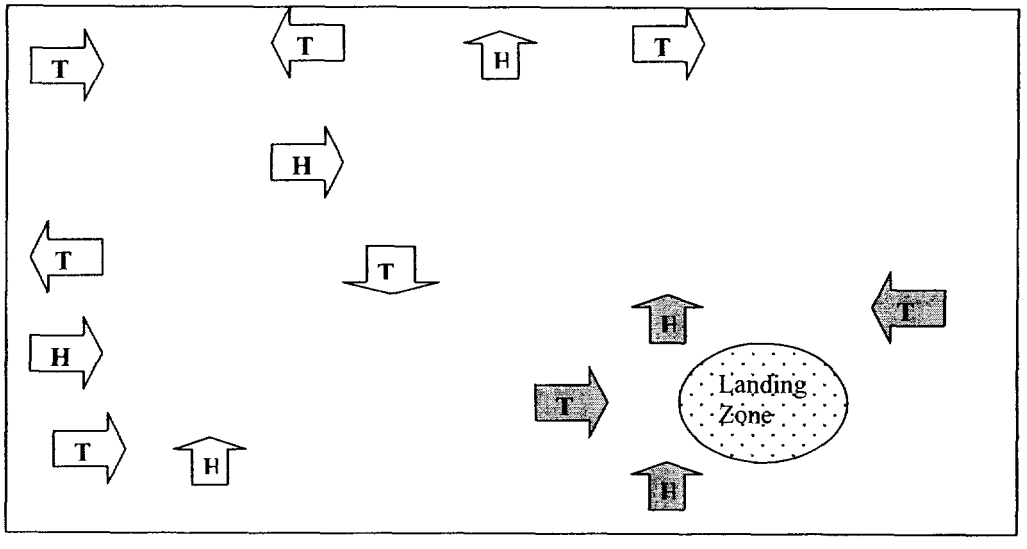

Figure 9-2: A Battle Area with Landing Zone and Enemy Tanks and Helicopters.

A commander in the field wants to be notified whenever an enemy unit poses a direct threat to a landing zone, where a direct threat is defined as a unit that is within its weapon's range of the zone. This condition can be expressed as the following query:

Threaten(UnitID, LZ) ←
LandingZone(Lz, Loc0) & EnemyUnit(UnitID) & UnitType(UnitID, Type)
& WeaponRange(Type, R) & Position(UnitID, Loc1) & Distance(Loc0, Loc1, D)
& $(D \leq R)$.

In this request, *LandingZone/2* tracks the location of landing zones of interest. *EnemyUnit/1* tracks enemy vehicles. *UnitType/2* maps a unit to its type. *WeaponRange/2* provides the maximum range of the weapons that a unit is equipped with by its type. *Position/2* is a sensing service that tracks the location of units in the field. Finally, *Distance/3* translates the distance between two locations. In Figure 9-2, enemy units that threaten the landing zone are shown in gray.

The central object set in this example is the set of enemy units, and the key changeable attribute for each object is its location. Change events, primarily in the form of additions and deletions, can occur at a number of the services involved in this condition (e.g., *LandingZone*, *EnemyUnit*), but the most frequent updates will be modifications to the location attribute in *Position*. Notice, however, that a unit's position cannot change arbitrarily; it is constrained by its maximum speed over the current terrain. In particular, suppose we are given the following constraint:

← *upd(Position(UnitID,* (Loc_0, Loc_1)*), T) & UnitType(UnitID, Type)*
& MaxVelocity(Type, V) & Distance(Loc_0*,* Loc_1*, D) &* $(D > V \times T)$.

This constraint is headless, meaning it derives *false*. It is interpreted as follows: If *Position(UnitID,*Loc_0*)* is true at time 0, and *Position(UnitID,*Loc_1*)* is true at time T, then the distance between Loc_0 and Loc_1 cannot be greater than the product of the maximum velocity of *UnitID*, V, and the time transpired, T.

Note that this constraint includes a special second-order predicate, *upd*/2. The *upd* (or *update*) predicate takes a time attribute, *T*, as its second argument. Its first argument is a special form of a first-order service predicate that includes a set of key attributes or (equivalently) an object identifier (*UnitID*, in this case), and one or more attribute pairs. Each pair represents distinct values for the corresponding (dynamic) attribute, such that the first variable represents the attribute value at time *0*, and the second variable represents the attribute value at Time *T*. The *upd* predicate allows us to describe constraints on object attributes over time. Note, further, that the constraint incorporates information from multiple services, many of which are involved in the condition of interest, but some of which are not. In particular, a supplemental service is required that provides information on the maximum velocity of enemy units by unit type. In general, constraints of interest can involve the full range of service capabilities available to an ASIS.

A *MinTTI* predicate includes an object identifier (or attributes encompassing a key) plus the time variable *T*. By merging the constraint above with the definition of *Threaten*/2, in a process somewhat similar to that of Semantic Query Optimization (SQO) (Chakravarthy, Grant et al. 1990), and performing a simple transformation to obtain the border condition for *T*, we can derive the following definition for $MinTTI_{\text{Threaten}}$:

$$MinTTI_{Threaten}(UnitID, T) \leftarrow$$
$$LandingZone(Lz, Loc0) \;\&\; EnemyUnit(UnitID) \;\&\; UnitType(UnitID, Type)$$
$$\&\; WeaponRange(Type, R) \;\&\; Position(UnitID, Loc1) \;\&\; Distance(Loc0, Loc1, D)$$
$$\&\; MaxVelocity(Type, V) \;\&\; (T = |(D - R)|/V).$$

This definition of *MinTTI* computes the time it will take for any enemy unit that is not within range of a landing zone to be within range if it travels at maximum velocity toward the landing zone. For an enemy unit that currently threatens a landing zone, the query computes the time it will take for the unit to be beyond a threatening range if it travels at maximum velocity away from the landing zone. Note that, in the latter case, we need to take an absolute value to prevent the *T* value from being negative. Should

multiple *T* values be associated with a single *UnitID*, we take the minimum such value for each *UnitID*, since we are computing a lower bound for each object. In some cases, we may want to handle satisfying objects differently than non-satisfying objects (for instance, if we expect the set of satisfying objects to be small, and if satisfying the condition is not likely to last long). But here we show the general case in which we handle both groups uniformly.

Note that the definition of $MinTTI_{\text{Threaten}}$ has a high degree of commonality with the *Threaten/2* predicate itself. Such commonality implies that whenever *Threaten/2* must be computed, $MinTTI_{\text{Threaten}}$ can be computed as well at little additional cost. This pattern of commonality between a condition and its *MinTTI* query is a result of the SQO-style merging of the condition with a relevant constraint, which leaves the body of the condition intact. *MinTTI* typically involves some additional filtering over the initial condition, which is easy to compute. As we will see, this observation implies that we can use multiple query optimization techniques to compute *MinTTI* with minimal overhead.

Once $MinTTI_{\text{Threaten}}$ has been derived, it is used to partition the set of units by their corresponding *T* values. Each partition (or bucket) is associated with a monitoring interval that is less than the least *MinTTI* value of any *UnitID* contained in it. One partition is monitored most frequently, at the monitoring rate required by the request (often as frequently as possible). This partition is referred to as the *urgent bucket*. Figure 9-3 shows our previous snapshot of the battle area with enemy units highlighted that fall within the urgent bucket. Items in the urgent bucket may appear farther away than other items that are not in the urgent bucket, even though they are temporally closer. In our example, two helicopters must be monitored urgently, even though tanks that are closer to the landing zone do not need to be monitored as frequently. This situation can only occur if the helicopters have a higher maximum velocity than the tanks. Note that one of the threatening helicopters must also be monitored in the urgent bucket.

Only changes to objects in the urgent bucket result in the incremental evaluation of the condition of interest, $\Delta Threaten$. Changes to all buckets, however, require the

incremental recomputation of *MinTTI* . We describe this process in detail and model the associated costs later in this chapter.

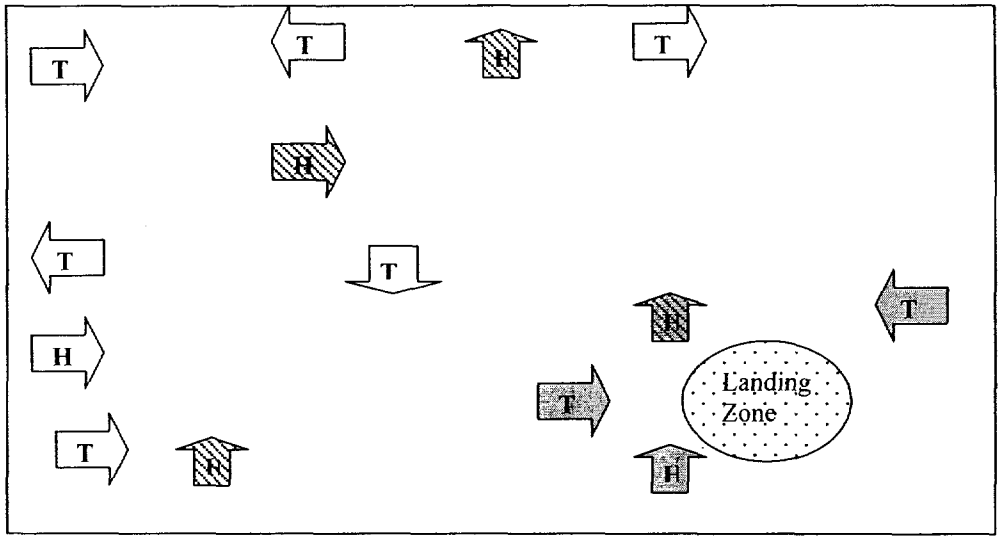

Figure 9-3: Striped Units are in the Urgent Bucket.

9.3 Partitioning Pragmatics

In general, we take an ASIS request that centers on changes to one or more attributes of a collection of objects, a temporal integrity constraint on the rate of change of these attributes that contains the *upd* predicate, and we attempt to manually derive an associated *MinTTI* predicate by inspection. But while any single *MinTTI* derivation may be straightforward and manageable, repeating the process for large numbers of requests is burdensome and error-prone. It is reasonable to ask whether the *MinTTI* can be derived automatically. In fact, we have performed some preliminary investigations of automatic

generation of *MinTTI* . We believe that an approach involving a combination of Semantic Query Optimization (Chakravarthy, Grant et al. 1990); (Levy and Sagiv 1995), whereby applicable constraints are merged with the original request, and a special program tranformation from *upd/2* to *MinTTI* holds promise. But this line of inquiry is not part of this thesis. In the absence of automatic generation, however, it is perfectly reasonable in real systems to write *MinTTI* directly by hand for problematic requests and special applications. The system can then use the definition to optimize the monitoring process.

As we have noted, given the definition of *MinTTI*, we compute a relation of *OID-Time* pairs that covers every object currently being monitored at a particular source. This relation provides the basis for dividing monitored objects into buckets based on each object's minimal time value. Another important issue is how many buckets we have, and how the bucket boundaries are defined. One extreme is to have a separate bucket for each object. The other extreme is to place all of the objects in a single (urgent) bucket. The latter is equivalent to the normal approach. We will describe an approach based on two buckets. But we believe other approaches should be investigated, including a clustering approach, in which a natural partition emerges from the data. An implicit design decision here is whether the buckets should have static boundaries, or if boundaries should be determined dynamically, based on the computed time values.

Note that while monitor buckets can be implemented at the mediator to good effect, even greater savings can be realized when services provide direct support for such buckets. In the former case, all updates will be sent to the mediator at the highest monitoring frequency. The mediator may then filter out all such updates but those pertaining to the appropriate buckets, and thus avoid computing the overall condition for some portion of these updates. In the latter case, in contrast, the bucket-based filtering of updates can occur upstream, at the service provider from which they originated; the mediator never needs to see them. Many of the triggering capabilities we have described earlier in this thesis, including the capabilities associated with full-fledged database

management systems, are insufficient for this purpose. We prescribe a "bucket-trigger" capability that operates as follows:

1. A bucket trigger can be created with a user-specified number of buckets, with a user-specified monitoring interval associated with each bucket, and with a user-specified table of object identifiers associated with each bucket.
2. The set of changes for the currently active interval is maintained for each bucket.
3. As object attributes change, the changed value is recorded in the appropriate change bucket. If a value is already present for the given OID, it is overwritten.
4. For each bucket, the associated set of changes for the current interval is emptied when the interval expires. The data is passed to the client of the trigger (the mediator, in our ASIS architecture).
5. Bucket triggers can be modified atomically, in a set-oriented fashion. For example, operations such as *Add(OidSet, Bucket)*, *Delete(OidSet, Bucket)*, *Move(OidSet, Bucket1, Bucket2)* are supported.

Implementing this capability given the building blocks of a modern data management system is not particularly difficult. In a relational database service, for example, an update trigger could be set on the original service call (*Position(UnitID, Loc)* in the *Threaten* example). The set of current changes for all buckets could be implemented as a relational table with an added *BucketID* field. Each triggered set of updates can be joined against an efficient OID-to-*BucketID* hash index and inserted into the change table. A daemon process could empty each change bucket at the appropriate interval (via a selection query on the change table, by *BucketID*), and pass the contents to the client of the trigger. Bucket creation and modification can occur by atomically modifying the hash index, and creating the change table if necessary (for a *Create()*).

The intuition behind our approach is straightforward. As long as an object is far from satisfying a condition of interest, we can delay and possibly ignore changes to it and the processing that goes with these changes. In our example, if we are concerned with

enemy units that threaten a given landing zone, we need not repeatedly compute the *Threaten* predicate in response to the movement of tanks, say, that are hundreds of miles away. By ignoring such changes, traffic from the service to the mediator is reduced, and we can avoid the expensive process of evaluating a complex, distributed condition for every change. The price of our scheme, however, is the need to do bucket maintenance. We now look more closely at the costs and benefits of our approach.

9.4. Performance Modeling

Finding a convincing battery of tests to demonstrate the performance of our methods is difficult. No widely accepted benchmark exists that captures a set of conditions to monitor or important data and system parameters relevant to distributed condition monitoring. We opt, instead, to present an analytical model that demonstrates the characteristics and sensitivities of our approach. Our model shows that our approach can yield significant cost savings and improved response time, even when implemented in the simplest manner.

Because complex condition monitoring is generally an ongoing, long-lived process, we focus on the steady-state costs involved. In particular, we ignore the one-time cost associated with the initial computation of the $MinTTI_Q$ relation, and the division of objects into buckets. Note that this computation is accompanied by the initial computation of Q, which must be computed regardless of the method chosen. But there will usually be a large degree of commonality between $MinTTI_Q$ and Q, so by exploiting multiple query optimization techniques we can compute $MinTTI_Q$ at little additional cost. By similar reasoning, we ignore the cost of view maintenance (for Q and $MinTTI_Q$) associated with objects that are added to or deleted from the set we are monitoring. We assume that this set is fixed, and consider only modifications to the objects in the set.

In addition, we do not consider the monitoring costs at the (remote) source. We justify this choice in two ways. First, while our methods involve a richer monitoring capability than traditional approaches, we show in a later section that this capability can be implemented efficiently, at little cost overhead beyond that of a traditional trigger. Second, the mediator and network costs are likely to dwarf the costs of monitoring at a single source.

9.4.1 Modeling Steady-State Costs

Given a query Q that describes a complex distributed condition involving dynamic attributes of a set of N objects, S, with monitoring interval M, the "naive" approach to monitoring Q with respect to S is shown in Figure 9-4 below:

1) At the source of S, at each monitoring interval, check if there is a change to S.
2) If there is a change, call it ΔS, send ΔS to the mediator.
3) At the mediator, compute ΔQ with respect to ΔS.

Figure 9-4: "Naïve" approach to monitoring Q with respect to S

As indicated, we ignore costs associated with Step 1. We will consider the cost of sending ΔS to the mediator as part of the cost of computing ΔQ (steps 2 and 3), and that cost is given by the cost function of the query optimizer. The optimizer is not likely to consider complex statistical properties of ΔS. Rather, it will assume ΔS follows the statistical properties of S (if it considers them at all) and only consider the cardinality of

ΔS (denoted as $|\Delta S|$). Hence the cost of computing ΔQ is a function of $|\Delta S|$, and the expected cost of the "naive" approach is given by:

$$E(C_{naive}) = \sum_{x=0}^{N} p(|\Delta S| = x) \cdot C(\Delta Q, x) / M \tag{9-1}$$

Here $p(|\Delta S| = X)$ is the probability of $|\Delta S| = X$, and $C(\Delta Q, X)$ is the cost, given by the cost function, of the query ΔQ with respect to ΔS, with $|\Delta S| = X$.

For our "bucketing" approach, assume we have J buckets. The expected cost is the sum of the expected costs of each bucket:

$$E(C_{buckets}) = \sum_{i=1}^{J} E(C_{B_i})$$

B_0 is distinguished as the *urgent bucket*. This bucket is monitored at the shortest monitoring interval, M_0. The processing related to this bucket is similar to that of the naive approach, except we have the additional process of bucket maintenance to worry about. That is, a change to an object in B_0 may result in a change to the $MinTTI_Q$ value associated with that object, which in turn may result in a bucket change for that object. The steps associated with B_0 are shown in Figure 9-5 below:

1. At the source of S, at each monitoring interval, check if there is a change to the members of S in B_0.
2. If there is such a change, call it ΔB_0, send ΔB_0 to the mediator.
3. At the mediator, compute ΔQ and perform bucket maintenance with respect to ΔB_0.

Figure 9-5: "Bucketing" approach. Monitoring Q with respect to ΔB_0

Let us assume, at this stage, that we use a simple bucketing strategy where the dividing points between buckets, once established, are fixed. Then the cost of bucket maintenance is dominated by the cost of computing a new $MinTTI_Q$ for each object in ΔB_0.[1] By a similar argument to that of the naive case above, the expected cost of the urgent bucket is given by:

$$E(C_{B_0}) = \sum_{x=0}^{N_0} p(|\Delta B_0| = x) \times C(\Delta Q + \Delta MinTTI_Q, x) / M_0$$

Here $C(\Delta Q + \Delta MinTTI_Q, X)$ is the cost, given by the cost function, of executing the two queries ΔQ and $\Delta MinTTI_Q$ with respect to ΔB_0, with $|\Delta B_0| = X$. Note that M_0, the monitoring interval for the urgent bucket, is the user-defined monitoring interval, which is the same M that applies to the naive case (Equation (9-1)). N_0 is the number of objects in B_0, which is less than N.

We now consider B_i, $i > 0$, the non-urgent buckets. Recall that $MinTTI_Q$ for any object in a non-urgent bucket is greater than the monitoring interval for that bucket, M_i. Thus, by definition, a tuple in ΔB_i cannot result in a change to Q. Therefore, for all non-urgent buckets, processing follows the same steps as the urgent bucket, except that the computation of ΔQ is not needed; only bucket maintenance is required. The expected cost for each non-urgent bucket is given by:

$$E(C_{B_i}) = \sum_{x=0}^{N_i} p(|\Delta B_i| = x) \times C(\Delta MinTTI_Q, x) / M_i \qquad \text{for } i = 1 \ldots (J-1)$$

9.4.2 Cardinalities of Deltas

[1] This is conservative in that bucket maintenance may be done more efficiently in some cases. It is generous in that we disregard the cost of notifying the remote source of bucket changes.

The equations we have derived thus far depend on the distribution of the cardinalities of the deltas, $|\Delta S|$ and $|\Delta B_i|$. To model these distributions, assume that changes to objects in S are independent, and that the probability of an object changing in a given time interval is independent of and identical to the probability of it changing in any other time interval of equal length.[2] This assumption implies that the number of changed objects for a given set of objects within a given monitoring interval, and thus the cardinalities of the deltas, follows a binomial distribution. Suppose, further, that the probability that a given object will change in the smallest monitoring interval, M, is θ. The distributions for $|\Delta S|$ and $|\Delta B_0|$ are given by:

$$p(|\Delta S| = x) = b(x; N, \theta)$$

$$p(|\Delta B_0| = x) = b(x; N_0, \theta)$$

where $b(x;n;\theta) = \binom{n}{x} \theta^x (1-\theta)^{n-x}$ is the x^{th} binomial coefficient.

For the non-urgent bucket, B_i, $i > 0$, let M_i be the monitoring interval for B_i, and θ_i be the probability that a given object will change that interval, then $M_i > M$, and thus $\theta_i > \theta$. Assume, without loss of generality, that $M_i = k_i * M$ for some integer k_i. Then the probability of x *changes* to a *single* object within M_i is $b(x; k_i, \theta)$, and θ_i, the probability of *some change* to a single object within M_i, is $1 - b(0; k_i, \theta) = 1 - (1-\theta)^{k_i}$. Thus the distribution of $|\Delta B_i|$ is given by:

$$p(|\Delta B_i| = x) = b(x; N_i; 1-(1-\theta)^{k_i}) \qquad \text{for } i = 1 \ldots (J-1)$$

Notice that if we monitored the objects in B_i at interval M, as we would in the naive case, we would expect to process $\theta k_i N_i$ updates to these objects every M_i time period. With bucketing we expect to process $(1-(1-\theta)^{k_i})N_i$ updates in the same period.

[2] This assumption is imperfect, but not damaging. Trying to model complex inter-update correlations is

That is, the expected volume of *Meaningless Updates Discarded* (*MUD*) for B_i is given by:

$$E(MUD_{B_i}) = \theta\ k_i N_i - (1-(1-\theta)^{k_i})N_i$$

MUD, and the avoidance of related processing costs, is the source of the savings produced by the bucketing approach. When these savings exceed the overheads introduced by bucket maintenance, our approach is beneficial.

9.4.3 Simple 2-Bucket Case

In the remainder of this section we will consider the simplest possible implementation of our methods: two buckets ($M = 2$), B_0 and B_1, with a fixed boundary between them, M_1. Assume, without loss of generality, that $M = M_0 = 1$. Then $M_1 = k_1$. For simplicity of analysis, we will also assume simple linear forms for the query cost functions as follows[3]:

$$C(\Delta Q, x) = Ax \tag{9-2}$$

$$C(\Delta MinTTI_Q, x) = Bx \tag{9-3}$$

$$C(\Delta Q + \Delta MinTTI_Q, x) = (A+B)x \tag{9-4}$$

Note that Equation (9-4) represents a worst case cost for computing ΔQ and $\Delta MinTTI_Q$ together. But we may be able to do much better. As we noted in Section 9.2, Q and *MinTTI* will have a high degree of commonality, and thus ΔQ and $\Delta MinTTI$ will have a high degree of commonality. In fact, $\Delta MinTTI$ is likely to include ΔQ intact, with some relatively inexpensive filtering on top of the computation of ΔQ. Therefore, handling ΔQ

---

difficult at best, and we posit that our methods tend to work better where such correlations exist.

[3] We choose not to use the AX+Y form since it does not capture the important notion, in our setting, that X=0 implies C=0.

and $\Delta MinTTI$ as a (relatively simple) multiple-query optimization problem may get us $\Delta MinTTI_Q$ for little added cost, in which case, based on Equation (9-2):

$$C(\Delta Q + \Delta MinTTI_Q, x) \approx Ax$$

In any event, the cost of the naive monitoring implementation is given by:

$$E(C_{naive}) = \sum_{x=0}^{N} b(x; N, \theta) \times C(\Delta Q, x) = \theta \times N \times A$$

We will now look at two distributions for $MinTTI_Q$, consider how a choice of M_1 can be made to minimize the costs of the 2-bucket approach, and compare these costs to the naive approach.

9.4.3.1 Uniform Distribution

Assume a *stable* uniform distribution of $MinTTI_Q$ values for the set of objects, S, over the range [0...R]. Given M_1, $N_0 = (M_1/R)N$ and $N_1 = (1 - M_1/R)N$. The cost of the 2-bucket approach in this case is given by:

$$\begin{aligned} E(C_{2\text{-}buckets}) &= \sum_{x=0}^{(M_1/R)N} b(x; (M_1/R)N, \theta) \times C(\Delta Q + \Delta MinTTI_Q, X) \\ &+ \sum_{x=0}^{(1-M_1/R)N} b(x; (1 - M_1/R)N, 1 - (1-\theta)^{(1-M_1/R)N}) \times C(\Delta MinTTI_Q, X)/M_1 \end{aligned} \quad (9\text{-}5)$$

$$= ((M_1/R)N \times \theta \times (A + B)) + ((1 - M_1/R)N \times (1 - (1-\theta)^{(1-M_1/R)N}) \times B)/M_1$$

As a final simplification, we will assume that the probability that an object in $B1$ will change within M_1 is 1. That is, we will round $\theta_1 = 1-(1-\theta)^{(1-M_1/R)N}$ to 1 in the Equation (9-5). Note that while θ_1 approaches 1 as θ goes to 1 and as M_1 gets large, this simplification is an overestimate of the costs associated with B_1. But it allows us to simplify the Equation (9-5):

$$E(C_{2-buckets}) = ((M_1/R)\times N\times\theta\times(A+B)) + ((1-M_1/R)\times N\times B)/M_1 \tag{9-6}$$

and $E(C_{2-buckets})$ is minimized by choosing:

$$M_1 = \sqrt{BR/(A+B)\theta} \tag{9-7}$$

Table 9-1 below shows analytical results for a range of values of R and θ. Here we assume that there are 10,000 objects to be monitored (N=10,000). We show cost comparisons for the pessimistic assumption that:

$$C(\Delta Q + \Delta MinTTI_Q, X) = (A+B)X = 2AX \tag{9-8}$$

and the optimistic assumption that:

$$C(\Delta Q + \Delta MinTTI_Q, X) \approx AX \tag{9-9}$$

In this setting, the objects are monitored over the values, R, ranging from 100 to 10,000. The probability that a given object will change in a specific monitoring interval, θ, varies between 0.01 and 0.25. We compute M_1, C_{naive}, and $C_{buckets}$ using equations (13), (11), and (12), respectively. In each table, the Gain is the difference between the cost of the bucketing approach and that of the naive approach.

In the pessimistic model, when $R = 100$ and $\theta = .01$ and .05, our bucketing approach is worse than the naive approach since the cost of bucket maintenance outweighs the (relatively small) savings due to MUD. In all other cases, however, under our assumptions, our method provides a more efficient monitoring capability.

		M_1			Pessimistic		Optimistic	
R	θ	Pess	Opt	C_{naive}	$C_{buckets}$	Gain	$C_{buckets}$	Gain
100	0.01	71	100	100*A*	182.8*A*	-82.8*A*	100.0*A*	0
100	0.05	32	45	500*A*	532.5*A*	-32.5*A*	347.2*A*	152.8*A*
100	0.25	14	20	2500*A*	1314.3*A*	1185.7*A*	900.0*A*	1600.0*A*
1,000	0.01	224	316	100*A*	79.4*A*	20.6*A*	53.2*A*	46.8*A*
1,000	0.05	100	141	500*A*	190.0*A*	310.0*A*	131.4*A*	368.6*A*
1,000	0.25	45	63	2500*A*	437.2*A*	2062.8*A*	306.2*A*	2193.8*A*
10,000	0.01	707	1000	100*A*	27.3*A*	72.7*A*	19.0*A*	81.0*A*
10,000	0.05	316	447	500*A*	62.2*A*	437.8*A*	43.7*A*	456.3*A*
10,000	0.25	141	200	2500*A*	140.4*A*	2359.6*A*	99.0*A*	2401.0*A*

Table 9-1: Gains under Optimistic and Pessimistic Assumptions

Figures 9-6 and 9-7 graph the pessimistic and optimistic cost savings, respectively, for a range of R values with θ fixed at .01, .10, and .5. In both figures, as θ increases, the cost savings of the bucketing approach increases as well. As object changes become more frequent, more MUD is generated by our methods, which translates to more savings. Note that in general we may, at best, only be able to estimate (or guess) at θ. But simple calculations based on the above show that even if we are wrong in our guesses, and our choice of M_1 is imperfect, we can still do better than the naive approach.

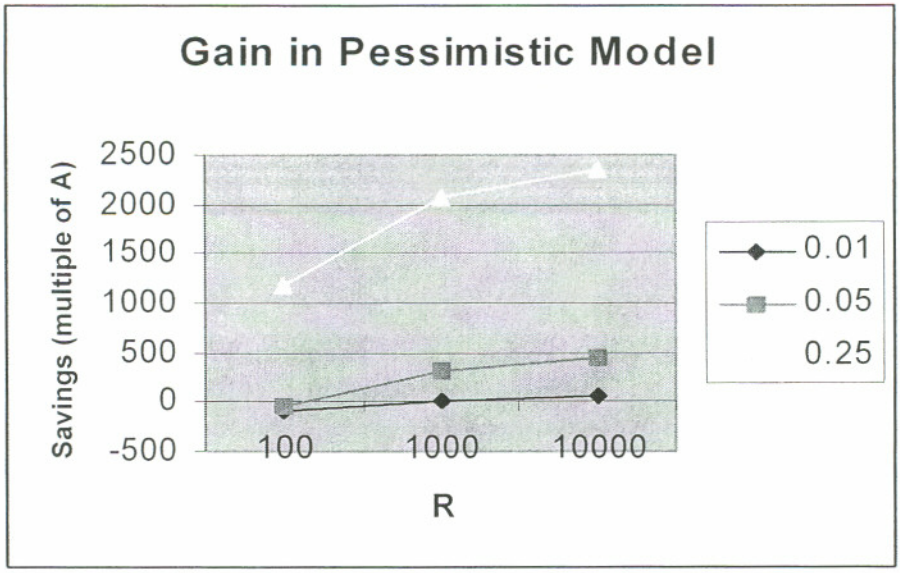

Figure 9-6: Savings versus R for Varying θ, Pessimistic

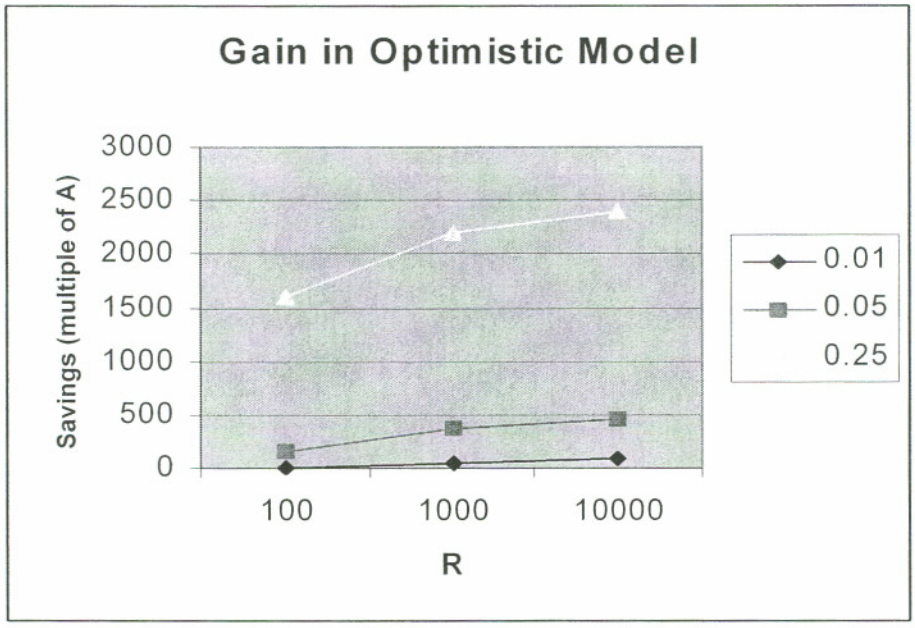

Figure 9-7: Savings versus R for Varying θ, Optimistic

9.5 Chapter Summary

We have presented a framework for exploiting constraints on information change in the setting of active service integration. We have presented strong evidence that, for a large class of applications, our techniques can greatly reduce the update rate seen by a ASIS, and thus improve the scalability and response time of such systems. These savings can be realized even where the simplest of bucketing schemes is adopted.

There are several clear areas for near-term expansion of these ideas. The requirement that *MinTTI* must be manually specified for each applicable ASIS request is burdensome and error prone. We are currently investigating techniques for automatically generating *MinTTI*. We believe that a combination of program transformation and semantic query optimization can feasibly be employed to this end. Another important issue is how bucket boundaries are defined and how they evolve over time. Near-term work involves investigating a range of approaches for doing this, including clustering techniques.

Interesting future avenues to pursue include trying to broaden the range of constraints we can work with. Often absolute constraints do not exist, but probabilistic ones do. A stock currently priced at $20, for example, is highly unlikely to drop to $2 in a matter of hours (recent events notwithstanding). How can constraints that apply (merely) with high probability be integrated into our scheme? What are the tradeoffs between efficiency and scalability in applying these techniques, and accuracy or consistency (a familiar theme in Internet-based systems)? Often applications can tolerate some inconsistency for greater scalability and efficiency. On the other hand, the truly unusual or spectacular change events may be just the sort of events we are most interested in. If the price of greater efficiency is that we miss being notified of such unusual events, which is what a probabilistic approach could imply, we could not accept that tradeoff. Another avenue of exploration involves studying the intent of objects we are monitoring, and incorporating knowledge of an object's intent into our constraint

reasoning. Employing intent reasoning may involve similar tradeoffs to that of employing probabilistic reasoning.

Our specific techniques may be applied to a variety of independent variables, provided the variables in question can be related to time. In our example, we related distance to time based on an object's maximum velocity, but many other variables can be related to time in a natural way, e.g., flow in a refinery, number of items processed in a machine, number of trades or transactions in an auction, etc. But beyond variables whose rate of change can be naturally derived, we believe that the general notion of using application semantics to produce MUD has wider applicability. Constraints based on some necessary path of change or an impossible transition, required intermediate steps, or some discrete number of update events might be used to deduce that changes to certain objects can be discarded or temporarily ignored.

In the Internet age, scalability and robustness are a constant challenge in computing. We believe semantics provide a key source of traction in meeting this challenge. The ideas in this chapter represent a first step toward tapping this source in the context of active service integration. Significant work in view maintenance has sought to determine when updates are syntactically independent of views (Levy and Sagiv 1993). The work we present here is the first attempt, to our knowledge, to isolate updates that are *semantically* independent of views. Many of the ideas in this chapter have also been presented at the AAAI Spring Symposium (Benninghoff and Noh 2001) and at the International Conference on Enterprise Information Systems (Benninghoff and Noh 2003).

Chapter 10

Contributions, Related Work, and Conclusions

The accelerating deployment of networked information sources and services creates a great opportunity for mediation systems of all stripes. As networked services proliferate, monitoring and event-based integration will become an increasingly important mediator capability. Yet the majority of work on mediation systems has focused narrowly on queries and transactions. As a result, the issues involved in providing monitoring and event-based integration are not well understood. The work presented in this dissertation is a step towards improving the understanding of these issues. In this concluding chapter, we reiterate our thesis, summarize the major contributions of this dissertation, discuss related work and future directions suggested by our work here, and highlight some important lessons learned from this effort that can help

to guide systems builders interested in implementing active service integration technology.

10.1 Contributions

We have described a powerful type of mediation system that we call an Active Service Integration System (or ASIS), whose central function is monitoring and event-based integration over autonomous, networked services and information sources. We have built a basic ASIS, deployed it in at least one running application, and used it to motivate advanced issues in scalability and performance. We have derived solutions for several of these issues.

We have sought to demonstrate our thesis that inter-task sharing together with the specification and exploitation of service characteristics and capabilities are key concerns in building a monitoring and event-based integration capability that is scalable and efficient. Techniques for sharing of result processing, data movement and planning are all important pieces of a solution that demands a multi-pronged approach. The exploitation of rich application semantics is another important avenue to improved scalability and performance. The semantics of information change, in particular, can be leveraged to improve the scalability and performance of a number of useful applications.

Besides providing evidence to support our thesis, this dissertation has made several concrete contributions:

- **The Basic ASIS:** We have presented a description and implementation of a basic ASIS that addresses the fundamental issues of construction, and many issues involved in the practical and efficient execution of ASIS requests.

  We have demonstrated the value of that exploiting the heterogeneous computational capabilities and data characteristics of services to efficient processing of ASIS requests. Our system captures such capabilities in a metadata

model that goes beyond those found elsewhere in the networked services world to describe computational capabilities, data characteristics, and change events of available services. Our request processing module extends query processing technology to exploit the capabilities and characteristics exposed in our metadata model. We create new logical and physical operators that support integration and monitoring, and that provide the basis for capability-driven optimization and execution of ASIS requests.

- **Cost-Based Cache Selection:** We described how the long-lived nature of ASIS requests can be leveraged to make effective caching decisions. We present a detailed model and framework for cost-based cache (or view) selection in an ASIS, supported by metadata on service change events and their associated data. An exhaustive approach to the resultant optimization problem, however, is doubly-exponential in complexity. We describe a multi-pronged attack to handling this problem in a tractable manner that features a combination of heuristics and inter-task sharing.

- **Multiplex Query Optimization:** We presented a description and implementation of Multiplex Query Optimization (MuxQO), a method for efficiently handling problems that can be cast as a group of overlapping query optimization problems. We characterized the applicability of MuxQO, and we described a performance evaluation that demonstrates the effectiveness of MuxQO in handling cost-based view selection in an ASIS. We describe how a top-down optimizer can be modified to support MuxQO. MuxQO can provide dramatic savings in ASIS view selection, particularly when the join arity of ASIS requests is large. MuxQO is applicable to a range of problems, including physical database design, multiple query optimization, and the physical representation of new data formats and models such as XML (Bohannon, Freire et al. 2002).

- **Exploiting the Semantics of Information Change:** We provided a description and an analytic evaluation of a novel approach to exploiting rich application semantics to improve the efficiency and scalability of an ASIS. In particular, we describe and evaluate a method for exploiting constraints on information change over time. We argue that application-level semantics are a rich vein to mine in improving the scalability and efficiency of active service integration.

10.2 Related Work

While there is a dearth of work in active service integration, per se, techniques developed in other areas are applicable to and have heavily influenced our work here. We have selectively referenced such work throughout this dissertation. Here we discuss related work in a more global and systematic fashion. We divide this section by major areas of related work, including classic database systems, distributed and federated databases, information and data integration, materialized views and physical database design, and semantic optimization.

10.2.1 Classic Database Systems

We would be remiss if we did not acknowledge the influence of certain classic works in relational database management systems on the work in this dissertation. Codd developed the relational model for data management (Codd 1970), which helped to spawn an industry and a sub-discipline of computer science. Our declarative request language shares the first-order logic base of the relational model, and is equivalent to the basic select-project-join queries of the relational model. System R (Astrahan, Blasgen et al. 1976) and Ingres (Stonebraker, Wong et al. 1976) were the among the earliest relational systems, dating back to the early 70's, and they have profoundly influenced all work on database implementation that has come since. The basics of cost-based request

processing in Paradox closely follows the query processing techniques originally developed in System R (Selinger, Astrahan et al. 1979), as do those of most major commercial database systems today. Our implementation of MuxQO, however, involved modifications to the Columbia optimizer (Graefe 1995), a descendent of Volcano (Graefe 1994) and Exodus (Graefe and DeWitt 1987; Carey, DeWitt et al. 1988), which utilizes a top-down approach to optimization, as opposed to the bottom-up approach of System R. Top-down optimizers are not as common as bottom-up ones, but commercial optimizers available in recent product offerings from Tandem and Microsoft have taken the top-down approach. The top-down approach, due to its use of a data structure that hashes the current-best-subplan by the logical multi-expression that it implements, was easy to extend to support MuxQO. Extending a bottom-up optimizer to support MuxQO would be more difficult. Proponents of the top-down approach have argued that top-down optimizers are more easily extensible than bottom-up ones. Our experience with MuxQO bolsters these claims.

10.2.2 Distributed and Federated Database Systems

The extensive body of work in query processing in distributed databases has also influenced the work in this dissertation (Epstein, Stonebraker et al. 1978; Bernstein, Goodman et al. 1981; Mackert and Lohman 1986). Much distributed database work has focused on extending techniques from the classic centralized DBMS's to multiple cooperating DBMS's distributed over a LAN. The most obvious examples of such systems are R* (Haas, Selinger et al. 1982; Lindsay, Haas et al. 1984) and Distributed Ingres (Stonebraker and Neuhold 1977), which came from the same organizations that developed System R and Ingres, respectively. The notion of *row blocking*, in which tuples are sent in chunks, rather than one by one, from one distributed operator to another over a network is ubiquitous in such systems. In Paradox we adopt a variant of row blocking in our wrapper model, which breaks with strict demand-driven dataflow in

request execution to create a supply-driven pool of data on the receiving side of a network connection. This technique is particularly important in the WAN environment where it not only reduces the number of message sends, but also helps to mitigate the effects of bursty network traffic (Amsaleg, Urhan et al. 1998) and the cumulative effect of latency through multiple levels of a demand-driven pipeline.

Work in distributed databases has resulted in many proposals for efficient join processing, where the data to be joined resides in different locations. The semijoin is one notable proposal (Bernstein and Chiu 1981). The idea of the semijoin is to minimize the data transfer costs of joining a Table A at Site 1 to a Table B at Site 2 by sending only the columns of A that appear in the join predicate to Site 2, joining B with those columns, and then shipping the result back to Site 1 where the join is completed. Performance studies in a LAN environment indicate that the additional CPU costs of completing the join is often greater than the savings in communication costs (Lu and Carey 1985; Mackert and Lohman 1986). But semijoins become more attractive in a WAN setting, and where tuples are very large. Our parajoin operator is a variant of a semijoin designed with networked services in mind, in which any number of columns can be shipped to the joining site (the columns sent can be determined in a cost-based manner), but the join always occurs at the site of the service providing the operator. More efficient variants of the semijoin have been proposed, notably the Bloom join, in which a hash-based signature of the join columns, rather than the columns themselves, is shipped to the semi-join site (Babb 1979; Valduriez and Gardarin 1984). But a Bloom join is impractical for the networked services environment, since it requires a shared hash function between the mediator and any service that supports it.

Closely allied with work in distributed databases is that in federated database systems (Fankhauser, Gardarin et al. 1998; Elmagarmid, Rusinkiewicz et al. 1999; Roantree, Hasselbring et al. 2000), which shares the goal of uniform access to multiple, network-accessible database systems. One key distinguishing characteristic of a federated database system is that the participating databases are *autonomous*. Autonomy in this context is a relative term. It implies that a database is under its own administrative

regime, that its primary mission may be other than participation in the federation, and that the system itself was not designed with distributed processing in mind. The federation normally cannot expect to use a participating database for more than query or transactional access to the island of data it manages. Distributed database techniques such as data partitioning between nodes, semijoin programs, and heavyweight transactional commit protocols (e.g., three-phase commit (Skeen 1983)) are thought to violate autonomy, and are therefore verboten. Note that services in an ASIS are autonomous only in the first sense listed above, and this distinction is crucial. The primary mission of a networked service is to be integrated and composed with other services, and it should be designed with these uses in mind. A central tenet of our work is that building efficient, scalable integration systems in the Internet environment requires that services actively assist in the processing tasks of their clients, even where the assistance is fundamental to integration or composition, and not to the service as such. In the network services world we are compelled to push the traditional bounds of autonomy. The parajoin operator is one example of pushing these bounds. Our taxonomy of distributed change notification and update dissemination capabilities is another example.

A recent and innovative approach for bringing federated databases to the WAN environment can be found in the Mariposa project (Litwin, Pfeffer et al. 1996). Mariposa developed a novel economic model for sharing and optimizing query-processing effort across the federation, which was heavily influenced by economic-based models developed in the AI and multi-agent systems community (Smith 1980; Wellman 1993; Wellman 1995; Clearwater 1996). Philosophically, Mariposa agreed with our thinking that scaling in the WAN environment demands that participants in a federation be designed with federation in mind, and that they offer significant computational assistance to the federation as a whole. Mariposa parted with us at a more fundamental level, however, in the belief that centralized query processing was impractical in the WAN environment and could not scale to myriad information sources. An attempt was made to commercialize Mariposa via the company Cohera, Inc. But Cohera was not successful, closing its doors late in 2001. Simpler, better-understood, R*-like extensions to

centralized DBMS's remain the commercial state of the art in distributed and federated DBMS's.

10.2.3 Heterogeneous Databases and Data Integration

Work in heterogeneous databases, which overlaps federated database work somewhat, dates back to the Computer Corporation of America's MultiBase project (Smith, Bernstein et al. 1981), but also includes systems such as IBM's Datajoiner, METU (Dogac, Halici et al. 1996), and IRO-DB (Gardarin, Sha et al. 1996). The distinguishing characteristic of a heterogeneous database system is that the participating databases can have differing query capabilities, schema or underlying data models. Source autonomy is generally assumed in such systems as well. A closely allied area is data (or information) integration. In a data integration system autonomous component sources are often not database systems at all; they can be flat files, web sites, directory services, spread sheets, or some other variety of information source. Notable data integration systems include DISCO (Tomasic, Raschid et al. 1998), Garlic (Carey, Haas et al. 1995; Roth, Arya et al. 1996; Josifovski, Schwarz et al. 2002), Hermes (Adali, Candan et al. 1996), Pegasus (Ahmed, Smedt et al. 1991), and the Information Manifold (Levy, Rajaraman et al. 1996). All of these systems share essentially the same three-tier architecture with the Paradox system. A client passes requests to a middle tier, which parses and decomposes the request into fragments that can be handled at individual sources and combines or integrates the resulting data. Wrappers or adapters provide a uniform interface between the middle tier and the sources. A catalog or some form of metadata repository provides information needed to guide the process.

The early focus in heterogeneous database systems and data integration systems was largely about coping with semantic heterogeneity between models, schema and domains. These difficult issues are relevant in the context of active service integration as well, but they are not the focus of this dissertation. The growing importance of the

Internet, the advent of the World Wide Web, and the promise of ubiquitous connectivity have resulted in intensified interest in data integration, and in an increasing focus on issues of performance and scalability (Florescu and Levy 1998).

A common goal in attempts to achieve better data integration performance and scalability has been to make the best use of advanced source capabilities while avoiding plans that sources of limited capability cannot process. Several researchers, for example, have considered limited source capabilities in which sources are modeled as predicates and capabilities are modeled as limited binding patterns (Levy, Rajaraman et al. 1996; Li, Yerneni et al. 1998; Florescu, Levy et al. 1999; Li and Chang 2000). Web sites that retrieved data based on simple HTML forms were a major impetus for this work. We consider binding patterns as well, though it is not a primary emphasis in our work. Our motivation for considering them is more to handle computational functions, which are important in the network services world, than to deal with web sites with simple forms interfaces.

One difficult issue in query optimization in heterogeneous databases and data integration systems is in estimating the cost of the parts of a plan that are executed by external sources. Three main approaches to this problem have been proposed. One approach is to require a customized cost model for each information source. This approach has the advantage of being as accurate as possible, but it places a large burden on wrapper development. A second approach is to have a generic cost model integrated into the mediator that relies on heuristics about the algorithms and computational complexity of the processing done at each source in a given plan. This is the approach taken by Paradox, which adjusts its estimates based on source capabilities. An enhancement to this approach allows a generic cost model to be tuned or calibrated at each source based on specific parameters. Several generic cost models have been proposed (Du, Krishnamurthy et al. 1995; Gardarin, Sha et al. 1996; Zhu and Larson 1998). Some researchers have taken a hybrid approach that uses a generic model by default, but allows customized cost models to override the generic model in critical instances (Naacke, Gardarin et al. 1998). This hybrid approach was eventually adopted

by Garlic (Haas, Kossmann et al. 1997; Roth and Schwarz 1997). A third approach is to observe query plans in action and learn the costs of sources based on their execution history, which was the approach taken by HERMES (Adali, Candan et al. 1996). Paradox allows for a learning approach to statistics that inform its cost model, which can be used where other sources of statistical information are not available. A system administrator is also able to tweak the Paradox cost model, which allows for a form of calibration.

While we have only highlighted a small portion of the voluminous literature from the database community on heterogeneous databases and data integration, the general trend in plan generation and optimization in this area has been to encapsulate the heterogeneity of information sources and otherwise apply techniques from distributed and federated databases to the degree possible (Kossman 2000). The work in this dissertation is in the same spirit. We are unique in extending these ideas to encompass monitoring and event-based processing, and our networked services orientation brings with it some subtle but important differences in terms of the capabilities of external information sources.

Blackboard systems and facilitators from the AI world seek to coordinate and often integrate the capabilities of multiple network-accessible agents. The notion of "agent" in such systems is fairly broad, and subsumes our concept of a networked service. The Flipside system (Schwartz 1993), and the Open Agent Architecture and its successor the Adaptive Agent Architecture (Cohen 1994; Kumar 2000) combine the capabilities of a blackboard and a facilitator, and are exemplars of such systems. While these systems provide integrated access to the capabilities of multiple agents, possibly including triggering or event-notification capabilities, they do not provide monitoring of complex conditions over multiple agents, and so they cannot be considered active service integration systems. Further, in many of these systems, emphasis is placed on agent communication and coordination, not on plan optimization or caching. Other systems from the AI community that integrate data include Ariadne (Ambite, Ashish et al. 1998), Occam (Kwok and Weld 1996), Razor (Weld 1997), InfoSleuth (Woelk, Bohrer et al.

1995) and Sim (Knoblock 1993). These systems focus more on planning issues, applying AI representation and planning techniques, rather than cost-based optimization, to the problem of finding feasible plans given source restrictions.

While most work on distributed databases and information integration has concentrated on queries and transactions, a couple of recent projects have dealt with continuous or continual queries, which is very similar to the spirit of an ASIS. The Continual Query project (CQ) has developed an update monitoring system for use in the WWW environment (Liu, Pu et al. 1998; Liu, Pu et al. 1999). CQ has concentrated on handling the specifics of the Web environment, including creating wrappers with update notification capability over HTML and XML-based web pages. CQ has also developed a simple but expressive update specification interface. But CQ does not handle updates over multiple sources, nor have they dealt with scaling issues such as caching. Another notable effort is the Niagara project at the University of Wisconsin and OGI (Naughton, DeWitt et al. 2001). One of the focuses of the Niagara project is to try to build a continuous query system that can scale to the Internet. They refer to the continuous query component of this project as NiagaraCQ (Chen, DeWitt et al. 2000). The focus of NiagaraCQ is on processing myriad monitoring queries that have a large degree of similarity. The basic approach is to incrementally merge query plans that overlap significantly. This approach is similar in spirit to our merging of request fragments on a single service. NiagaraCQ also focuses on handling XML data.

10.2.4 Materialized Views and Physical Database Design

Materialized views became a hot topic in the early 90's as their value to a wide variety of advanced database applications became evident along with the sorry state of commercial facilities to support them. The importance of data warehousing and advanced applications such as OLAP, in particular, has provided impetus for a great deal of work on materialized views (Hammer, Garcia-Molina et al. 1995; Widom 1995; Harinarayan,

Anand A. Rajaraman et al. 1996; Kimball, Reeves et al. 1998). A large body of work has dealt with algorithms for incrementally updating materialized views (Blakeley, Larson et al. 1986; Qian and Wiederhold 1991; Gupta, Mumick et al. 1993; Gupta and Mumick 1995; Mumick, Quass et al. 1997). As a first approximation, an ASIS request is conceptually quite close to a view that must be incrementally updated. The Paradox system considers incremental plans for ASIS requests. Our incremental algorithms are based on work on centralized, deductive databases (Gupta, Mumick et al. 1993), but we adapt this work to the network services environment in a manner that tolerates some temporary (but necessary) inconsistencies, and that avoids inherent problems in the distributed environment such as update anomalies. Many of the pitfalls of incremental maintenance in a distributed environment, including update anomalies, were described in work from the WHIPS data-warehousing project (Zhuge, Garcia-Molina et al. 1995; Zhuge, Garcia-Molina et al. 1998). The efficiency of incrementally updating a materialized view depends on the "heuristic of inertia", which is the assumption that updates will be small compared to the size of base relations (Gupta and Mumick 1995). This is not a safe assumption in the networked services environment. Paradox considers both incremental algorithms and complete recomputation in the plan space for computing request results.

The problem of cache or view selection for an ASIS is similar in nature to the problem of physical database design, and in particular to the problem of materialized view and index selection. A substantial body of work exists in this area, dating back to the early 70's. The natural breakdown of view selection that we present into a choice of candidate views followed by combining candidates into configurations is echoed in much of this work. For candidate selection the two main approaches taken are syntactic analysis of the workload (Hammer and Chan 1976; Finkelstein, Schkolnick et al. 1988), and a knowledge or rule-based approach that makes heuristic judgments about complex view and index interactions (Hobbs and England 1991; Rozen and Shasha 1991; Choenni, Blanken et al. 1993). The Paradox approach is to optionally lay rule-based, heuristic restrictions over the set of possible intermediate results generated by the

optimizer. We suggest using a component of the optimizer itself (if the optimizer has been designed properly for this task) in the candidate generation process. The Microsoft *AutoAdmin* project generates candidate indexes and views by finding the best index and view configuration for each query independently. They refer to this technique as the *query-specific-best-configuration* candidate selection algorithm. Note that where a query can be expressed in the view language, the best configuration will consist of materializing the entire query. In an ASIS, the cache or view set must be adjusted incrementally, as each request is encountered. As a result, we generate candidates and configurations by analyzing each plan suite independently. Materializing an entire query in the suite is not possible in the ASIS setting, of course, since each query is based on data changes that are not yet available. We then attempt to merge the optimal configuration for the latest suite with the existing view set to get a better global configuration.

Selecting configurations requires assessing the value of a configuration, and a search method to work through the space of possible configurations. Some researches have developed approximate, "stand-alone" cost models for this purpose (Harinarayan, Rajaraman et al. 1996; Gupta, Harinarayan et al. 1997). Others have used the actual cost-based query optimizer of the target database to do "what-if" analysis of the costs of the workload if a given configuration is chosen (Finkelstein, Schkolnick et al. 1988; Chaudhuri and Narasayya 1997). We endorse the latter method. A key aspect of the Paradox approach is that candidate and configuration generation and selection is unified with the optimizer, and therefore with the process of plan generation and optimization.

In searching the space of possible configurations, given a set of candidates, the obvious greedy and exhaustive algorithms predominate in the literature. In the context of selecting materialized views in a data cube setting, researchers at Stanford have shown that a greedy algorithm is within 63% of optimal in all cases, and that in many realistic scenarios the difference between the greedy and optimal solutions is essentially nothing (Harinarayan, Rajaraman et al. 1996). The Microsoft *AutoAdmin* project applies a combination of exhaustive and greedy configuration enumeration. Their algorithm, *Greedy(m,k)*, searches exhaustively for configurations of size m, and from there

progresses to a configuration of size k in a greedy fashion (Agrawal, Chaudhuri et al. 2000). All of these approaches assume that each unique optimization problem will be handled separately by the query optimizer.

Ross et al. discuss the problem of choosing additional views to materialize and maintain in order to make the maintenance of a given set of materialized views more efficient (Ross, Srivastava et al. 1996). This problem is very similar, in essence, to cache or view selection in an ASIS. They present a cost-based formulation that is similar to that of the physical database design work presented above, and is similar in many respects to our formulation for ASIS view selection. They suggest using metadata estimating the frequency of base-table changes, which is a similar notion to our modeling of the frequency of change event types. But their approach to computing the optimal set of views is inefficient. It does not recognize the large overlap of the optimization problems induced by their approach. Also, while they mention the possibility of utilizing multiple query optimization in their solution, they do not describe a method for doing so.

Recently, others have also noticed the intimate connection between multiple query optimization and view selection. Mistry et al. present methods for efficiently finding a plan for the maintenance of a set of materialized views that exploits multiple query optimization and the materialization of additional views (Mistry, Roy et al. 2001). In their approach, as with ours, incremental maintenance plans, complete recomputation plans, multiple query optimization and ancillary view materialization are all considered in an integrated fashion within the same search space. But there are several distinctions between our work and theirs. First, their approach is tailored to the centralized DBMS environment, whereas ours is oriented towards the distributed, network services environment. Second, they consider a complete workload as a single unit, whereas, in the ASIS environment, we must adapt incrementally as requests are added to and deleted from the system. Third, their approach involves a new optimizer framework that requires fleshing out a *query DAG*, which represents every possible plan for every possible configuration of permanently or transiently materialized views. Our approach, in contrast, is much easier to integrate into an existing optimizer, and will be more efficient

because it can fully exploit search optimizations within the optimizer such as group pruning (Shapiro, Maier et al. 2001). Fourth, their search space considers a set of views to materialize either permanently or transiently, and then considers, in isolation, whether each view in the set should be permanently or transiently materialized. This approach misses important interactions between choices of permanence and transience within a given view set that are not missed in our approach. Finally, their approach only handles the specific problem of maintaining a set of views. We have presented a general technique, MuxQO, which can be applied to a wide array of optimization problems.

10.2.5 Rich Application Semantics

It is difficult to find work that is closely related to the material we presented in Chapter 9 on using rich application semantics to optimize ASIS request processing. But we will briefly mention three areas.

Moving objects databases is a new area of study that has garnered considerable recent interest in the database research community (Chon, Agrawal et al. 2002; Jenson and Saltenis 2002; Papadias, Tao et al. 2002; Venkata, Kanth et al. 2002). Like geographic information systems (Tomlin 1990), moving object databases extend database technology to provide a specialized framework for a data-intensive application domain. In this case, a moving object database represents and processes information specific to tracking the location and juxtaposition of objects in motion. Work in moving object databases includes new algorithms and new query language constructs specifically designed for spatio-temporal reasoning. The techniques we present in Chapter 9 are simpler and more general, in that they apply to ASIS applications beyond the realm of moving objects, and they can be implemented on top of current database technology without a major effort. But moving objects databases provide a much more complete data management solution for the specific domain of moving objects.

Semantic query optimization (SQO) is an area of research at the juncture of database query processing and logic programming (King 1981; Chakravarthy, Grant et al. 1988; Chakravarthy, Grant et al. 1990; Levy and Sagiv 1995; Grant, Gryz et al. 1997). In SQO, given a query and semantic knowledge about the database represented as a set of integrity constraints involving base relations or extensional predicates within the body of the query, the query is transformed into a semantically equivalent query that is (hopefully) more efficient to process. Semantic query optimization has been shown to be very effective for a wide variety of applications. Nonetheless, SQO techniques have not been integrated into any commercial systems to our knowledge.

The semantic optimization work we present in Chapter 9 uses semantic query optimization as a starting point. To apply our technique, we need an ASIS request, Q, that involves tracking attributes of an object captured by a service call, R, in the body of Q, and an integrity constraint, C, on the rate of change of attributes in R. An SQO-style transformation of $\Delta Q/\Delta R$ based on C gets us much of the way to the generation of $MinTTI_Q$. We have done preliminary research on using a combination of semantic query optimization and program transformations to automatically generate *MinTTI*. We believe this approach will be effective for an important class of requests and constraints, and that it is a promising area for future work. Auto-generation of *MinTTI* would fill an important gap in applying application semantics to the optimization of ASIS requests.

Finally, returning to work on materialized views, significant work in view maintenance has dealt with determining when an update can be categorized as independent of a view, without requiring that view maintenance queries be evaluated (Blakeley 1989; Elkan 1990; Sagiv 1993). Researchers have applied various forms of syntactic analysis to this problem, including analysis of query reachability and of uniform equivalence. Our goal is similar, in that we too want to determine when an update cannot possibly affect the result of a view. But we employ semantic knowledge to make that determination. So while we would characterize all of the previous work in this area as determining when an update is *syntactically* independent of a view, this is the first work,

to our knowledge, that attempts to determine when an update is *semantically* independent of a view.

10.3 Lessons Learned

We close this dissertation with a narrative highlighting lessons we have learned in studying and implementing active service integration technology. Future implementers of active service integration systems would do well to heed these lessons.

Lesson 1: Hardware alone does not an efficient or scalable ASIS make.

Intel and other hardware vendors will love active service integration. An active service integration system is resource-intensive in terms of both CPU and memory. But while you can expect an ASIS to run on a large box, creating a system that is scalable and efficient is all about the software. An ASIS requires a complex software infrastructure. Building that infrastructure is a significant undertaking, along the lines of building other complex server software such as a database management system. Careful system design for scalability and efficiency is essential.

Lesson 2: Ignore costs at your peril.

Our experience building an ASIS demonstrates that the approach of adapting and extending cost-based query processing technology to service integration is both feasible and beneficial. The declarative request syntax enabled by this approach is convenient for the client. More importantly, however, adaptive plan choice based on costs is essential to efficiency, throughput, and scalability. Many networked services are data intensive, and for many requests large differences in the processing demands and responsiveness of

alternative feasible plans can be observed. Moreover, a given plan can be cheap or expensive depending on service characteristics that may change over time. To a large degree, the performance characteristics of an ASIS depends on the plan choices the system makes. Bad choices can have severely deleterious effects beyond the request being planned for, since an ASIS shares resources over large numbers of requests. Avoiding extremely bad plans is particularly important, which is one of the strengths of cost-based optimization.

Lesson 3: Expect inertia, but don't count on it.

Incremental processing of ASIS requests can often be very efficient. In general, the efficiency of incremental processing relies on the "law of inertia", which states that the change of information in a body of data will be small compared to the body of data as a whole. But the law of inertia is merely a heuristic. While this heuristic may hold overwhelmingly in the world of large corporate databases, it can not be depended on as consistently in the network services environment, which will expose a greater variety of data, having a wider range of update characteristics. An ASIS should consider a plan search space that includes both incremental evaluation of a request and complete, from-scratch re-computation.

Lesson 4: Services are autonomous, but they shouldn't be oblivious.

Efficient ASIS request processing is aided tremendously by services that are complicit in the integration process. When services provide accurate and extensive metadata on their characteristics, and when they provide advanced "integration-friendly" operations, they enable an ASIS mediator to create and choose plans that are as efficient and scalable as possible. Some useful services can be all but unusable in the integration context without advanced operations. Services that hold very large volumes of data, for example, can often be integrated far more efficiently if they support some variation on a

semi-join operator, such as the parajoin operator we have described here. Services that support multiple methods of event notification, including the separation of event notification from event-related data retrieval, enable an ASIS mediator to consider a wider plan space that may include more efficient and scalable plans.

In the traditional view of federated systems, autonomy is paramount, and participating systems are not designed with federation in mind. In that setting, much of the integration support we have suggested here would be considered unreasonable. But the network services world is different. Much of the power of network services is derived from the view of services as components that can be freely integrated and combined. Network services exist to be integrated with other services, so they should provide explicit support for integration. One contribution of this dissertation is to describe some forms of integration support that are useful.

Lesson 5: Take what services give you.

If services offer capabilities that enable integration, it is incumbent upon integrators to exploit those capabilities. ASIS plan generation and optimization must be capability driven, it must incorporate and adapt to the processing capabilities and limitations of services. Useful network services can be counted on to display a wide range of heterogeneous capabilities. Some will have very limited capabilities, while others will offer advanced operational support for integration. A mediation system that cannot adapt to limited capabilities will be unable to employ many useful services. A system that cannot take advantage of advanced processing support will miss opportunities for more efficient and scalable plan execution. In the context of an ASIS, it is important to model and exploit notification and monitoring capabilities as well as more traditional request processing capabilities. In particular, the separation of the notification of an event and the retrieval of data associated with it enables more flexible and efficient plan generation.

In the Paradox system, we looked at the capabilities of a wide variety of information sources and services, and we attempted to reduce and abstract these capabilities to a generic model with wide applicability. We also explored capabilities that could be useful to efficient ASIS request processing, and we added a number of these with an eye towards what can be reasonably expected from an autonomous service. Our architecture relies on wrappers, when needed, to accomplish the final bit of adaptation to any specific source, with the caveat that wrappers should be lightweight and easy to implement. But the capability model alone should be sufficient for capturing all major cost elements in processing an ASIS request.

Our approach allowed us to adapt easily to services with unusual characteristics in the battleground domain, and to capture the cost characteristics of the environment. We emphasize that our specific metadata model is not the last word, far from it. System builders will want to explore the services of interest to them and, in a fashion similar to the one outlined above, adapt or extend our model to suit their primary domains of interest.

Lesson 6: If you know the future, use that knowledge.

As with other distributed information systems, caching is vitally important to the scalability and efficiency of an ASIS. Unlike most other information systems, however, an ASIS has excellent information available to it for making caching decisions, particularly if metadata on the frequency and data characteristics of change events can be obtained. We have outlined a feasible approach to cost-based cache selection that exploits the long-lived nature of ASIS requests, and we have described techniques for making it more efficient. Automated, cost-based approaches to physical database design have proved very useful based on less reliable heuristics on future workloads (i.e., that future workloads will mimic past work loads). The evidence strongly suggests that a cost-based approach to ASIS cache selection is not only feasible, but that it is highly effective.

Lesson 7: Share and share alike.

In this dissertation, we have repeatedly come back to the theme of inter-task sharing for implementing a scalable and efficient active service integration technology. We have demonstrated many opportunities for inter-task sharing that arise in the ASIS environment, and we have described methods for exploiting these opportunities that make use of heuristic techniques and incremental processing for efficiency. We have presented effective methods for sharing monitoring costs by merging monitoring requests, for sharing data materialized and maintained in a cache among multiple distinct requests and among multiple evaluations of the same request, for sharing transiently materialized data between simultaneously executing requests (MQO), and for sharing planning and optimization effort (MuxQO) in the context of cost-based cache selection and MQO. Each of these techniques represents a piece of the puzzle in building a scalable and efficient ASIS.

Lesson 8: Keep a human in the loop.

Several aspects of the design of the Paradox system leave room for manual intervention on the part of a system administrator. The metadata collection procedure, for example, allows a service to explicitly provide metadata on its capabilities and data characteristics. But where important metadata elements are missing or unreliable, they can be overridden by a system administrator. A plan suite is generated when a request is issued, but generated plans may become sub-optimal over time. An administrator controls when replanning is invoked. Experience with database query optimization indicates optimization may not always generate the best plans, and hand-optimization of plans may occasionally be needed. This task falls to a system administrator. Our initial scheme for exploiting the semantics of information change calls for an administrator to

specify the definition of *MinTTI* by hand based on service integrity constraints. A system administration is integral to an ASIS; an ASIS should be designed with this role in mind.

Lesson 9: Physics is your friend.

Scalability and efficiency will be a never-ending challenge in service integration. Even where all of the techniques described in this dissertation are applied, the complexity and volatility of communications networks, increasing numbers of services and volumes of data, and the increasing demands and sophistication of users will continue to stress the limits of our algorithms and computing infrastructure. One of the final and most useful places to turn to extend the applicability of integration technologies is to rich application semantics. In this dissertation, we present a framework for exploiting constraints on information change to improve the efficiency and scalability of an ASIS. Information change is rarely random. For the class of applications where information change can be constrained, whether by the laws of physics or by other semantic properties, our approach has the potential to provide dramatic performance benefits.

REFERENCES

Adali, S., K. S. Candan, et al. (1996). "Query Caching and Optimization in Distributed Mediator Systems." ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada:137-148.

Agrawal, S., S. Chaudhuri, et al. (2000). "Automated Selection of Materialized Views and Indexes for SQL Databases." Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt: 496-505.

Ahmed, F., P. De Smedt, et al. (1991). "The Pegasus heterogeneous multidatabase system." IEEE Computer **24**(12): 19-27.

Aho, A., J. Hopcroft, et al. (1983). Data Structures and Algorithms, Addison-Wesley.

Ambite, J. L., N. Ashish, et al. (1998). "Ariadne: A system for constructing mediators for Internet sources." ACM SIGMOD International Conference on Management of Data, Seattle, WA: 561-563.

Amsaleg, L., M. Franklin, et al. (1996). "Scrambling Query Plans to Cope With Unexpected Delays." Proceedings of the 4th International Conference on Parallel and Distributed Information Systems, Miami Beach, FL: 208-219.

Amsaleg, L., M. Franklin (1998). "Cost Based Query Scrambling for Initial Delays." ACM SIGMOD International Conference on Management of Data, Seattle, WA: 130-141.

Astrahan, M., M. Blasgen, et al. (1976). "System R: Relational Approach to Database Management." IEEE Transactions on Database Systems **1**(2): 97-137.

Babb, E. (1979). "Implementing a Relational Database by Means of Specialized Hardware." IEEE Transactions on Database Systems **4**(1): 1-29.

Benninghoff, P., and S. Noh (2001) "Improving Robustness of Distributed Condition Monitoring by Exploiting the Semantics of Information Change," AAAI Spring Symposium, Stanford, California: 10-17.

Benninghoff, P., and S. Noh (2003), "Scaling Up Information Updates in Distributed Condition Monitoring." Proceedings of the 5th International Conference on Enterprise Information Systems, Angers, France: 31-40.

Bernstein, P. A. and D.-M. W. Chiu (1981). "Using Semi-Joins to Solve Relational Queries." Journal of the ACM **28**(1): 25-40.

Bernstein, P. A., N. Goodman, et al. (1981). "Query Processing in a System for Distributed Databases (SDD-1)." IEEE Transactions on Database Systems **6**(4): 602-625.

Blakeley, J. A., P.-Å. Larson, et al. (1986). "Efficiently Updating Materialized Views." ACM SIGMOD International Conference on Management of Data, Washington, D.C.: 61-71.

Blakeley, J. A., N. Coburn, Larson, P. A. (1989). "Updating derived relations: detecting irrelevant and autonomously computable updates." IEEE Transactions on Database Systems **14**(3): 369-400.

Bohannon, P., J. Freire, et al. (2002). "From XML Schema to Relations: A Cost-based Approach to XML Storage." International Conference on Data Engineering, San Jose, CA: 1-12.

Bradley, S. P. and R. L. Nolan, Eds. (1998). Sense & Respond: Capturing Value in the Network Era, Harvard Business School Press.

Bray, T., J. Paoli, et al. (2000). Extensible Markup Language (XML) 1.0, World Wide Web Consortium. Url: http://www.w3.org/TR/REC-xml.

Carey, M. J., D. J. DeWitt, et al. (1988). "A Data Model and Query Language for EXODUS." ACM SIGMOD International Conference on Management of Data, Chicago, IL: 413-423.

Carey, M. J., L. M. Haas, et al. (1995). "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach." RIDE-DOM, Taipei, Taiwan: 134-141.

Chakravarthy, U. S., J. Grant, et al. (1988). "Foundations of semantic query optimization for deductive databases." In Foundations of DeductiveDatabases and Logic Programming. J. Minker. Los Altos, CA, Morgan Kaufmann: 243-273.

Chakravarthy, U. S., J. Grant, et al. (1990). "Logic-Based Approach to Semantic Query Optimization." IEEE Transactions on Database Systems **15**(2): 162-207.

Chaudhuri, S. and V. Narasayya (1997). "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server." Proceedings of the 23rd International Conference on Very Large Databases, Athens, Greece: 146-155.

Chen, C. and N. Roussopoulos (1994). "Adaptive Selectivity Estimation Using Query Feedback." ACM SIGMOD International Conference on Management of Data, Minneapolis, MN: 161-172

Chen, J., D. DeWitt, et al. (2000). "NiagaraCQ: A scalable continuous query system for internet databases." ACM SIGMOD International Conference on Management of Data, Dallas, TX: 379-390.

Choenni, S., H. Blanken, et al. (1993). "Index Selection in Relational Databases." Fifth International IEEE Conference on Computing and Information, Sudbury, Canada: 491-496.

Choenni, S., H. Blanken, et al. (1993). "On the Automation of Physical Database Design." ACM Symposium on Applied Computing, Indianapolis, IN: 358-367.

Chon, H., D. Agrawal, et al. (2002). "Data Management for Moving Objects." Data Engineering Bulletin **25**(2): 41-47.

Clearwater, S., Ed. (1996). Market-Based Control: A Paradigm for Distributed Resource Allocation, World Scientific Press.

Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM **13**(6): 377-387.

Cohen, P. R., Cheyer, A., Wang, M., and Baeg, S. C. (1994). "An open agent architecture." AAAI Spring Symposium: 197-204.

Czerwinski, E., B. Zhao, et al. (1999). "An Architecture for a Secure Service Discovery Service." International Conference on Mobile Computing and Networking, Seattle, WA: 24-35.

Dar, S., M. Franklin, et al. (1996). "Semantic Data Caching and Replacement." Proceedings of the 22nd International Conference on Very Large Databases, Bombay, India: 330-341.

Dogac, A., U. Halici, et al. (1996). "METU Interoperable Database System." ACM SIGMOD International Conference on Management of Data, Montreal, Canada: 552-561.

Du, W., R. Krishnamurthy, et al. (1995). "Query optimization in heterogeneous DBMS." Proceedings of the 21st International Conference on Very Large Databases, Vancouver, Canada: 277-291.

Elkan, C. (1990). "Independence of Logic Database Queries and Updates." ACM SIGACT-ACM SIGMOD-SIGART Symposium on Principles of Database Systems, Atlantic City, NJ: 154-160.

Elmagarmid, A., M. Rusinkiewicz, et al., Eds. (1999). Management of Heterogeneous & Autonomous Database Systems, Morgan Kaufmann.

Epstein, R. S., M. Stonebraker, et al. (1978). "Distributed query processing in a relational data base system." ACM SIGMOD International Conference on Management of Data, Austin, Texas: 169-180.

Fallside, D. C. (2001). "XML Schema Part 0: Primer." World Wide Web Consortium. Url: http://www.w3.org/TR/xmlschema-0.

Fankhauser, P., G. Gardarin, et al. (1998). "Experiences in Federated Databases: From IRO-DB to MIRO-Web." Proceedings of the 24th International Conference on Very Large Databases, New York, NY: 655-658.

Finkelstein, S., M. Schkolnick, et al. (1988). "Physical Database Design for Relational Databases." ACM Transactions on Database Systems. **13**(1): 91-128.

Florescu, D., D. Koller, et al. (1997). "Using Probabilistic Information in Data Integration." Proceedings of the 23rd International Conference on Very Large Databases, Athens, Greece: 216-225.

Florescu, D. and A. Y. Levy (1998). "Recent Progress in Data Integration - A Tutorial." East European Symposium on Advances in Databases and Information Systems, Poznan, Poland: 1-2.

Florescu, D., A. Y. Levy, et al. (1999). "Query Optimization in the Presence of Limited Access Patterns." ACM SIGMOD International Conference on Management of Data, Philadephia, PA: 311-322.

Fox, A., S. Gribble, et al. (1997). "Scalable Network Services." ACM Symposium on Operating Systems Principles, St. Malo, France: 78-91.

Franklin, M. J. and S. B. Zdonik (1998). "Data In Your Face: Push Technology in Perspective." ACM SIGMOD International Conference on Management of Data, Seattle, WA: 516-519.

Fulghum, R. (1993). All I Really Need to Know I Learned in Kindergarten: Uncommon Thoughts on Common Things, Ivy Books.

Garcia-Molina, H., J. D. Ullman, et al. (2000). Database System Implementation, Prentice Hall.

Gardarin, G., F. Sha, et al. (1996). "Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System." Proceedings of the 22nd International Conference on Very Large Databases, Bombay, India: 378-389.

Godfrey, P. and J. Gryz (1997). "Semantic Query Caching for Heterogeneous Databases." Proceedings of the 4th Knowledge Representation Meets Databases Workshop, Athens, Greece: 6.1-6.6.

Graefe, G. (1990). "Encapsulation of Parallelism in the Volcano Query Processing System." ACM SIGMOD Conference on the Management of Data: 102-111.

Graefe, G. (1993). "Query Evaluation Techniques for Large Databases." ACM Computing Surveys **25**(2): 73-170.

Graefe, G. (1994). "Volcano - An Extensible and Parallel Query Evaluation System." Transactions on Knowledge and Data Engineering **6**(1): 120-135.

Graefe, G. (1995). "The Cascades Framework for Query Optimization." Data Engineering Bulletin **18**(3): 19-29.

Graefe, G. and D. J. DeWitt (1987). "The EXODUS Optimizer Generator."ACM SIGMOD International Conference on Management of Data, San Francisco, CA: 160-172.

Graefe, G. and W. J. McKenna (1993). "The Volcano Optimizer Generator: Extensibility and Efficient Search." IEEE International Conference on Data Engineering, Vienna, Austria: 209-218.

Grant, J., J. Gryz, et al. (1997). "Semantic Query Optimization for Object Databases." IEEE International Conference on Data Engineering: 444-453.

Gupta, A. and I. S. Mumick (1995). "Maintenance of Materialized Views: Problems, Techniques, and Applications." Data Engineering Bulletin **18**(2): 3-18.

Gupta, A., I. S. Mumick, et al. (1993). Maintaining Views Incrementally. ACM SIGMOD International Conference on Management of Data, Washington, D.C.: 383-394.

Gupta, H. (1997). Selection of Views to Materialize in a Data Warehouse. International Conference on Database Theory. Delphi, Greece: 98-112.

Gupta, H., V. Harinarayan, et al. (1997). Index Selection for OLAP. International Conference on Data Engineering. Manchester, UK: 208-219.

Gupta, H. and I. S. Mumick (1999). Selection of Views to Materialize Under a Maintenance-Time Constraint. International Conference on Database Theory. Jerusalem, Israel: 156-165.

Haas, L. M., D. Kossmann, et al. (1997). Optimizing queries across diverse data sources. Proceedings of the 23rd International Conference on Very Large Databases, Athens, Greece: 276-285.

Haas, L. M., P. G. Selinger, et al. (1982). "R*: A Research Project on Distributed Relational DBMS." Data Engineering Bulletin **5**(4): 28-32.

Halevy, A. Y., A. O. Mendelzon, et al. (1995). "Answering Queries Using Views." ACM SIGACT-ACM SIGMOD-SIGART Symposium on Principles of Database Systems, San Jose, California: 95-104.

Hammer, J., H. Garcia-Molina, et al. (1995). "The Stanford Data Warehousing Project." IEEE Data Engineering Bulletin: 41-48.

Hammer, M. and A. Chan (1976). "Index Selection in a Self-Adaptive Database Management System." ACM SIGMOD International Conference on Management of Data. Washington, D.C.: 55-64.

Harinarayan, V., A. Rajaraman, et al. (1996). "Implementing Data Cubes Efficiently." ACM SIGMOD International Conference on Management of Data, Montreal, Quebec: 205-214.

Hobbs, L. and K. England (1991). Rdb/MVS A Comprehensive Guide, Digital Press.

Howes, T., M. C. Smith, et al. (1999). Understanding and Deploying Ldap Directory Services, MacMillan.

Ives, Z., D. Florescu, et al. (1999). "An Adaptive Query Execution System for Data Integration." ACM SIGMOD International Conference on Management of Data, Philadephia, PA: 299-310.

Jenson, C. and S. Saltenis (2002). "Towards Increasingly Update Efficient Moving-Object Indexing." Data Engineering Bulletin **25**(2): 35-40.

Josifovski, V., P. M. Schwarz, et al. (2002). "Garlic: A New Flavor of Federated Query Processing for DB2." ACM SIGMOD International Conference on Management of Data, Madison, WI: 524-532.

Joy, B., G. Steele, et al. (2000). The Java Language Specification, Second Edition, Addison-Wesley.

Kabra, N. and D. DeWitt (1998). "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans." ACM SIGMOD International Conference on Management of Data, Seattle, WA: 106-117.

Keller, A. M. and J. Basu (1996). "A Predicate-based Caching Scheme for Client-Server Database Architectures." VLDB Journal **5**(1): 35-47.

Kimball, R., L. Reeves, et al. (1998). The Data Warehouse Lifecycle Toolkit : Expert Methods for Designing, Developing, and Deploying Data Warehouses, John Wiley & Sons.

King, J. J. (1981). "Query Optimization by Semantic Reasoning." PhD Thesis, Computer Science Department, Stanford University, Stanford, CA.

Knoblock, Y. A. (1993). "SIMS: Retrieving and integrating information from multiple sources." ACM SIGMOD International Conference on Management of Data, Washington, D.C.: 562-563.

Koch, G. and K. Loney (1997). Oracle8: The Complete Reference, Osborne McGraw-Hill.

Korth, H. and A. Silbershatz (1991). Database System Concepts, 2nd Edition, McGraw-Hill, Inc.

Kossman, D. (2000). "The State of the Art in Distributed Query Processing." ACM Computing Surveys **32**(4): 422-469.

Kumar, S. C., P.R.; Levesque, H.J. (2000). "The Adaptive Agent Architecture: Achieving Fault-Tolerance Using Persistent Broker Teams." International Conference on Multi-Agent Systems, Boston, MA: 159-166.

Kwok, C. T. and D. S. Weld (1996). "Planning to Gather Information." Proceedings of the Conference of the American Association of Artificial Intelligence, Portland, OR: 32-39.

Levy, A. and D. Lomet, Eds. (June 2000). Data Engineering Bulletin, Special Issue on Adaptive Query Processing.**23**(2).

Levy, A., A. Rajaraman, et al. (1996). "Querying Heterogeneous Information Sources Using Source Descriptions." Proceedings of the 22nd International Conference on Very Large Databases, Bombay, India: 251-262.

Levy, A. Y., A. Rajaraman, et al. (1996). "Answering Queries Using Limited External Processors." ACM SIGMOD International Conference on Management of Data, Montreal, Canada: 227-237.

Levy, A. Y. and Y. Sagiv (1993). Queries Independent of Updates. Proceedings of the 19th International Conference on Very Large Databases, Dublin, Ireland: 171-181.

Levy, A. Y. and Y. Sagiv (1995). "Semantic Query Optimization in Datalog Programs." ACM SIGACT-ACM SIGMOD-SIGART Symposium on Principles of Database Systems. San Jose, CA: 163-173.

Li, C. and E. Chang (2000). "Query Planning with Limited Source Capabilities." IEEE International Conference on Data Engineering, San Diego, CA: 401-412.

Li, C., R. Yerneni, et al. (1998). "Capability Based Mediation in TSIMMIS." ACM SIGMOD International Conference on Management of Data, Seattle, WA: 564-566.

Lindsay, B. G., L. M. Haas, et al. (1984). "Computation and Communication in R*: A Distributed Database Manager." IEEE Transactions on Computer Systems **2**(1): 24-38.

Lipton, R. J., J. F. Naughton, et al. (1990). "Practical Selectivity Estimation through Adaptive Sampling." ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ: 1-11.

Litwin, W., A. Pfeffer, et al. (1996). "Mariposa: A Wide-Area Distributed Database System." VLDB Journal **5**(1): 48-63.

Liu, L., C. Pu, et al. (1996). "Differential Evaluation of Continual Queries." IEEE Conference on Distributed Computing Systems, Hong Kong: 458-465.

Liu, L., C. Pu, et al. (1999). "Continual Queries for Internet Scale Event-Driven Information Delivery." IEEE Transactions on Knowledge and Data Engineering **11**(4): 610-628.

Liu, L., C. Pu, et al. (1998). "CQ: A Personalized Update Monitoring Toolkit." ACM SIGMOD International Conference on Management of Data, Seattle, WA: 547-549.

Lohman, G. (1988). "Grammar-like Functional Rules for Representing Query Optimization Alternatives." ACM SIGMOD International Conference on Management of Data, Chicago, IL: 18-27.

Lowe-Norris, A. G. (2000). Windows 2000 Active Directory, O'Reilly.

Lu, H. and M. J. Carey (1985). "Some Experimental Results on Distributed Join Algorithms in a Local Network." Proceedings of the 11th International Conference on Very Large Databases, Stockholm, Sweden: 108-121.

Luo, Q., J. Naughton, et al. (2000). "Active Query Caching for Database Web Servers." WebDB Workshop, Dallas, TX: 92-101.

M. Franklin, L. A., A. Tomasic (1998). "Dynamic Query Operator Scheduling for Wide-Area Remote Access." Journal of Distributed and Parallel Databases **6**(3): 217-246.

Mackert, L. F. and G. M. Lohman (1986). "R* Optimizer Validation and Performance Evaluation for Distributed Queries." Proceedings of the 12th International Conference on Very Large Databases, Kyoto, Japan:84-95.

Michies, D. (1968). "Memo Functions and Machine Learning." Nature(218): 19-22.

Mistry, H., P. Roy, et al. (2001). "Materialized View Selection and Maintenance Using Multi-Query Optimization." ACM SIGMOD International Conference on Management of Data, .Santa Barbara, CA: 201-212.

Mumick, I. S., D. Quass, et al. (1997). "Maintenance of Data Cubes and Summary Tables in a Warehouse." ACM SIGMOD International Conference on Management of Data, Tucson, AZ: 100-111.

Naacke, H., G. Gardarin, et al. (1998). "Leveraging Mediator Cost Models with Heterogeneous Data Sources." IEEE International Conference on Data Engineering. Orlando, FL: 351-360.

Naughton, J., D. DeWitt, et al. (2001). "The Niagara Internet Query System." IEEE Data Engineering Bulletin **24**(2): 27-33.

Ono, K. and G. M. Lohman (1990). "Measuring the Complexity of Join Enumeration in Query Optimization." Proceedings of the 16th International Conference on Very Large Databases, Brisbane, Australia: 314-325.

Orfali, R., D. Harkey, et al. (1995). The Essential Distributed Objects Survival Guide, Wiley & Sons.

Papadias, D., Y. Tao, et al. (2002). "Indexing and Retrieval of Historical Aggregate Information about Moving Objects." Data and Knowledge Engineering **25**(2): 10-17.

Patterson, D. A. (1997). "How to Have a Bad Career in Research/Academia." Presentation at the CRA Academic Careers Workshop, Denver, CO. Url: http://www.cs.berkeley.edu/~pattrsn/talks/research.pdf.

Pitt, E., K. McNiff, et al. (2001). Java.rmi : The Remote Method Invocation Guide, Addison Wesley.

Pottinger, R. and A. Y. Halevy (2000). "A Scalable Algorithm for Answering Queries Using Views." Proceedings of the 26th International Conference on Very Large Data Bases, Cairo, Egypt: 484-495.

Qian, X. and G. Wiederhold (1991). "Incremental Recomputation of Active Relational Expressions." Transactions on Knowledge and Data Engineering **3**(3): 337-341.

Rabinovich, M. (1998). "Caching and Replication on the Internet." Tutorial presented at the 24th International Conference on Very Large Databases, New York, NY. Url for presentation slides: http://www.research.att.com/~misha/tutorial/slides.ps.gz.

Raggett, D., A. L. Hors, et al. (1998). HTML 4.0 Specification (W3C Recommendation). Url: http://www.w3.org/TR/1998/REC-html40-19980424.

Ranadive, V. and S. McNealy (1999). The Power of Now: How Winning Companies Sense and Respond to Change Using Real-Time Technology, McGraw-Hill.

Roantree, M., W. Hasselbring, et al., Eds. (2000). Engineering Federated Information Systems, IOS Press.

Ross, K., D. Srivastava, et al. (1996). "Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time." ACM SIGMOD International Conference on Management of Data, Montreal, Canada: 447-458.

Roth, M. T., M. Arya, et al. (1996). "The Garlic Project." ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada: 557-567.

Roth, M. T. and P. M. Schwarz (1997). "Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources." Proceedings of the 23rd International Conference on Very Large Databases, Athens, Greece: 266-275.

Roussopoulos, N., C. Chen, et al. (1995). "The ADMS Project: Views R Us." IEEE Data Engineering Bulletin **18**(2): 19-28.

Rozen, S. and D. Shasha (1991). "A Framework for Automating Physical Database Design." Proceedings of the 17th International Conference on Very Large Databases, Barcelona, Spain: 410-411.

Russel, S. and P. Norvig (1995). Artificial Intelligence: A Modern Approach. Prentice Hall.

S. Chawathe, et. al (1994). "The TSIMMIS Project: Integration of Heterogeneous Information Sources." Proceedings of the 16th Meeting of the Information Processing Society of Japan. Tokyo, Japan: 7-18.

Sagiv, Y. and A. Levy (1993). "Queries independent of updates." Proceedings of the 19th International Conference on Very Large Databases, Dublin, Ireland: 171-181.

Salem, K., K. S. Beyer, et al. (2000). "How To Roll a Join: Asynchronous Incremental View Maintenance." ACM SIGMOD International Conference on Management of Data, Dallas, Texas: 129-140.

Scheuerman, P., J. Shim, et al. (1996). "WATCHMAN: A Data Warehouse Intelligent Cache Manager." Proceedings of the 22nd International Conference on Very Large Data Bases, Bombay, India: 51-62.

Schwartz, D. G. (1993). "Cooperating Heterogeneous Systems: A Blackboard-based Meta Approach." Technical Report 93-112, Center for Automation and Intelligent Systems Research, Case Western Reserve University, Cleveland, OH.

Selinger, P., M. Astrahan, et al. (1979). "Access Path Selection in a Relational Database Management System." ACM SIGMOD International Conference on Management of Data, Boston, MA: 23-34.

Sellis, T. K. (1988). "Multiple-Query Optimization." IEEE Transactions on Database Systems **13**(1): 23-52.

Shapiro, L., D. Maier, et al. (2001). "Exploiting Upper and Lower Bounds In Top-Down Query Optimization." International Database Engineering and Applications Symposium, Grenoble, France: 20-33.

Shukla, A., P. M. Deshpande, et al. (1998). "Materialized View Selection for Multidimensional Datasets." Proceedings of the 24th International Conference on Very Large Databases, New York, NY: 488-499.

Skeen, D. (1983). " A Formal Model of Crash Recovery in a Distributed System." IEEE Transactions on Software Engineering **9**(3): 219-228.

Smith, J. M., P. A. Bernstein, et al. (1981). "MULTIBASE: Integrating Heterogeneous Distributed Database Systems." National Computer Conference: 487-499.

Smith, R. (1980). "The Contract Net Protocol: High Level Communication and Control in Distributed Problem Solver." IEEE Transactions on Computers **29**(12): 1104-1113.

Steinbrunn, M. (1996). Heuristic and Randomized Optimization Techniques in Object-Oriented Database Systems. Infix-Verlag, Germany.

Stonebraker, M. and E. J. Neuhold (1977). "A Distributed Database Version of INGRES." Berkeley Workshop on Distributed Data Management and Computer Networks: 19-36.

Stonebraker, M., E. Wong, et al. (1976). "The Design and Implementation of INGRES." TODS **1**(3): 189-222.

Tomasic, A., L. Raschid, et al. (1998). "Scaling Access to Heterogeneous Data Sources with DISCO." IEEE Transactions On Knowledge and Data Engineering **10**(5): 808-823.

Tomlin, C. (1990). Geographic Information Systems and Cartographic Modeling, Prentice Hall.

Ullman, J. D. (1988). Principles of Database and Knowledge-Base Systems, Volumes 1 and 2, Computer Science Press.

Ullman, J. D. (1997). "Information Integration Using Logical Views." International Conference on Database Theory, Delphi, Greece:19-40.

Valduriez, P. and G. Gardarin (1984). "Join and Semijoin Algorithms for a Multiprocessor Database Machine." IEEE Transactions on Database Systems **9**(1): 133-161.

Vance, B. (1998). Join-order Optimization with Cartesian Products. Ph.D. Thesis, Computer Science and Engineering Department, Oregon Graduate Institute of Science and Technology, Beaverton, OR.

Venkata, K., R. Kanth, et al. (2002). "Spatio-Temporal Indexing in Oracle: Issues and Challenges." Data Engineering Bulletin **25**(2): 56-60.

Weld, M. F. (1997). "Efficiently Executing Information-Gathering Plans." International Joint Conference on Artificial Intelligence, Nagoya, Japan: 785-791.

Wellman, M. P. (1993). "A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems." Journal of Artificial Intelligence Research **1**: 1-23.

Wellman, M. P. (1995). "The Economic Approach to Artificial Intelligence." ACM Computing Surveys **27**(3):360-362.

Widom, J. (1993). "Deductive and Active Databases: Two Paradigms or Ends of a Spectrum?." International Workshop on Rules in Database Systems, Edinburgh, Scotland: 306-315.

Widom, J. (1994). "Research Issues in Active Database Systems: Report from the Closing Panel at RIDE-ADS'94." ACM SIGMOD Record **25**(3): 41-43.

Widom, J. (1995). "Research Problems in Data Warehousing." International Conference on Information and Knowledge Management, Baltimore, MD: 25-30.

Widom, J. and S. Ceri (1996). Active Database Systems: Triggers and Rules for Advanced Database Processing. Prentice Hall.

Wiederhold, G. (1992). "Mediators in the Architecture of Future Information Systems." IEEE Computer **25**(3): 38-49.

Winer, D. (2002). "The Meaning of the Google API." Url: http://www.davenet.com.

Woelk, D., B. Bohrer, et al. (1995). "Carnot and InfoSleuth: Database technology and the World Wide Web." ACM SIGMOD International Conference on Management of Data, San Jose, CA: 443-444.

Zhu, Q. and P.-Å. Larson (1998). "Solving Local Cost Estimation Problem for Global Query Optimization in Multidatabase Systems." Distributed and Parallel Databases **6**(4): 373-421.

Zhuge, Y., H. Garcia-Molina, et al. (1995). "View Maintenance in a Warehousing Environment." ACM SIGMOD International Conference on Management of Data,. San Jose, CA: 316-327.

Zhuge, Y., H. Garcia-Molina, et al. (1998). "Consistency Algorithms for Multi-Source Warehouse View Maintenance." Journal of Distributed and Parallel Databases **6**(1): 7-40.

Biographical Note

Paul Benninghoff has worked as a house painter in Connecticut, a prep cook in Vail, Colorado, and an olive picker in Crete. He has also held a variety of jobs related to the software industry, including positions in marketing, management consulting, research, and development. Paul holds a bachelor's degree in math and economics from Middlebury College, and a master's degree in computer science from New York University. He is currently employed as a Senior Developer at Oracle Corporation.