Efficient Query Processing for Data Integration

Zachary G. Ives

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2002

Program Authorized to Offer Degree: Department of Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Zachary G. Ives

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

_____

Alon Halevy

Reading Committee:

_____

Alon Halevy (chair)

_____

Daniel Weld

_____

Dan Suciu

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, or to the author.

Signature_____

Date_____

University of Washington

Abstract

Efficient Query Processing for Data Integration

by Zachary G. Ives

Chair of Supervisory Committee:

Professor Alon Halevy
Computer Science and Engineering

A major problem today is that important data is scattered throughout dozens of separately evolved data sources, in a form that makes the "big picture" difficult to obtain. Data integration presents a unified virtual view of all data within a domain, allowing the user to pose queries across the complete integrated schema.

This dissertation addresses the performance needs of real-world business and scientific applications. Standard database techniques for answering queries are inappropriate for data integration, where data sources are autonomous, they generally lack mechanisms for sharing of statistical information about their content, and the environment is shared with other users and subject to unpredictable change. My thesis proposes the use of *pipelined* and *adaptive* techniques for processing data integration queries, and I present a unified architecture for adaptive query processing, including novel algorithms and an experimental evaluation. An operator called *x-scan* extracts the relevant content from an XML source as streams across the network, which enables more work to be done in parallel. Next, the query is answered using algorithms (such as an extended version of the *pipelined hash join*) whose work is adaptively scheduled, varying to accommodate the relative data arrival rates of the sources. Finally, the system can adapt the ordering of the various operations (the *query plan*), either at points where the data is being saved to disk or in mid-execution, using a novel technique called *convergent query processing*. I show that these techniques provide significant benefits in processing data integration queries.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Portions of this dissertation have previously been published in SIGMOD [IFF+99] and in the IEEE Data Engineering Bulletin [IHW01, ILW+00]. However, these portions have been significantly revised and extended within this dissertation. Additionally, portions of Chapter 4 have been submitted concurrently to the *VLDB Journal*.

It's amazing to look back and see how things have changed over the past few years, how others have influenced me. I've been blessed with a brilliant and incredibly generous group of collaborators and advisors, who've given me great ideas and shaped my research ideas and my thesis work. Special thanks to my advisors, Alon Halevy and Dan Weld, who let me be creative but kept me on track, and with whom I've had countless stimulating discussions that led to new ideas. Even more importantly, they taught me choose worthwhile problems and aim high. I have come to understand just how critical this is in the research world.

Thanks also to Steve Gribble, Hank Levy, and Dan Suciu for showing me a broader perspective on my research topics — it really helps to see a problem from an outsider's perspective in many cases — and for many fruitful discussions and arguments. And I am greatly appreciative of the amount of work they put into preparing me for the interview circuit, despite the fact that I did not have a formal advisee relationship with them.

I'm also grateful to Igor Tatarinov, Jayant Madhavan, Maya Rodrig, and Stefan Sariou for their contributions to the various projects in which I have participated these past few years. I greatly enjoyed their ideas and their enthusiasm, and I learned a great deal from working with them. Many of these people have also been invaluable sources of comments on my papers.

A special thank-you is warranted for Rachel Pottinger, who has been a fellow "databaser" and constant source of encouragement since the beginning — and

to Steve Wolfman as well, who, while not a database person, has been a good friend and source of feedback. They have been perhaps the best exemplars of why this department has a stellar reputation as a place to work.

I'd also like to express my appreciation for those who contributed suggestions and feedback to my work, even if they weren't officially affiliated with it: Corin Anderson, Phil Bernstein, Luc Bouganim, Neal Cardwell, Andy Collins, AnHai Doan, Daniela Florescu, Dennis Lee, Hartmut Liefke, David Maier, Ioana Manolescu, Oren Zamir, and the anonymous reviewers of my conference and journal submissions. There is no doubt that it is richer and more complete as a result.

This research was funded in part by ARPA / Rome Labs grant F30602-95-1-0024, Office of Naval Research Grant N00014-98-1-0147, by National Science Foundation Grants IRI-9303461, IIS-9872128, and 9874759, and by an IBM Research Fellowship.

DEDICATION

To Mom, Dad, and Joyce, my first teachers, who inspired me to learn about the world and teach others; to my many teachers, professors, and advisors since, who helped mold me into what I am today; and to Long, who has given me a new level of inspiration, support, and motivation. May God grant me the privilege of making as positive an impact on others as you've all had on me...

Chapter 1

INTRODUCTION

The processing of queries written in a declarative language (e.g., SQL or XQuery) has been a subject of intense study since the origins of the relational database system, with IBM's System-R [SAC+79] and Berkeley's Ingres [SWKH76] projects from the 1970s. The standard approach has been to take a declarative, user-supplied query and to try to select an order of evaluation and the most appropriate algorithmic implementations for the operations in the query — these are expressed within a *query plan*. The query plan is then executed, fetching data from source relations and combining it according to the operators to produce results.

System-R established a standard approach to query processing that is still followed today. This approach is very similar to compilation and execution of traditional languages: a *query optimizer* statically compiles the query into a plan, attempting to pick the most efficient plan based on its knowledge about execution costs and data, and then a *query execution engine* runtime is responsible for executing the query plan. This paradigm relies on having a rich set of information available to the optimizer: disk and CPU speeds, table sizes, data distributions, and so on are necessary for good cost estimation. These statistics are computed offline, and they are expected to stay relatively constant. They become inputs into the query optimizer's *cost model*, which consists of functions that estimate the cost of each operation given its expected input data, and which also have a model for composing separate costs. The plan that has the cheapest estimated cost is generally chosen (although, for reasons of efficiency, not all optimizers exhaustively enumerate all possible plans).

The System-R model has been extremely successful in practice, and it is well-suited for domains where computation and I/O costs are predictable, the cost model is highly accurate, representative statistics about the data are available, and data characteristics are essentially regular throughout the span of query execution. It is poorly suited for any situations in which these criteria are not satisfied.

One area that, unfortunately, falls into this area, is that of data integration. In data integration (described in more detail in the next subsection), we focus on the problem of querying across and combining data from multiple autonomous, heterogeneous data sources, all under a common virtual schema. The data sources typically are not designed to support external querying or interaction with other sources, and each source is often a shared resource used by many people and organizations. Furthermore, most interesting data integration applications tend to include data sources from different administrative domains, scattered through a wide-area network or even the Internet. As a result, I/O costs are often unpredictable, statistics about the data are difficult to obtain, and data characteristics could even change as execution progresses.

This dissertation presents and evaluates a set of techniques for *adaptively* processing queries, which allows a query processor to react to changing conditions or increased knowledge. In the remainder of this introduction, I motivate the problem of data integration (below), present my thesis and explain how it addresses the query processing problem (Section 1.2), and provide an outline of the dissertation (Section 1.3).

1.1 The Motivations for Data Integration

Over the past twenty years, the basic operating paradigm for data processing has evolved as computing technology technology itself has changed. We have moved from mainframe-based, centralized data management systems to networks of powerful PC clients, group servers, and the Internet. Recent trends in research suggest that we may ultimately be moving to an even more extreme, peer-based model in which all machines both consume and provide data and computation in a fully decentralized architecture.

Motivation for these changes has come not merely from more advanced hardware and networking technologies, but from a natural desire to decentralize control and administration of data and compute services. Not only does a centralized system generally form a bottleneck in terms of scaling *performance*, but the centralized computing model can be a scalability bottleneck in terms of *administration*. When data is "owned" and managed by numerous heterogeneous groups with different needs, a central schema is difficult to design, it typically relies on the development of standards *before* it can be constructed, and it is slow to adapt to the needs of its members. A decentralized collection of autonomous systems, however, can be much more dynamic, as

individual components can be separately designed and redesigned to meet the needs of their small user populations.

It is unsurprising, then, that today most enterprises, institutions, and formal collaborations — which typically are comprised of groups that are at least partly autonomous from one another — seldom operate with only centralized, common data management systems. Instead, individual groups often create their own separate systems and databases, each with the schema and data most relevant to their own needs. A recent study indicates that a typical large enterprise has an average of 49 different databases [Gro01]. Moreover, an organization's databases seldom represent all of the data it owns or accesses: in many cases, additional data is encoded in other formats such as documents, spreadsheets, or custom applications, and often today's organizations have collaborations with external entities (or make acquisitions of new groups) that may share certain data items.

Unfortunately, the common data management model, a decentralized collection of autonomous, heterogeneous systems typically suffers from a major shortcoming: there is no longer a single point of access to organizational data that can be queried and analyzed in a comprehensive form. The decentralized computation model provides great flexibility, but the centralized model provides uniformity and a global perspective.

Two solutions have been proposed to this problem, both of which are end-points along a broad continuum of possible implementations: data warehousing lies at one end of the spectrum, and "virtual" data integration at the other. Both approaches take a set of pre-existing decentralized data sources related to a particular domain, and they develop a single unified (*mediated*) schema for that domain. Then a series of transformations or *source mappings* are specified to describe the relationship between each data source and the mediated schema.

Data Integration vs. Warehousing

The primary philosophical difference between a data warehouse and a virtual data integration system is essentially one of "eager" versus "lazy" evaluation (see Figure 1.1). In data warehousing, the expectation is that the data changes infrequently or that the integrated view does not need to be current — and that large numbers of expensive queries will be posed over the integrated view of the data. Hence, the full contents of the global schema are precomputed (by evaluating all source mappings), they are

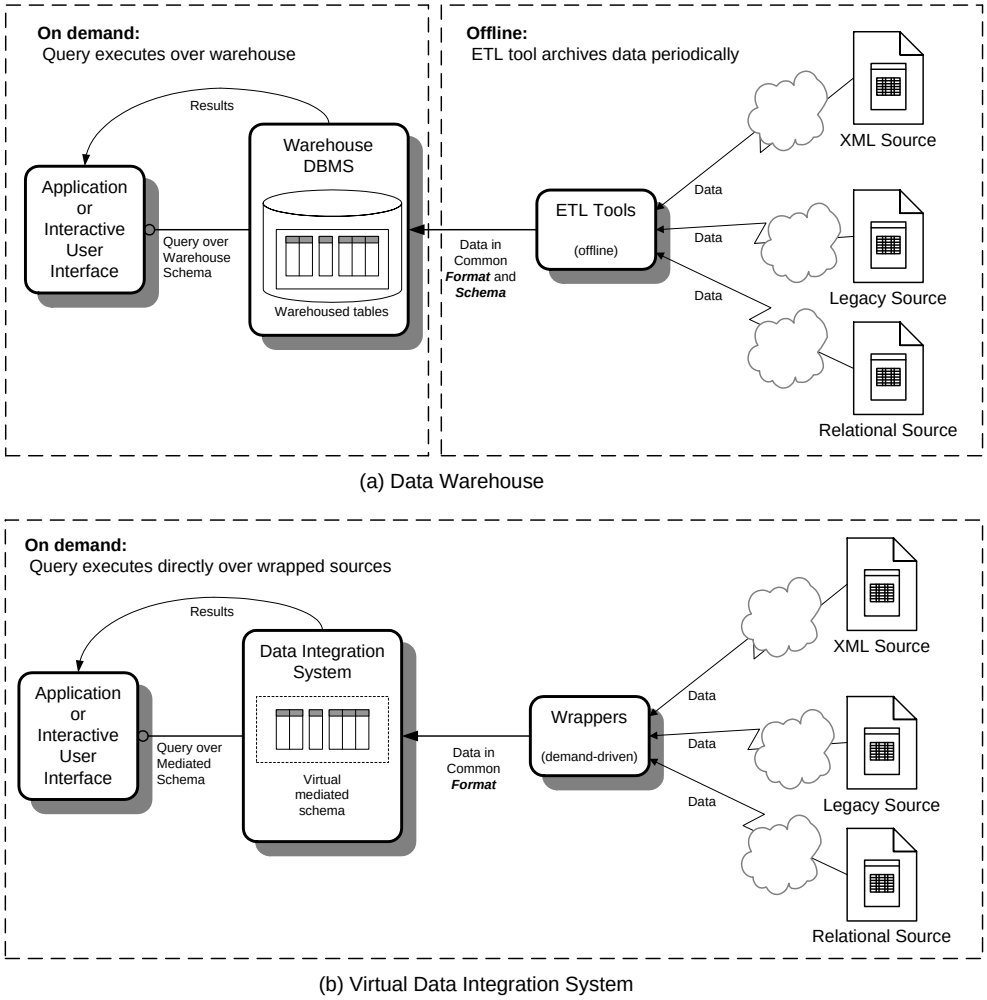


Figure 1.1: Data warehousing (**a**) replicates data from sources offline and executes its queries over the warehoused data. Virtual data integration (**b**) presents a virtual, mediated schema but fetches the actual data on-demand from the underlying data sources.

stored in a separate "warehouse" database that will be used for querying, and significant attention is given to physical warehouse design and indexing, in order to get the best possible query performance. Refreshing the warehouse is typically relatively expensive and done offline, using *ETL* (extract, transform, and load) tools.

Data integration addresses the case where warehousing is impractical, overly expensive, or impossible: for instance, when queries only access small portions of the data, the data changes frequently, "live" data is required, data-providing partners are

only willing to grant limited access to their data, or the global schema itself may be changed frequently. In fully virtual data integration, the global schema is strictly a logical or virtual entity — queries posed over it are dynamically rewritten at runtime to refer to actual data sources (based on the source mappings), and data is fetched from the sources (via *wrappers*) and combined. Data integration has become of increasing interest in recent years as it has matured, because it has several benefits to the implementer versus warehousing: it supports data sources that may only allow limited access to data; it supports a "live" view of the data environment; and it can present multiple versions of a mediated schema at the same time (for instance, to maintain compatibility with "legacy" queries).

One potential drawback of the virtual data integration approach is that certain data cleaning and semantic matching operations between sources are too expensive to perform "on the fly," and must be computed offline; another is that virtual data integration may heavily load the data sources. To handle these issues, an implementation whose characteristics fall between the fully precomputed model of the data warehouse and the fully virtual model of data integration may be desirable: certain data items or matching information may be cached, prefetched, or precomputed. In this thesis, I will consider all of these points (including full warehousing) to fall within the general category of data integration, but my interests lie in handling those sources that are not cached.

1.2 Query Processing for Data Integration

Until very recently, the emphasis of research in data integration was on developing models [BRZZ99, ZRZB01, ROH99], mappings [DDH01, MBR01, HMH01], and translators or "wrappers" [KDW97, BFG01] for data sources; with additional work on the problem of translating or "reformulating" [LRO96, Qia96, DG97, FLM99, PL00, HIST02] queries over the mediated schema into queries over the real sources. These problems have been addressed well enough to provide a level of functionality that is sufficient for solving many real-world problems. Now that there are established algorithms and methodologies for data integration, there are two important challenges remaining to be addressed to make data integration a widespread technology: one is the problem of defining correspondences between entities at different data sources (i.e., mappings between different concepts or schema items, but also mappings be-

tween entities that appear in different sources with different representations); the other is the problem of developing system-building techniques that allow a data integration system to perform well enough that it can be useful in practice. Other researchers [MBR01, BHP00, DDH01, PR01] have started to address aspects of the first problem; my focus is on the second problem, developing techniques for efficiently answering data integration queries.

As discussed earlier, traditional query processing deals with an environment in which statistics are computed offline and used to statically optimize a query, which can then be executed. This generally is effective in a standard database environment because the data and computing environment are under the strict control of the database, and they are thus fairly predictable and consistent. Yet even in this context, many simplifying assumptions must be made during optimization, and there are many situations in which the traditional model does poorly (e.g., in many circumstances, the optimizer accumulates substantial error in modeling complex queries).

Even worse, the data integration domain has a number of features that make it substantially harder than the conventional database management system context. Data integration typically interacts with autonomous sources and externally controlled networks: it becomes difficult to model the expected performance of such data sources (sometimes even the amount of data it will return is unknown), and the performance of the network may be unexpectedly bursty. Unpredictability and inconsistency are the norm in querying remote, autonomous data sources.

Furthermore, traditional query processing tends to optimize for complete results (i.e., batch-oriented queries). Many data integration applications are interactive, suggesting that early incremental results are particularly important in this context — suggesting the need for a different metric in query optimization, or at least a different set of heuristics.

1.2.1 Thesis Statement and Overview

My thesis is that since the data integration domain is inherently filled with uncertainty, static optimization and deterministic execution based on existing knowledge are ill-suited for this domain — *adaptive* techniques can provide greatly improved performance. The central question I sought to answer is whether it is possible to develop adaptive query processing techniques that yield better performance in our domain:

they must be sufficiently low-overhead that they will compensate for the fact that adaptivity is inherently more expensive. I have answered this question by proposing and experimentally validating novel techniques for utilizing *pipelining* (defined in the next section) and adaptivity, which do indeed provide faster and more efficient query processing for data integration.

In particular, I have found that three techniques are highly beneficial for integration of XML data, and in fact a necessity for good performance: (1) providing pipelined execution for streaming XML data as well as relational data; (2) using adaptive operators, such as the pipelined hash join, to accommodate variations in source data transfer rates; and (3) re-optimizing queries and dynamically splitting execution pipelines in mid-stream. I motivate, define, and briefly describe each technique below.

1.2.2 The Need for Pipelining

Query execution occurs over a *query plan*, which describes a sequence or expression evaluation tree of query operators (as well as the algorithms to use in executing each operator). In general, any fragment of a query plan can be executed in one of two ways: as a *pipeline*, where control flow propagates up the operator evaluation tree one tuple at a time; or as a series of batch-oriented or *blocking* operators, where each operator consumes a table and outputs a table that gets *materialized* to disk and then fed into the next operator.

Blocking has one substantial benefit: it executes a plan fragment as a sequence of separate stages, so not all of the operators' state information must be in memory simultaneously. Furthermore, certain query operations (for instance, coalescing groups of tuples into sets with the same attribute values) must inherently be blocking, because they typically must "see" the entire table before they can compute output tuples. However, when it is feasible, pipelining has a very desirable characteristic for interactive applications like data integration: it returns initial answers much more quickly.

In a traditional database, query plans are typically divided into a series of pipelines, where each pipeline is divided from the next by a blocking operation that materializes its results to disk. This allows the query processor to best use its resources to compute an answer. For an interactive application, overall speed is typically less a concern than speed in returning initial answers — so in this context, a good heuristic is to begin execution with a single pipeline, and develop a strategy for handling memory

overflow after some initial answers have been returned.

Extending this concept slightly beyond the flow of tuples *between* operators, data should be incrementally processed as source data is streaming *into* the query processor's operators — it should not be a requirement that the data be prefetched from all sources before query execution can begin. Data transfer time is often a significant factor in processing network-based queries.

Finally, pipelining provides another benefit beyond returning fast initial answers: it enables more query operators to be running concurrently, and as a result, a query execution system has more flexibility in determining which operations should get resources at a particular time. I discuss techniques for *adaptive scheduling* in the next section.

Pipelining has been a standard technique employed for standard relational databases, but the methodology had not been fully translated to the XML realm: in particular, an XML document sent across the network would need to be read and parsed in its entirety before any query processing could begin. One of my research contributions is a model for pipelining XML data as it is being streamed across the network — resulting in significantly improved performance, as well as more possibilities for adaptive scheduling and optimization.

1.2.3 The Need for Adaptive Scheduling

An important characteristic of the physical query plans used by query execution engines is that, because they express both an order of algebraic evaluation and specify a set of deterministic algorithms to be used, they encode a *physical scheduling*. Generally, the scheduling of compute cycles to each query operator is done via an *iterator* execution model: an operator calls its child's `getNext()` method when it needs a new tuple from the child, and the child in turn calls its child, etc. A given query operator algorithm consumes and returns tuples according to a predefined order that is hard-coded into the algorithm.

The iterator model works very well for situations where I/O is relatively inexpensive and inherently sequential, and where free compute cycles are rare. In a standard database, most I/O operations come from the buffer manager (which often prefetches data) rather than directly from disk, and typically disk I/Os are not fully parallelizable (at least if they come from the same disk). Furthermore, if an operation is primar-

ily compute-bound, then context-switching from one operation to the next should only be done in an operation-aware fashion. Thus, in a standard database, execution of operations according to their own deterministic, embedded logic makes sense.

The network-based query processing domain is considerably different. First, the I/O bottleneck typically resides at the data source or at some point within the middle of the network, rather than at the query engine itself — hence, overlapping of I/O operations from different sources can be done with significant benefit. Second, network I/O delays are often relatively long, which tends to increase the number of idle cycles within a particular query. Here, it makes sense not only to overlap multiple I/O operations with one another, but also to provide flexible or adaptive scheduling: the query processor should schedule any available work when a particular part of a query plan is I/O-bound and free cycles are available.

I have proposed two enhanced algorithms for the standard query operations of join and union, both more appropriate for data integration because they support adaptive scheduling of work. Combined with a query optimization step that emphasizes pipelined execution, my techniques allow the query processor to significantly overlap I/O and computation for good performance.

1.2.4 The Need for Adaptive Re-optimization

Conventional database query processors must make static decisions at optimization-time about which query plan will be most efficient. A database query optimizer relies on a set of parameters defining CPU and I/O costs, as well as statistics describing general data characteristics, to calibrate its cost model so it can choose an inexpensive query plan. These cost values and statistics are typically recorded and periodically updated offline, and assumed to be stay relatively consistent between the update intervals.

However, cost values and statistics may not stay consistent — CPU and I/O costs vary as DBMS server load changes, and statistics are often updated rather infrequently. As a result, the optimizer may actually choose poor plans. However, most database implementers have felt that these potential pitfalls were acceptable, especially since alternative techniques would be too difficult to design and implement.

In the data integration context, the weaknesses of the standard query processing approach are exacerbated. First, data transfer speeds are unpredictable: network

congestion may delay data as it is sent from autonomous sources, and additionally the sources themselves may be overloaded at unexpected moments. Second, it is typically difficult to maintain statistics about the data within the sources: the sources were typically not designed to cooperate with a data integration system, and thus they do not export any sort of statistics-sharing interfaces. Ideally, the data integration system can gather statistics on its own, but this is not always possible: some sources do not hold relational data, and the amount and type of data they return will vary depending on how they are queried; some sources only expose small portions of their data on demand, in response to a specific query, making it difficult to estimate the overall amount of data at the source; some sources are updated frequently, rendering past statistics irrelevant very quickly.

The problems caused by the data integration environment clearly suggest that static optimization decisions, based on pre-existing knowledge, are inadequate. Instead, I believe that a query plan must be chosen *adaptively*. In particular, the query processor can choose an initial query plan, which it will continue to refine as it monitors actual query execution costs and statistics. This allows the system to deal with factors that vary over the course of execution, as well as factors that stay consistent but are unknown prior to execution.

My major contribution in this area is a novel set of techniques called *convergent query processing*, which allows an existing query plan to be changed in mid-stream, or even split into multiple stages, at virtually any point during execution. This flexibility is attained without imposing significant overhead, and I demonstrate how standard database query processing techniques can be leveraged in this context.

Together, the techniques I propose in this dissertation provide great flexibility in the scheduling of query processing work, and do so with minimal overhead and early initial answers. They are especially well-suited to our target domain of data integration, and certain techniques such as convergent query processing may also be useful in more traditional database contexts.

1.3 Outline of Dissertation

The remainder of this dissertation is structured as follows.

Chapter 2 provides background about data integration and XML. It introduces the basic components of a data integration system and cites relevant work in these areas.

It also describes the basics of the most popular XML data models and query languages.

Chapter 3 provides an overview of the considerations for adaptive query processing and the specific contributions of my thesis. It also presents an overview of the Tukwila data integration system, which utilizes the proposed techniques and has formed an evaluation platform for my work.

In Chapter 4, I describe an XML query processing architecture for pipelined query execution, including query operators that incrementally consume data from an XML stream and produce tuples that are fed into pipelined query plan. I also present the complementary operators that convert from tuples back into XML.

Chapter 5 describes the components of a query execution engine that are required for an adaptive query processing architecture. I describe the infrastructure that supports my adaptive query optimization techniques, and I present a basic set of query operators for performing joins and unions in a wide area context, with emphasis on adaptive scheduling. I present an extension to the pipelined hash join that supports overflow resolution, as well as a variant of the union for handling mirrored data sources.

Chapter 6 then describes and evaluates techniques for performing adaptive re-optimization to improve a query plan at runtime. I detail an approach to re-optimizing at pipeline boundaries, and I present *convergent query processing*, a novel means of re-optimizing an executing pipeline.

Chapter 7 describes the impact of the Tukwila system: I provide an overview of how my query processor has been used in a number of research projects and real applications at the University of Washington and in the Seattle area.

I discuss related work in Chapter 8 and conclude with some ideas for future exploration in Chapter 9.

Chapter 2

BACKGROUND: DATA INTEGRATION AND XML

The remainder of my thesis assumes a working knowledge of the basic concepts of data integration and XML. This chapter sets the context by providing the necessary background information. I begin with a discussion of the components of a data integration system and follow with a discussion of querying XML.

2.1 Data Integration System Architecture

As discussed in the introduction, the key attributes of a data integration system are its ability to present an integrated, *mediated* schema for the user to query, the ability to translate or *reformulate* this query to combine information from the various data sources according to their relationships with the mediated schema, and the ability to execute this query over the various local and remote data sources. A data integration system varies somewhat from a standard database system: it has little or no physical storage subsystem and usually no capability for expressing updates, but it needs query translation capabilities and the ability to fetch data from remote sources. The typical components of a data integration system are illustrated in Figure 2.1, and they consist of the following:

Application or Interactive User Interface

Typically, the initiator of queries and consumer of answers will be an interactive, GUI-based user interface, custom application logic, or a web-based application. In general, this data consumer requires early initial answers so it can provide early feedback to the user. Furthermore, many applications will be posing ad-hoc, exploratory queries that may be terminated before they complete.

Query Reformulator

The initial query will typically be posed in terms of a *mediated schema*, a single unified schema. Schema mediation relies on expressing the relationships between the

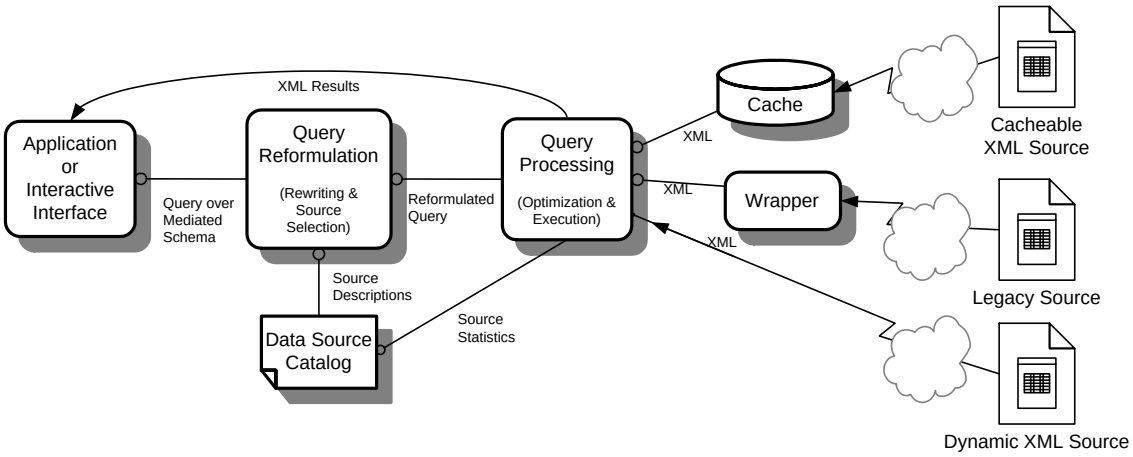


Figure 2.1: Data integration architecture diagram. The application or user interface poses a query over the mediated schema. The reformulator uses data from the data source catalog to rewrite this query to reference real data sources. The query processor finds an optimal plan for executing this query, then fetches data from the sources (in some cases through a wrapper or cache) and combines it to return the maximal set of answers.

global schema and the data sources through view definitions. Two classes of techniques have been proposed: *local-as-view*, which defines sources as views over the mediated schema, and *global-as-view*, which defines the mediated schema as a view over the sources (see [Hal01] for more details). Global-as-view mediation has the advantage that mediated-schema queries can be simply merged with the view definitions ("unfolded") to get the full query. Local-as-view requires more complex *query reformulation* [LRO96, Qia96, DG97, FLM99, PL00, HIST02], but has been shown to be more expressive — thus most modern data integration systems use it (or a hybrid of the two techniques, as in [FLM99, HIST02]). Work in query reformulation algorithms has been a major focus of database researchers at the University of Washington: UW authors have contributed the bucket [LRO96] and MiniCon [PL00] algorithms.

The existing work in query reformulation has been on conjunctive queries over purely relational data. However, the most natural data model for integrating multiple schemas is that of XML (see [FMN02a]), since this data model is general enough to accommodate hierarchical, object-oriented, document, and relational data sources. We expect, therefore, that a data integration query will typically be posed in XQuery [BCF+02], the standard query language being developed for XML. Based on recent trends in the database theory community, I expect that query reformulation will soon be extended

to handle more general XML data. Thus, for the purposes of my work, I have assumed the eventual creation of a query reformulator for XQuery (or a subset thereof). In fact, recent research on the Piazza project has begun to adapt the MiniCon algorithm to work with a conjunctive subset of XQuery [HIST02].

Data source catalog

The catalog contains several types of metadata about each data source. The first of these is a semantic description of the *contents* of the data sources. A number of projects, including [DDH01, MBR01, HMH01], have focused on developing techniques for automatic or semi-automatic creation of mappings between data sources and the mediated schema of the data integration system.

Data source sizes and other data distribution could also be recorded alongside the information about the mappings, but this will only feasible if the data source changes infrequently and can be "probed" in a comprehensive way; I do not expect this to be the common case. In a few cases, a system may have even further information describing the overlap between data values at different sources. A model for reasoning about overlap is described in [FKL97], and describes the probability that a data value d appears in source S_1 if d is known to appear in source S_2. This can be used alongside cost information to prioritize data from certain data sources.

Attempts have also been made to profile data sources and develop cost estimates for their performance over time [BRZZ99, ZRZB01], and to provide extensive models of data source costs and capabilities [LRO96, CGM99, LYV+98, ROH99].

Query processor

The query processor takes the output of the query reformulator — a query over the actual data sources (perhaps including the specification of certain alternate sources) — and attempts to *optimize* and *execute* it. Query optimization attempts to find the most desirable operator evaluation tree for executing the query (according to a particular objective, e.g., amount of work, time to completion), and the query execution engine executes the particular query plan output by the optimizer. The query processor may optionally record statistical profiling information in the data source catalog. In this dissertation, I describe how adaptive techniques and pipelining can be used to produce an efficient query processor for network-bound XML data.

Wrappers

Initial work on data integration predates the recent efforts to standardize data exchange. Thus, every data source might have its own format for presenting data (e.g., ODBC from relational databases, HTML from web servers, binary data from an object-oriented database). As a result, one of the major issues was the "wrapper creation problem." A *wrapper* is a small software module that accepts data requests from the data integration (or *mediator*) system and fetches the requested data from the data source; then converts from a data source's underlying representation to a data format usable by the mediator. Significant research was done on rapid wrapper creation, including tools for automatically inducing wrappers for web-based sources based on a few training examples [KDW97, BFG01], as well as toolkits for easily programming web-based wrappers [SA99, LPH00]. Today, the need for wrappers has diminished somewhat, as XML has been rapidly adopted as a data exchange format for virtually any data source. (However, the problem has not completely disappeared, as legacy sources will still require wrappers to convert their data into XML.)

Data Sources

The data sources in an integration environment are likely to be a conglomeration of pre-existing, heterogeneous, autonomous sources. Odds are high that none of these systems was designed to support integration, so facilities for exchanging metadata and statistics, estimating costs, and offloading work are unlikely to be present. Sometimes the sources will not even be truly queryable, as with XML documents or spreadsheets. Additionally, in many cases the sources are controlled by external entities who only want to allow limited access to their data and computational resources. At times, these access restrictions may include *binding patterns*, where the source must be queried for a particular attribute value before it returns the corresponding tuples[1].

The data sources may be operational systems located remotely and running with variable load — hence, the data integration system will often receive data from a given source at an indeterminate and varying rate. The data integration system must be able to accommodate these variations, and should be able to "fall back" to mirrored sources or caches where appropriate.

[1] An instance of this is a web form from Amazon.com, where the user must input an author or title before the system will return any books

Another important issue related to external data sources is that of semantic matching of entities. Multiple data sources may reference the same entity (for instance, a user may be a member of multiple web services). Frequently, these different data sources will have a different representation of identity, and an important challenge lies in determining whether two entities are the same or different. This problem is far from being solved, but some initial work has been done on using information retrieval techniques for finding similarity [Coh98] and on probabilistic metrics [PR01]. In many cases, however, semantic matching is straightforward enough to be done in exact-match fashion, or users may supply matching information. For these situations, we merely need a common data format.

XML has emerged as the de facto standard for exchanging information, and virtually any demand-driven data integration scenario is likely to be based on XML data requested via HTTP. Most applications already support some form of XML export capabilities, but legacy applications may require *wrappers* to export their data in XML form. A key characteristic of XML is its ability to encode structured or semi-structured information of virtually any form: relational and object-oriented data, text documents, even spreadsheets and graphics images can be encoded in XML.

2.2 The XML Format and Data Model

XML originates in the document community, and in fact it is a simplified subset of the SGML language used for marking up documents. At a high level, XML is a very simple hierarchical data format, in which pairs of open- and close-tags and their content form *elements*. Within an element, there may be nested elements; furthermore, each element may have zero or more *attributes*. From a database perspective, the difference between an attribute and an element is very minor — attributes may only contain scalar values, an attribute may occur only once per element, and attributes are unordered with respect to one another. Elements, on the other hand, may be repeated multiple times, may include element or scalar data, and are ordered with respect to one another. An example XML document representing book data is shown in Figure 2.2.

An XML document is fully parsable and “well-formed” if every open-tag has a matching close-tag and the details of the XML syntax are followed. However, typically the XML document structure is constrained by a schema. There are two standards for specifying schemas in XML: DTDs and XML Schemas.

```
<db>
 <book publisher="mkp">
   <title>Readings in Database Systems</title>
   <editor>Stonebraker</editor>
   <editor>Hellerstein</editor>
   <isbn>123-456-X</isbn>
 </book>
 <book publisher="mkp">
   <title>Transaction Processing</title>
    <author>Bernstein</author>
    <author>Newcomer</author>
    <isbn>235-711-Y</isbn>
  </book>
  <company ID="mkp">
    <name>Morgan Kaufmann</title>
    <city>San Mateo</city>
    <state>CA</state>
  </company>
</db>
```

Figure 2.2: Sample XML document representing book and publisher data.

The DTD, or Document Type Definition, establishes constraints on XML tags, but does so in a very loose way. A DTD is essentially an EBNF grammar that restricts the set of legal XML element and attribute hierarchies. The DTD also explicitly identifies two special types of attributes: `ID`s and `IDREF`s. An `ID`-typed attribute has a unique value for each document element in which it is contained — this forms, in essence, a key within the scope of a document. `IDREF` and `IDREFS`-typed attributes, correspondingly, form references to particular `ID`s within a document. An `IDREF` contains a single reference to an `ID`; an `IDREFS` attribute contains a list of space-delimited `ID`s. The XML specification mandates that no `IDREF`s within a document may form dangling references.

XML Schema is a more recent standard intended to supplant the DTD. The primary benefits of Schema are support for element types and subclassing, support for richer type information about values (e.g., integers, dates, etc.), and support for keys and foreign keys. Unfortunately, Schema is a very complex standard with many related

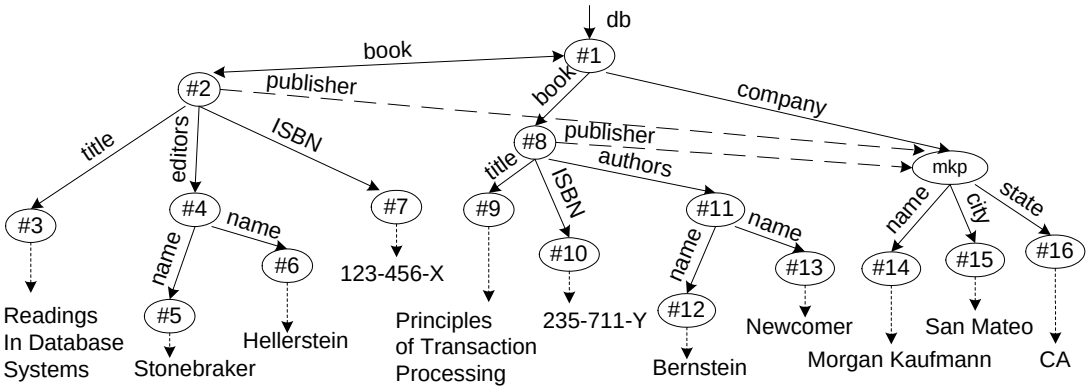


Figure 2.3: XML-QL graph representation for Figure 2.2. Dashed edges represent `IDREF`s; dotted edges represent `PCDATA`.

but slightly different idioms (e.g., different types of inheritance, support for structural subtyping and name subtyping), and there is no clean underlying data model that comes with the standard.

There have been numerous proposals for an XML data model, but two are of particular interest to us. The first proposal, a mapping from XML into the traditional semistructured data model, is interesting because it provides graph structure to an XML document; however, it does not capture all elements of the XML specification. The second proposal, the W3C XML Query data model, is tree-based but incorporates all of the different details and idiosyncrasies of the XML specification, including processing instructions, comments, and so forth. The XML Query data model is an attempt to define a clean formalism that incorporates many of the core features of XML Schema.

2.2.1 The XML-QL Data Model

Today, with the advent of proposed standards, the XML-QL query language [DFF+99] has lost favor, so one could argue that its data model is irrelevant. However, the data model is a mapping from XML to a true semi-structured data model, and thus it has some interesting features that are missing from recent standards — features that may be important for certain applications.

In the XML-QL data model, each node receives a unique label (either the node's

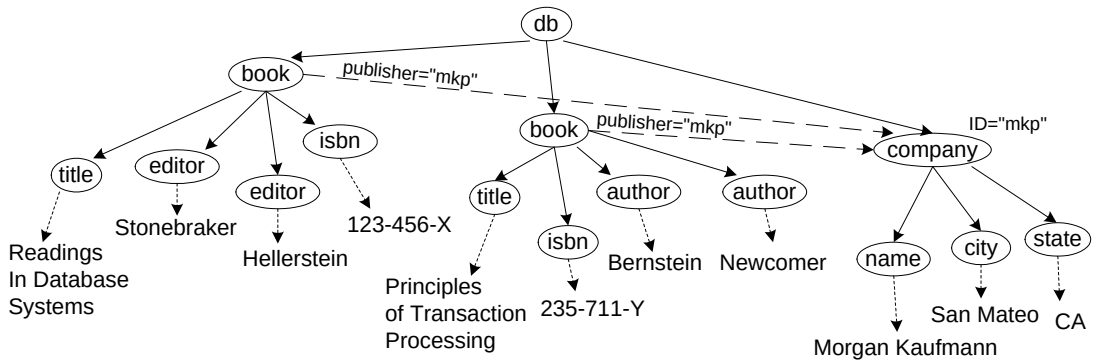


Figure 2.4: Simplified XQuery data model-based representation for Figure 2.2. Dashed edges illustrate relationships defined by IDREFs; dotted edges point to text nodes.

ID attribute or a system-generated ID). A given element node may be annotated with attribute-value pairs; it has labeled edges directed to its sub-elements and any other elements that it references via IDREF attributes. Figure 2.3 shows the graph representation for the sample XML data of Figure 2.2. Note that IDREFs are shown in the graph as dashed lines and are represented as edges labeled with the IDREF attribute name; these edges are directed to the referenced element's node. In order to allow for intermixing of "parsed character" (string) data and nested elements within each element, we create a PCDATA edge to each string embedded in the XML document. These edges are represented in Figure 2.3 as dotted arrows pointing to leaf nodes.

XML-QL's data model only has limited support for order: siblings have indices that record their *relative* order, but not absolute position, so comparing the order of arbitrary nodes is difficult and requires recursive comparison of the ancestor nodes' relative node orderings (starting from the root). Furthermore, XML-QL's data model does not distinguish between subelement edges and IDREF edges — therefore, a given graph could have more than one correct XML representation.

2.2.2 The XML Query Data Model

The World Wide Web Consortium's XML Query (XQuery) data model [FMN02b] is based on node-labeled trees, where IDREFs exist but must be dereferenced explicitly. The full XQuery model derives from XML Schema [XSc99], or at least a cleaner ab-

straction of it called MSL [BFRW01]. This data model has many complexities: elements have types that may be part of a type hierarchy; types may share portions of their substructure ("element groups"); elements may be keys or foreign keys; scalar values may have restricted domains according to specific data types; and the model also supports XML concepts such as entities, processing instructions, and even comments. Furthermore, any node in the XQuery data model has implicit links to its children, its siblings, and its parent, so traversals can be made up, down, or sideways in the tree.

For the purposes of this thesis, most of XQuery's type-related features are irrelevant to processing standard, database-style queries, so I will discuss only a limited subset of the XQuery data model, which focuses on elements, attributes, and character data and which includes order information. Figure 2.4 shows a simplified representation of our example XML document using this data model, where a left-to-right preorder traversal of the tree describes the order of the elements.

2.3 Querying XML Data

During the past few years, numerous query languages for XML have been proposed, but recently the World Wide Web Consortium has attempted to standardize these with its XQuery language specification [BCF+02]. The XQuery language is designed to extract and combine subtrees from one or more XML documents. The basic XQuery expression consists of a `For-Let-Where-Return` clause structure (commonly known as a "flower" expression): the `For` clause provides a series of XPath expressions for selecting input nodes, the `Let` clause similarly defines collection-valued expressions, the `Where` clause defines selection and join predicates, and the `Return` clause creates the output XML structure. XQuery expressions can be nested within a Return clause to create hierarchical output, and, like OQL, the language is designed to have modular and composable expressions. Furthermore, XQuery supports several features beyond SQL and OQL, such as arbitrary recursive functions.

XQuery subsumes most of the operations of SQL, and general processing of XQuery queries is extremely similar, with the following key differences.

```
<result> {
  FOR $b IN document("books.xml")/db/book,
      $t IN $b/title/data(),
      $n IN $b/(editor|author)/data()
  RETURN <item>
           <person>{ $n }</person>
           <pub>{ $t }</pub>
         </item>
} </result>
```

Figure 2.5: XQuery query that finds the names of people who have published and their publications. The `For` clause specifies XPath expressions describing traversals over the XML tree, and binds the subtrees to variables (prefixed with dollar signs).

Input Pattern-Matching (Iteration)

XQuery execution can be considered to begin with a *variable binding* stage: the `For` and `Let` XPath expressions are evaluated as traversals through the data model tree, beginning at the root. The tree matching the end of an XPath is *bound* to the `For` or `Let` clause's variable. If an XPath has multiple matches, a `For` clause will iterate and bind its variable to each, executing the query's `Where` and `Return` clause for each assignment. The `Let` clause will return the collection of all matches as its variable binding. A query typically has numerous `For` and `Let` assignments, and each legal combination of these assignments is considered an *iteration* over which the query expression should be evaluated.

An example XQuery appears in Figure 2.5. We can see that the variable `$b` is assigned to each `book` subelement under the `db` element in document `books.xml`; `$t` is assigned the title within a given `$b` book, and so forth. In my thesis work, I assume an extended version of XPath that allows for disjunction along any edge (e.g., `$n` can be either an `editor` or `author`), as well as a regular-expression-like Kleene star operator (not shown).

In the example, multiple match combinations are possible, so the variable binding process is performed in the following way. First, the `$b` variable is bound to the first occurring `book`. Then the `$t` and `$n` variables are bound in order to all matching `title` and `editor` or `author` subelements, respectively. Every possible pairing of `$t` and `$n` values for a given `$b` binding is evaluated in a separate iteration; then the

```
<result>
  <item>
    <person>Stonebraker</person>
    <pub>Readings in Database Systems</pub>
  </item>
  <item>
    <person>Hellerstein</person>
    <pub>Readings in Database Systems</pub>
  </item>
  <item>
    <person>Bernstein</person>
    <pub>Transaction Processing</pub>
  </item>
  <item>
    <person>Newcomer</person>
    <pub>Transaction Processing</pub>
  </item>
</result>
```

Figure 2.6: The result of applying the query from Figure 2.5 to the XML data in Figure 2.2. The output is a series of person-publisher combinations, nested within a single result root element.

process is repeated for the next value of `$b`. We observe that this process is virtually identical to a relational query in which we join books with their titles and authors — we will have a tuple for each possible $\langle$`title`, `editor`|`author`$\rangle$ combination per book. The most significant difference is in the terminology; for XQuery, we have an "iteration" with a binding for each variable, and in a relational system we have a "tuple" with a value in each attribute.

XML Result Construction

The `Return` clause specifies a tree-structured XML constructor that is output on each iteration, with the variables replaced by their bound values. Note that variables in XQuery are often bound to XML subtrees (identified by their root nodes) rather than to scalar values. The result of the example query appears in Figure 2.6.

One of the primary idioms used in XQuery is a correlated flower expression nested within the `Return` clause. The nested subexpression returns a series of XML elements

that satisfy the correlation for each iteration of the outer query — thus, a collection of elements gets nested within each outer element. This produces a "one-to-many" relationship between parent and child elements in the XML output. Note that this particular type of query has no precise equivalent in relational databases, but it has many similarities to a correlated subquery in SQL.

Traversing Graph Structure

Finally, as we pointed out earlier in our discussion of the XML data model, some XML data makes use of `IDREF` attributes to represent links between elements (the dashed lines in Figure 2.3). `IDREF`s allow XML to encode graph-structured as well as tree-structured data. However, support for such capabilities in XQuery is limited: while XQuery has "descendant" and "wildcard" specifiers for selecting subelements, its `->` dereference operator only supports following of single, specified edges. These restrictions were presumably designed to facilitate simpler query processing at the expense of query expressiveness. For applications that require true graph-based querying, extensions to XQuery may be required, and these extensions will give us a data model that looks more like that for XML-QL. We discuss these features in more detail in Chapter 4.

2.3.1 Querying XML for Data Integration

To this point, we have discussed the general principles of querying an XML document or database, including data models, query languages, and unique aspects. Our discussion has assumed that the XML data is available through a local database or a materialized data source, e.g., an XML document on a local web server.

However, the data integration context provides an interesting variation on this pattern: sometimes XML data may be accessible only via a specific *dynamic* query, as the result of a "virtual XML view" over an underlying data source (e.g., an XML publishing system for a relational database, such as [FTS99, CFI+00], or a web forms interface, such as those at Amazon.com). This query may need to request content from the data source according to a specific value (e.g., books from Amazon with a particular author). In other words, a data integration system may need to read a set of values from one or more data sources, then use these to generate a dynamic query to a "dependent" data source, and then combine the results to create a query answer. XQuery currently

supports a limited form of dynamic queries, based on CGI requests over HTTP, and the Working Group is considering the addition of special functions for fully supporting these features under other protocols such as SOAP.

Now that the reader is familiar with the basic aspects of data integration and XML query processing, we focus on the problem of XML query processing for data integration.

Chapter 3

QUERY PROCESSING FOR DATA INTEGRATION

The last chapter presented the basic architecture of a data integration system and discussed several motivations for adopting the XML data format and data model standards. The construction of data integration systems has been a topic of interest to both of my advisors, who developed the Information Manifold [LRO96] and Razor [FW97]. These systems focused on the problems of mapping, reformulation, and query planning for relational data. My thesis project, Tukwila, builds upon the techniques developed for these systems and the rest of the data integration community, with a new emphasis: previous work established the set of capabilities that a basic data integration system should provide and developed techniques for effective semantic medation. However, they typically focused on issues other than providing the level of scalability and performance necessary to be used in real applications. In the data integration domain, queries are posed over multiple autonomous information sources distributed across a wide-area network. In many cases, the system interacts directly with the user in an ad-hoc query environment, or it supplies content for interactive applications such as web pages. The focus of my thesis has been on addressing the needs of this type of domain: building a data integration query processing component that provides good performance for interactive applications, processes XML as well as relational data, and handles the unique requirements of autonomous, Internet-based data sources.

As I discuss in previous chapters, modern query processors are very effective at producing well-optimized query plans for conventional databases, and they do this by leveraging I/O cost information as well as histograms and other statistics to compare alternative query plans. However, data management systems for the Internet have demonstrated a pressing need for new techniques. Since data sources in this domain may be distributed, autonomous, and heterogeneous, the query optimizer will often not have histograms or any other quality statistics. Moreover, since the data is only accessible via a wide area network, the cost of I/O operations is high, unpredictable, and variable.

I propose a combination of three techniques, which can be broadly classified as

adaptive query processing, to mitigate these factors:

1. Data should be processed as it is **streaming across the network** (as is done in relational databases via pipelining)[1].

2. The system should employ **adaptive scheduling** of work, rather than using a deterministic and fixed strategy, to accommodate unpredictable I/O latencies and variable data flow rates.

3. The system should employ **adaptive plan re-optimization**, where the query processor adapts its execution strategy in response to new knowledge about data sizes and transfer rates as the query is being executed.

In the remainder of this chapter, I provide a more detailed discussion of how these three aspects interrelate. I begin with a discussion of the adaptive query processing space and where my work falls within this space, then in Section 3.2 I provide a sketch of how my thesis contributions in adaptive query processing solve problems for data integration, and I conclude the chapter with an overview of my contributions in the Tukwila system, which implements the adaptive techniques of my thesis.

3.1 Position in the Space of Adaptive Query Processing

Adaptive query processing encompasses a variety of techniques, often with different domains in mind. While I defer a full discussion of related work to Chapter 8, it is important to understand the goals of my thesis work along several important dimensions:

Data model To this point, most adaptive query processing techniques have focused on a relational (or object-relational) data model. While there are clearly important research areas within this domain, other data models may require extensions to these techniques. In particular, XML, as a universal format for describing data, allows for hierarchical and graph-structured data. I believe that XML is of particular interest

[1]Note that our definition of "streaming" does *not* necessarily mean "infinite streams," as with some other recent works, e.g. [DGGR02a].

within a data integration domain, for two reasons. First, XML is likely to be the predominant data format for integration, and it poses unique challenges that must be addressed. Second, the basic concepts from relational databases often are insufficient for XML processing. For instance, there has been no means of "pipelining" XML query processing as the data streams across the network[2]. As a second example, the notions of selectivity and cardinality have a slightly different meaning in the XML realm, since the number of "tuples" to process depends on the XPath expressions in the query.

Remote vs. local data Traditional database systems have focused on local data. Recent work in distributed query processing has focused on techniques for increasing the performance of network-bound query applications, including [UFA98, UF99, IFF+99, AH00, HH99]. The focus of my thesis has been on remote data, although many of the techniques can be extended to handle local data as well.

Large vs. small data volumes Data warehouses and many traditional databases concentrate on handling large amounts of data (often in the tens or hundreds of GB). In contrast, most previous work on data integration systems and distributed databases has concentrated on small data sources that can easily fit in main memory. My focus is on "mid-sized" data sources: sources that provide tens or hundreds of MB of data, which is enough to require moderate amounts of computation but which is still small enough to be sent across typical local-area and wide-area networks. I believe that typically, most data sources will be able to filter out portions of their data according to selection predicates, but that they will be unlikely to perform joins between their data and that of other sources — hence, the amount of data sent is likely to be nontrivial.

First vs. last tuple For domains in which the data is used by an application, the query processor should optimize for overall query running time — the traditional focus of database query optimizers. Most of today's database systems do all of their optimization in a single step; the expectation (which does not generally hold for large query plans [AZ96] or for queries over data where few statistics are available) is that the optimizer has sufficient knowledge to build an efficient query plan. The INGRES optimizer [SWKH76] and techniques for mid-query re-optimization [KD98] often yield

[2]SAX parser events can be viewed as a form of pipelining, but they do not do query processing.

better running times, because they re-optimize later portions of the query plan as more knowledge is gained from executing the earlier stages. I have studied re-optimization both as a way to speed up total running time (last tuple) and as a way of improving time to first tuple. My work has also emphasized pipelined execution as a heuristic that typically speeds time to first tuple.

Re-use of results and statistics Data may often be prefetched and cached by the query processor, but the system may also have to provide data freshness guarantees. Caching and prefetching are well-studied areas in the database community, and the work on rewriting queries over views [Hal01] can be used to express new queries over cached data, rather than going to the original data sources. My thesis work does not directly investigate caching, but our work on Piazza (see Chapter 7) investigates the use of caching mechanisms in network-based query processing. Related to caching of data, it is also possible to "learn" statistical models of the content at data sources, as in [CR94], or patterns of web-source access costs, as in [BRZZ99, ZRZB01]. My thesis work can take advantage of cached results or statistics when they are available, but the emphasis is on working well even when they are not.

3.2 Adaptive Query Processing for Data Integration

As mentioned above, I have proposed a three-pronged strategy to providing a comprehensive solution to network-based query processing. I describe these dimensions in the following subsections, and I describe how my contributions differ from previous techniques within each dimension. A more comprehensive section on related work appears in Chapter 8.

3.2.1 Pipelining XML Data

XML databases [Aea01, XLN, Tam, GMW99, FK99b, SGT+99, AKJK+02, KM00, BBM+01] typically map XML data into an underlying local store — relational, object-oriented, or semistructured — and do their processing within this store. For a network-bound domain where data may need to be re-read frequently from autonomous sources to guarantee "freshness," this materialization step would provide very poor performance. Furthermore, deconstructing and reconstructing XML introduces a series of complex operations that limit opportunities for adaptivity. Thus it is imperative that an XML

data integration system not materialize the data first, but instead support processing of the data directly.

Contrasting with databases, the tools from the XML document community (in particular, XSLT processors) have been designed to work on XML data without materializing it to disk — they typically build a Document Object Model (DOM) parse tree of the single XML document to be processed, and then they match XSLT templates against this document and return a new DOM tree. Note that they are limited by main memory and they typically only work over a single document. More significantly, they do not provide incremental evaluation of queries: first the document is read and parsed, then it is walked, and finally output is produced. Since network data transfer and XML parsing are both slow, it is desirable to incrementally pipeline the data as it streams in, much as a relational engine can pipeline tuples as they stream in. This capability would improve pure performance, and it would also enhance the number of possibilities for adaptive scheduling of work and for re-optimization, since more concurrent operations would be occuring.

In order to provide incremental, pipelined evaluation of XML queries, a system must be able to support efficient evaluation of *regular path expressions* over the incoming data, and incremental output of the values they select. Regular path expressions are a mechanism for describing traversals of the data graph using edge labels and optional regular-expression symbols such as the Kleene-star (for repetition) and the choice operator (for alternate subpaths). Regular path expressions bear many similarities to conventional object-oriented path expressions, and can be computed similarly; however, the regular expression operators may require expensive operations such as joins with transitive closure.

The first contribution of my thesis is a query processing architecture that focuses on pipelining XML data in a way that is similar to relational query execution. A key contribution is the *x-scan* operator, which evaluates XPath expressions across incoming XML data, and which binds query variables to nodes and subgraphs within the XML document. X-scan includes support for both tree- and graph-structured documents, and it is highly efficient for typical XML queries. It combines with the other adaptive query features proposed in my thesis to rapidly feed data into an efficient query plan, and thus return initial data to the user as quickly as possible.

3.2.2 Adaptive Operator Scheduling

A physical query plan generally contains both an algebraic order of evaluation (i.e., an evaluation tree specifying operator precedence) for the query data, as well as a set of physical algorithms that establish a physical order of evaluation (e.g., nested loops join defines that the "inner" table be iterated over before the "outer" table). Adaptive operator scheduling techniques preserve the current algebraic query plan, but modify which operators are physically scheduled first or which tuples are processed earliest. An example of an adaptive operator is the pipelined hash join, or double pipelined join [RS86, WA91], which runs the child subtrees of the join and their parent in separate concurrent threads (which are scheduled by the operating system or the database itself). Another example is one aspect of query scrambling [UFA98], which reschedules a portion of a query plan when an initial delay is encountered. Finally, dynamic pipeline scheduling [UF01] allows a query plan to prioritize the processing of certain tuples to produce early output.

Adaptive scheduling produces no benefits if the query is completely I/O-bound and all data has been fully processed. On the other hand, it is extremely beneficial for queries that are what might be termed "scheduling-bound" — where the default scheduling prevents the plan from doing work it might otherwise be able to perform — and it is also helpful if the query is CPU-bound and the system wishes to prioritize certain results.

A key characteristic of operator scheduling is that it affects query performance over the currently available tuples, but does not affect the overall amount of work the CPU must perform to complete the query. Hence, a scheduling change is in some sense a "low-risk" decision, as it does not incur a future penalty.

In data integration, adaptive scheduling can be extremely beneficial, because a complex pipelined query plan typically encounters situations in which a particular query operator is I/O-bound, but some other operator could be doing work. Here, the query processor can block the I/O-bound operator and switch to the one that can make progress, and it can do this without needing to consider the possibility of incurring a penalty.

My contributions in the area of adaptive operator scheduling are a variant of the pipelined hash join that supports dynamic destaging when it runs out of memory, as well as a query operator called the *dynamic collector*, which adaptively reads tuples

from mirrored or overlapping sources according to source priority and availability.

3.2.3 Adaptive Query Plan Re-Optimization

Changing the algebraic plan order (or the physical algorithms being used within the query plan) is much more significant than changing plan scheduling, because this has a long-term effect on overall plan performance. This is because future tuples must be processed not only within the new plan, which has a set of intermediate results with which it must be combined, but they must also be combined with all data in previous plans. Depending on the sizes of the intermediate results in all of the plans, some computation may be extremely inexpensive, and other computation may be extremely costly. The decision to change a plan should generally be made when new selectivity or cardinality values are discovered and the original cost estimates for the query are known to be based on invalid assumptions and likely to lead to high cost.

Instances of plan modification include the *operator synthesis* stage of query scrambling [UFA98], dynamic re-optimization [KD98], the eddy [AH00], and convergent query processing [IHW].

Changing a query plan can be a costly operation, and it tends to waste work, so most prior techniques enabled adaptivity *between* pipelines — when a fragment of a query plan completed, techniques such as choose nodes or dynamic re-optimization could change the remainder of the query plan. This prevented the possibility of redoing computation, and meant that the emphasis of these efforts was on choosing the optimal points at which to break pipelines, as well as on deciding when to re-optimize. The initial Tukwila system built similar capabilities into the query optimizer and execution engine. However, my co-authors and I found several significant problems:

- Re-optimization is required because the query optimizer generally has too few statistics to select a good query plan. Unfortunately, in many cases, it also has too few statistics to select a good point at which to break a pipeline.

- One of the desiderata for data integration is early initial answers. Breaking a query plan into multiple pipelines increases the amount of time before a first answer is delivered.

- Even with re-optimization, query execution might take a very long time if the initial pipeline was poorly chosen.

Clearly, a better solution would be one that could re-optimize the current pipeline, rather than merely a future one. Not only would this eliminate the need for an *a priori* decision about where to break the pipeline, but it would actually allow the entire query to be executed in a single pipeline (producing faster initial answers), and it would also allow the system to switch from a bad initial plan to a better one. It is important, however, that this set of adaptive capabilities be provided in a way that does not detract from average-case performance. More specifically, we would also like the following features:

- The ability to take advantage of any existing statistics or samples from tables when they are available.
- Low overhead beyond that of standard query processing techniques.
- Efficient re-use of available data, and minimal re-execution of work.
- The ability to support not only standard select, project, and join operators, but also aggregation optimizations, and potentially to use more sophisticated query rewrite techniques (e.g., [MP94]).

The eddy algorithm of [AH00] supports frequent dynamic re-optimization of a query plan, but focuses on re-optimizing as each new tuple that enters the system. As a result, it relies on a greedy heuristic that does not support aggregation or complex queries, and one which cannot take advantage of existing statistics. Furthermore, eddies, at least in their current implementation, are less efficient than traditional query processing techniques because of the inherent overhead in re-optimization and in routing tuples through operators.

The third contribution of my thesis is a technique called *convergent query processing*, which allows for adaptive re-optimization that meets the basic desiderata listed above. It can take advantage of existing cost information and data statistics, but can also infer statistics and costs for situations where it does not have them. It provides great flexibility in the kinds of re-optimizations that are possible, but minimizes redundant work. Finally, the decision making can be done with low overhead, as the system can re-optimize in the background and at periodic intervals.

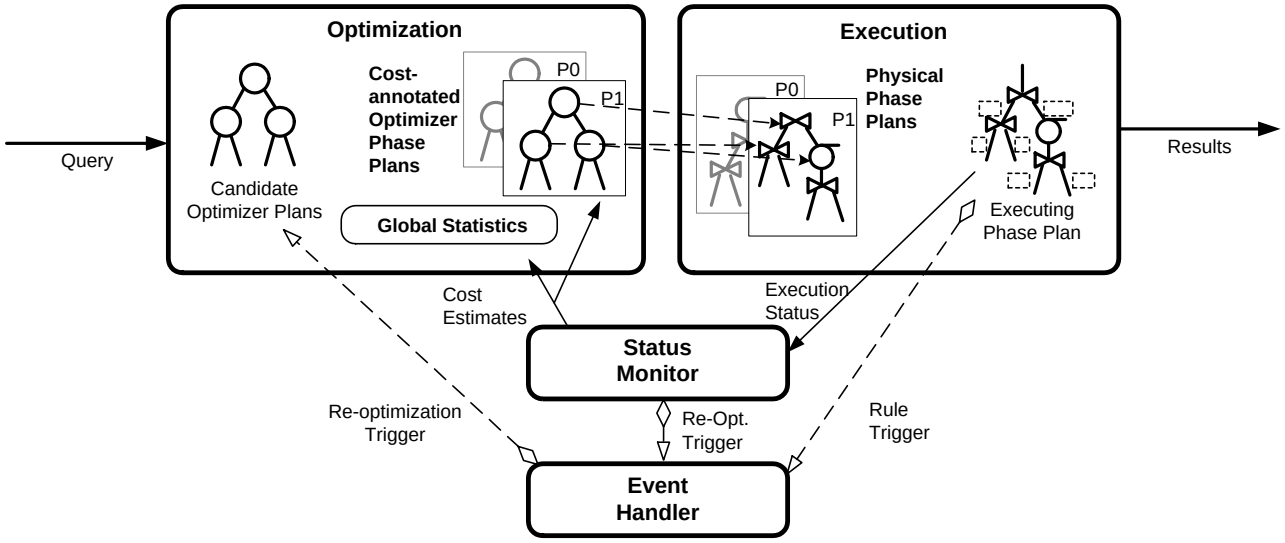


Figure 3.1: Architecture diagram for Tukwila query processor: Tukwila contains tightly interlinked optimization and execution components. The optimizer can be re-invoked by triggering certain rules, or when a status monitor determines that a plan should be re-optimized in mid-stream.

3.3 The Tukwila Data Integration System: An Adaptive Query Processor

In order to validate the techniques proposed in my thesis, both experimentally and for real-world applications, I have built an adaptive query processor as the core of the Tukwila data integration system. The current Tukwila implementation is a complete data integration environment except that it is not typically coupled with a query reformulator module: it accepts reformulated XQueries over XML data sources, it requests data from these sources via HTTP or NFS requests, and it combines the source XML to form new results. In other work, my co-authors and I have also combined the Tukwila system with the MiniCon reformulator algorithm of [PL00] to form a complete system with a mediated schema, which works for conjunctive relational queries. (Tukwila has also been the query processor for a number of other projects at the University of Washington, as I discuss in Chapter 7.)

3.3.1 Novel Architectural Features

The focus of my thesis work has been on the query processing component of Tukwila, illustrated in Figure 3.1. So far as I am aware, the Tukwila query processor is the first system to be engineered with the express goal of building an adaptive query processor. Key architectural aspects of Tukwila include the following:

- An architecture that emphasizes pipelined query execution, in order to produce early first results. Our system typically begins with a single pipelined query plan, and it can read and process XML as it streams across the network. Since a single pipeline may run out of resources, my architecture include capabilities for handling memory overflow and for adaptively splitting a single pipeline into multiple stages once a significant portion of computation has completed.

- Support for multithreaded execution of queries, in order to facilitate adaptive scheduling. Unlike most traditional, iterator-based query processors, my system is not forced to rely on deterministic scheduling, and instead it can selectively attempt to overlap I/O and computation to more effectively utilize available data and resources.

- Novel algorithms for network-based and pipelined query processing, including: (1) a variant of the pipelined hash join [RS86] that supports overflow resolution and the *dynamic collector*, (2) a specialized version of the relational union that, upon data source failure, can switch from the original data source to its mirrors or alternates, (3) an XPath pattern-matching operator, *x-scan*, which produces variable bindings in pipelined fashion over streaming XML data, and (4) an operator, *indirect-scan* that generalizes the relational dependent join operator for a web- and XML-oriented environment.

- Mechanisms for efficiently switching query plans in mid-execution of a pipeline, as well as at the interface between two separate pipelines. One of the key challenges in building the Tukwila execution engine was providing mechanisms for efficiently passing data along pipelines (using techniques such as sharing space in the buffer pool between operators), but also enabling a query plan to be modified in mid-execution.

- Close coupling between the query optimization and execution components. The optimizer and execution engine expose their data structures to a *status monitor* that periodically reads query execution cost and selectivity information and uses this to update the query optimizer's cost model. If the actual values diverge significantly from the estimates, a query re-optimization operation may be triggered in the background.

- An event handler for responding to important runtime events. The event handler provides a way of responding to data source failures, network delays, and memory overflow. The event handler is capable of initiating a complete re-optimization of the query. Note that the event handler operates at a lower granularity than a handler for traditional database triggers or active rules: it responds to events at the physical level rather than the logical level, and it can also modify the behavior of query operators or initiate a change in query plans.

- An extended version of the System-R cost-based optimizer algorithm ([SAC+79]) that can be used not only to create initial query plans, but also to periodically re-optimize an executing query plan when better statistical information is available.

- Support within both the cost-based query optimizer and the query execution system for sharing of data structures *across* different query plans within an execution sequence. These capabilities are used to avoid repeating computation of results in a query execution sequence that consists of multiple plans.

Adaptive behavior during query execution is key in situations where I/O costs are variable and unpredictable. When data sizes are also unpredictable, it is unlikely that the query optimizer will produce a good query plan, so it is important to be able to modify the query plan being executed. As a result, Tukwila supports incremental re-optimization of queries during particular plan execution points.

In the following chapters, I provide a detailed discussion of each of the novel aspects of my work, and supplement this discussion with an experimental evaluation of the techniques within the Tukwila system.

Chapter 4

AN ARCHITECTURE FOR PIPELINING XML STREAMS

For many years, a wide variety of domains, ranging from scientific research to electronic commerce to corporate information systems, have had a great need to be able to integrate data from many disparate data sources at different locations, controlled by different groups. Until recently, one of the biggest obstacles was the heterogeneity of the sources' data models, query capabilities, and data formats. Even for the most basic data sources, custom *wrappers* would need to be developed for each data source and each data integration *mediator*, simply to translate mediator requests into data source queries, and to translate source data into a format that the mediator can handle.

The emergence of XML as a common data format, as well as the support for simple web-based query capabilities provided by related XML standards, has suddenly made data integration practical in many more cases. XML itself does not solve all of the problems of heterogeneity — for instance, sources may still use different tags or terminologies — but often, data providers come to agreement on standard schemas, and in other cases, we can use established database techniques for defining and resolving mappings between schemas. As a result, XML has become the standard format for data dissemination, exchange, and integration. Nearly every data management-related application now supports the import and export of XML, and standard XML Schemas and DTDs are being developed within and among enterprises to facilitate data sharing (instances of these are published at the BizTalk and OASIS web sites[1]). Language- and system-independent protocols such as the various web services standards, Microsoft's .NET [NET01] initiative, and Sun's JXTA [JXT01] peer-to-peer protocols use XML to represent transactions and data.

Processing and integrating XML data poses a number of challenges. In many data integration applications, XML is merely a "wire format," the result of some view over a live, dynamic, non-XML source. In fact, the source may only expose subsets of its data as XML, via a query interface with access restrictions, e.g., the source may only

[1]See www.biztalk.org and www.xml.org.

return data matching a selection value, as in a typical web form. Since the data is controlled and updated externally and only available in part, this makes it difficult or impossible to cache the data. Moreover, the data sources may be located across a wide-area network or the Internet itself, so queries must be executed in a way that is resilient to network delays. Finally, the sources may be relatively large, in the 10s to 100s of MB or more, and that may require an appreciable amount of time to transfer across the network and parse. We refer to these types of data sources as "network-bound": they are only available across a network, and the data can only be obtained through reading and parsing a (typically finite) stream of XML data.

To this point, integration of network-bound, "live" XML data has not been well-studied. Most XML work in the database community has focused on constructing XML repositories and warehouses [Aea01, XLN, Tam, GMW99, FK99b, AKJK+02, KM00, BBM+01, SGT+99], exporting XML from relational databases [FTS99, FMS01a, CFI+00], adding information retrieval-style indexing techniques to databases [NDM+01, FMK00], and on supporting query subscriptions or continuous queries [Aea01, CDTW00, AF00] that provide new results as documents change or are added.

Clearly, both warehousing and indexing are useful for storing, archiving, and retrieving file-based XML data or documents, but for many integration applications, support for queries over dynamic, external data sources is essential. This requires a query processor that can request data from each of the sources, combine this data, and perhaps make additional requests of the data sources as a result. To the best of my knowledge, no existing system provides this combination of capabilities. The Web community has developed a class of query tools that are restricted to single-documents and not scalable to large documents. The database community's web-based XML query engines, such as Niagara and Xyleme, come closer to meeting the needs of data integration, but they are still oriented towards smaller documents (which may be indexable or warehoused), and they give little consideration to processing data from slow sources or XML that is larger than memory.

As discussed in previous chapters, query processing for data integration poses a number of challenges, because the data is not tightly controlled or exclusively used by the data integration system. The query optimizer will often have little or no access to statistics about the data, and it likely has a very limited model of the data source's performance. The lack of initial knowledge and the continued variation in conditions

suggest that static query optimization is inappropriate for the data integration domain. Later in this dissertation, I present techniques for improving performance by adapting the query plan in mid-execution and for exploiting parallelism and flexible scheduling of its constituent operators. However, even without a detailed discussion of these techniques, it is evident that the best execution model for network-bound data is one in which data can be read off the input stream and immediately pipelined through the query plan. This model presents a number of significant benefits for data integration and for enabling adaptivity:

- A single execution pipeline does not require materialization operations, or preparsing or preprocessing of an XML document, so initial answers will be returned more quickly. This satisfies an important desideratum for interactive data integration applications.

- A single pipeline provides the most opportunities for exploiting parallelism and for flexibly scheduling the processing of tuples.

- A single pipeline processes all of the data sources at once, providing the most information about the data distributions and performance of the data sources, and about the selectivity of query subplans. Hence, it provides the most opportunities for learning about and adapting to query execution costs.

Pipelining and adaptive query processing techniques have largely been confined to the relational data model. One of the contributions of this chapter is a new XML query processing architecture that emphasizes pipelining the XML data streaming into the system, and which facilitates a number of adaptive query processing techniques.

As described in Chapter 2, XML queries operate on *combinations of input bindings*: patterns are matched across the input document, and each pattern-match binds an input tree to a variable. The query processor iterates through all possible combinations of assignments of bindings, and the query operators are evaluated against each successive combination. At first glance, this seems quite different from the tuple-oriented execution model of the relational world, but a closer examination reveals a useful correspondence: if we assign each attribute within a tuple to a variable, we can view each legal combination of variable assignments as forming a tuple of binding values (where the values are XML trees or content). In this chapter, I describe an XML query

processing architecture, implemented in the Tukwila system, which exploits the correspondence between the relational and XML processing models in order to provide adaptive XML query processing capabilities, and thus to support efficient network-bound querying, even in the presence of delays, dynamic data, and source failures. This architecture includes the following novel features:

- Support for efficient processing of scalar and structured XML content. Our architecture maps scalar (e.g., text node) values into a tuple-oriented execution model that retains the efficiencies of a standard relational query engine. Structured XML content is mapped into a Tree Manager that supports complex traversals, paging to disk, and comparison by identity as well as value.

- A pair of streaming XML input operators, *x-scan* and *web-join*, that are the enablers of our adaptive query processing architecture. Each of these operators transforms an incoming stream of XML data into an internal format that is processed by the query operators. X-scan matches a query's XPath expressions against an input XML stream and outputs a set of tuples, whose elements are bindings to subtrees of XML data. Web-join can be viewed as a combination of an x-scan and a dependent join — it takes values from one input source, uses them to construct a series of dynamic HTTP requests over Internet sources, and then joins the results.

- A set of physical-level algebraic operators for combining and structuring XML content and for supporting the core features of XQuery [BCF+02], the World Wide Web Consortium XML query language specification, which is nearing completion.

In this chapter, I describe Tukwila's architecture and implementation, and I present a detailed set of experiments that demonstrate that the Tukwila XML query processing architecture provides superior performance to existing XML query systems within our target domain of network-bound data. Tukwila produces initial results rapidly *and* completes queries in less time than previous systems, and it also scales better to large XML documents. The result is the first scalable query processor for network-bound, live XML data. I validate Tukwila's performance by comparing with leading XSLT and data integration systems, under a number of different classes of documents

and queries (ranging from document retrieval to data integration); I show that Tukwila can read and process XML data at a rate roughly equivalent to the performance of SQL and the JDBC protocol across a network; I show that Tukwila's performance scales well as the complexity of the path expressions is increased; and we show that Tukwila's *x-scan* operator can scale well to large (100's of MBs) graph-structured data with `IDREF`s.

The remainder of this chapter is structured as follows. Section 4.1 begins by describing standard techniques for XML query processing, and finishes by presenting the Tukwila architecture and emphasizing its differences. I then describe the XML query operators in Section 4.4, and how they can be extended to support a graph data model in Section 4.5. Section 4.6 provides experimental validation of our work. I conclude in Section 4.7.

4.1 Previous Approaches to XML Processing

As discussed in Chapter 2, the XML data model and XQuery language are considerably more complex than simple relational query processing because of their reliance on path expressions. In particular, the hierarchical nature of XML typically means that a document can be normalized to a single relational table, but a set of tables that have parent-child foreign-key relationships.

People have generally attempted to handle the XML processing problem using one of four methods: (1) focus on techniques for "shredding" XML into tables, combining the tables, and re-joining the results to produce XML output; (2) make a few modifications to object-oriented or semi-structured databases, which also inherently support hierarchy, so they support XML; (3) use a top-down tree-traversal strategy for executing queries; (4) use a custom wrapper at the source end for index-like retrieval of only the necessary content. Before I describe the Tukwila architecture, it is useful to briefly examine these previous approaches and assess their strengths and weaknesses. For a more comprehensive discussion of related work, please see Chapter 8.

Relational databases Research projects at INRIA [FK99a, MFK+00], AT&T Labs (STORED [DFS99] and SilkRoute [FTS99, FMS01b]), IBM Almaden [CFI+00, SKS+01, TVB+02], and the University of Wisconsin [SGT+99] focused on the problems of mapping XML data to and from relational databases. Today, all of the major relational

DBMS vendors build upon this work and provide support some form of XML export (e.g., [Rys01, BKKM00]). In general, results suggest that a relational database is generally not ideal for storing XML, but when the XML data either originates from relational tables or is slow to change, it may be an acceptable solution. Significant benefits include scalability and support for value-based indexes; drawbacks include expensive document load times and expensive reconstruction of XML results. The relational query optimizer can improve performance significantly if the XML query maps to simple SQL, but it frequently makes poor decisions for more complex queries, since it does not understand XML semantics [ZND+01].

Object-oriented and semi-structured databases Several major commercial OODBs, including Poet and ObjectStore, have been adapted to form new XML databases. They provide some benefits over strictly relational engines because their locking and indexing structures are designed for hierarchical data; however, OO query optimizers are still generally relatively weak. The Lore semi-structured database [GMW99], which has a number of special indexing structures, has also been adapted to XML (though performance was shown to be poor relative to a relational system storing XML [FK99a]). Numerous native XML databases [KM00, BBM+01, AKJK+02, MAM01] are also under development. Most of these systems focus on issues relating to efficiently storing and traversing hierarchical objects, as well as on indexing. For more details, please see the discussion of related work in Chapter 8.

Web-oriented DOM processors The techniques mentioned above focus on storage and retrieval of XML content. Of course, XML is expected to also be a format for content *transmission* across networks, and some of this content will be of a transient nature — there is a need for systems that format, query, combine, and present it without storing it. For this domain, an entirely different class of query processors has been developed. These processors, such as the XT, Xalan, and MSXML XSLT engines and the Niagara system from the University of Wisconsin [NDM+01] typically work by parsing an XML document into an in-memory DOM tree; then they traverse the tree using XPath expressions, extract the specified content, and combine it to form a new document. For transient data of small size, this performs much better than storing the data on disk and then querying it; however, it is limited by available memory, and it cannot begin producing answers until after the input document has been parsed. (For

a large document over a slow Internet connection, this may be a significant delay.)

Other web-oriented processors The MIX system from the University of California-San Diego [BGL+99] is web-based, but has a "pull-based" lazy XML evaluation method where the query processor can request specific regions from the data from the mediator as required. This allows for better scalability than the DOM approach, but suffers from two potential problems. First, it requires a custom wrapper at the source, which processes the pull-based messages. Second, one of the major costs in wide-area communication is round-trip time, and the pull-based method requires considerable communication between data source and consumer.

4.2 The Tukwila XML Architecture

Our intent in designing the Tukwila system is to provide the scalability and query optimization of a full-fledged database while maintaining the interactive performance characteristics of the web-based systems. We want to be able to support naive producers of XML content, but also to take advantage of more complex systems that can process queries (or portions of queries) directly.

I defer a discussion of Tukwila's query optimization features to Chapter 6, and instead I shall focus on the query execution architecture and operators in the remainder of this chapter.

The Tukwila architecture is based on the following observations:

1. The basic execution model of XQuery is very similar to that for relational databases: XQuery evaluates the `WHERE` and `RETURN` clauses over every possible combination of input bindings, and each combination of bindings can be viewed as a tuple.

2. The `FOR` and `LET` clauses bind input variables using XPath expressions, which typically are traversals over XML parse tree structure, occasionally with selection or join predicates. The majority of XPath expressions traverse in the downward ("forwards") direction, which matches the order in which a parser encounters the XML elements as it reads an input stream.

3. The majority of selection and join predicates in XQuery tend to involve scalar (text node) data, rather than complex XML hierarchies. Bindings to hierarchical

XML data are most commonly used only in the RETURN clause.

4. The majority of XML processors use DOM-based parsers, which must construct the entire XML parse tree before query processing begins. Incremental parsing, combined with pipeline-based execution[2] as in relational databases, would produce significant benefits. First, it can reduce the time to first answers, as results percolate through the query plan more quickly. Second, the increased parallelism of pipelined operators allows for *adaptive scheduling*, which allows the query processor to overlap I/O with computation [IFF+99] and prioritize important work [UF01].

Based on these observations, we have designed an architecture that is particularly efficient for common-case query execution.

4.2.1 The Tukwila Execution Engine

The core operations performed by most queries are path matching, selecting, projecting, joining, and grouping based on *scalar* data items. Our engine can support these operations with very low overhead, and in fact it can approximate relational-engine performance in simple queries. Our query execution engine also emphasizes a relational-like pipelined execution model, where each "tuple" consists of bindings to XML content rather than simple scalar attributes. This gives us the time-to-first-tuple benefits cited previously, and it has the benefit of leveraging the best techniques from relational query processing.

A high level view of the Tukwila architecture is illustrated in Figure 4.1. The query optimizer passes a plan to the execution engine; at the leaf nodes of this plan are x-scan operators. The x-scan operators (1) retrieve XML data from the data sources, (2) parse and traverse the XML data, matching regular path expressions, (3) store the selected XML subtrees in the XML Tree Manager, and (4) output tuples containing scalar values and references to subtrees. The tuples are fed into the remaining operators in the query execution plan, where they are combined and restructured. As it flows through the operators near the top of the query plan, each tuple is annotated with information describing what content should be output as XML, and how that content should be

[2]Note that while not all operators are pipelineable, a fairly large class of queries can be answered with pipelined operators.

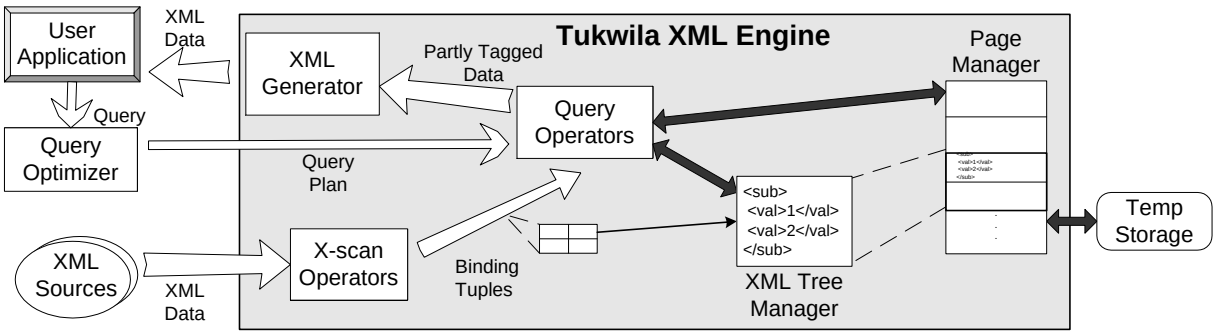


Figure 4.1: Architecture of the Tukwila query execution engine. After a query plan arrives from the optimizer, data is read from XML sources and converted by x-scan operators into output tuples of subtree bindings. The subtrees are stored within the Tree Manager (backed by a virtual page manager), and tuples contain references to these trees. Query operators combine binding tuples and add tagging information; these are fed into an XML Generator that returns an XML stream.

tagged and structured. Finally, the XML Generator processes these tagged tuples and returns an XML result stream to the user.

In a sense, the "middle portion" of our architecture (represented by the "Query Operators" box and the Page Manager) resembles a specialized object-relational database core. Tuples contain attribute values that have been bound to variables; these values can be scalar, and stored directly within the tuple, or they can be XML structures, similar to CLOBs (character large objects) in an OR database — XML structures are stored separately from the tuple, in an *XML Tree Manager*, which is a virtual memory manager for XML subtrees. (Note that we do not attempt to support any other object-oriented types, nor do we implement methods.) The tuples being pipelined through the query plan contain references to subtrees within this Tree Manager, so if multiple variables are bound to an XML tree, the data does not need to be duplicated. Our query operators can manipulate both references within the Tree Manager and values embedded within the tuple, so both object-based and value-based operations are possible — including grouping, nesting, and aggregation. XML subtrees can be reference-counted and garbage-collected when all tuples referring to them have been processed by the system.

The Tukwila architecture allows us to leverage a number of components from the relational world, such as most of the basic memory management strategies and oper-

ators; it is also straightforward to make use of adaptive query processing operators when these are appropriate for the query semantics. We discuss Tukwila's query operators later in this chapter.

4.2.2 Pipelining XML Data

One of the virtues of the flat relational model is its extreme flexibility as a representation. For example, since relations are order-independent, joins can be commuted and non-order-preserving algorithms can be used. Techniques for query decorrelation can be used. Predicates can be evaluated early or late, depending on their selectivity.

A hierarchical data model, such as XML, is often more intuitive to the data consumer (since it centers around a particular concept), but the natural model of execution — breaking a query by levels in the hierarchy — is not necessarily the most efficient. Even more restrictive than hierarchy is ordering: by default, XQuery is very procedural, specifying an order of iteration over bindings, an implicit order of evaluating nested queries, and so forth.

One possible execution model for XQuery would resemble that for nested relations, and in fact "recursive" algebras for nested relations, in which all operators can operate at any level of nesting in the data, have been proposed and implemented (e.g., [HSR91, Col89]). However, we have a preference for mapping XML — even hierarchical XML — into something more resembling the "flat" relational model: an XML document gets converted into a relation in which each attribute represents the value of a variable binding, and position is encoded using counter or byte-offset information. Each such binding may contain arbitrary XML content; but unlike in a nested relational model, the query may only manipulate the top level of the structure. Nested structure must be expanded before it can be manipulated.

This architecture allows us to directly leverage relational query execution and optimization techniques, which are well-understood and provide good performance. Moreover, we believe that, particularly in the case of data integration, we can get better performance from an execution model that preserves structure but has "flat" query operators, for three key reasons. First, many data integration scenarios require significant restructuring of XML content anyway — hence, it makes little sense to spend overhead maintaining structure that will be lost in the end. Second, we can make the unnesting and re-nesting operations inexpensive: our x-scan algorithm provides a

```
FOR $b IN document("books.xml")/db/book,
     $pID IN $b/@publisher,
     $t IN $b/title/data(),
    $pub IN
    document("amazon.xml")/book/item,
     $t2 IN $pub/title/data(),
     $p IN $pub/source,
      $pi IN $p/@ID,
     $pr IN $pub/price/data()
WHERE $pr < 49.95
      AND $pID2 = $pID
      AND $t = $t2
RETURN  <book>
         <name>{ $t }</name>,
         <publisher>{ $p }</publisher>
        </book>
```

Figure 4.2: Query returning titles and publishers for books priced under $49.95 at Amazon. The plan for this query is shown in Figure 4.3.

---

low-overhead way to unnest content, and we can insert additional metadata into every tuple to make it easy to re-nest or re-order values. Third, we believe that there is an inherent overhead in building algorithms that preserve multiple levels of hierarchy, and as a result we feel a "RISC" philosophy is most appropriate.

Example 4.2.1 Figure 4.3 shows a physical query plan and the tuple encoding for the simple XQuery of Figure 4.2. The x-scan operators at the leaves convert XML to streams of tuples by binding variables to the nodes matched by regular path expressions. General query operators such as selects and joins are performed over these tuples: first we select Amazon `publications` priced less than $49.95, and then we join the results with `books` on the `publisher` and `title` values. Once the appropriate binding values have been selected and joined, an output XML tree must be generated with the variables' content. The **output** operator is responsible for replicating the subtree value of a given binding to the query's constructed output. The **element** operator constructs an element tag around a specified number of XML subtrees. In the figure, the output subtree is shown at different stages of construction — first we output $t and insert a `name` element above it; then we output $p and a `publisher` element tag

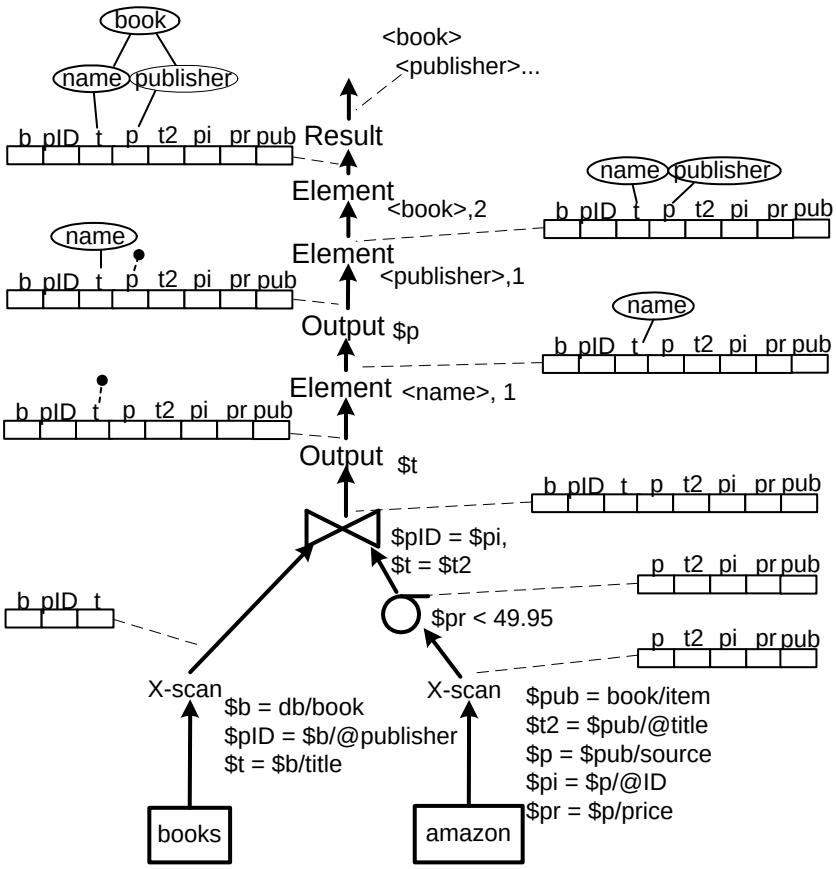


Figure 4.3: Query plan for Figure 4.2 includes a pair of x-scan operators to compute the input bindings, a join across these sources, and a series of **output** and **element** operators that copy the desired variables to the output and construct XML elements around them.

around it; finally, we take both of these subtrees and place them within a `book` element. As a last step, the stream of tuples is converted back into a stream of actual XML. □

In subsequent sections, we describe in detail how Tukwila encodes XML structural information, including tags, nested output structure, and order information.

Encoding XML Tags

In XQuery, a single `RETURN` clause builds a tree and inserts references to bindings within this tree. The tree is in essence a template that is output once for each binding

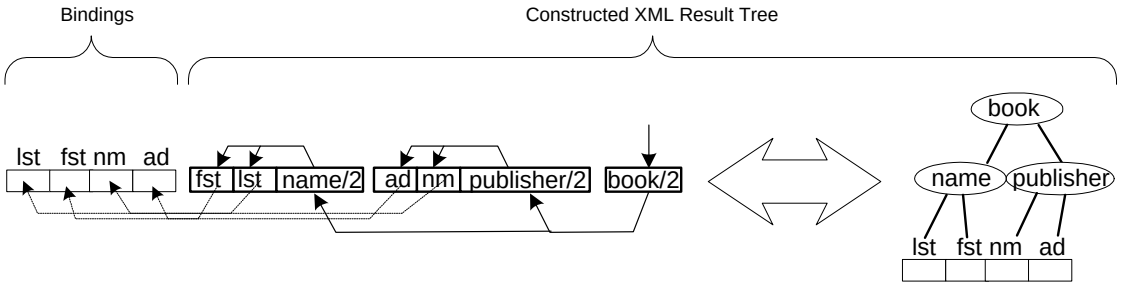


Figure 4.4: Encoding of a tree within a tuple, for the query of Figure 4.3. The encoding is done in bottom-up fashion, so each parent element needs only to specify its number of children. (The arrows are for visualization only.) The tree "leaves" are references to attributes containing bindings.

tuple.

In Tukwila, we need to encode the tree structure and attach it to each tuple. We do this by adding special attributes to the tuple that describe the structure in a right-to-left, preorder form. The benefit of this encoding is that we do not need pointers from parent nodes to children — instead, each non-leaf node specifies a count of how many subtrees lie underneath it, and the decoding algorithm simply expands each subtree recursively.

Figure 4.4 shows this schematically: the tree in the right half of the figure is encoded as the tuple in the left half. The leftmost 4 entries in the tuple are the values of the variable bindings, which contain data values but are not directly part of the XML document under construction. The XML fragment represented by this tuple can be decoded as follows: we start at the rightmost item in the tuple (`book`); this represents a book element with two children (indicated by the "/2" in the figure), and we output a `<book>` tag. We traverse to the leftmost child of the element by moving left by 2 attributes; this yields a `<name>` with 2 children. Again, we traverse the left child – here, we are instructed to output the **fst** attribute. Next we visit the sibling, **lst**, and output its value, and so on.

Of course, the encoding mentioned above assumes that there are no $1 : n$ parent-child relationships in the returned output (every element occurs once for every combination of input bindings). It is very common for XQueries to contain correlated nested

subqueries, which embed many results within each iteration of the outer query.

Encoding Nesting

As mentioned previously, although we want to capture hierarchical nesting of XML results, we do not actually encode it using nested relations. Instead, we flatten or denormalize the results: a single parent tuple with n child tuples is represented by n "wide" tuples with both parent and child information. The XML hierarchy could be decoded by grouping together all tuples with the same parent content. However, that approach does not support proper bag semantics, since duplicate parents will be combined, and it is fairly costly since all parent attributes must be matched. Instead of adopting that approach, we insert an additional attribute that encodes the parent's sequence ID, and group tuples by this ID to find all of the children with a common parent.

Note that this flattened encoding gives the query processor the opportunity to arbitrarily re-order tuples at any point, potentially distributing consecutive data items anywhere in the tuple stream, as long as it performs a sort at the end. It is worth noting that this tuple encoding approach has some similarities to the "outer union" encoding implemented in [CFI+00, SSB+00] and in Microsoft SQL Server's `FOR XML EXPLICIT` mode; however, we encode the branches of the *subquery hierarchy* rather than the *XML data hierarchy*. As a result, we seldom have null values in our tuple stream.

Encoding Order

All of Tukwila's path-matching algorithms can insert attributes that record both the *position* of a binding, by encoding its byte offset within the XML stream, and its *ordering* relative to any other matches for the same variable. Note that these are two distinct concepts, especially when references are used. By adding an ordinal attribute, Tukwila may use non-order-preserving join operators but still maintain XQuery ordered semantics: it simply sorts the data before outputting it.

Generating XML Output

Converting from a tuple stream to an XML stream requires several steps: (1) traverse the XQuery RETURN clause constructor embedded within a tuple, outputting

the appropriate structure, (2) retrieve and embed any referenced XML subtrees, and (3) correctly output hierarchical XML structure which may span multiple tuples. The first step, traversing the tree structure embedded within a tuple consists of starting at the rightmost output attribute and recursively traversing the tuple-encoded tree, as described in Section 4.2.2. Each time a leaf node is encountered, the second step is performed: the referenced XML subtree is retrieved from the Tree Manager and replicated to the output.

The first two steps above are used when all values encoded within a tuple are to be output; this is not necessarily the case if grouping or nesting attributes are present. If nested structure is being represented, then each tuple will actually consist of data for the parent relation followed by data for the child relation. Clearly, the parent data should only be output once for the entire group. This is easily determined by testing whether the parent ID attribute has changed between successive tuples.

Groups can be grouped or nested, so this process scales to arbitrary depth. Moreover, XQuery semantics are outer-join-like, so it is possible to have a publisher with no books. In this case, the book attributes in the tuple are set to null values, and the XML decoder simply outputs the publisher attributes with no book content.

In the next two sections, we describe the Tukwila query operators, which make use of this tuple-based encoding. We begin with the operators that produce the tuple stream: the *x-scan* and *web-join* operators.

4.3 Streaming XML Input Operators

It is Tukwila's support for streaming XML input that most differentiates it from other XML query processors. This support is provided by two different operators that take an input XML stream and a set of XPath expressions, and they return "tuples of trees" representing the combinations of variable bindings that match the XPaths. The simpler operator, *x-scan*, performs XPath matching over a specified input document. The *web-join* operator adds further mechanisms for supporting data-dependent queries: like the dependent join in a relational system, it is provided with a stream of "independent tuples." A web-based (e.g., HTTP) query string is generated by inserting values from the current tuple into a *query generating expression*; this query request is performed, and the resulting XML document is then pattern-matched against XPath expressions. Finally, the matching bindings are combined with the original independent tuple to

produce a cartesian product. X-scan is used for querying static or predetermined web sources, and web-join allows Tukwila to dynamically query and combine numerous sources.

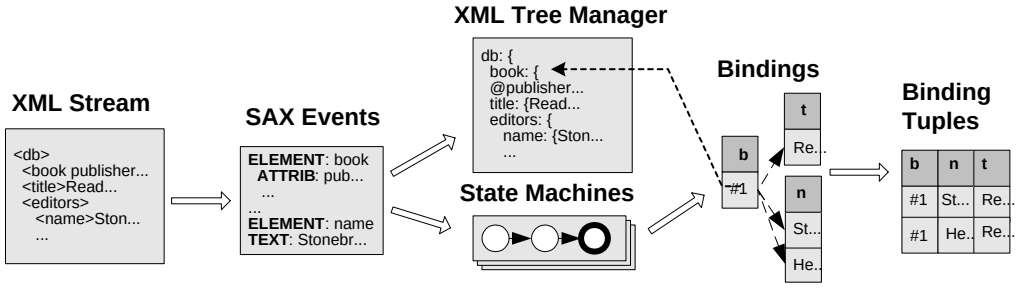
The intuition behind the streaming XML input operators is that an XPath expression greatly resembles a regular expression (where the alphabet consists of element and attribute labels), and this can be simulated by a finite state machine. Tukwila uses an event-driven (SAX) XML parser to match input path expressions as an XML stream is being parsed; a variable binding is created each time a state machine reaches an accept state. Bindings are combined to form tuples, which are pipelined through the system, supporting output of XML results as the data stream is still being read.

While using a finite state machine to match XPath expressions seems natural, the algorithms for supporting the details of XPath, combining the bindings, and supporting efficient execution are quite complex. To the best of our knowledge, Tukwila is unique in creating pipelinable XML query results directly from a data stream, and in using finite state machines to do so — and as a result it shows significant performance and scalability benefits over other systems. Systems such as Niagara fetch and parse an entire input XML document, construct a complete DOM representation in memory, and finally match path expressions across the tree and pass the results through query operators. XSLT processors such as Xalan, MSXML, and XT are similar, but use a recursive pattern-matching semantics rather than a set of query operators. Most other XML query processors are designed to operate on XML in a local warehouse. One interesting system that is not a query processor but bears some resemblance to Tukwila is the DBIS system and its XFilter [AF00] operator[3]. DBIS takes XML documents and determines whether they meet specific XPath expressions, and it "pushes" those that do to "subscribing" users. DBIS performs document filtering rather than query processing, so XFilter, an operator with a binary (match/non-match) return value, differs substantially from x-scan in its functionality. The XML Toolkit [GMOS02] builds upon the XFilter work, but proposes a "lazy" approach to building deterministic finite state machines from nondeterministic path expressions.

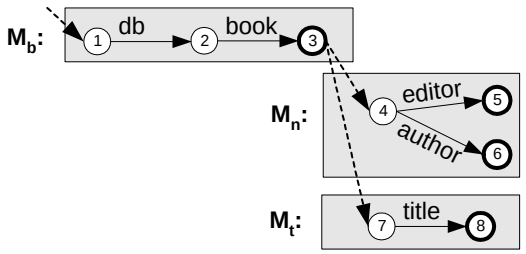
We now present the details of the streaming XML input operators, beginning with x-scan.

---

[3]In fact, the XFilter and x-scan operators were developed concurrently.



(a) X-scan process



(b) State machines for Fig. 2.5 query

Figure 4.5: X-scan takes an XML document and maps it into the XML Tree Manager, while simultaneously running state machines over the parse tree. Each state machine creates variable bindings, and these must be combined to produce binding tuples. Solid arcs in (b) denote state transitions on the label; dashed arcs denote dependencies between machines.

4.3.1 X-scan Operator

Given an XML text stream and a set of regular path expressions as inputs, x-scan incrementally outputs a stream of tuples assigning binding values to each variable. A binding value is typically a tree — in which case the tuple contains a reference to data within the Tukwila XML Tree Manager — but if it is a scalar value, this value may be "inlined" directly within the tuple. A depiction of x-scan's data structures appears in Figure 4.5: the XML stream is processed by an event-driven SAX parser, which creates a series of event notifications. The XML data is stored in the XML Tree Manager and is also matched against a series of finite state machines (responsible for XPath pattern matching). These state machines produce output binding values, which are then combined to produce binding tuples.

Basic XPath expressions are a restricted form of regular path expressions[4]. Thus x-scan converts each XPath expression into a regular expression and generates its corresponding nondeterministic finite state machine, which it then converts into a deterministic machine, for reasons discussed later in this section. XPath expressions originating at the document root are initialized to the *active* mode, and the active machines' states are updated as x-scan encounters subelements and attributes during document parsing. Figure 4.5(b) shows the state machines created for the example query of Figure 2.5.

Initially, only the top-level machine (M_b in our example) is active. When any machine reaches an accepting state, it produces a binding for the variable associated with it. The machine then activates all of its dependent state machines, and they remain active while x-scan is scanning the value of the binding. In our example, the machines M_n and M_t remain active while we scan children of b.

Associated with each machine is a table for binding values. As a machine reaches an accept state, it adds an entry containing its bound subtree value, and also an association with the entry's parent binding (shown in Figure 4.5(a) as a dashed arrow from parent to child)[5]. In our example, M_b's table would just store values of b, while M_n and M_t would store author/editor names and titles, respectively, and these would be associated with their corresponding b values. The final output of x-scan is essentially a join of the entries maintained by the machines, done for matching parent-child pairs (this is done in a data-driven, rather than iterator, model, as with a pipelined hash join [WA91]).

We illustrate the execution of x-scan on our example data of Figure 2.3. Suppose M_b is initialized to machine state 1, which takes the XML `root` as its start position. The `root` node is a virtual node representing the entire document, and its only child is the `db` node. X-scan follows the edge to the `db` node, setting M_b to state 2. Next, x-scan can follow one of two outgoing edges to `book` nodes. It chooses the leftmost one (*Readings in Database Systems*), causing it to set M_b to state 3. M_b is now in an accepting state, so x-scan writes the reference to the current node into M_b's table, suspends M_b, and activates M_n and M_t. The `editor` element takes M_n from state 4

[4]We shall discuss additional, non-path-oriented XPath features later in this section

[5]The implementation can store subtrees by value or reference. For expository simplicity, we write as though nodes are stored by `ID`-based reference and attributes are stored by value.

to 5, which is an accepting state for M_n. Hence, x-scan writes "`Stonebraker`" and a pointer from the current book. In the meantime, M_t follows the arc to the `title` element, putting its machine into state 8, which is also an accepting state. Hence, the tuple $\langle$`title1`, `book1`$\rangle$ will be written into M_t's table. From this node, no (non-text) children remain for exploration, so x-scan pops the stack and backs up the state machines. It sets M_b to state 2, where it can continue to explore the second `book` node, proceeding as before. □

To this point, we have described how x-scan performs simple path expression matching. However, XPath supports capabilities beyond mere path matching, and these features are also provided by x-scan.

Querying order (node indexing): XPath expressions may restrict bindings based on ordering information, such as a constraint on a subelement's index number (e.g., "2nd paragraph subelement") or on the relative positions of bindings (e.g., $a BEFORE $b). X-scan supports both capabilities: the x-scan state machines are annotated with counters to keep track of element indices, and the output of the x-scan can include both a binding and its index or its absolute position. A select operator can then filter out tuples based on order.

Selection predicates: Another useful capability in x-scan is the ability to apply certain selection predicates over the variable bindings and their subtrees. These can be simple predicates over values (e.g., "bind $b to book titles with the value 'Transaction Processing' ") — similar to "sargable predicates" [SAC+79]. Additionally, x-scan supports existential path tests (e.g., return books only if they have titles). Existential quantification of a path is similar to any other path expression, except that its binding is not returned. (Other forms of existential quantification are possible, and they can be implemented using correlated subqueries and traditional relational techniques.)

Node test functions: XPath expressions often include node tests, which restrict the type of XML node being selected (common instances include `text()`, `comment()`, and `processing-instruction()`). Similarly, an XPath edge with an at-sign prefix () represents an attribute node. All of these conditions are expressed within the x-scan state machines as restrictions on the XML nodes to be matched.

Traversing in reverse: Our current implementation of x-scan does not evaluate the XPath "parent" axis, i.e., it does not traverse backwards through the tree. Instead, the Tukwila query optimizer rewrites path expressions with the parent operator by splitting them into a parent-binding and a child-binding. Conditions are evaluated on the child, and if they are met, the parent is used. (While this process may at times be less efficient than supporting a true "parent" traversal, we expect use of the parent axis to be uncommon.)

Efficiency enhancements: In x-scan, we include a number of optimizations to boost XML parsing and processing performance. First, we avoid processing XML content (i.e., handling SAX parser messages) when the state machines are inactive — it is important to avoid unnecessary copying and handling of string data. Additionally, the instant it becomes evident that a subtree cannot satisfy an XPath expression (e.g., it does not meet a sargable predicate or is missing an attribute), we deactivate the state machines until the next subtree is reached.

Expected complexity of state machines: While x-scan uses deterministic finite state machines — which can be exponentially larger than the nondeterministic machines from which they are derived — XPath expressions tend to be short (queries to depth of more than 6-8 seem to be rare). Furthermore, XPath only supports a restricted version of regular path expressions: instead of Kleene closure, XPaths are limited to simpler "wildcard" and "descendent" operations.

Handling memory overflow Typically, x-scan needs very little working space — it outputs a stream of binding tuples (i.e., sets of subtrees) and little state needs to be maintained between the production of any two tuples. However, there are two cases where it may run out of memory.

First, the XML data that is still being referenced may be larger than memory. Since the XML Tree Manager is a paged data structure, segments of this data are swapped to and from disk as needed. Of course, as a result, a large XML file could produce "thrashing" in the swap file during query processing. However, our experiments in Section 4.6 suggest that this is typically avoided, which we attribute to two factors. First, the system supports "inlining" of scalar values: string, integer, or other "small" data items are embedded directly in the tuple, avoiding the dereferencing operation.

Typical query operations in XQuery are done on scalar rather than complex data (e.g., joining or sorting are frequently based on string values); thus these operations often only need data that has been inlined. Large, complex tree data is typically only required at the XML generation stage, when the final results are returned. A second mitigating factor is that many XML queries tend to access the input document in sequential order, and the Tree Manager therefore can avoid re-reading data that has been paged out. For purposes of comparison, we point out that a paged DOM-based approach would have similar behavior to our scheme (except that in-memory representation of XML is larger in a DOM tree, typically at least 2-4 times larger, because of DOM's heavyweight nature); a mapping from XML to relations ("shredded XML") typically requires a significant amount of materialization in the first place, and often incurs heavy costs whenever it needs to perform joins to recreate irregular structure.

The second memory overflow case, which may occur for trees with high fanout, is when sibling XPaths each have many bindings, and x-scan must return all combinations. To take the query of Figure 2.5 as an example, we might somehow have many authors and alternative titles per book, and x-scan would have to return every possible title-author pairing for each book. To accomplish this, x-scan maintains the current value of b, plus tables for n and t bindings. As values of n are added, they are combined with b and all existing values of t; and the process works similarly for new values of t. Each time a new value of b is encountered, the tables can be flushed and the process restarted. In an extreme case, the tables may grow larger than memory — this case can be handled in a manner similar to the pipelined hash join overflow strategies of [UF00, IFF+99].

4.3.2 Web-Join Operator

The x-scan operator is analogous to the sequential table scan in relational databases, and to the "wrapper fetch" operation in relational data integration: it allows the query processor to read through an XML document and extract out the relevant content. If the source has more sophisticated query capabilities, certain operations may in fact be "pushed" into it via the x-scan HTTP query request.

In distributed query processing, sometimes it is beneficial to make use of a *dependent join* operator instead of more traditional table scan and join operators. Instead of requesting data independently from two sources and then joining it, the depen-

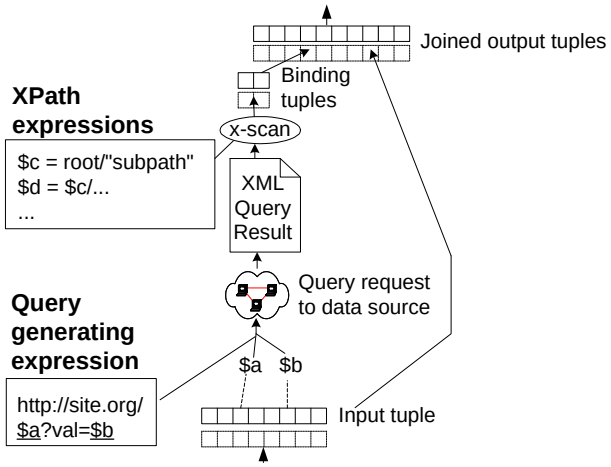


Figure 4.6: The web-join operator takes each input tuple and substitutes its values into a query generating expression. This expression becomes a web request that queries a data source; its results are matched against a set of XPath expressions by an x-scan operator. The resulting tuples are joined with the original input tuple to produce a set of results for later query processing.

dent join reads data from one source, sends this data to the other source and requests matching values, and then combines the data from the two sources. This operation is particularly useful in two cases: one is if the join with the second source is highly selective, so much less data is transferred using the dependent join. The second is when the source requires input values before it will return an answer (e.g., the source may be an online bookseller with a web forms interface that requires an author or title), this is equivalent to the notion of "binding patterns" in relational data integration [RSU95, KW96, LRO96].

In a web context, a query to a data source is generally provided using one of two means: via an HTTP request (GET or POST) or via a SOAP call with some form of query (perhaps an XQuery). For both of these domains, we propose the *web-join* operator. Web-join (Figure 4.6) is intuitively similar to the combination of an x-scan operator with a relational-style dependent join: it receives an input tuple stream and a query generating expression (shown in the lower left of the Figure as a string with two underlined parameters, `$a` and `$b`). The parameters in the query generating expression will be instantiated with values from the input tuple stream, and the resulting query string will be evaluated as a URI string, HTTP POST sequence, or SOAP envelope.

The XML resulting from the request is now evaluated against XPath expressions by an embedded x-scan operator. Now, each of the resulting binding tuples is joined with the original tuple and output. The process repeats for each tuple of the original input stream.

Web-join is an important operator for querying dynamic sources, especially ones with embedded Xlinks or URIs. It also allows our query processor to do "lazy" evaluation: Tukwila can request some initial data, execute filtering operations on it, and then request additional content for those elements that remain.

4.4 Tukwila XML Query Operators

The previous section presented the query operators that are responsible for mapping an XML data stream into a stream of tuples. Now we describe the query operators that process this data. A logical query algebra usually is designed to be expressive and minimal. In contrast, the set of physical query operators needs to have predictable performance (to make the optimizer's cost model easier to build) and in efficient implementations for specific contexts (where the optimizer should choose the most appropriate implementation).

As we have constructed the physical algebra for Tukwila, we have focused on providing efficient support for executing a relatively expressive "core" of XQuery: our focus to this point has not been on supporting the full language. Currently, we do not support recursion, typechecking, or conditional assignments. We have also implemented only a small subset of the proposed XQuery function library. However, we feel that the current implementation is sufficient to demonstrate how common-case queries can be executed quickly, and that it can eventually be extended to include the absent features. The complete list of operators is summarized in Table 4.1, and I provide more detail below.

Streaming Input The *x-scan* and *web-join* streaming input operators were already discussed in Section 4.3.

Path Evaluation The *follow* operator is a path traversal operator. It takes as input a binding tuple, evaluates an XPath (which may involve following an `IDREF` or even an XLink) originating at one of the bindings, and returns a sequence of 0 or more binding

Table 4.1: Physical query operators and algorithms in Tukwila

Name	Class	Function
x-scan	streaming input	Match input path expression
web-join	streaming input	Query based on bound vars.
follow	path evaluation	Evaluate XPath over binding
select	combination/filter	Filter tuples by predicate
project	combination/filter	Discard bindings
hybrid hash join	combination/filter	Equijoin
pipelined hash join	combination/filter	Equijoin
merge join	combination/filter	Ordered equijoin
nested loops join	combination/filter	Order-preserving join
union	combination/filter	Relational-style union
collector	combination/filter	Union with fail-over
assign	combination/filter	Evaluate expression
distinct	2nd-order	Remove duplicates
sort	2nd-order	Reorder tuples
aggregate	2nd-order	Compute aggregate over group
nestChild	nesting	Correlated nesting of elements
group	nesting	Group and restructure sets of elements
output	result	Output binding to XML
element	result	Create XML element
attribute	result	Create XML attribute

tuples. Since x-scan has very little overhead, *follow* is primarily useful when following XLinks or references within a graph-structured document[6].

Combination/filter Most of these operators are almost identical to the standard relational equivalents. One notable exception is the *collector* operator, which we first proposed in [IFF+99]: it starts reading from one or more data sources, but can switch to alternate sources depending on availability and performance. We have one additional operator, *assign*, which adds a new attribute (and binding) to a tuple, assigning it the result of some scalar expression. This expression may be posed in terms of other bindings (e.g., a string concatenation).

[6]We expect that XLink reference traversal will be less frequently used than the other operations, and hence we have defined but not yet implemented the follow operator.

Second-order The second-order operators all process sets (or bags) of tuples. The only nonstandard operator is *aggregate*, which takes a stream of tuples representing subquery content nested within parent query content and, for each parent, computes an aggregate value across all of its children. This is very similar to the relational GROUP BY operator, with two exceptions: (1) the grouping information is already present, as the result of a *group* operator as discussed below, and (2) the nested data within the group is preserved rather than discarded.

Nesting These operators are also second-order, but we separate them because they have a special role in our XML encoding. The *group* operator hierarchically restructures tuples: for each set of tuples that have an identical set of grouping attribute values, the operator conceptually outputs a single tuple with these grouping attributes, *plus* an embedded subtable with tuples of the non-grouped attributes. In Section 4.2.2, we described how this nested structure is encoded within "flat" tuples; we provide each tuple-group with a unique ID, and this becomes the identifier for the "parent tuple," while all non-grouped attributes are the "child tuple." *Group* is primarily useful for providing a relational-style group-by, or for extracting common structure from "flat" XML.

Nested FLWR query expressions are a basic idiom in XQuery, and we handle this case with our *nestChild* operator, which has semantics very similar to a relational left outer join. *NestChild* takes a parent and a child tuple stream, plus a correlation predicate. For each parent tuple, *nestChild* finds the set of child tuples meeting the predicate and groups them with the parent tuple. At the same time, it groups the parent's XML subtree together with all of the children's XML subtrees. (We note that many nested relational algebras and their derivatives include a *unary* operator called "nest" which is closer in nature to our *group* operator than our *nestChild*. Systems with that type of algebra must perform least two operations — join and "nest" — to achieve the same effect as our *nestChild*, and end up doing redundant work.)

Whereas the join operator is typically allowed to output results in any order, *nestChild* semantics require a nested loops join-like ordering, where all child values are returned with their parent. We encode the "hierarchical tuple" as described in Section 4.2.2, which preserves enough information to determine whether any two "flat" tuples contain the same parent tuple. Using this approach, if we use order-preserving operators, we can pipeline the encoded structure all the way to the output result; otherwise, we

must use a hashing or sorting algorithm to cluster tuple groups together before we convert them to XML.

Result These operators are responsible for creating the output for the XQuery. They construct the output XML tree and are applied using a postfix ordering. An *output* operator always creates a leaf node in the output; it simply outputs the result of a binding as a string value. *attribute* wraps the result of the last *output* node within the specified XML attribute name (which may be a literal or the value of a binding). *element* constructs an XML element around the last k nodes (which may be the result of previous *output*, *attribute*, or *element* operations), where k is a constant specified by the query and the attribute's label may be either a literal or the value of a binding.

With this basic set of operations, Tukwila can execute the core, database-like subset of XQuery that avoids conditionals, recursive functions, and type information. Additionally, whereas XQuery is a heavily tree-oriented query language, we can also support graph-structured data in Tukwila, as I describe in the next section.

4.5 Supporting Graph-Structured Data in Tukwila

To this point, we have presented the Tukwila query processing system under the assumption that our data is completely tree-structured and that this structure is mirrored in the XML element/attribute hierarchy. However, the XQuery data model and language do support limited forms of encoding graphs in XML, through the use of `IDREF` attributes (within a document) and XLinks (outside a document). In this section, we briefly describe some of the issues involved in supporting these operations.

4.5.1 Join-Based Traversal

The conventional way to evaluate an `IDREF` is to use a join operation: for example, suppose we allow only a single `IDREF` in each XPath. To evaluate these expressions, take all XPaths and separate them into "pre-`IDREF`-traversal" and "post-`IDREF`-traversal" steps. Do an x-scan of the input document with the pre-`IDREF` XPaths. In parallel, do an x-scan over the same document for all elements that have `ID`s, and evaluate the post-`IDREF` XPaths. Now join the results based on matches between the last `IDREF` of the first x-scan and the originating `ID` of the second x-scan. Similar techniques can

be used to support k IDREFs in each XPath. XQuery does not support Kleene closure over IDREFs, so a query must have a fixed number of reference traversals and this technique can always be made to work.

The join-based traversal method is effective for following links in many situations, but it has two potential drawbacks. First, standard join algorithms will not "short-circuit" once an IDREF is matched to its target ID, i.e., they do not "know" that there should be precisely one match to every IDREF. Alternative means of traversing IDREFs, which we discuss next, can move to the next reference as soon as the current reference has been matched once — fully pipelining the results. Second, the join-based traversal only works for IDREFs or XLinks that all belong to the *same* target document.

4.5.2 Follow-Based Traversal

A second option, which supports both IDREFs and XLinks, is to use the Tukwila *follow* operator. In following an IDREF, *follow* does an XPath match against an *in-memory* XML document that was output from a prior x-scan and returns a set of bindings. For IDREF traversal, *follow* makes use of an index of ID elements that was created by the x-scan operation. This index is further described below.

Follow is intuitively an x-scan that operates on "tuples of trees" rather than on XML documents. Given a set of path expressions and an input tuple stream (as well as the XML trees it references), *follow* adds new variable bindings to each of its input tuples by evaluating the path expressions against the trees within the tuple. If a pattern matches multiple subtrees within the tuple, a set of tuples will be returned, one for each possible binding combination. (This operator is essentially a special case of the *map* operator in some object-oriented algebras.)

Follow is the only reasonable option for evaluating XLinks. At each link, it opens the referenced document and evaluates the XLink path expression to select out the desired XML data, then matches the remainder of the query's XPath against this document fragment, in a manner similar to x-scan or web-join.

4.5.3 Graph Traversal with X-scan

As we shall see in our experimental evaluation, x-scan's state machine infrastructure adds very little overhead in performing XPath matching against an XML tree. Hence, any XPath traversal across a document's tree structure should generally be done at

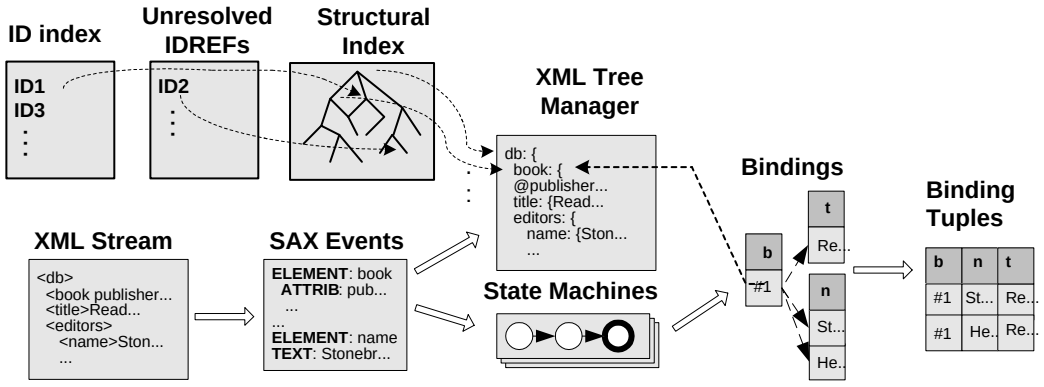


Figure 4.7: Graph-based execution of x-scan uses 3 new data structures (upper left). The ID index records the positions of each ID within the XML data graph; the unresolved index maintains a list of IDREFs that have not been resolved; the structural index physically encodes element, attribute, and reference relationships.

the x-scan level. Traversals across IDREFs can also be done at the x-scan level, and as we shall see later, this performs reasonably for moderately sized documents that do not contain large numbers of references. We now discuss the extensions necessary for traversing IDREFs in x-scan.

The first difference is the addition of three new data structures, shown in the upper left corner of Figure 4.7:

- ID index: records the IDs of all elements and their matching locations in the XML data. It is used to facilitate resolution of IDREFs in the graph.

- Unresolved-ID index: maintains a list of references to not-yet-seen element IDs (to be resolved as they are found later in the input).

- Structural index: provides an index of the XML graph, corresponding to Figure 2.3, but without the data values at the leaves. This is not necessary, but speeds x-scan's traversal through the graph in memory.

When x-scan is run on a graph-structured XML source, it generates a structural index, which is a trie-like index of the XML graph structure (i.e. the subelement and IDREF links). This index allows x-scan to quickly traverse back through XML structure

in evaluating references. In addition, as we explain below, the construction of the structural index continues even when we need to suspend the state machines because of unresolved IDREFs. This index is intended only to last for the lifespan of the query, so it is built in memory and paged out only as necessary. (We expect that x-scan will generally only be used to traverse moderately-sized graph data, and will be supplanted by *follow* or joins for larger documents, so paging of this index should seldom occur.)

Each node in the index contains information about an element (its ID and an offset into the original XML data file so that the node's source can be accessed quickly) as well as pointers to all subelements, attributes, and IDREFs of the element. Essentially, the index structure looks like the graph of Figure 2.3 except that data values such as those in the leaf (PCDATA) nodes are not stored.

In addition, x-scan creates the ID index, which records all the IDs that it has encountered so far, mapping from ID to entry in the structural index, and the unresolved-ID index which records all IDs that have not yet been seen in the input, and lists all referrers to each such ID.

X-scan's general execution proceeds similarly to the tree-structured case, except when an element with references is encountered, and the references are to be traversed by the regular path expression. If the reference is to an element that has already been parsed (a backward reference), the state machines are run over the reference's target in the structural index, and then parsing continues.

Forward References

On occasion x-scan will encounter an IDREF edge which points "ahead" to a node which has not yet been parsed. When x-scan hits a forward reference, it pauses all state machines [7] and adds an entry to the list of unresolved IDREF symbols, specifying the desired ID value and the referrer's address. However, x-scan continues reading the XML source and building the structural index. Once the target element is parsed, x-scan fills its address into each referring IDREF in the structural index, removes the entry from the list of unresolved IDREFs, and awakens the state machines and proceeds. Although this approach causes x-scan output to stall as it waits for a reference to be resolved, our empirical results have shown that with the help of the structural

[7] Conceptually, x-scan could continue state machine operation until the reference target is found, then insert the target, return all of the matches found afterwards, and continue normal operation; but for simplicity of coding, our implementation does not do this.

index, x-scan "recovers" quickly. In the worst case, x-scan should still do at least as well as a DOM-based query processor — as with DOM, it builds a structure in memory that can be quickly traversed; however, unlike the DOM implementations with which we are familiar, x-scan can still execute when this structure must be paged to disk because it exceeds virtual memory.

Cycles

In order to avoid cyclic traversals of references, x-scan maintains a history of nodes visited by each automaton state in a given path traversal. X-scan uses deterministic automata, so if a machine re-visits a node that it has encountered *in the same state along the same path*, this is a cycle and can be aborted.

4.5.4 General Guidelines for Reference Traversal

There are a number of ways of supporting graph-structured data within the Tukwila system. Each of these methods has different capabilities and performance results; we now present a set of guidelines by which an optimizer can choose the best mechanism for evaluating XPaths in the graph context.

We begin by noting that the x-scan operator is very efficient on strictly tree-structured data, so we believe it will seldom make sense to use either the join or *follow* methods to traverse anything but `IDREF`s or XLinks. Thus, the query processor should use x-scan to evaluate the segment of an XPath before (or after) a reference traversal.

The type of reference being evaluated now becomes important: as was noted earlier, the join method does not work for evaluating XLinks. Our x-scan implementation does not follow XLinks, either, because such a traversal is quite expensive and probably should not be done as a leaf-level operation. Thus, for XLinks, the *follow* operator is the only option.

For documents with a low number of `IDREF`s, the x-scan traversal approach works well. Once a large number of `IDREF`s must be evaluated, however, the join and *follow* alternatives look more promising. The *follow* operator is a unary operator, and only requires one scan of a given document; however, it traverses through the XML data (which may result in thrashing if the document is larger than memory). The join operator is less likely to cause thrashing, since it combines tables that are each completed in a single pass — but it requires two separate scans of the input document.

4.6 Experimental Results

Now that we have seen the details of the Tukwila query engine for both tree-structured and graph-structured data, we move to our experimental validation of the system. Our implementation was written in C++. We originally wrote the system for Windows 2000 using the Apache Xerces-C XML parser at the core of our x-scan implementation. Later, we migrated to a slightly slower Linux machine using James Clark's expat 1.95.1, which performed faster XML parsing. In the experiments below, we used the expat-based implementation for comparing XML pattern matching experiments (Section 4.6.1), and we relied on the Windows machine for the compute-bound and memory scalability experiments, since it was faster and had more memory.

Our system architecture is based on a client-server model, with a Java client that submits queries using SOAP over HTTP, then reads and times the XML results. Most experiments measured the performance of the Tukwila engine on an 866MHz Pentium III machine with 1GB RAM (of which we allocate only a subset to Tukwila) under Windows 2000 server; but as mentioned above, for the studies of XML pattern matching performance in Section 4.6.1, we instead ran Tukwila on an 800MHz Pentium III with 256MB RAM under Red Hat Linux 7.1. In all cases, XML documents were served via HTTP from our web server, a dual Pentium II 450MHz system with 512MB RAM, running Windows 2000 and IIS 5. The web server and query processing machine communicated via 100Mb fast Ethernet, with each machine on a separate subnet within a larger-scale network. Experiments were run once for "warm-up" and repeated at least 7 times, and error bars are included for queries where the confidence interval is less than 95%.

Experimental data sets were chosen to encompass a range of different XML data classes, and are listed in Table 4.2. They include real documents, real semistructured data, semistructured data generated with the recent XMark XML query benchmark [SWK+02], synthetic data with references, and relational tables saved in XML format. The synthetic data with references was the only data set that we created ourselves; it was designed to have random variation in depth and distribution of `IDREF`s. The data set was generated using the following process: replicate a "core" XML subtree a specified number of times, and then randomly attach it to different points within the current document, with probability 15% that it attaches to the root. Afterwards, the designated number of `IDREF` edges were added between random pairs of endpoints.

Table 4.2: Data sets used in experiments.

Name	Size	Description
religion	7MB	Concatenation of Bosak's collection of religious texts (bible, quran, Book of Mormon)
xmark-50	59MB	0.5-scale-factor XMark auctions file
xmark-1000	118MB	1.0-scale-factor XMark auctions file
xmark-500	596MB	5.0-scale-factor XMark auctions file
dmoz	341MB	Open directory (dmoz) RDF hierarchy
dblp-proc	155KB	DBLP list of conference proceedings
dblp-pubs	8.9MB	DBLP list of conference publications
dblp-conf	39MB	DBLP complete conference information
dblp-cj	61MB	DBLP complete conference and journal information
customer-10	0.5MB	TPC-H 10MB (0.01-scale-factor) customer table in XML
orders-10	5.4MB	TPC-H 10MB (0.01-scale-factor) orders table in XML
lineitem-10	32MB	TPC-H 10MB (0.01-scale-factor) lineitem table in XML
customer-100	5.2MB	TPC-H 100MB (0.1-scale-factor) customer table in XML
orders-100	53MB	TPC-H 10MB (0.1-scale-factor) orders table in XML
lineitem-100	324MB	TPC-H 100MB (0.1-scale-factor) lineitem table in XML
synth	100K-100MB	Data from synthetic generator (see text)

Since we are proposing a new model for query execution, we begin by comparing Tukwila's performance with that of systems using more traditional approaches. Later, we look at scalability and the performance of Tukwila on database-style operations including join; we examine how hierarchically nesting XML content limits performance because it restricts order; and we look at how Tukwila's x-scan algorithm can be used to support `IDREF` traversal for graph-structured data.

4.6.1 XML Extraction Queries

Clearly, the core operation at the heart of any XML processor is the pattern-matching and XML content extraction step, and in fact this is where Tukwila's approach differs from other implementations. Our first set of experiments focuses on comparing the relative performance of Tukwila with other systems when extracting XML content with XPath expressions. Our suite of queries is described in Table 4.4, and consists of a mix of text-oriented and path-oriented queries over different types of hierarchical documents and semistructured data. (We examine performance on more regular XML

Table 4.3: Systems compared in Section 4.6.1.

Name	Implemented	Domain	Description
Xalan 1.1	C++	Doc	Apache XSLT processor, built over Xerces-C parser
XT 19991105	Java	Doc	James Clark's XSLT processor
MSXML 4.0	C++	Doc	Microsoft parser and XSLT processor toolkit
Niagara 1.0	Java	XML-DB	University of Wisconsin XML integration system
Tukwila 1.0	C++/Java	XML-DB	XML engine described in this paper

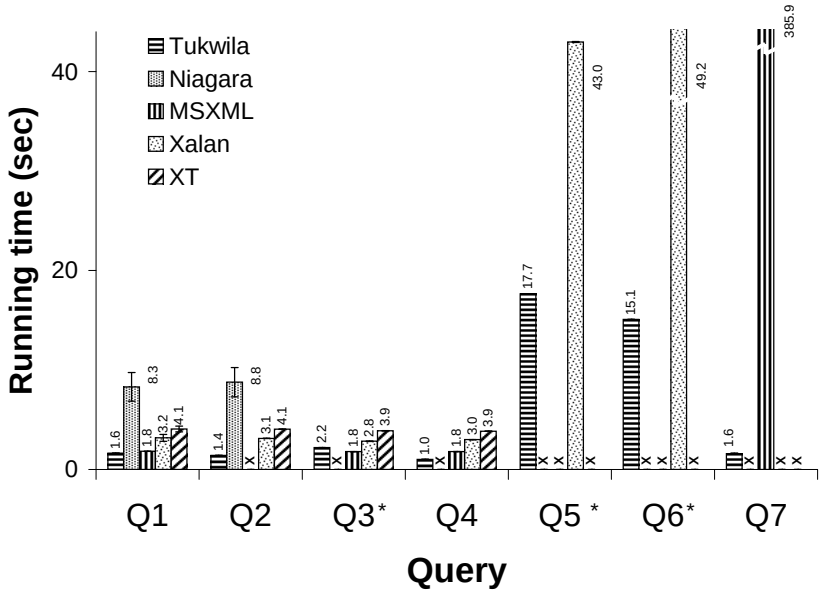
Table 4.4: List of queries used for comparing pattern-matching performance.

Nbr.	Input	Query
Q1	religion	All chapter 5's in Book of Mormon (medium trees)
Q2	religion	All chapters 8+ in Book of Mormon (medium trees)
Q3	religion	Sura titles with "Mormon" from Book of Mormon (single result)
Q4	religion	Suras in Quran with "The" in title (large trees)
Q5	xmark-50	XMark query Q1 (extract person0 data)
Q6	xmark-50	XMark query Q2 (extract bidder 1's bid increases)
Q7	dmoz	Return all topic IDs in Open Directory structure RDF file

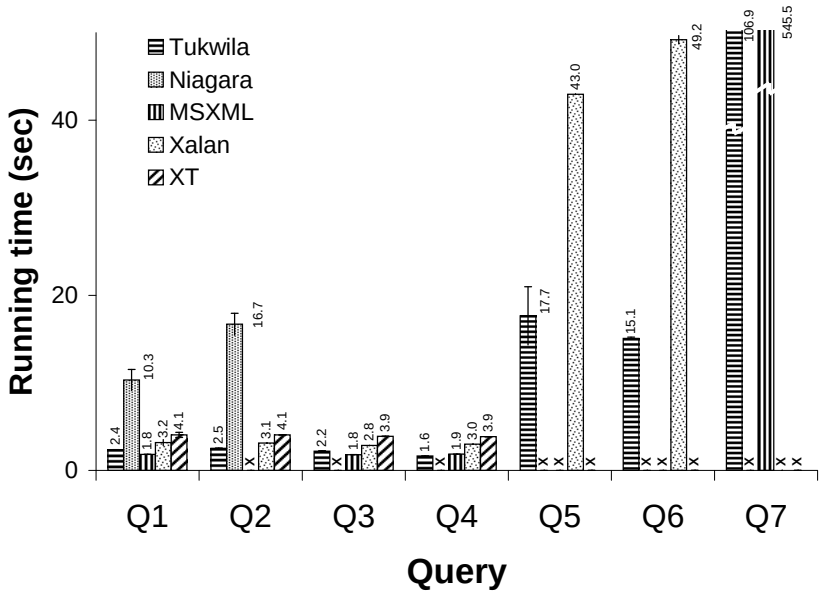
data from relational systems in the next section.)

See Table 4.3 for details on the systems in our comparison; all except for Tukwila are main-memory-only XML engines. We included three popular XSLT processors in our study: the Apache Xalan-C system, James Clark's XT engine (which was generally rated as one of the faster XSLT engines), and the XSLT processor in Microsoft's MSXML 4.0 toolkit (which has been heavily optimized and is considered to have the fastest parser and XSLT engine available). We also wanted to compare with data integration systems, so we included the December 2000 version of the Niagara system (as of this time, the latest version that is publicly available). Early in the development of Tukwila, we also compared our performance against the Lore System [GMW99], an XML repository; at the time, Tukwila significantly outperformed Lore. Unfortunately, Lore is no longer being distributed, and therefore we omit it from our comparison, because it would be unfair to compare with an outdated version of Lore.

Figure 4.8 shows the results for the queries in two graphs: part (a) shows the time to the initial 5 answers, as a way of measuring quick feedback to the user; part (b)



(a) First 5 tuples returned (queries marked with a * have fewer than 5 tuples total).



(b) Total query time

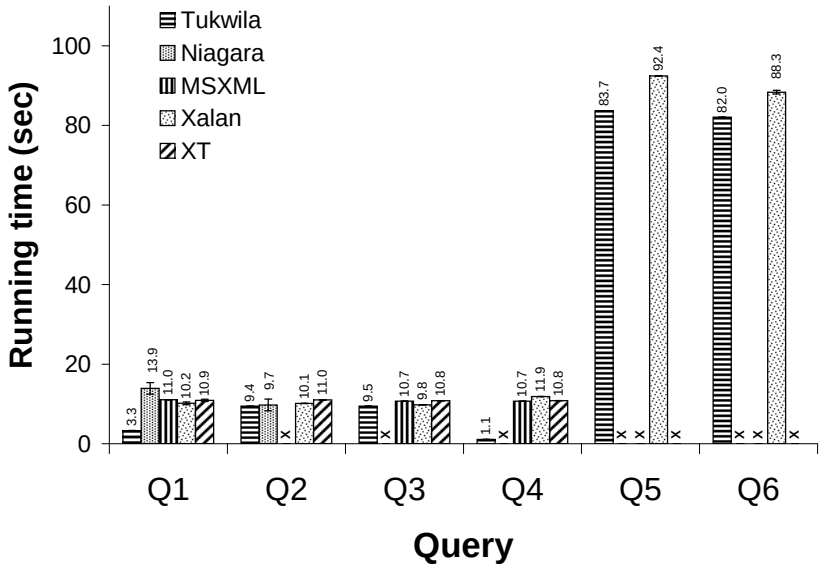
Figure 4.8: Experimental comparison of XML queries shows that Tukwila has equal or better total running time (and better time to first tuples) for a variety of XML extraction queries.

shows the overall query completion time. Note that queries Q3, Q5, and Q6 all had fewer than 5 answers, so they have identical timings.

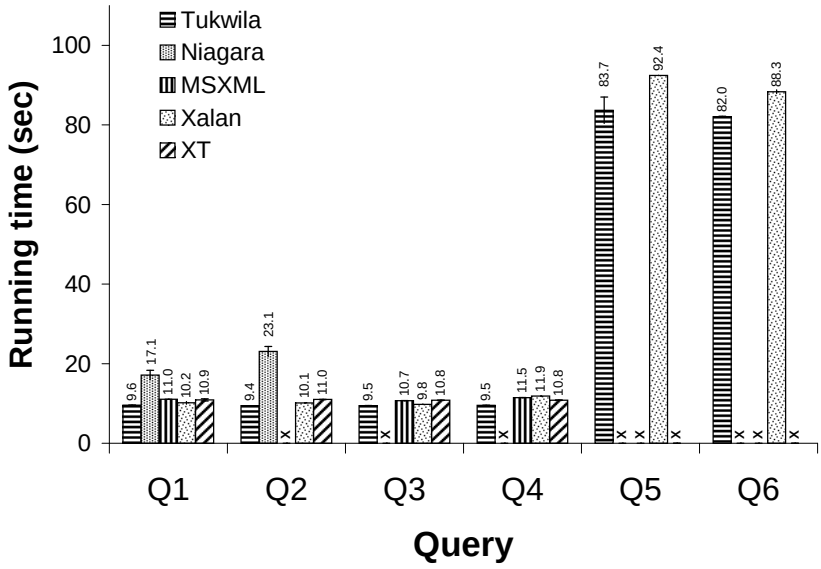
We make several observations about the results. First, although Tukwila was run on a slower machine (800MHz vs. 866MHz) with less memory (256MB vs. 1GB) than all of the other systems, it nearly matched or significantly outperformed all of the other engines documents across the entire suite of queries. Microsoft's MSXML processor lives up to its reputation as being a very fast engine, and it is actually faster by a margin of half a second for the queries over the relatively small `religion` document — we attribute this to the additional overhead Tukwila incurs to optimize its queries. For larger documents, however, such as the XMark document, Tukwila is substantially faster overall, and is especially faster for Query Q7. Q7 clearly demonstrates that Tukwila is the only processor to scale to large XML data files: our system comfortably processed the 324MB `dmoz` XML document on a 256MB machine in less than a quarter the time that MSXML (needing most of the 1GB of RAM in its experimental configuration) did. No other systems were able to accommodate the large document.

Surprisingly, although our suite of queries was relatively simple, some of the queries could not be executed on all systems. Niagara does not support the XML-QL **LIKE** predicate or index variables, so we could not express queries Q3, Q4, and Q5. MSXML executed query Q2 with incorrect results (returning no answers). Several query processors failed with the XMark document (generating what appear to be spurious parse errors), and nearly all failed on the large `dmoz` document (running out of memory even on a 1GB system).

Overall, x-scan's support for pipelined operation over data streams results in much better time to initial tuple (in general returning 5 answers in approximately 2 seconds, except in the cases where there were fewer than 5 answers to be returned), and in fact the incremental processing model improves overall execution time as well. We also observe that the Niagara system, which has largely focused on producing partial answers in order to return early results, can *only* produce those results after it has finished loading and parsing an XML document — Niagara would benefit significantly from the x-scan operator.



(a) First 5 tuples



(b) Total query time

Figure 4.9: In the wide-area context, Tukwila's architecture provides even greater performance improvements when compared to the other systems.

Slow links

Our first experiment measured general query processing performance across a local area network; however, wide-area query processing is one of the focal points of the Tukwila project. Thus our second experiment repeats the previous queries in a bandwidth-constrained environment. We simulated these conditions by artificially adding a 50ms delay to the initial request for a document (representing a slightly longer round-trip time), plus a 15ms delay per 16KB of data sent (limiting the throughput of the connection). This delay was sufficient to inject 960 msec of delay per MB of data transferred, giving us about 1MB per second or 8Mbits per second as our approximate transfer rate. We repeated all of the queries of the previous section except for the `dmoz` query, which we judged to be too huge for anyone to want to transfer in this situation.

Performance results are in Figure 4.9. As expected, Tukwila's incremental output greatly improves the time to initial answers, but the overall query completion time also shows a relative performance gain versus the other query processors. Since Tukwila does filtering and construction of content in parallel with reading, it manages to use the network delay times to help compute answers; in contrast, the other query engines are idle during delays, since they cannot process results until after the parse is complete.

Scale-up

A point of emphasis in our design of the Tukwila architecture has been scalability to large XML documents. While most XML files on the Web are currently only tens of KB in size, as XML matures, querying and integration of data between groups or enterprises is expected to become commonplace — and such data will be considerably larger. In many of these situations, the query processor may be servicing many outstanding requests simultaneously, so each query must run with limited resources. Moreover, current query processors' in-memory representations of XML data are substantially larger than the original XML data — e.g., the XT processor required over 260MB of memory to load and scan the 39MB DBLP XML file in query Q4 of the previous subsection; even a server with 1GB of memory cannot handle many such queries simultaneously.

Tukwila avoids this pitfall by supporting out-of-core execution. Many aspects of the Tukwila architecture (e.g., external sorts, grouping operators, hash and pipelined hash joins) will scale in predictable ways, as they are well-understood components of

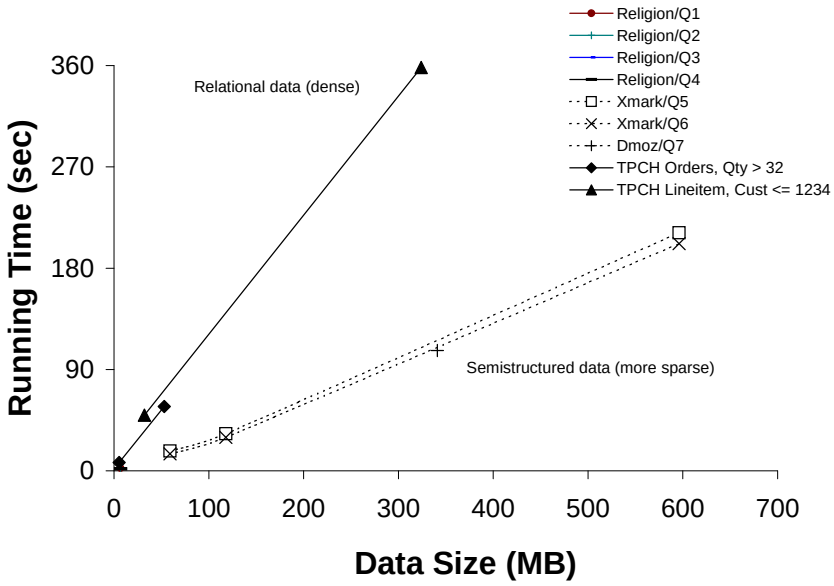


Figure 4.10: An X-Y plot of running times versus data sizes shows that Tukwila yields relatively consistent and linear performance. Note that queries over relational data, which is typically more "dense," result in a higher slope than more sparse semi-structured data.

relational query engines. As observed in Section 4.4, most query operations take place over scalar data values rather than subtrees, and these values are likely to be inlined within the tuple — hence page faults in the XML Tree Manager are not likely to greatly affect performance.

The main concern for scalability, then, is the x-scan operator and the data structures it uses. We investigated the performance of x-scan for both simple path expressions and more complex ones (i.e., those with more bindings and a Kleene-star operator in them), across a variety of document sizes.

We took all of the queries from the previous section, plus two selection queries over relational data and plotted the running times versus the data sizes in Figure 4.10. We note that an interesting dichotomy emerges: the relational tables, which are quite "dense" with many tuples and many XPath matches, seem to yield running times that all fall on the approximately same line at the left of the plot. Likewise, the other queries over sparser semi-structured data seem to follow a different line with a lower slope. As we would hope, Tukwila's performance appearance appears to scale approx-

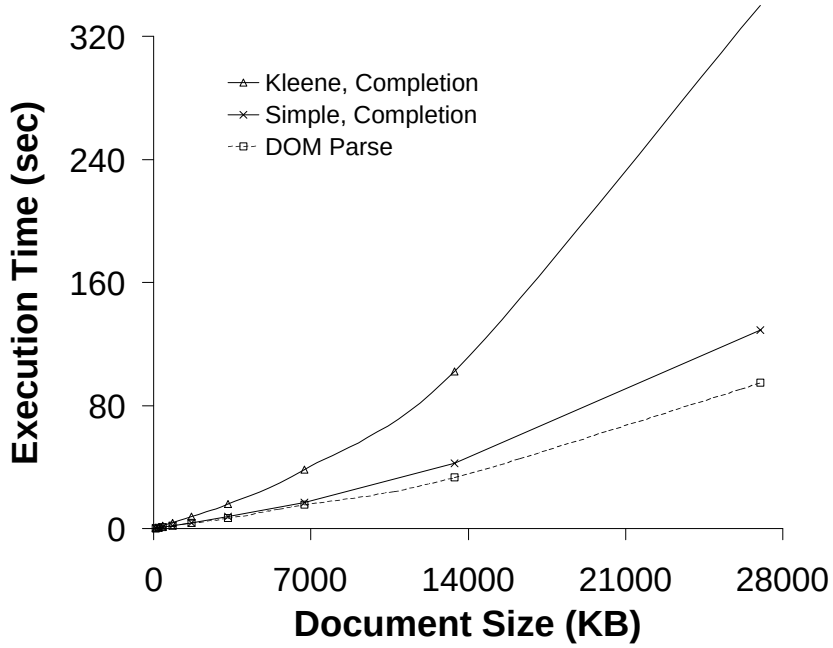


Figure 4.11: Scale-up results for query completion time on synthetic data for simple path query, Kleene-* query, and DOM parse. (Time to first 5 tuples was under 2 seconds.)

imately linearly, with the slope determined by the number of pattern matches that occur.

Figure 4.11 shows performance over a range of synthetic data, generated as described at the beginning of this section. We observe that the time required to process a simple query grows at a rate only slightly faster than it takes to parse the XML and build a DOM tree (the approach taken by previous systems); x-scan state-machine operation and Tree Manager overhead within Tukwila is fairly low. Kleene query execution times grow at a significantly faster rate than the simple query, but this query produces many more tuples because it consists of two sibling Kleene-star path expressions — the cartesian product of these two bindings must be returned for each common subtree. The increase in execution times is closely approximated by the growth rate in the number of tuples produced.

Table 4.5: Queries with database-style selection (Q8-Q9) and join (Q10-Q11) operations using relational data mapped into XML.

Nbr.	Class	Input	Query
Q8	Rel. Sel.	5MB	TPC-H Orders for Customer "1234"
Q9	Rel. Sel	31MB	TPC-H LineItems with Quantity $>$ 32
Q10	Rel. Join	5MB x 0.5MB	Join TPC-H Orders for Customer key $<$ "1234" with all Customers
Q11	Rel. Join	31MB x 7MB	Join TPC-H LineItems with Orders

4.6.2 Database-Style Operations

One of the major sources of data is, of course, relational databases, and there is significant interest in sharing relational data in the XML format. An important concern is the amount of overhead incurred by "adding XML into the loop." Do we lose a great deal by querying over an XML view, rather than over traditional relational data? To answer this question, we compared three different means of processing selection and join queries:

- **XMLified SQL**, where we sent a SQL selection or join query to a database at the server (DB2 UDB 6.1 running on a 450MHz web server), read the data via Java JDBC and sent it as tuples across the network to our mediator, which added XML tags around the tuples and returned the results to our client. The relational database was fully indexed. This approach is similar to those adopted by the SilkRoute [FTS99] and XPERANTO [CFI+00] mediator layers, which wrap an XML view interface over relational systems, except that we do not translate queries.

- **Relational Mediator**, in which all tuples from the tables were simply read from JDBC and returned to the original Tukwila system, which executed a relational query and then converted the data to tagged XML.

- **Tukwila**, which took materialized XML views of the relational tables, read them via HTTP, and did XML query processing over the data using the techniques described in this paper.

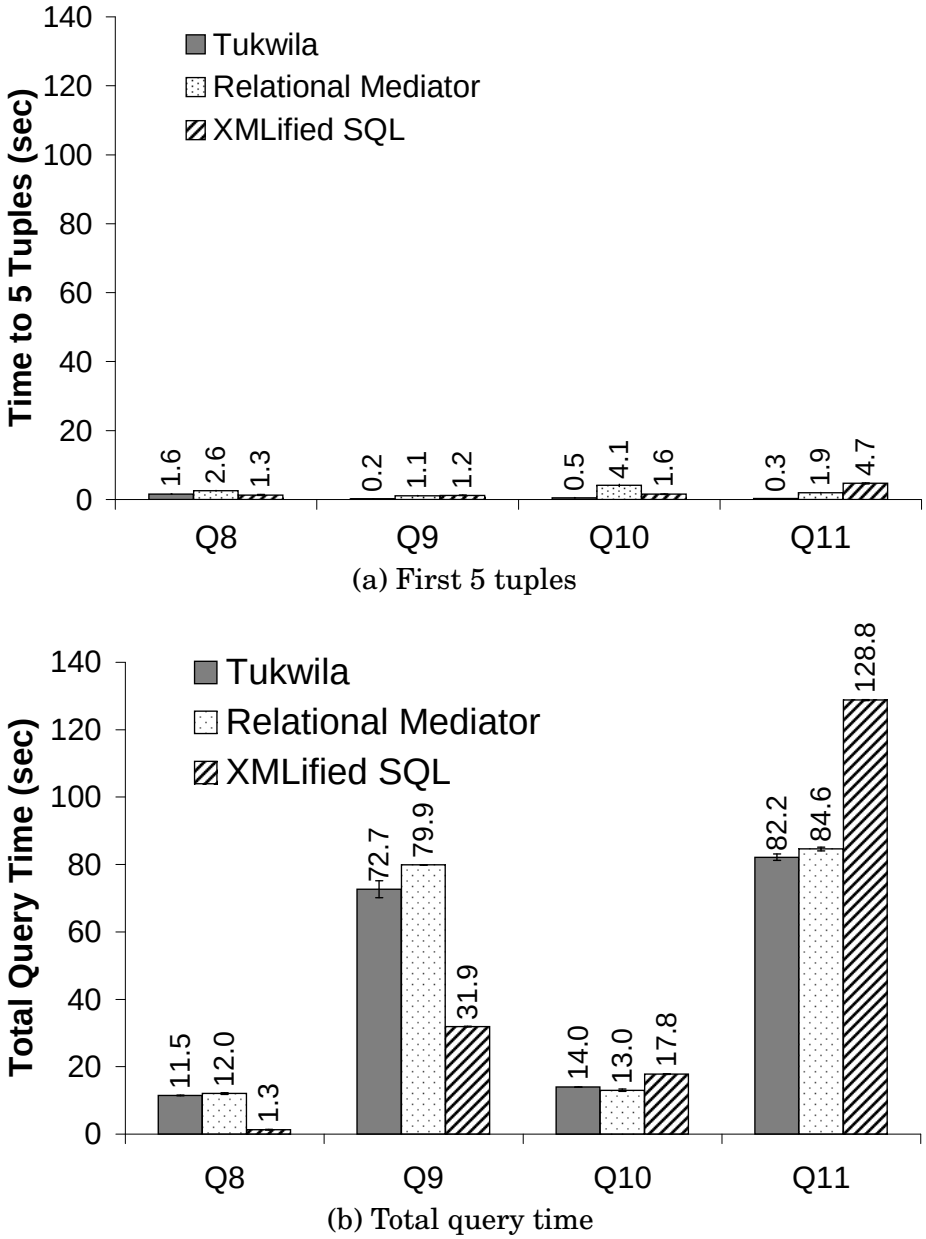
As Figure 4.12 illustrates, the Tukwila and Relational Mediator approaches tended to have very comparable running times, despite the fact that the XML-ified input tables were considerably larger. Moreover, the overhead inherent in JDBC and Java socket I/O (even given the fast 100Mbps network) appear to be more substantial than we had anticipated, so processing the query at the server was not necessarily a win. As expected, selection queries are significantly faster when done within the database engine. However, both join queries execute more slowly when done inside the relational engine. We attribute this to the fact that JDBC was a bottleneck in our experiments and the join results were larger than the sum of the combined inputs — as a result, it was more efficient to read the original tables separately and join them within the mediator. Likewise, it was essentially as efficient to read and process the XML version of the data as it was to read the data through JDBC. We conclude that the choice of whether to push an operation into a data source depends greatly on the communication-link costs, even when we are choosing between querying data in its original relational form or converting it to XML first.

While we do not claim that JDBC is the fastest method for exchanging relational data (and we acknowledge that many modern databases provide other means of exporting XML), we observe that its performance is acceptable for many business and scientific applications. Since Tukwila performs similarly for the equivalent queries, we believe that x-scan-based XML data exchange also provides sufficient performance for real-world applications. Moreover, the Tukwila XML-based engine provides greater interoperability because it can combine relational and non-relational data.

Nesting Data

As we observed in Section 4.2.2, the operation of hierarchically nesting XML child elements within a parent element is very similar to a left outer join in relational databases. However, a nesting operation has an important constraint, which is that the elements must appear contiguously, clustered by parent. Clearly, maintaining this grouping incurs some overhead, and we wanted to examine how significant this was.

A general practice in query optimization, especially for network-based data, is to use the *smaller* join relation as the inner relation, and the larger as the outer relation. Not only does this reduce memory overhead in algorithms such as the hash join, but it also produces initial results earlier (assuming roughly equivalent transfer rates be-



(a) First 5 tuples

(b) Total query time

Figure 4.12: Experimental comparison of relational queries shows that Tukwila performs nearly as well over data mapped into XML as the comparable relational-model integration system. In-SQL execution, included for comparison only, was better for the selection query but not for the joins.

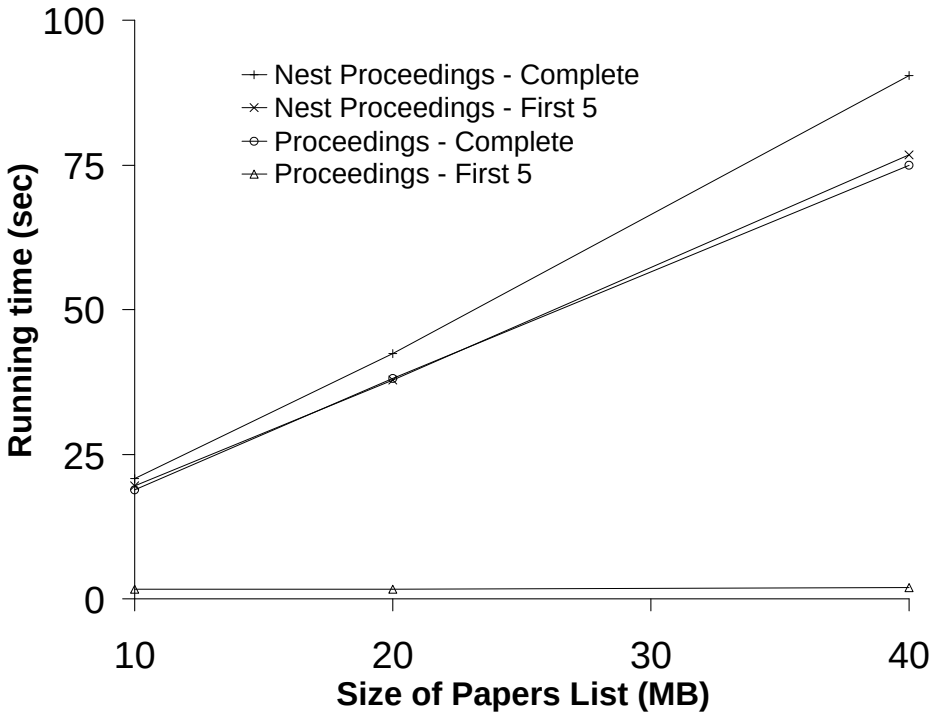


Figure 4.13: Comparison of nest and join operations combining DBLP papers and proceedings. Nest requires the (larger) inner relation to be read first, thus it has much longer time-to-first-results and slower overall time than the optimal join.

tween sources) because the hash join must block until it has finished reading its inner relation. Unfortunately, since a **nest** operation is used to create a $1 : n$ hierarchical relationship, it must place the *larger* join relation as the inner relation so it can iterate over it for each parent tuple. We can see in Figure 4.13 that as a result, **nest** performs more slowly than a hash join that has been commuted to the opposite configuration. In fact, the hash join completes its execution in the same amount of time as **nest** takes to output the first 5 tuples.

This suggests that performance in interactive applications, where first answers are most important, would be considerably improved if it were possible to do the **nest** the same manner as the join, i.e., if we did not have to maintain the parent-based ordering constraints on its output tuples. However, if we output results without preserving order, we must ultimately sort the data to get it into its proper form. We are exper-

imenting with a user interface in which the final sort operation is performed at the client-side on a periodic basis, which frees the query processor to stream out results in any order and provide faster feedback to the user.

4.6.3 Supporting Graph-Structured Data

Although most of today's XML queries traverse the document as a tree, there are many potentially interesting uses of XML as a representation for semistructured graphs, encoding edges as both elements and `IDREF`s. Thus, the x-scan operator has a number of features designed for querying graphs. Previous work on `IDREF` traversal has typically been done using the join or **follow** approaches described in Section 4.5, but we now examine the use of x-scan as an alternative.

X-scan Traversal of `IDREF`*s*

In our comparison of strategies for evaluating graph-style references, we suggested that x-scan could be used on moderately sized documents that had low numbers of references. In Figure 4.14, we see execution times of x-scan across synthetic documents of different sizes. The different lines represent execution times when the ratio of `IDREF`s to elements is 1:8, 1:4, 1:3, and for comparison we include the execution time for a typical tree-traversing query, which does not build the structural index, over the mid-sized (1:4-ratio) documents. For proportionately low numbers of references, we see that the overhead in supporting graphs is relatively low; and even with fairly high numbers of traversed `IDREF`s, running times are reasonable, especially since initial results are output quickly. With a 1:3 ratio of `IDREF`s to elements, Tukwila takes 90 seconds to return 193,000 leaf nodes from a 7MB synthetic graph. In contrast, the tree version of the same query yields only 55,000 leaf nodes. As the ratio of `IDREF`s gets even higher — not shown in the graph — the XML graph begins to approach full connectivity, and x-scan spends large amounts of time doing repeated evaluations. Clearly, in these situations, the join- or follow- based approach is more appropriate.

Graph Traversal with Limited Memory

We also examined in detail the performance characteristics of x-scan, particularly those related to paging data to disk. For simple tree-based queries, memory constraints are typically not an issue — Tukwila needs only to maintain state and sub-

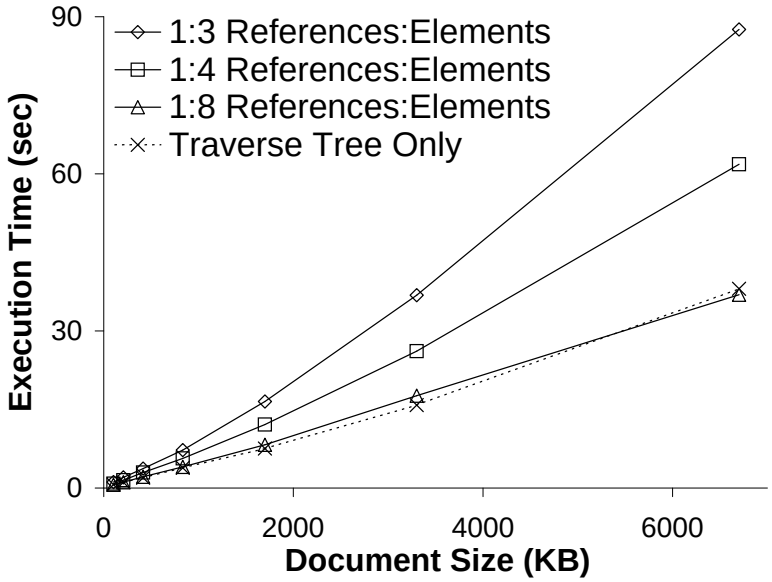


Figure 4.14: Scale-up results for Kleene-* graph query on synthetic data, with tree query shown for reference.

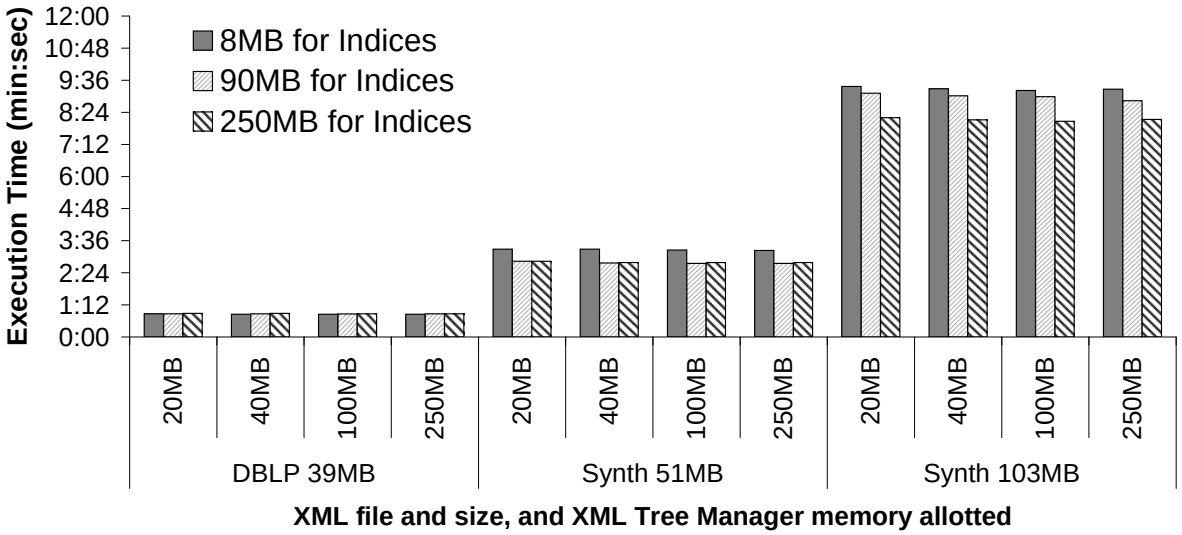


Figure 4.15: Query processing times with restrictions on XML tree memory (x-axis) and index buffer memory (bar shades). Index buffer size impacts performance more than tree memory.

trees for a limited amount of time, i.e. until all tuples referencing the subtrees have passed through the pipeline. Thus, for example, when we queried the the 159MB Open Directory Project topic hierarchy for all topic aliases, query processing times were approximately 7 minutes 43 seconds whether Tukwila was given 20MB of memory or 250MB. Results were similar for tree-style queries over other data sets such as DBLP. Additional experiments demonstrated that the performance bottleneck was clearly in the areas of network I/O and parsing — saving a locally cached copy of the input XML document to disk (from a separate thread) added no perceptible time overhead to the query.

Our final experiment, in Figure 4.15 measures the performance of x-scan graph traversal across large XML data files when the amount of memory available to the Tree Manager and the structural index are constrained. Data sets on the graph include two synthetic data sets of 103MB and 51MB, each with a 1:8 element-to-`IDREF` ratio, and the DBLP conference data set with cross-references from papers to conferences as `IDREF`s[8]. Our experiments do include a data set in which most of the referenced items are relatively clustered (DBLP) and one in which they are randomly distributed throughout the document (the synthetic data). In all cases, the structural index ranged in size from two to three times the data set size. We separately adjusted the size of the index's memory allocation and the Tree Manager's allocation, to see how greatly each affected performance. In general, the variations in memory had less of an impact than one might expect — we attribute this to the fact that the query processor is generally network-bound, and hence can make use of free CPU and disk cycles. Moreover, as expected, the size of the index buffer affects performance more than the size of the Tree Manager. A final observation is that, as expected, the DBLP data set, with a fairly strong locality of references, is basically not impacted by memory, whereas the synthetic data with its randomized reference targets is somewhat more affected.

4.7 Conclusions

In order to build an effective data integration system, we need the ability to query remote data sources whose content is sent across the network in XML. This requires

[8]We also attempted to use the Open Directory data file, but were unable to successfully "clean" the document by removing elements unacceptable to the Xerces parser, while still maintaining `IDREF` link integrity.

the following capabilities:

- the ability to query, combine, and restructure the content of XML documents of arbitrary size,
- the ability to combine data from multiple sources, including data that is the result of dynamically computed queries
- support for a "streaming" or pipelined query processing model that produces results as soon as possible.

In this chapter, I have described the Tukwila XML query engine, which meets the above requirements with the following features:

- an architecture which extends tuple-oriented, relational techniques such as pipelining, as well as recently developed adaptive query processing techniques for network-based relational data, to work efficiently on XML;
- two key operators, x-scan and web-join, that map XML data (from both static and dynamically queried sources) into tuples in a streaming fashion;
- and a set of basic operators for combining and restructuring tuples of XML subtrees into new XML content.

In subsequent chapters, I discuss techniques for adaptive scheduling and query optimization, and make use of this basic execution system as the experimental platform. Note that adaptivity is especially important in the XML context because not only are statistics generally unavailable in data integration, but the database community does not have good cost models and statistical models for XML query processing.

Chapter 5

EXECUTION SUPPORT FOR ADAPTIVITY

In the previous chapter, I presented a basic architecture for processing network-bound XML data in a pipelined fashion. Pipelining is an important capability in a network-based query engine, because it allows the query engine to process data much earlier, and allows the engine to be running multiple tasks in parallel. Moreover, Tukwila's pipelined architecture allows us to leverage existing techniques from the relational database literature.

However, pipelining of XML is not itself sufficient to provide good performance for data integration query processing when coupled with a standard database query engine. Standard relational engines use deterministically scheduled query operator algorithms and an iterator architecture to execute the query plan specified by a static query optimizer. There are three reasons why this approach is inappropriate for data integration:

- **Absence of statistics:** statistics about the data (e.g., cardinalities, histograms) are central to a query optimizer's cost estimates for query execution plans. Since data integration systems manipulate data from autonomous external sources, the system has relatively *few* and often *unreliable* statistics about the data. As a result, a query optimizer that selects a plan before runtime may choose a plan that is poorly scheduled or sub-optimal.

- **Unpredictable data arrival characteristics:** unlike traditional systems, data integration systems have little knowledge about the rate of data arrival from the sources. Two phenomena that occur frequently in practice are significant initial delays before data starts arriving, and bursty arrivals of data thereafter. Hence, even if the query optimizer is able to determine the best plan based on total work, the data arrival characteristics may cause the query plan and the iterator-based execution model to be inefficient in practice [UFA98].

- **Overlap and redundancy among sources:** as a result of the heterogeneity of

the data sources, there is often significant overlap or redundancy among them. Hence, the query processor needs to be able to efficiently collect related data from multiple sources, minimize the access to redundant sources, and respond flexibly when some sources are unavailable. These capabilities are not supported in standard database systems, which have a fixed, generally small number of tables without overlap or the expectation of failure.

These problems can only be solved by using *adaptive* query processing techniques. Adaptive optimization techniques can replace a poor query plan in mid-execution, and they can be combined with adaptive execution techniques that can change query plan scheduling — which also addresses the second problem listed above. Adaptive techniques can also be used to make use of overlapping and redundant data sources: they can switch from one set of sources to an alternate set depending on runtime conditions.

In this chapter, I focus on the problem of **execution engine support for adaptivity**. This consists of two components: (1) adaptivity within the execution engine itself, which may include rescheduling of work or reallocation of resources; and (2) mechanisms for supporting adaptive query optimization, including infrastructure for monitoring costs and statistical information, as well as mechanisms by which an optimizer can replace a running query plan. Specific features described in this chapter and implemented in Tukwila include:

- **Event handling:** Many types of exceptions and other events (e.g., source failure, memory overflow) may occur during query execution, and an event handler is a means of responding to these events and adapting to them. Responses may include invoking the optimizer, altering memory allocations, enabling or disabling query operators, or even modifying operator behavior.

- **Interleaving planning and execution:** Since the initial query plan may be quickly discovered to be sub-optimal, a useful capability is execution engine support for beginning execution, monitoring performance, and incrementally re-optimizing the plan as cost and statistical information is learned. In Chapter 6, I describe techniques for re-optimizing queries in mid-pipeline. In this chapter, I describe how the query execution engine is built to support such re-optimizations, as well as re-optimization between pipeline stages.

- **Cost and statistics monitoring:** Tukwila query operators include instrumentation that allows the optimizer to determine the cost of each operation and the cardinalities of each operator's inputs and outputs.

- **Adaptive operators:** Tukwila incorporates operators that are especially well suited for adaptive execution and for minimizing the time required to obtain the first answers to a query. Specifically, it employs an enhanced version of the pipelined hash join [RS86, WA91] (a join implementation which executes in a symmetric, data-driven manner, also called the pipelined hash join) and techniques for adapting its execution when there is insufficient memory. In addition, the Tukwila execution engine includes a *collector* operator whose task is to efficiently union data from a large set of possibly overlapping or redundant sources. Finally, Tukwila query execution plans can contain conditional nodes in the spirit of choose nodes [CG94] in order to adapt to conditions that can be anticipated at optimization time.

Adaptive behavior in Tukwila is coordinated in a uniform fashion by a set of event-condition-action rules. An event may be raised by the execution of operators (e.g., out of memory, data source not responding) or at materialization points in the plan. The possible actions include modifying operator execution, reordering of operators, or re-optimization. The system includes default rules for handling events, and the query optimizer may also include custom rules.

Example

A simple example demonstrates the breadth of Tukwila's adaptive behavior. Suppose that the same query (Figure 5.1a) is issued to the system under three extreme conditions: when the source tables are of unknown size, are small, or are large. Each time, assume that the relative statistics are such that a traditional optimizer would construct the join tree in Figure 5.1b. In a traditional query engine, the join implementations, memory allocations, and materialization points will be fixed at compile time, and the tree will be executed in a predetermined order. Tukwila implements mechanisms needed to behave more adaptively. Consider its response to the three cases:

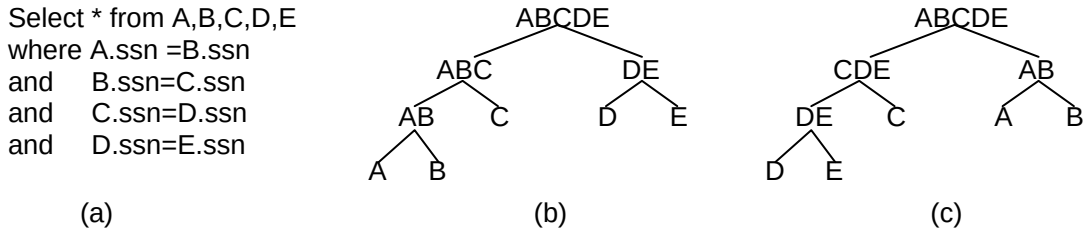


Figure 5.1: Sample query, initial join tree, and join tree produced by re-optimization.

No size information: With no information there is no point in traditional optimization. The optimizer picks a query plan randomly and begin executing it. As cost information becomes available from execution, it may refine the plan in mid-pipeline.

Small tables: Tukwila chooses the pipelined hash join (double pipelined join) implementation for joins of small cardinality, and pipelines the entire query. When source latencies are high, this type of join has a large advantage over traditional joins, but it demands considerably more memory. To handle the "unlucky" case that memory is exceeded, the join operator has an overflow resolution mechanism.

Large tables: Since early initial answers are often desirable, Tukwila's optimizer may build a single pipelined query plan, using pipelined hash joins. However, at some point the plan may run out of memory. Now, Tukwila may either rely on the overflow resolution mechanisms of the join operator, or it may adaptively change the query plan. Two possible changes are to replace the pipelined hash joins with other join algorithms or to break the plan into multiple separate pipelines. In either case, many answers will already have been returned.

Alternatively, Tukwila's optimizer may choose standard hash joins, and it may break the pipeline, perhaps after join AB in Figure 5.1b. Now, depending on the rules in force, one of two things may happen during execution:

- **Rescheduling:** If all sources respond, and table AB has a cardinality sufficiently close to the optimizer's estimate, execution continues normally. Should some

sources respond slowly, however, Tukwila can reschedule as with query scrambling [UFA98]. If the connection to data source A times out, join DE will be executed preemptively. Should that time out as well, the optimizer is called with that information to produce a plan reordered to use the non-blocked sources first.

- **Re-optimization at materialization points:** After the AB join completes and materializes, Tukwila compares the actual cardinality with the optimizer's estimate. As in [KD98], if this value significantly differs from the optimizer's estimate, the optimizer is awakened to find a cheaper plan (perhaps the one in Figure 5.1c) given more accurate information.

- **Re-optimization in mid-pipeline:** As the AB join begins executing, suppose it becomes apparent that the actual selectivity of the operation is substantially different from the optimizer's estimate. The optimizer can suspend that plan and replace it with the one in Figure 5.1c and begin executing that plan over the portions of the data sources *not* processed by the initial plan. For mid-pipeline changes, a further, *cleanup plan* must also be executed to provide complete answers; details are discussed in Chapter 6.

This chapter is organized as follows. Section 5.1 presents our query execution architecture and its mechanisms, as well as its interactions with the query optimizer. Section 5.2 describes the new query operator algorithms used in Tukwila. Section 5.3 describes our experimental results. Section 5.4 concludes the chapter.

5.1 An Adaptive Execution Architecture

The goal of the Tukwila execution engine is to be adaptive and flexible without significantly compromising performance. In this section, I describe major features of the engine, focusing on the basic query execution model, event handling, and support for inter-pipeline and intra-pipeline adaptivity.

5.1.1 Execution Model

The basic Tukwila execution model follows the standard *iterator* approach of [Gra93], in which each query operator in the plan passes control to its children in a deterministic fashion, calling the `next()` function to read a tuple. The iterator-based execution

model is a low-overhead means of supporting pipelining and processing of data in many situations.

For increased flexibility, however, Tukwila also includes support for multithreading within a query plan. Operators such as the pipelined hash join (discussed in Section 5.2) can make use of multiple threads to perform several tasks "in parallel" and to mask I/O delays with alternate computation. Multithreaded operators are based on a *producer-consumer* model: the child operator runs in a separate thread from its parent, and the child writes into a queue from which the parent consumes data.

Wherever possible, Tukwila makes use of a "synchronized pipeline" model: we attempt to re-use the same tuple memory space between operators where possible, simply passing the same data structure from one operator to the next. For instance, a selection operator simply passes through tuples that satisfy its predicates — there is no need to copy the data. Similarly, a projection operator reduces the number of attributes in a tuple, but does not modify their values, so it can simply change the list of available attributes, without modifying the tuple data.

Finally, another notable feature of the Tukwila execution system is that all query operators monitor the cardinality of their output and the CPU time consumed by the operator (obtained by consulting the clock before and after key operations are performed). These monitoring operations allow the query optimizer to determine costs and selectivities for query operators. For additional statistical information, histogram-generating operators could be inserted into a query plan (although these would incur additional memory and compute overhead).

5.1.2 Event Handling for Controlling Adaptivity

When a particular state transition is made or an exception occurs, an event notification is generated by the Tukwila execution system. These events are fed into an event queue, which imposes an ordering on the rule evaluation process. Events are processed in FIFO order by either a *default handler* supplied by the system or by an optimizer-specified event-condition-action rule. For each active rule, Tukwila evaluates the conditions; if they are satisfied, all of the rule's actions are executed before the next event in the queue is processed. Actions supported by the rule system include:

- **Re-optimization:** Re-optimization can be triggered in the middle of query execution, e.g., if memory overflow occurs, or at the end of a pipeline, if the optimizer's

cardinality estimate for the last stage is significantly different from the actual size. (This second case is in the same spirit as [KD98]).

- **Contingent planning:** At the end of a pipeline stage, the execution engine can check properties of the result in order to select the next execution step from among several alternatives (thus implementing choose nodes [CG94]).

- **Altering operator functionality:** The policy for memory overflow resolution in the double pipelined join (Section 5.2.2) is guided by a rule. Collectors (Section 5.2.1) are also implemented using rules.

- **Rescheduling:** Rules are used for specifying when a plan should be rescheduled if a source times out (as in query scrambling [UFA98]).

Tukwila rules have the form **when** *event* **if** *condition* **then** *actions*. For example, the following rule calls the optimizer to replan the subsequent fragments if the estimated cardinality is significantly different from the size of the result.

```
when closed(frag1)
if card(join1) > 2 * est_card(join1) then replan
```

Formally, a rule in a Tukwila plan is a quintuple $\langle event, condition, actions, owner, is\_active \rangle$. An event can *trigger* a rule. If the rule's condition is true, it *fires*, executing the actions. The *owner* is the query operator or plan fragment that the rule controls or monitors. Only active rules with active owners may trigger. Firing a rule once deactivates it by default.

The execution system generates events in response to important changes in the execution state, such as:

Event Name	*Function*
`state_becomes(open|closed|error)`	operator state changes
`end_of_fragment`	fragment completes
`wrapper_timeout(`n`)`	data source has not responded in n msec.
`out_of_memory`	join has insufficient memory
`every_n_tuples(`n`)`	n tuples processed by operator
`every_n_ms(`n`)`	n msec passed

Once an event has triggered a set of associated rules, each rule's *condition*s are evaluated in parallel to determine whether any actions should be taken. Conditions are propositional formulas, with comparator terms as propositions. The quantities that can be compared include integer and state constants, states, values precomputed by the optimizer (e.g., estimated cardinality or memory allocated), and various dynamic quantities in the system:

Function Name	*Returns*
`global_memory`()	memory available
`state`(*operator*)	operator's current state
`card`(*operator*)	number of tuples produced so far
`estimated_card`(*operator*)	optimizer's cardinality estimate
`memory_allocated`(*operator*)	memory given to operator
`memory_used`(*operator*)	actual memory consumed by operator
`time_since_last_tuple`(*operator*)	time waiting since last tuple
`timeout`(*operator*)	number of timeouts that have occurred
`next_child_index`(*collector*)	index of next alternative source in collector (see Section 5.2)
`num_open_children`(*collector*)	count of open children in collector operator

After all rule conditions corresponding to a given event have been evaluated, *actions* are executed for those rules whose conditions are met. Most actions change some operator's memory allocation, implementation, or state. Tukwila actions include:

- set the overflow method for a double pipelined join
- alter a memory allotment
- deactivate an operator or fragment, which stops its execution and deactivates its associated rules
- reschedule the query operator tree
- re-optimize the plan
- return an error to the user
- activate or deactivate a data source in a collector

Naturally, the power of the rule language makes it possible to have conflicting or non-terminating rules. It is ultimately the responsibility of the optimizer to avoid generating such rules. However, in order to avoid the most common errors we impose a number of restrictions on rule semantics: (1) All of a rule's actions must be executed before another event is processed. (2) Rules with inactive owners are themselves inactive. (3) No two rules may ever be active such that one rule negates the effect of the other and both rules can be fired simultaneously. (This final aspect is a condition that can be statically checked.)

Perhaps the most common uses of rules will be to trigger changes to the query plan by re-invoking the optimizer (i.e., interleaving planning and execution). We describe two possible forms of this in the next two sections.

5.1.3 Support for Inter-Pipeline Changes

The simplest form of interleaving planning and execution occurs during a materialization point between two query execution pipelines: the first pipeline is processed and its results are materialized, and then the remaining pipeline is re-optimized, using the results from the first pipeline. This particular approach was first proposed by Kabra and DeWitt in [KD98], and is especially useful for situations in which the query optimizer has good statistics on the data sources, but optimizer error increases significantly as more and more operations are performed. For that situation, the initial pipeline would be close to optimal and would contain only a few operations. Once it completed, the next stage could be re-optimized with the actual statistics from the data output by the first stage.

Early versions of the Tukwila optimizer focused on the problem of breaking the query into appropriate stages at optimization time, adding rules to tell the execution system when it needed to re-optimize, and how to preserve and reuse optimizer state. Details of this version of the optimizer are described in our SIGMOD 99 paper [IFF+99] and in Marc Friedman's dissertation [Fri99].

Key features in the execution engine to support this form of interleaved planning and execution are as follows. First, the execution system needs support from the rule system to determine when to re-invoke the optimizer. The rules must be able to examine the cardinality of a materialized result, as well as other optional statistics such as selectivities, and must be able to re-invoke the optimizer as a result. The execution

system must maintain the materialized results while the optimizer is being re-invoked, then it must use these results (plus any unprocessed sources) as the inputs to the new query plan. Tukwila supports all of these features, and later in this chapter we see experimental results showing that this form of interleaving planning and execution can produce significant performance benefits.

However, inter-pipeline changes are limited in their applicability, because it is highly inefficient to materialize frequently, and it is difficult to determine where to place materialization points in a query plan. Thus our focus has shifted to supporting *intra-pipeline* re-optimization.

5.1.4 Support for Intra-Pipeline Changes

The techniques used for mid-pipeline query re-optimization are discussed in Chapter 6, but they require substantial support from the query execution engine. Key features required of the execution system are a means of modifying or replacing the executing current plan, capabilities for directly accessing the (hash or list) data structures that hold tables within join and other operators, and the ability to trigger mid-pipeline re-optimization (e.g., when the current plan runs out of memory).

When the query optimizer wishes to replace or modify the current query plan, it sends a "suspend" message to the execution engine. This message causes the execution engine to stop consuming input and compute the effects of the input read so far, emptying all queues within the operators of the pipeline. In the Tukwila implementation, a query plan can be replaced only at a "consistent" point during execution. Informally, the plan is consistent if each pipeline has returned *all* the answers for the portion of the data consumed by the plan — the "effects" of every input tuple are computed, and no tuples are queued up within the pipeline. "Query plan consistency" can be more formally defined as follows:

Definition 5.1.1 (Query Plan Consistency) Let $\mathcal{Q}(R_1, ...R_n)$ represent a query plan tree with input relations $R_1...R_n$. Let $Q_i(R_i, ..., R_j), 1 \leq i \leq j$, be a pipelined subtree within $\mathcal{Q}(R_1, ..., R_n)$, combining relations $R_i, ..., R_j$. Let $R_{i\lfloor_x^y}$ be the subset of R_i read between start time x and end time y.

A plan is *consistent between times* t_0 *and* t_1 if every pipelined subtree $Q_i(R_i, ..., R_j)$ executed between times t_0 and t_1 produces all answers in the expression $Q_i(R_{i\lfloor_{t0}^{t1}}, ..., R_{j\lfloor_{t0}^{t1}})$.
□

An example of an inconsistent query plan is one with a sort-merge join that has buffered tuples but not yet sorted them and produced output; it will become consistent once it percolates the "suspend" message's effects. In the eddy algorithm of [AH00], there is a conceptually similar notion of *moments of symmetry* that restrict when the eddy can adapt the query plan; however, they are considerably more restricted in when they can make changes. A join in a standard eddy can only be interrupted and commuted when it has completed its innermost loop. The pipelined hash join has a moment of symmetry every time it reads a tuple and finishes probing the opposite relation; but the hybrid hash join has a moment of symmetry only after the build relation has been consumed, and then one occurs at the point another probe tuple is about to be read. Note that as a result, an eddy cannot change a hybrid hash join until it has consumed the entire build relation — whereas Tukwila can send a suspend message at any point, and it only needs to wait for the effects to be computed for the portion of the relation that has been read.

As soon as the executing plan is consistent within Tukwila's execution system, the query optimizer can selectively replace operators in the query plan — it generally never replaces the leaf-level x-scan operators (these are simply attached to the next plan so input can resume where the last plan left off), nor the topmost aggregation or XML structuring operators (these can continue reading results from the new query plan).

Once the query plan has been replaced, an additional operation must be performed. Since tuple memory is often shared between operators (as discussed before), but the new query plan consists of some new and some old operators, a "re-bind" operation must be performed to re-link the memory spaces along the pipeline. Now execution may resume using the modified plan. Generally, the state from the old join operators is maintained until full query execution has completed, because the joins data structures may be needed in a "cleanup" plan (as discussed in Chapter 6). Their state is accessed through a hash table keyed on the "signatures" (subexpressions) represented by each join's subtree.

The general case, described above, is that the query optimizer runs in the background and invokes a plan replacement operation when it finds a better query plan. However, sometimes the converse can occur: re-optimization operation can be triggered by the execution engine. Typically, this happens because an event is triggered

(e.g., a source fails or memory overflows).

The memory overflow problem actually requires special care. In particular, if a query operator runs out of memory, this typically means that the query plan is *not* stable: the effects of an input tuple have percolated up through the query plan, but have not fully been processed by the time the system runs out of memory. Furthermore, there are insufficient resources to get into a stable state. Here, we adopt the opposite approach: in order to reach a stable state, we *unwind* the effects of the last tuple consumed, re-queueing it at the leaf node and removing all of its effects in the intermediate-node query operators.

In order to achieve this, every query operator that maintains state has a *commit* and an *abort* operation. An operator *commits* its state each time a new tuple is read by a leaf-level operator (which is the most recent state we can guarantee the ability to reach stability). An operator *aborts* by restoring its state to the last commit point. The abort message is issued by the operator that runs out of resources to the child operator that returned the most recent tuple, and this process is repeated down to the leaf of the tree (the leaf will return the most recent tuple the next time it is consulted for data). Once the stable state has been reached, the out-of-memory event is finally generated and ultimately handled by the rule system.

Together, this combination of features allows both optimizer-initiated and execution engine-initiated plan changes even in the midst of executing a query — the result is a highly flexible architecture in which many forms of adaptivity can be explored.

Now that I have presented our basic Tukwila query execution architecture and the infrastructure for supporting event-driven adaptivity, I now focus on adaptive features within the query operator algorithms themselves.

5.2 Adaptive Query Operators

Tukwila plans include the standard relational query operators: join (including dependent join), selection, projection, union and table scan. In this section, I highlight Tukwila's adaptive operators: the *dynamic collector* and the *pipelined hash join* or *double pipelined join* operator.

```
when opened(coll1)
     if true then activate(coll1,A); activate(coll1,B)
when threshold(A,10)
     if true then deactivate(coll1,B)
when threshold(B,10)
     if true then deactivate(coll1,A)
when timeout(A)
     if true then activate(coll1,C); deactivate(coll1, B);
     deactivate(coll1, A)
```

Figure 5.2: Rules specifying a collector policy: initially, sources A and B are active. When the first of these operators returns 10 tuples, the other is switched off. If source A times out, source B is disabled and source C is activated.

5.2.1 Dynamic collectors

A common task in data integration is to perform a union over a large number of overlapping sources [YPAGM98, FKL97]. Common examples of such sources include those providing bibliographic references, movie reviews and product information. In some cases different sites are deliberately created as mirrors.

For these reasons, we expect the Tukwila query reformulator to output queries using disjunction at the leaves. We could potentially express these disjunctions as unions over the data sources. However, a standard union operator simply attempts to gather all information from all sources, which is not the best semantics if the important data can be obtained using only a subset of the sources. The standard union also has no mechanism for handling errors. In general, it simply does not provide the flexibility needed in the data integration context. In Tukwila we treat this task as a primitive operator into which we can program a policy to guide the access to the sources.

An optimizer that has estimates of the overlap relationships between sources can provide guidance about the order in which data sources should be accessed, and potential fallback sources to use when a particular source is unavailable or slow (as in [FKL97]). This guidance is given in the form of a *policy*. The query execution engine implements the policy by contacting data sources in parallel, monitoring the state of each connection, and adding or dropping connections as required by error and latency conditions. A key aspect distinguishing the collector operator from a standard union is flexibility to contact only some of the sources.

Formally, a collector operator includes a set of children (wrapper calls or table scans of cached or local data) and a policy for contacting them. A policy is a set of triples $\langle i, a_i, t_i \rangle$, associating with the ith child of the collector an activation condition a_i and a termination condition t_i. The conditions are propositional boolean formulas constructed from `true`, `false`, `and`, `or`, and four kinds of predicates on children: `closed`(c), `error`(c), `timeout`(c) and `threshold`(c). The policy is actually expressed in Tukwila as a set of event-condition-action rules, which are implemented using the normal rule-execution mechanisms.

In the example of Figure 5.2, we have a fairly complex policy. Initially we attempt to contact sources A and B. Whichever source sends 10 tuples earliest "wins" and "kills" the other source. (Note that we take advantage of the fact that a rule owned by a deactivated node has no effect.) If Source A times out before Source B has sent 10 tuples, Source C is activated and the other sources are deactivated.

5.2.2 Pipelined Hash (Double Pipelined) Join

Conventional join algorithms have characteristics undesirable in a data integration system. For example, sort-merge joins (except with pre-sorted data) and indexed joins cannot be pipelined, since they require an initial sorting or indexing step in this context. Even the pipelined join methods — nested loops join and hash join — have a flaw in that they follow an asymmetric execution model: one of the two join relations is classified as the "inner" relation, and the other as the "outer" relation. For a nested loops join, each tuple from the outer relation is probed against the entire inner relation; we must wait for the entire inner table to be transmitted initially before pipelining begins. Likewise, for the hash join, we must load the entire inner relation into a hash table before we can pipeline.

We now contrast these models with the double pipelined join (also known as the pipelined hash join), which was originally proposed in [WA91] for parallel database systems.

Conventional Hash Join

As was previously mentioned, in a standard hash join, the database system creates a hash table from the inner relation, keyed by the join attributes of the operation. Then one tuple at a time is read from the outer relation and is used to probe the hash table;

all matching tuples will be joined with the current tuple and returned [Gra93]. If the entire inner relation fits into memory, hash join requires only as many I/O operations as are required to load both relations. If the inner relation is too large, however, the data must be partitioned into smaller units that are small enough to fit into memory. Common strategies such as recursive hashing and hybrid hashing use overflow resolution, waiting until memory runs out before breaking down the relations.

In recursive hashing, if the inner relation is too large, the relation is partitioned along bucket boundaries that are written to separate files. The outer relation is then read and partitioned along the same boundaries. Now the hash join procedure is recursively performed on matching pairs of overflow files.

Hybrid hashing [Gra93] uses a similar mechanism, but takes a "lazy" approach to creating overflow files: each time the operation runs out of memory, only a subset of the hash buckets are written to disk. After the entire inner relation is scanned, some buckets will probably remain in memory. Now, when the outer relation is read, tuples in those buckets are immediately processed; the others are swapped out to be joined with the overflow files. Naturally, hybrid hashing can be considerably more efficient than recursive hashing.

A hash join has several important parameters that can be set by an optimizer based on its knowledge of the source relations' cardinalities. Most important is the decision about which operand will be the inner relation: this should be the smaller of the two relations, as it must be loaded into a memory. Other parameters include the number of hash buckets to use, the number of buckets to write to disk at each overflow, and the amount of memory to allocate to the operator. In a conventional database system, where the optimizer has knowledge about cardinalities, and where the cost of a disk I/O from any source is the same, the join parameters can be set effectively. However, a data integration environment creates several challenges:

- The optimizer may not know the relative sizes of the two relations, and thus might position the larger relation as the inner one.

- Since the time to first tuple is important in data integration, we may actually want to use the larger data source as the inner relation if we discover that it sends data faster.

- The time to first tuple is extended by the hash join's non-pipelined behavior when it is reading the inner relation.

Double Pipelined Hash Join

The double pipelined hash join is a symmetric and incremental join, which produces tuples almost immediately and masks slow data source transmission rates. The trade-off is that we must hold hash tables for both relations in memory.

As originally proposed, the double pipelined join is data-driven in behavior: each of the join relations sends tuples through the join operator as quickly as possible. The operator takes a tuple, uses it to probe the hash table for the opposite join relation, and adds the tuple to the hash table for the current relation[1]. At any point in time, all of the data encountered so far has been joined, and the resulting tuples have already been output.

The double pipelined join addresses many of the aforementioned problems with a conventional hash join in a data integration system:

- Tuples are output as quickly as data sources allow, so time to first output tuple is minimized.

- The operator is symmetric, so the optimizer does not need to choose an "inner" relation.

- Its data-driven operation compensates for a slow data source by processing the other source more quickly. This also allows the query execution system to make more efficient use of the CPU, as it may process data from one join relation while waiting for the other.

On the other hand, the double pipelined join poses two problems as we attempt to integrate it into Tukwila. The first is that the double pipelined join follows a data-driven, bottom-up execution model. To integrate it with our top-down, iterator-based system, we make use of multithreading: the join consists of separate threads for output, left child, and right child. As each child reads tuples, it places them into a small

[1]Once the opposite relation has been read in its entirety, it is no longer necessary to add tuples to the hash table unless the matching bucket has overflowed.

tuple transfer queue. The join output thread then takes a tuple from either child's queue, depending on where data is present, and processes that tuple. For greater efficiency, we ensure that each thread blocks when it cannot do work (i.e., when transfer queues are empty for the output thread, or full for the child threads).

The second problem with a double pipelined join is that it requires enough memory to hold both join relations, rather than the smaller of two join relations. To a large extent, we feel that this is less of a problem in a data integration environment than it is in a standard database system: the sizes of most data integration queries are expected to be only moderately large, and we may also be willing to trade off some total execution time in order to get the initial results sooner. Additionally, we expect an optimizer to use conventional joins when a relation is known to be especially large, or when one input relation is substantially smaller than the other. Nevertheless, we have identified several strategies for efficiently dealing with the problem of insufficient memory in a double pipelined join, and report on experiments with each of these methods (see Section 5.3).

Handling Memory Overflow

When a hash join overflows, the only feasible recovery strategy is to take some portion of the hash table and swap it to disk. With the double pipelined hash join, there are at least four possibilities. First, it is possible to use statically sized buckets which are flushed and refilled every time they overflow, but this would not perform well if the relation were slightly larger than memory. Another alternative would be a conversion from double pipelined join to hybrid hash join, where we simply flush one hash table to disk.

The two algorithms we implemented in Tukwila are considerably more sophisticated and efficient. To give a feel for the algorithms' relative performance, we include an analysis here of a join between two unsorted relations A (left child) and B (right child) of equal tuple size and data transfer rate, and of the same cardinality s. For simplicity, we count tuples rather than blocks, and we further assume even distribution of tuples across hash buckets, and that memory holds m tuples. Note that our emphasis is on the disk I/O costs, and that we do not include the unavoidable costs of fetching input data across the network or writing the result.

Incremental Left Flush Upon overflow, switch to a strategy of reading only tuples from the right-side relation; as necessary, flush a bucket from the left-side relation's hash table each time the system runs out of memory. Now resume reading and joining from the left side. This approach allows the double pipelined join to gradually degrade into hybrid hash, flushing buckets lazily. If memory is exhausted before the operation completes, we proceed as follows. (1) Pause reading tuples from source A. (2) Flush some buckets from A's hash table to disk. (3) Continue reading tuples from source B, entering them into B's hash table, and using them to probe A's (partial) table; if a B-tuple belongs in a bucket whose corresponding A-bucket has been flushed, then *mark* the tuple for later processing. (4) If source B's hash table runs out of memory after A's table has been flushed completely, then write one or more of B's buckets to disk. (5) When all of B has been read, resume processing tuples from source A. If these tuples belong in a bucket which has been flushed, then write the tuples to disk; otherwise probe source B's hash table. (6) Once both sources have been processed, do a recursive hybrid hash to join the bucket overflow files. To avoid duplicates, the unmarked tuples from A should only be joined with marked tuples from B, whereas marked tuples should be joined with both unmarked and marked tuples. We calculate total costs for this algorithm as follows:

- **Suppose** $\mathbf{\frac{m}{2}} < \mathrm{s} \leq \mathbf{m}$, so B does not overflow. We flush $s - \frac{m}{2}$ tuples from A, giving a cost of $2s - m$.

- **Suppose** $\mathbf{m} < \mathrm{s} \leq \mathbf{2m}$, so B is too large to fit in memory. In reading B, we overflow $(\frac{m}{2}) + (s - m)$ tuples. Reading the rest of A flushes $s + \frac{m^2}{2s} - \frac{3}{2}m$ more tuples. Our total cost becomes $4s - 4m + \frac{m^2}{s}$.

Incremental Symmetric Flush In this case, we pick a bucket to flush to disk, and flush the bucket from both sources. Steps to resolve overflow are as follows: (1) Upon memory exhaustion, choose a bucket and write that component of both A and B's hash tables to disk. (2) Continue reading tuples from both source relations. (3) If a newly read tuple belongs to a flushed bucket, mark the tuple as new and flush it to disk; otherwise, add the tuple to the appropriate hash table, and use it to probe the opposite hash table. (4) Once both sources have been processed, do a recursive hybrid hash to join the bucket overflow files. Note that the join must consider the tuple markings:

unmarked tuples should only be joined with marked tuples; marked tuples should be joined with both unmarked and marked tuples. The disk I/O costs of this algorithm can be derived as follows:

- **Suppose** s $\leq$ **2m**. After reading the entire contents of both tables, we have overflowed $2s - m$ tuples. After reading them back, we get a total cost of $4s - 2m$.

Our analysis suggests that incremental left-flush will perform fewer disk I/Os than the symmetric strategy, but the latter may have reduced latency since both relations continue to be processed in parallel. Section 5.3.3 evaluates this assessment empirically.

5.3 Experiments

We report the highlights of our experiments in four areas, showing that (1) the double pipelined join outperforms hybrid hash, (2) the preferred output behavior dictates optimal memory overflow strategy, (3) interleaved planning and execution produces significant benefits, and (4) having the optimizer save state in order to speed subsequent re-optimizations yields substantial savings.

5.3.1 Implementation and Methodology

The experiments in this section were performed with an early version of the Tukwila system, which was based not on XML, but on a socket-based, JDBC interface to a standard relational database. (As seen in the last chapter, JDBC over a socket provides roughly the same level of performance as x-scan over XML.)

A this point, the execution engine was approximately 25,000 lines of C++ code, with a dedicated memory manager for hash joins (we have since extended the memory management capabilities to be significantly more general). The initial optimizer and wrappers were written in Java 1.1. A key feature of this optimizer was the ability to save optimization state; this optimizer was used in our experiments involving interleaving of planning and execution. For the other experiments, we used hand-coded query plans for greater control.

Experiments were performed using scaled versions of the TPC-D data set, at 50MB and 10MB, created with the **dbgen** 1.31 program. This data was stored in IBM DB2 Universal Database 5.20 on a dual-processor 450MHz Pentium II server with 512MB

RAM, running Windows NT Server. The wrappers used IBM's DB2 JDBC driver, and were run directly on the server with JIT v. 3.10.93. The execution engine was run on a 450MHz Pentium II machine under NT Workstation with 256MB RAM. Our machines were connected via a standard 10Mbps Ethernet network.

For each of the experiments, we initially ran the query once to "prime" the database, then repeated it 3 times under measurement conditions. We show the average running times in our experimental results.

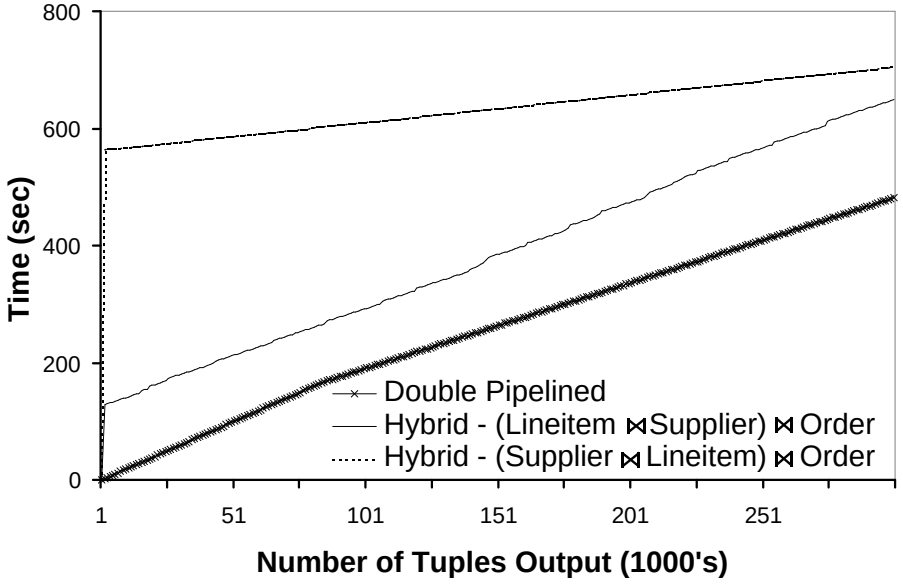
5.3.2 Performance of Double Pipelined Join

In order to compare the overall performance of the double pipelined join versus a standard join, we ran all possible joins of two and three relations in our 50MB TPC-based data set.

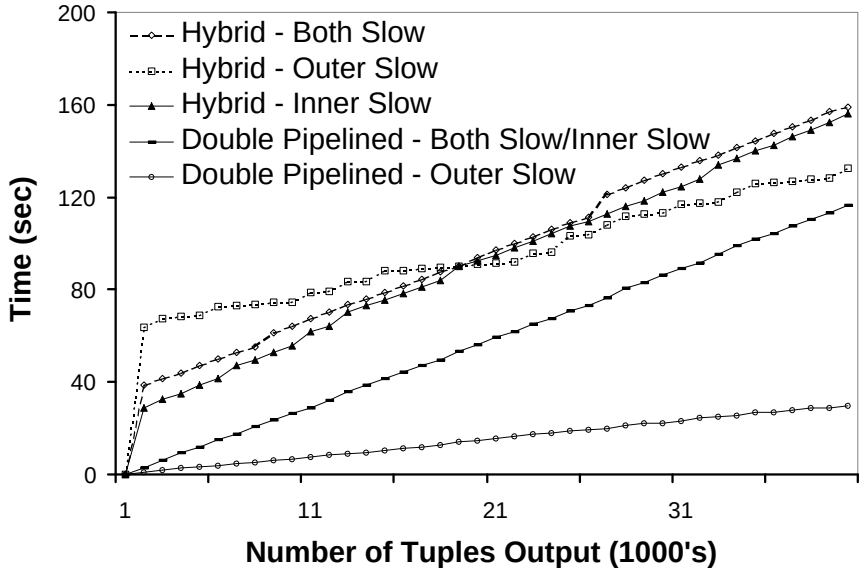
The results are very much in favor of the double pipelined join. In each of the experiments, we saw the same pattern: not only did the double pipelined join show a huge improvement in time to first tuple, but it also had a slightly faster time-to-completion than the hybrid hash join. This is explained by the double pipelined join's use of multithreading, which allows it to perform useful work as it is waiting for data to arrive. The exact performance gain of the double pipelined join varied depending on the sizes of the tables (since a small inner relation allows the hybrid hash join to perform well), but in all cases there was a measurable difference. Additional preliminary experiments suggest that adding prefetching to the hybrid hash join can almost remove the gap in total execution time between the two join methods, but that the double pipelined hash join still has an advantage in time-to-first-tuple.

Figure 5.3.2a shows a typical plot of tuples vs. time for the 3-relation join **lineitem** $\bowtie$ **order** $\bowtie$ **supplier** with different configurations of the join tree. **lineitem** is larger than the combined **order** $\bowtie$ **supplier** result, so clearly it should be joined last. However, since the hybrid hash join is not symmetric, our assignment of inner and outer relations at each join impacts the performance for this join. In contrast, the double pipelined join performs equally well in all of these cases.

Next, we analyze the performance of the double pipelined join in a wide-area domain. In order to get realistic performance, we redirected wrapper data originating at the University of Washington to a Java "echo server" located at INRIA in France, which "bounced" the data back to the wrapper, which in turn forwarded the delayed data to



(a) Join Performance: Lineitem ⋈ Supplier ⋈ Order



(b) Wide Area Performance: Partsupp ⋈ Part

Figure 5.3: Double pipelined join produces initial results more quickly, is less sensitive to slow sources, and completes faster than the optimal hybrid hash join.

the execution engine. A measurement of link bandwidth with the **ttcp** network measurement tool yielded an estimate of 82.1KB/sec, and **ping** returned a round-trip time of approximately 145msec.

Figure 5.3.2b shows the performance of a sample join, **partsupp** $\bowtie$ **part**, under conditions where both connections are slow, the inner relation is slow, the outer relation is slow, and at full speed. As expected, we observe that the double pipelined join begins producing tuples much earlier, and that it completes the query much faster as well.

5.3.3 Memory Overflow Resolution

The first experiment assumed ample memory, but since double pipelined join is memory intensive, we now explore performance in a memory-limited environment. In order to contrast our double pipelined overflow resolution strategies, we ran experiments to measure the performance of these strategies under different memory conditions.

Figure 5.4 shows one such result. Here we are executing the join **part** $\bowtie$ **partsupp**, which requires approximately 48MB of memory in our system. The graph shows how the number of tuples produced by a given time varies as we run the same join with full memory, 32MB of memory, and 16MB of memory.

From the figure it is apparent that the Left Flush algorithm has a much more abrupt tuple production pattern, as it runs smoothly only until the first overflow, after which it must flush and read in the right child before resuming fully pipelined operation. Note that this is still superior to the hybrid hash join, because our algorithm may still produce output as it reads the right child if there is data in the left child's hash table.

In contrast, the Symmetric Flush algorithm continues to pipeline as it overflows, but the number of buckets in memory decreases. The result is a a somewhat smoother curve which is dependent on the skew of the data.

Our experiments suggest that overall running time for the two strategies is relatively close, and that the primary basis for choosing the overflow resolution strategy should be the desired pattern of tuple production. Left Flush must operate for a period in which few tuples are output, but after which it begins pipelining the left child against most or all of the right child's data. Symmetric Flush produces tuples more steadily, but its performance slows as memory is exceeded, up until the point at which

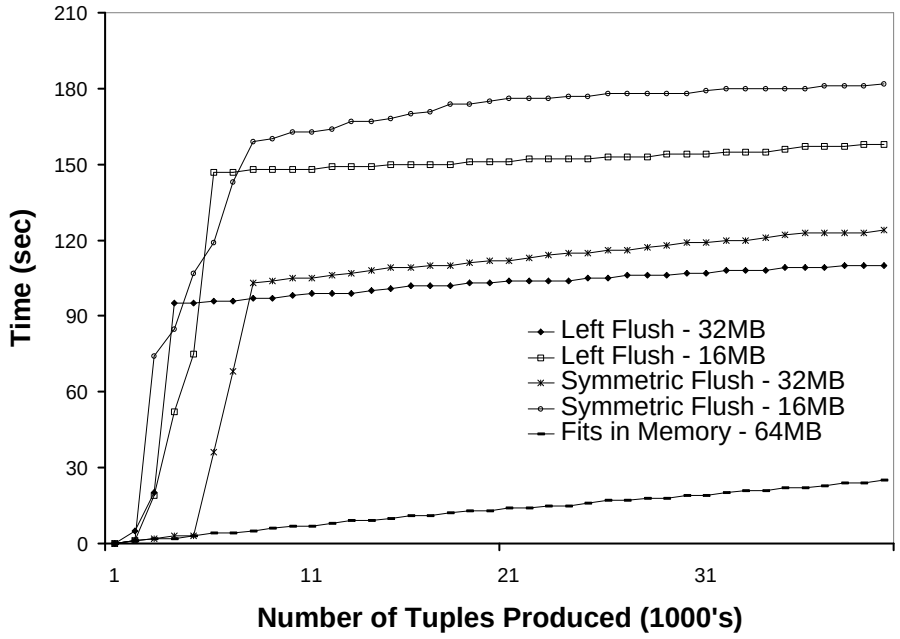


Figure 5.4: Symmetric Flush outputs tuples more steadily, but the rate tapers off more than with Left Flush. Overall performance of both strategies is similar.

the sources have been read and the overflow files can be processed.

The results also suggest that, while there is a noticeable penalty for overflowing memory with the double pipelined join, the operator's ability to produce initial tuples quickly may still make it preferable to the hybrid hash join in many situations.

5.3.4 Interleaved Planning and Execution

For complex queries over data sources with unknown selectivities and cardinalities, an optimizer is likely to produce a suboptimal plan. In this experiment, we demonstrate that Tukwila's strategy of interleaving planning and execution can slash the total time spent processing a query. We find that replanning can significantly reduce query completion time versus completely pipelining the plan.

For the 10MB data set, we ran all seven of the four-table joins that did not involve the **lineitem** table (which was extremely large). The optimizer was given correct source cardinalities, but it had to base its intermediate result cardinalities on estimates of join selectivities, since no histograms were available. We used the double

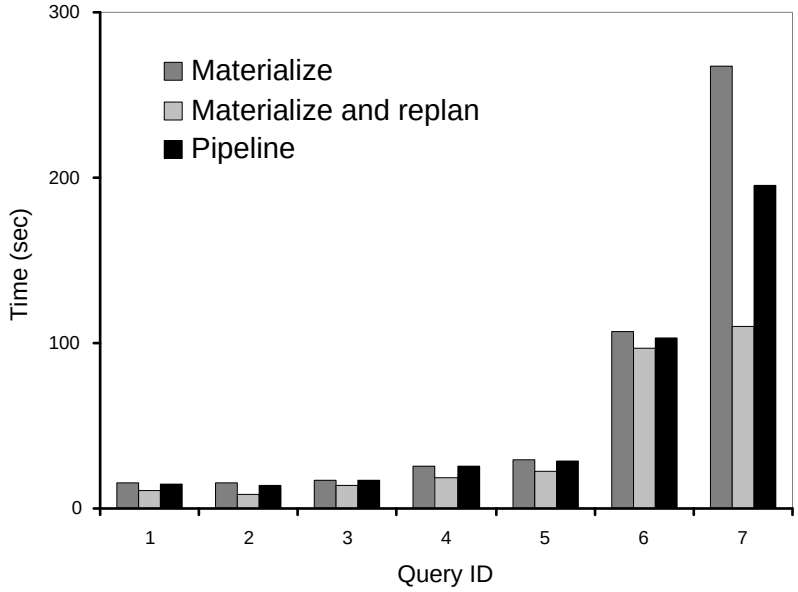


Figure 5.5: Even counting the cost of repeated materialization, interleaved planning and execution runs faster than a fully pipelined, static plan.

pipelined join implementation in all cases.

In Figure 5.5 we see the comparison of running times for three different strategies using the same queries. The baseline strategy is simply to materialize after each join and go on to the next fragment. The second strategy added a rule to the end of each fragment, which replans whenever the cardinality of the result differs from the estimate by at least a factor of two. The third strategy is to fully pipeline the query.

In *every case*, the materialize and replan strategy was fastest, with a total speedup of 1.42 over *pipeline* and 1.69 over the naïve strategy of materializing alone. This is somewhat surprising, since the benefit of replanning based on corrected estimates overwhelms the costs of both replanning and extra materializations in each case. The most likely reason is that many of the join operations were given insufficient memory because of poor selectivity estimates, and this caused them to overflow. In practice, both cardinality and selectivity estimates of initial table sizes will be inaccurate, favoring replanning even more.

5.3.5 Saving Optimizer State

As the results from the previous experiment illustrate, re-optimization can yield significant performance improvements. Hence, it is common for the Tukwila execution system to re-invoke the optimizer after finishing a fragment. The optimizer then needs to correct its size estimate for the fragment's result, and update the cost estimate to reflect the cost of reading the materialization. A dynamic-programming optimizer can either replan from scratch each time, or save its state for reuse on the next re-optimization.

For the case of replanning from scratch, the query gets smaller by one operation after each join, thereby halving the size of the dynamic program. However, reuse has the advantage that any new information about the completion of a fragment can only impact half of the entries in the original table.

The advantage of saving state is that half of the useful entries in the rebuilt table have already been computed. Our stored-state algorithm visits none of these nodes. To facilitate this search strategy during re-optimization, we introduce *usage pointers* into the dynamic program from each subquery to every larger subquery that *can* use it as a left or right child. We also keep a usage pointer from every subquery to every subplan that *does* use it as a left or right child. In our final experiment, we compare replanning from scratch to re-optimization based on saved state as optimized with usage pointers. Here we realize a speedup of up to 1.64 over replanning from scratch. In separate experiments (not shown) we compare re-optimization using saved state without usage pointers and the resulting performance is worse than replanning from scratch. See Marc Friedman's thesis [Fri99] for more details on the implementation.

5.4 Conclusions

This chapter presents the features necessary for building an adaptive query processing system. There are two primary points of emphasis: (1) providing support mechanisms for an optimizer to monitor and replace the currently executing query plan, and (2) features by which the execution engine can itself adapt scheduling or behavior within query execution. My contributions include identifying basic mechanisms for achieving adaptive behavior, incorporating them into a unified framework, and presenting evidence of their utility. Specifically:

- I describe the architecture of the Tukwila query engine, which builds numerous adaptive features directly into its core. The execution engine features provide significant feedback to the query optimizer so it can create more effective query plans. It also supports replacement or modification of a query plan at pipeline boundaries or in mid-execution, and does this while maintaining good performance (for instance, by sharing data between operators).

- I describe the design and implementation of query operators that are especially suited for adaptive behavior — the double pipelined join and the dynamic collector. I also demonstrate two useful techniques Tukwila uses to adapt the execution of a double pipelined join when there is insufficient memory for its execution.

- I use Tukwila to measure the impact of adaptive execution on data integration performance. I show that the double pipelined join outperforms the hybrid hash join when given sufficient memory, and I experimentally demonstrate the efficiency gains of interleaving optimization and execution over the traditional approach of computing the entire plan before execution begins. I describe how we efficiently resolve memory overflow for the double pipelined join.

The adaptive execution engine described in this chapter has a great deal of flexibility, and can be used with many different optimizer policies and re-optimization intervals. In the next chapter, I present a framework building query re-optimization in mid-execution, and I present a specific query re-optimization policy that builds on existing database techniques to provide good performance in data integration scenarios.

Chapter 6

ADAPTIVE OPTIMIZATION OF QUERIES

Beginning with the first cost-based query optimizer in System-R [SAC+79], databases have typically answered queries in a two-stage process. First, the query optimizer examines statistics on the data, incorporates knowledge of CPU and I/O speeds, and uses a cost model to estimate which plan will complete with least cost. Then, the query execution system takes this query plan and executes it to completion, exactly as specified. While this approach has been extremely successful in many applications, it has several important limitations:

1. Because detailed data statistics are often sparse, it may not be possible to find histograms to help estimate the size of a particular join. In some cases even cardinality estimates may be unavailable — for instance, if data is remotely controlled and frequently updated, or if it is provided by a system with an incompatible internal representation (e.g., a keyword search engine). In these cases, optimizers typically make assumptions about the data distribution and use arbitrary heuristics or "magic numbers."

2. Since actual runtime conditions may vary (e.g., due to concurrent queries), the optimizer may make inappropriate scheduling decisions and cost estimates.

3. It is essentially impossible to build a cost model that accurately models the current environment and data. Even with good statistical information, the error in optimizer estimates grows exponentially with the number of joins [IC91], and it is even worse when selection and join predicates include conjunctions and disjunctions [Ant93]. Furthermore, it is extremely difficult and expensive to precompute enough statistical information to accurately represent correlations and selectivities between attributes in large query expressions (for instance, using probabilistic relational models [GTK01] or multidimensional histograms [BCG01]).

All three of these problems are being encountered with increasing regularity in database applications, especially in areas such as data integration, wide-area dis-

tributed databases, and even centralized databases in which statistics are limited and queries are posed in *ad hoc* fashion. As a result, one of the fundamental problems in query processing is techniques for providing efficient answers in limited-knowledge situations.

Previous Adaptive Methods

Many researchers have developed adaptive techniques in response to the problems. These have generally fallen into two categories, *inter-query* adaptivity and *intra-query* adaptivity.

Inter-query adaptivity attempts to future queries by "learning" from past queries. Obviously, there must be overlap between old and new queries for these techniques to work. The techniques used here mostly consist of improving statistics on certain regions of the queried data. One of the earliest such pieces of work was by Chen and Roussoupoulos [CR94], and it attempted to improve estimates for selection predicates. Recently, the commercial DB2 system added a "learning optimizer," LEO [SLMK00], which attempts to add adjustment factors for histograms that have been found to be inaccurate. Finally, SQL Server's AutoAdmin feature now includes a capability for generating additional histograms on subplans within a query, for use in computing future cost estimates [BC02].

Techniques for intra-query adaptivity interleave optimization and execution, and generally must deal with a harder problem: they must compromise between time spent trying to find a better plan and time spend executing. A simple approach to handling unpredictable intermediate result sizes, given good initial statistics, is the choose node [CG94], which allows the optimizer to precompute several alternative subplans and have the execution system choose one based on initial runtime conditions — in essence, this is eager generation of the most likely alternative plans. Kabra and DeWitt [KD98] dealt with mis-estimated intermediate result sizes in a lazy-evaluation way, by extending a traditional execution system so that it could re-invoke the optimizer after completion of any pipelined segment of the initial plan. Their system inserts statistical monitors where they incur low overhead but provide information about when re-optimization is useful. Query scrambling [UFA98] handles a different type of unpredictability: when data from a source is delayed, they attempt to reschedule or even re-optimize un-executed portions of the query plan. The XJoin [UF00] and double

pipelined join [IFF+99] handle unpredictable I/O rates in query execution, by allowing the system to schedule work while other parts of a query plan are I/O-bound. Finally, eddies [AH00] address the zero-knowledge problem in the case of SPJ queries by *eliminating* cost-model-based optimization altogether — instead using data flow rates and operator selectivity values to route tuples through a dynamic query plan.

Unfortunately, while all of these adaptive approaches provide certain benefits, they each address different subproblems, and it is not easy to combine them to create a comprehensive solution. Furthermore, many of these techniques are restricted to the domain of SPJ queries and cannot be easily generalized to more expressive query languages.

Convergent Query Processing

This chapter presents a logical query optimization framework and a set of adaptive techniques, together called *convergent query processing*, which provides a more comprehensive solution to the three challenges cited above. At a high level, our approach is similar to several previous methods: during execution, we continuously monitor the costs of operations and size of intermediate results; if the plan is poor, we replace it with one that is expected to perform better. Convergent query processing provides two very important benefits: first, it allows standard query optimization techniques, and even standard optimization infrastructure, to be used to re-optimize a query plan *in mid-pipeline*, and second, our framework supports re-optimization at any level of granularity, from heuristic-based per-tuple re-optimization to more sophisticated plan rewriting. Convergent query processing is so flexible primarily because of the way it partitions work.

Kabra and DeWitt's mid-query re-optimization partitions the *query plan*: statistics from early pipelined stages inform adaptation, which can only happen at *pipeline boundaries*. A bad choice for the initial pipeline can lead the system down an arbitrarily bad execution strategy.

Eddies effectively partition *data*, routing tuples through a dynamic plan. A tuple router consults data-flow and selectivity statistics for each operator, and it alters routes at *moments of symmetry*. Eddies use a greedy strategy based on recent operator selectivity values, and thus they cannot handle blocking operators or incorporate pre-existing (partial) knowledge about costs.

Convergent query processing partitions either or both *data and the query plan*: the statistics from processing initial data inform adaptation, which can happen at an arbitrary point in time. Convergent query processing can incorporate pre-existing information about source cardinalities to make more globally optimal decisions. Our method tends to quickly replace poor initial plans, limiting their adverse impact, and to "converge" towards a more optimal plan. Furthermore, by restricting the points at which re-optimization is considered or the types of plans considered, one can use our convergent query processing framework to emulate most previous methods.

The specific contributions of this chapter include:

- A novel *phased execution* framework for adaptive query processing, which enables more frequent re-optimization opportunities than previous approaches and provides more flexibility in how work is partitioned.

- A set of algebraic transformations that can be used to rewrite a query into a logically equivalent union of "phases" — distinct query plans, each with its own data partition. A *cleanup phase* is added to ensure that the union of phase results produces the complete answer set without introducing duplicates.

- An optimizer and execution architecture, implemented within the Tukwila, which exploits these transforms to power convergent query processing in a data integration context. Our implementation leverages established techniques from traditional optimization where appropriate, and demonstrates that a drastic re-architecting is not necessary.

- A set of experiments demonstrating that Tukwila's use of convergent query processing improves performance versus traditional query processing techniques (for several TPC-H and other queries); that convergent query processing performs acceptably in memory-constrained environments; that the overhead of monitoring and re-optimization (as well as the addition of the cleanup phase) are typically low and are easily regained through improved execution times; and that convergent query processing tends to stabilize on a plan that performs well.

The remainder of this chapter is structured as follows. I begin with an overview of the convergent query processing and present an example illustrating the basic princi-

ples. Section 6.2 describes how various query operators are supported in *phased query execution*, which is the basis for convergent query processing. Section 6.3 describes the key aspects of our implemented system, and Section 6.4 presents experimental results. Section 6.5 concludes the chapter.

Note that this chapter describes query processing using relational concepts, but our implementation maps XML data into a relational-style architecture and directly uses the techniques presented here.

6.1 Convergent Query Processing

While the problem of limited statistics appears in many different contexts, it is particularly an issue in data integration, the focus of this dissertation, where the data sources are likely to be autonomous and heterogeneous. We assume the following requirements:

- A rich query language (a subset of either SQL or XQuery) with grouping, joins, and aggregation.
- Interactive users, who want quick access to initial results as well as fast overall processing time.
- Few statistics on data sources, as well as the possible presence of dynamic data sources.
- Irregular data access characteristics due to remote system load and network latency.

The need for fast initial answers requires us to start query execution with a single pipeline, although this decision may need to be reconsidered after some number of answers have been returned (for instance, because system resources may be too constrained to execute the entire query in one pipeline). The response time requirement also precludes the system from spending much time sampling *before* execution begins; however, we fully support the use of pre-existing knowledge from statistics or prior executions, and our model supports monitoring of statistics during or alongside query execution. We rely on flexibly scheduled operators such as the double pipelined

join [IFF+99, UF00] to handle scheduling operators within a pipeline and mask I/O delays, so our instead focuses on reducing query execution cost (i.e., work done processing intermediate results). The remainder of this section describes the main features of CQP.

6.1.1 Phased Query Execution

In the convergent query processing model, every time the processor switches a plan, execution enters a new *phase*. The process of executing a sequence of phases in time-contiguous order is called *phased query execution*. At the beginning of each phase, the query processor can use the knowledge gained in prior phases to choose a different (hopefully better) evaluation plan ("*phase plan*") for the query. Since this new phase plan is applied to the data that is read from the inputs during the new phase, the first n phases partition each input table into n non-overlapping sub-tables. Finally, the $n+1$st phase, called the *cleanup phase*, combines subresults from the previous phases to produce all of the answers to the full query that have not been returned by the previous phases. Intuitively, phased query execution is possible because we can distribute unions over joins and other operators.

Convergent query processing's phased execution model allows arbitrary changes to the executing query plan in mid-stream. The hope is that each successive plan will be closer to optimal than the previous phase's[1]. Naturally, one would like to maintain all previously computed results, including intermediate results, even if the query plan changes between phases; this requires careful bookkeeping. Furthermore, since frequent switches may be required to adapt to a dynamic environment or to increasing knowledge, it is important that switching plans during phase transitions be extremely fast.

The development of convergent query processing can be broken up into three independent sets of issues, which we discuss below: (1) a set of policies and algorithms for deciding when to switch phases, (2) a set of techniques for handling multiple phases and combining data across phases, and (3) optimization algorithms for quickly determining what form the next phase should take.

---

[1]Note that if data characteristics are not uniform, the optimal execution of a query may not be a *single* static query plan, but rather a *sequence* of query plans, each near-optimal for some portion of query execution.

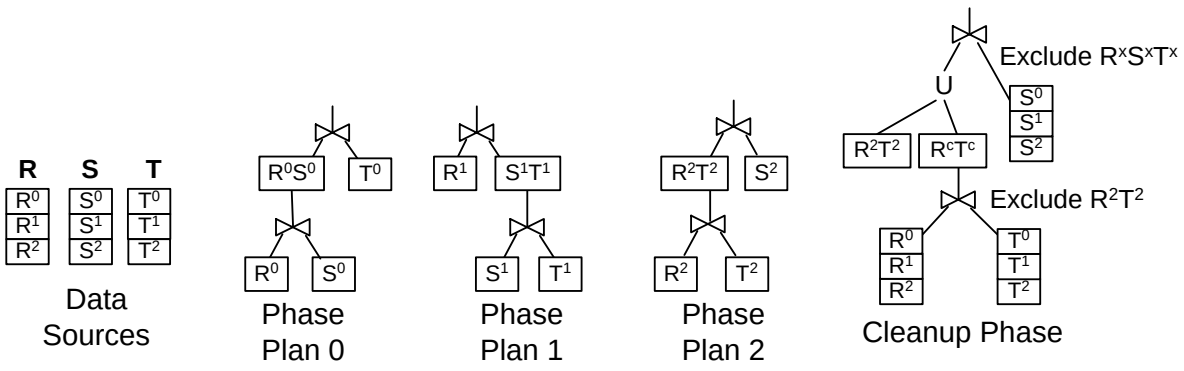


Figure 6.1: An example query joining R, S, and T. Suppose that during the course of execution, the query processor uses three different phases as it gains knowledge. Each of these phases will process a subset of the original relations — we designate these R^x, S^x, and T^x for phase x. Each of the separate phase plans produces some output tuples and some intermediate results. At the end, we must perform a final cleanup phase that joins the data *across* phases; this phase reuses data from the previous phases, even attempting to make use of intermediate results that have already been computed.

A Simple Example

The following simple example illustrates the main concepts of convergent query processing. Suppose the data sources consist of three relations, $R(x, y)$, $S(x, z)$, and $T(y, z)$, and the user asks for the natural join of R, S, and T. In SQL, this would be:

```
SELECT *
FROM R, S, T
WHERE R.x = S.x AND R.y = T.y AND S.z = T.z
```

Initially, convergent query processing proceeds in the conventional fashion. The query optimization module uses whatever cost and selectivity estimates are available and produces an initial query plan; suppose this "phase 0" plan is the join tree $(R \bowtie S) \bowtie T$. In our discussion we denote by R^x the data of relation R that was processed in phase x.

6.1.2 *Execution Monitoring & Phase Transitions*

As the query processor executes a phase plan, monitor running in the background periodically updates its estimates of costs, cardinalities, and selectivities for the currently executing plan and the query, as well as global costs such as CPU and disk I/O. If the re-estimated cost of the query plan deviates significantly from the original estimates, the query optimizer is re-invoked as a secondary thread with the updated selectivity and cost information. From this, the optimizer generates a *candidate phase plan*, and the query processor compares the candidate plan with the currently executing plan, which continues to do work. If the estimated cost of the candidate is sufficiently better than the current plan, the system performs a phase transition by pausing execution of the old plan in a stable state and starting execution of the new plan over the portions of the data sources that have not yet been consumed. The data structures of the old plan are maintained for later use, and in fact the query operators may also be retained for potential future re-activation.

Continuing our example (see Figure 6.1), suppose that during query execution, the monitor detects that its estimates were poor and triggers a re-optimization. It may then create a new phase in which S and T are joined first, and then the results are joined with R. When the query processor switches to this plan, it starts applying it to *new* data that it reads for the sources. Hence, we depict the plan by $R^1 \bowtie (S^1 \bowtie T^1)$. In a similar fashion, at a later stage, after seeing more data, the system may decide to switch to a third phase executing $(R^2 \bowtie T^2) \bowtie S^2$.

Note that a key benefit of convergent query processing is the ability to change join *implementations* from phase to phase, but to keep the example simple, we do not illustrate that here.

6.1.3 *The Cleanup Phase*

When the last phase runs until all data sources are exhausted, the system will have succeeded in joining the subsets of tables *within* each phase. Now the cleanup phase must join all combinations of subsets *across* phases: $R^0 \bowtie S^1 \bowtie T^0$, $R^0 \bowtie S^0 \bowtie T^2$, etc.

A common concern stems from the fact that if there are k relations and n phases, the cleanup plan must perform $O(n^k)$ joins between the different phase subtables. Fortunately, however, the subtables are proportionately smaller and thus in the worst case the work performed by the cleanup phase can never be greater than that for the

initial query as a whole. And if the processor successfully avoids computing redundant intermediate results, the cleanup plan will involve substantially less work. Thus the dominating factor becomes the *quality* of the plan being used. Fortunately, by the time the query processor has reached the cleanup phase, it will likely have excellent estimates for the costs and selectivities of the query plan; thus the cleanup plan should be extremely efficient.

The central challenge for the cleanup plan is minimizing the redundant computation of intermediate results. The previous phase plans all buffered their partitions of the original data sources, and these are combined (e.g., $R^0 \cup R^1 \cup R^2$) to form the source relations for the cleanup plan. However, since previous phase plans may already have computed some intermediate results needed by the cleanup plan, these intermediate results should be reused whenever possible.

Thus to ensure efficiency and provide correctness, joins in the cleanup plan are given an associated *exclusion list* which describes source-tuple combinations that should not be regenerated, because their results are available from previous phases. As tuples flow through join operators they are annotated with information describing the phase from which each of their source tuples originated, and these annotations are compared against the exclusion lists in subsequent joins. Continuing our example, note that exclusions in the cleanup plan of Figure 6.1 prevent regeneration of R^2T^2 or $R^xS^xT^x$ tuples for any phase x. As a result, the cleanup phase *only generates results that have not been previously produced*, and since it has access to highly accurate statistics, it uses a highly optimized plan.

Note that the phased execution process produces a highly beneficial workload for query execution: early "normal" phases, during which the system is reading data and acquiring knowledge, are less likely to be CPU-bound or to produce inefficient intermediate results — since either of these conditions would probably cause a phase transition. Most of the computation that results in expensive CPU costs is postponed until the system has stabilized on a good phase plan, i.e., to the cleanup phase, which will have the best plan the optimizer can create.[2]

[2]Note that erroneous selectivity statistics have a much greater effect on techniques like mid-query re-optimization [KD98] than on convergent query processing. A bad decision about the first join in a plan can have arbitrarily bad consequences, and mid-query re-optimization must complete the pipeline. In contrast, if the convergent query processor makes a bad decision, the execution monitor will quickly discover the problem and initiate a new plan.

6.1.4 Optimization and Re-optimization

Convergent query processing selects plans in an iterative fashion: the optimizer chooses a plan, the system begins executing it, the monitor updates the optimizer's cost model, and the optimizer re-optimizes the query. Since the optimizer is invoked repeatedly, it makes sense to maintain certain data structures throughout phased execution. The monitor may thus directly access the data structures of the execution system and the optimizer.

The optimizer maintains a tree, in which each interior node represents a combination of select, project, join, and group operations; each optimizer node is annotated with expected cost information as well as algorithm selection. The physical plan generator creates the set of physical operators described by each optimizer node and links these to it.

When execution begins, the operators begin recording cost and cardinality information. Periodically, the status monitor polls the executing plan; it updates the CPU cost of each operator and the expected sizes of the source relations. For each node in the optimizer tree, the monitor examines the output cardinality of the corresponding physical operator. From this, it computes *subtree selectivity* — the selectivity of applying all operators in the current subplan to its data sources.[3]

If the monitor detects that costs are significantly different from the original optimizer estimates, background re-optimization is triggered. This optimization step should yield better CPU cost and data size estimates. Furthermore, for certain join combinations, it also has accurate selectivity values obtained from the current phase plan execution. The use of subtree selectivity is critical here: if the plan $(R \bowtie S) \bowtie T$ is being executed, the system cannot accurately infer the selectivity of $S \bowtie T$, but it knows the selectivity of $R \bowtie (S \bowtie T)$. If the optimizer finds a plan which appears superior, the current plan is suspended and replaced. Then execution and monitoring resume and the process repeats. Ultimately, the system gains enough cost and selectivity information to avoid bad plans and hopefully to "converge" on a plan that produces good performance. Note that our current strategy is "reactive" rather than "proactive": it will change plans if execution appears to be proceeding poorly, but it would continue executing a plan that appears to be performing up to expectations, even if some better plan exists. In Chapter 8 I discuss possible strategies for finding truly optimal plans.

[3]The monitor makes no changes to the estimates for subtrees that have not yet output any data.

6.1.5 *Using Incomplete Plans*

In our previous example (Figure 6.1), each phase produced tuples which satisfied the user's query, but this is neither necessary nor always the best strategy. One useful alternative may be to start execution with an *incomplete* plan intended for "information gathering" purposes (e.g., we may wish to execute $R \bowtie T$ early, simply to get information about its selectivity), rather than as the final executable plan. Such a plan might gather selectivity information; once the plan fragment completes, the query processor can create a subsequent plan that advances closer towards producing a final query result. If this strategy were used in the first phase of the previous example, then the cleanup phase *would* need to generate $R^0S^0T^0$ tuples, and so the exclusion list would not include this combination. Instead, R^0T^0 would be excluded.

In summary, every phase contributes something towards processing the query: (1) answers, i.e., output from the root plan, (2) intermediate results that will be used in cleanup, and/or (3) knowledge, e.g., improved selectivity or cost information. The more of these that a phase produces, the greater its utility.

6.2 Operators for Phased Execution

In this section, we describe the principles of phased execution. We consider some important algebraic and physical-level aspects of the traditional relational operators, and see how they are adapted to the context of phased execution. Naturally, we start with joins, since they set the stage for the rest.

6.2.1 *Join*

Consider an equijoin-only query plan $J = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$, over the relations $R_1, \ldots, R_m$. Suppose there are n phases before the cleanup phase, and hence each $R_j, 1 \leq j \leq m$, has been partitioned into n subsets, one per phase: $R_j = R_j^1 \cup R_j^2 \ldots \cup R_j^n$. Using the distributive property of unions over joins, we can write the join as follows.

$$R_1 \bowtie \ldots \bowtie R_n = \bigcup_{1 \leq c_1 \leq n, \ldots, 1 \leq c_m \leq n} (R_1^{c_1} \bowtie \ldots \bowtie R_m^{c_m})$$

We can now rearrange the terms in the union into two sets. The first set is the union of the results from the first n phases: $\cup_{1 \leq i \leq n}(R_1^i \bowtie \ldots \bowtie R_m^i)$, and the second set defines what needs to be done in the cleanup phase:

$$\{t|t \in (R_1^{c1} \bowtie \ldots \bowtie R_m^{cm}),\ 1 \leq c_i \leq n,\ \neg(c_1 = \ldots = c_m)\}$$

Since the cleanup plan must join the subtables in all possible cross-phase combinations, one might conclude that the system needs to re-scan all source relations in order to perform cleanup.

However, if we require that every join operation buffers each of its inputs to perform the join, then the same effect can be obtained by taking the buffered inputs from each of the leaf-level join operators, and joining across those. Note that in terms of query operator algorithms, the pipelined hash family of joins [WA91] has precisely this property — and also has the benefit that the "build" portion of the hash join operation is already complete. For other join implementations, such as the nested loops join or the hybrid hash join, the "inner" or "build" relation is typically stored in memory, and the "outer" or "probe" relation is typically not. In this case, the external data source must be re-scanned or buffered and reused.

As mentioned previously, additional query processing performance can be gained by observing that certain subexpressions within the cleanup phase (e.g., joins between pairs of tables from the same phase) may have already been executed in previous phases. These intermediate join results can be reused within the cleanup process, reducing the amount of work that must be performed. In our example, there is no need to re-compute tuples of R^2T^2. (Note, however, that if phase plans differ substantially in order of execution, few intermediate results may be shared between the plans.)

6.2.2 Selection and Projection

Algebraically, the addition of selection and projection requires only a small modification to the join-splitting strategy described in Section 6.2.1. Selection and projection operators can initially be applied over the union of the different phases' join trees. Using the distributive laws, they can be pushed over the unions and joins — typically in query optimization, they will be pushed to the point of earliest evaluation.

At the physical level, however, predicate push-down requires some care in the cleanup stage. The join process during cleanup may require some combination of hash, nested loops, and double pipelined hash joins — as is required to join the data structures from prior phases. In some of these cases, the selection predicates may be evaluated within the join itself; in others, a separate selection operation must take place on

the join results. Hence, some predicates that can be pushed into a join operator may also be evaluated immediately afterwards in a separate selection operator.

Fortunately, projection generally causes fewer problems in terms of phased execution: conventional projection-push-down strategies ensure that identical subqueries within different phases will have the same projected attributes, even if different subplans are used. Thus no special precautions need to be taken to maintain homogeneity of intermediate results for re-use in the cleanup stage.

6.2.3 Grouping and Aggregation

Algebraically, grouping and aggregation are slightly more complicated, because one needs to be careful how to combine aggregate values from the different phases. In the cases of `MIN` and `MAX`, one can combine the minimum or maximum values computed in the different phases by simply another `MIN`/`MAX` operation. Likewise, for `SUM` one can add the sums of the different phases to obtain the correct result. The `COUNT` operation requires slightly different logic: the counts of each group from the individual phases need to be *summed* to produce the desired result. Finally, in order to handle `AVERAGE` one must compute for each phase both the sum and count of each group, and then average the results of the different phases at the end. (These techniques are the same as those suggested by [CS94].) A few additional adjustments are necessary in order to combine results that have pushed-down grouping and those that are not grouped, but we defer our discussion of these until Section 6.3, when we describe techniques for optimizing GROUP BY operations.

6.2.4 Other Operators

The phased execution model can also be extended to support other operators, such as union and outer join. Union, of course, distributes over join and other unions following the standard rules of the relational algebra — hence disjunctive queries can be decomposed into phases. The *nestChild* operator of Tukwila, which is similar to a left outer join operation, is commonly used in XQueries. It can be implemented in the following way: for all phases other than the cleanup phase, the *nestChild* is replaced with a standard join (which will not output any parents that do not have children, and which may not maintain contiguous order between sibling child elements). All results are fed into a grouping operator that clusters child elements in contiguous order under their

parents. Finally, during the cleanup phase, any parents that appear without children are also output.

6.2.5 Ordered Results

Up to this point, I have presented the phased execution model without mentioning ordered execution. In general, dependence on the preservation of "interesting orders" throughout plan execution, as is typically done for sorted inputs in a relational system, can be done within the phased execution model, but care must be taken to ensure that phases are broken along "natural boundaries" across the data. Otherwise, there may not be a total ordering on the data returned by the query. If the query needs to return data following a particular order, it needs to sort the final output[4].

6.3 Implementation within Tukwila

In previous chapters, I discussed the details of Tukwila's query engine, how it maps XML into a tuple-oriented, pipelined execution model, and its basic adaptive features. Now I present some of the important optimizer-related features.

While convergent query processing can benefit from proactive techniques, such as executing small subqueries alongside the main query for information gathering, our implementation focuses on showing the benefits of the convergent query processing approach in a context where all execution work is being done to return query answers, and where the same query optimizer is being used to generate both initial and final plans.

Tukwila consists of two modules. The core convergent query processor, which takes a query parse tree as input and returns tuples as output, is written in C++. The graphical user interface and query language parser are written in Java. Tukwila comprises approximately 80,000 lines of code. Since convergent query processing relies on tight interaction between query optimization, execution, and status monitoring, all of our query processing components run in the same memory space and share data structures. The monitor and optimizer can suspend query execution to replace a subplan, and the execution engine can trigger a re-optimization. In the next four subsections,

[4] If the first stage was over sorted relations and completely order-preserving, all of its tuples are guaranteed to precede those from subsequent phases, and thus they may "bypass" this final sort operation.

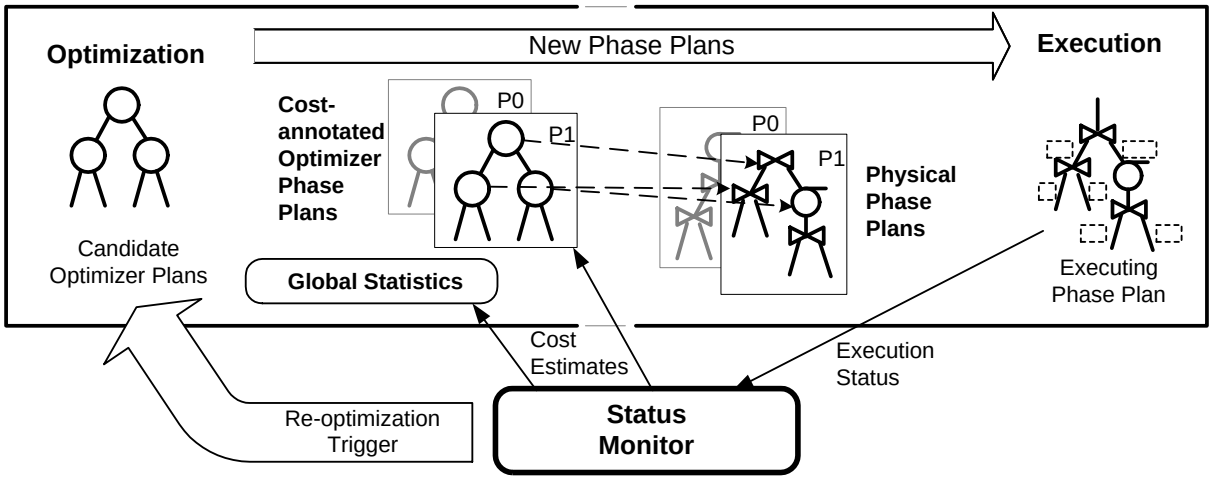


Figure 6.2: Tukwila query processor. The optimization component chooses a plan and annotates it with expected costs; as the plan executes, a status monitor updates cost-model estimates with real values. If costs diverge significantly, re-optimization may be triggered.

we describe the execution engine, our query optimizer, efficiency considerations for the cleanup phase, and Tukwila's novel mechanism for handling memory overflow.

6.3.1 Review of Query Execution Features

In order to support convergent query processing, Tukwila's query engine includes several unconventional features. The Tukwila implementations of the operators (including the standard array of joins, a double pipelined join, and a hash-based grouping operator) all must support arbitrary interruption and replacement. Since Tukwila uses an iterator-based execution model and attempts to share memory among operators to reduce copying, this requires some non-trivial bookkeeping and coordination among interlinked operators. Furthermore, every operator in the system records the cardinality of its output, plus the amount of time it spends doing its own "work." These allow the query processor to update selectivity and CPU cost information. Another key feature is that leaf-level "scan" operators read data from across the web using the HTTP protocol, which typically includes a header specifying the total number of bytes to be transferred. This information can be exploited to derive approximate values for source cardinalities.

Finally, in order to support the cleanup phase, join operators are slightly modified. The hybrid hash join and nested loops join operators typically only buffer their "build" or "inner" relations — in our system, the "probe" or "outer" relations are also buffered (this buffer may overflow to disk, as it is only needed by the cleanup phase). All of the data structures of the join are made accessible for the cleanup phase, which we discuss later in this section.

More details on the query execution engine are discussed in Chapter 5.

6.3.2 Phase Plan Selection

As explained in Section 6.1.4, plan selection is an iterative process: after the optimizer chooses a plan, execution begins, the monitor updates the cost model, and the optimizer re-optimizes the query. Initial query optimization occurs using conventional techniques. The Tukwila optimizer is a System-R-style dynamic-programming optimizer. The default selectivity heuristics of [SAC+79] are used whenever a predicate is encountered, and selections and projections are evaluated at the earliest opportunity. We slightly deviate from the traditional System-R design by supporting bushy tree enumeration: Tukwila is designed for network-bound data, and bushy enumeration is often beneficial in this context [HKWY97].

Recall that the optimization process creates a tree of optimizer nodes; each interior node represents a combination of logical operations, annotated with expected cost information and algorithm selection. A key characteristic of our optimizer is that, for any subset of tables within a query, it will always push the same set of predicates down to any subplan — so any two subplans can be identified as logically equivalent if they query over the same tables. This is important for the cleanup stage (discussed in the next subsection), which attempts to find equivalent subexpressions.

Our optimizer supports push-down of pre-aggregation, as described in Section 6.2 and [CS94], but we wish to maintain identical schemas between plans with pre-aggregation and those without. Thus we introduce a "pseudo-group" operator that computes aggregate values for "groups" of one tuple, i.e., a pass-through operator that injects aggregate values that can be directly combined with the output from a "true" grouping operator. We insert a pseudo-group operator at any point where a grouping operator *could* be pushed, but has not been. The logic for determining whether to insert pre-aggregation is based on an aggressive strategy: early aggregation is generally inserted

initially, and then removed if groups are found to be small.

6.3.3 Cleanup Phase Creation

There are two important points about optimization for cleanup. The optimizer will have the best possible statistics at this stage, and it can produce an optimal plan for the query (whose data is now locally available). However, note also that the query plan that would be optimal for executing the complete query over the original data sources may *not* be the optimal plan for answering the query when re-usable intermediate results are considered. Our optimizer considers the availability of intermediate results as it selects the cleanup query plan.

Once all re-usable data structures have been attached to the appropriate cleanup joins, they must all be converted to a form usable by the cleanup join (which works similarly to a ripple join [HH99]). Hash tables from previous phase plans have tuples that can be used directly by the cleanup join — but they may not have the appropriate hash key. For example, suppose that the phase zero plan uses the join expression $A \bowtie (B \bowtie C)$, but the cleanup plan chooses $(A \bowtie B) \bowtie C$. The A hash table from phase zero includes a useful subset of the A table — but it is keyed on the equijoin attributes between B **and** C. Since the cleanup join is attempting to join with B attributes **only**, it cannot probe the table. Our solution is to ensure that hash structures can also be scanned sequentially, i.e., they can be treated as lists instead of hash tables, and that they can be "re-keyed." Furthermore, any data that was saved as a simple buffered list (e.g., from a nested loops join) is converted to a hash table — this only incurs the overhead of one list traversal, and significantly improves performance in practice.

There are times when it is advantageous to perform some of the cleanup work before all other phases have completed. For example, suppose that the system is processing a query over n relations, and just after the first phase exhausts the data in R_j, the optimizer generates an improved plan for the next phase. Now, the next phase cannot perform any significant work, since it has no data to join from relation R_j. To alleviate this problem, we bring the data from a previous phase (e.g., R_j^0) into the current phase. A more comprehensive solution would be to allow the current phase to read data from earlier phases whenever delays were encountered, similar to the XJoin [UF00] method of processing overflow tuples during idle time. However, this would require considerably more bookkeeping than XJoin's timestamps strategy, and the overhead incurred

may be significant.

6.3.4 Handling Memory Overflow

Since it retains intermediate results and buffers all inputs, the convergent query processing strategy sounds quite memory intensive. It may initially seem counter-intuitive, then, that convergent query processing provides a natural partitioning on the data that has desirable qualities for overflow handling.

With phased execution, no data needs to be shared across phase boundaries until cleanup. This means that data from one phase can be flushed to disk to accommodate the next phase. Of course, during cleanup, many intermediate results may be needed to complete the cross-phase computation. However, no more memory will be required than in the equivalent plan under traditional query execution (ignoring the minor overhead of having the data partitioned across multiple structures). Moreover, during cleanup we should have good estimates of intermediate result sizes, and we can partition cleanup phase execution into an efficient sequence of pipelines with intervening materialization points, keeping only the immediately relevant data in memory.

In fact, there is a natural parallel between join overflow resolution and phase boundaries, particularly with regard to the double pipelined join or XJoin [IFF+99, UF00]. When such a join overflows, certain hash buckets (and future values that would fall into these buckets) are swapped to disk, and their space is allocated to the remaining buckets. Because of the double pipelined join's characteristics, all tuples prior to overflow have already been joined with all other data encountered before overflow. Subsequent tuples which get paged to disk must later be rejoined with (1) other post-overflow tuples that are immediately paged to disk, and (2) all pre-overflow tuples that were initially kept in memory but later paged to disk. This is very similar to a two-phase execution sequence: all pre-overflow tuples are joined (analogous to Phase 0); then all post-overflow tuples must be combined (Phase 1); finally, all post-overflow tuples must be combined with pre-overflow tuples (cleanup). This similarity suggested an overflow handling strategy which is different from that used in conventional query processing: upon encountering overflow, Tukwila initiates a new phase, swapping subresults from the current phase to disk. This gives Tukwila the opportunity to create a better plan that produces smaller subresults, and it can take advantage of the cleanup phase to produce the complete set of answers.

The overall Tukwila architecture performs quite well (as we shall see next in the experimental section). In general, most query output production occurs in the last pre-cleanup phase, and the optimizer seems to create a good cleanup phase plan to produce the remaining tuples. The memory overflow mechanism is also quite efficient.

6.4 Experiments

Tukwila is primarily a data integration system for the local and wide area. However, the principles of convergent query processing (CQP) are fully applicable to centrally managed data as well — although more statistical data may be available about data sources, it is well known that cost estimate errors can still be high for complex queries. Hence, our experimental evaluation considers both the local and remote query processing scenarios.

The experiments address five main questions: (1) how does CQP compare with the traditional optimize-then-execute strategy when execution is CPU bound? (2) how do the approaches compare when execution is largely I/O bound? (3) can CQP scale beyond memory? (4) how does CQP perform relative to Kabra and DeWitt's mid-query re-optimization approach [KD98]? and (5) how sensitive is CQP to parameter variations?

6.4.1 Experimental Setup

For data sources, we used the TPC-H benchmark, with tables generated at the 1% scale factor. These were converted to XML and stored on a web server so they could be fetched across the network by the query processor. We selected those queries that have SPJ and/or group operators, multiple output attributes, and at least two tables. These were queries 3, 5, and 10, which all offered opportunities for (re-)optimization of join ordering (group optimizations were not interesting because the joins were on foreign keys). Additionally, since our system supports dynamic push-down of aggregation operations, we wanted to see the performance impact on a query with a grouping operation that *could* be pushed. For this, we created a table encoding a set of people and a table encoding meetings and participants, and we posed a query that counts the number of person-to-person interactions one particular person could have had in all meetings (this results in a pair of joins and a grouping operation). Table 6.1 summarizes the relevant source information.

Table 6.1: Data sources for experiments. The first 6 tables are from the TPC-H benchmark, at the 1% scale factor.

Source	XML Size	Tuples	Queries
orders	53MB	150K	Q3, 5, 10, A, B
lineitem	32MB	60K	Q3, 5, 10, A, B
customer	5.2MB	15K	Q3, 5, 10, A, B
supplier	29KB	100	Q5, B
nation	4.5KB	25	Q5, 10, A, B
region	787B	8	Q5, B
people	1.6KB	9	PeopleMet
meetings	1.4MB	8.1K	PeopleMet

Our experimental setup included three machines: a 450MHz dual Pentium II data source server. For the experiments on local data we used an 866MHz Pentium III machine with 1GB of Rambus RDRAM memory (but with the query processor's memory pool limited to only 64MB), connected to the source server via a 100Mbps Ethernet. For the experiments on remote data we used a 1GHz Pentium III with 256MB of conventional RAM, connected to the Internet via an AT&T Broadband cable modem, with round-trip times of 30-60ms and bandwidth constrained to a maximum of 1Mbps. We ran all experiments a minimum of 7 times to ensure consistency in the results; 95% confidence bars are also provided.

When comparing CQP with traditional processing (one-pass optimization followed by execution), both approaches used the same cost model and optimization algorithms. This optimizer used bushy tree enumeration in all cases, since we found that left-linear trees never gave performance benefits. In some cases we gave one or both methods cardinality information; in the "unknown cardinality" case, the optimizer assumes a default size of 4000 tuples (roughly the averaged table size for the data set) for every relation. We set our convergent query processor to re-evaluate progress every 10 seconds, and to change plans only if an improvement of 50% is predicted. At each re-evaluation point, full bushy-plan enumeration is performed (in a background thread).

Note that in the experiments presented below, an alternative optimizer might make different decisions based on the input statistics provided, and thus it would have different performance results. The important consideration here is that we used the *same* optimizer and cost modeling algorithms throughout both optimization and convergent

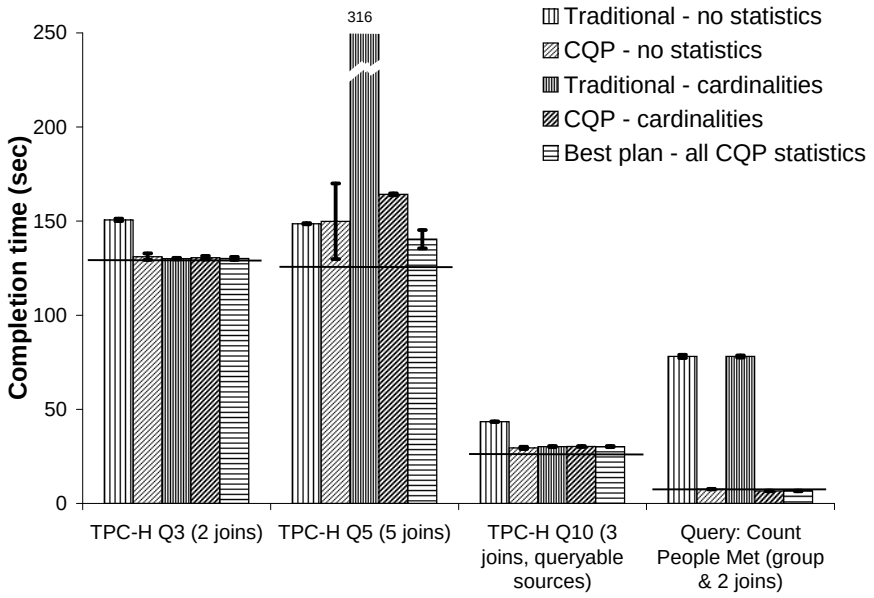


Figure 6.3: Query running times over 100Mb LAN for 10%-scale-factor TPC-H queries (with data converted to XML), plus a query with potential for aggregate push-down. Convergent query processing (CQP) adds little overhead in the worst case, and can produce substantial benefits (even with no *a priori* statistics). CQP performance often approaches that of the best single plan which can be generated given advance knowledge of the complete statistics recorded by CQP execution.

query processing. Every optimizer will make poor decisions in certain cases, but convergent query processing allows it to detect those mistakes and make corrections.

6.4.2 Fast-Network Performance

The first set of experiments focuses on an environment in which query costs are likely to be dominated by CPU speed, and in which costs are highly stable and predictable. Query processing performance should be dependent only on the optimizer's choice of join algorithms and the order in which it applies operations; hence, performance relies almost exclusively on the optimizer's cost model.

For each query, we compared performance of the two strategies with and without cardinality information. The running times are shown in Figure 6.3. As a point of comparison, we also show the performance of the best single execution plan which could be generated using all the information which would have been gathered by a complete execution of the convergent query processor, and we show the XML parse

times as a horizontal line across each set of bars. Currently, the XML parse time is quite significant, but these times will shrink as XML parsing technology is refined — hence, the focus of these experiments is on the running time above the parser line.

== [[[Note to reader: I have already made substantial improvements in Tukwila's parsing and path expression evaluation modules, and plan to re-run these experiments under these conditions.]]]

Queries Q3 and Q10, which have two and three joins, respectively, show that with few joins, a query optimizer is likely to do well when given source cardinality information (with few joins, its estimates of intermediate results should be reasonably accurate), and likely to perform poorly otherwise. We can see that as we would hope, CQP equals the optimizer's best performance, even if it starts with no source knowledge.

Query Q5, which includes 5 joins, illustrates a case in which the query optimizer introduces significant error when modeling complex queries. Here, since the standard optimizer relies on standard System-R heuristics and "magic numbers," it misestimates intermediate result sizes, and chooses a poor plan (whereas, coincidentally, for this query the optimizer does much better when given no source cardinality information). Fortunately, CQP provides good performance regardless of whether it starts with cardinality information. These four queries, which all focus on join re-ordering, demonstrate that although CQP always starts with the same plan as the traditional strategy, it can quickly change its plan as it acquires information, resulting in little performance penalty.

The "Count People Met" query illustrates the possible benefits of Tukwila's support for push-down of GROUP BY aggregation. (The previous TPC-H queries mostly involved foreign-key joins, so GROUP BY optimization makes little difference.) The original work on GROUP BY optimization by Chaudhuri and Shim [CS94] took a conservative approach to push-down, since overly-eager push-down could incur a cost without producing any savings. Tukwila can be much more aggressive in performing early aggregation, because it can monitor the performance and remove superfluous operators. In this query, the "meetings" table had many repeat attendees on its join attributes, so these could be clustered before the join was performed. In addition to the "Count People Met" query, we also ran a several similar count-based queries over the TPC-H data sets, where the joins were over non-key attributes. In these cases, only the CQP algorithm was able to answer the queries in a reasonable amount of time (1.5 to 2

minutes); the other approaches failed to complete within our 10 minute time limit.

A key point about this experiment is that the grouping optimization is one of many techniques that have previously been only applied heuristically, and only in a few cases. On the other hand, the optimization has significant potential benefit. I am currently investigating whether we can leverage other query optimization techniques that are seldom applied; if these can be integrated into CQP, they might be used more frequently, resulting in significant performance benefits.

In summary, CQP never added much overhead, and sometimes yielded significant speedup, even with zero knowledge. We observe that in the network-based query context, cardinalities are likely to be the highest level of statistics that could be made available, since histograms are unlikely to be generated. Even in a local database context, the query optimizer can typically only use histograms to estimate selectivities for the initial join that is performed, and thereafter must rely on predicate independence assumptions and heuristics [SLMK00]. Thus I believe that the results of this experiment also show promise for local queries with full statistics.

6.4.3 Wide-Area Sources

When queries are posed over data sources across a wide-area network, execution tends to become heavily I/O bound, except when intermediate result sizes are large. Wide-area data sources are often bursty, with high latencies and low throughput rates. Under this scenario, the query processor needs not only to produce a plan that minimizes the amount of work needed to answer the query, but it also needs to produce a plan that is "flexibly scheduled" (multithreaded) so the CPU can perform useful work, while waiting for I/O. We make use of the double pipelined join algorithm [WA91, IFF+99] to support flexible scheduling of SPJ queries in convergent query processing.

Even with flexible scheduling and a mostly-I/O-bound system, a good plan can make a difference, since it produces fewer intermediate results and thus requires less CPU overhead. Figure 6.4 shows the results of running our queries over a combination of wide-area and local sources (the `lineitem` table was local, and all others were accessed across a cable modem). While CQP has less effect on the smaller queries (Q3 and Q10), it still significantly speeds Q5 and the Count People Met query.[5] For Query

[5]Note that the absolute numbers in this graph are not directly comparable to those of Figure 6.3, because this experiment was run on a faster machine.

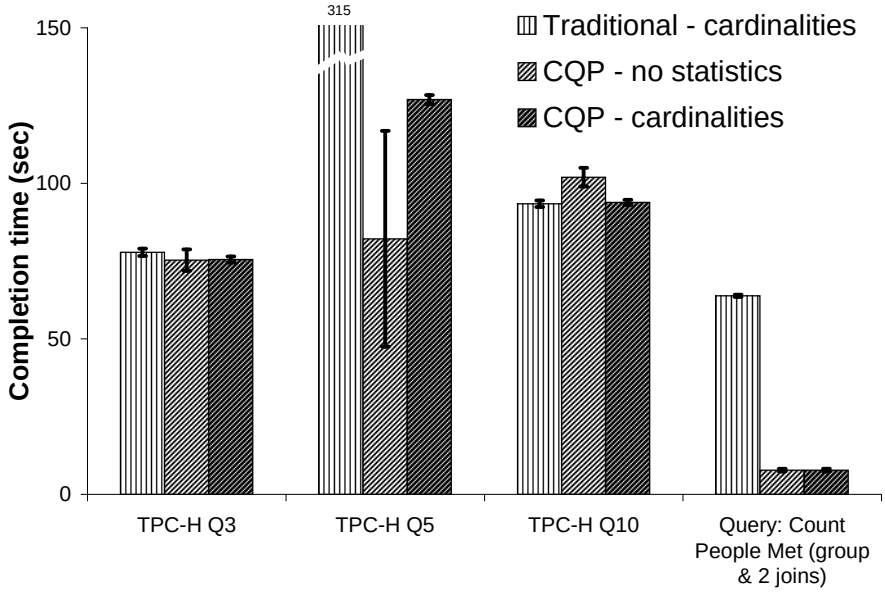


Figure 6.4: Wide-area query running times over AT&T cable modem connection for the queries of Figure 6.3 (with the `lineitem` data source local). In situations where intermediate results are large, even double pipelined joins benefit from convergent query processing.

5, it is interesting to note that CQP performs appreciably better without statistics than with them; in this case the system determines early that the no-statistics plan is bad, but takes somewhat longer to switch away from the original plan generated using statistics.[6]

6.4.4 Constrained Memory

One of the key concerns about convergent query processing is whether it can scale beyond memory; we ran several experiments that suggest that it does. In Figure 6.5, we see a comparison between traditional and convergent query processing strategies for overflow resolution. In both cases, we began with a single pipelined query and attempted to execute it within a particular amount of buffer pool space. For the traditional system, we used the *symmetric flush* overflow resolution strategy of [IFF+99] with 8KB buffer pages. Tukwila, on the other hand, handles overflow by (1) flushing its data structures to disk, (2) re-optimizing, and (3) initiating a new phase. During

[6] Recall that the monitor was set only to switch phases if an improvement of 50% is predicted; hence a better initial plan can hurt overall performance.

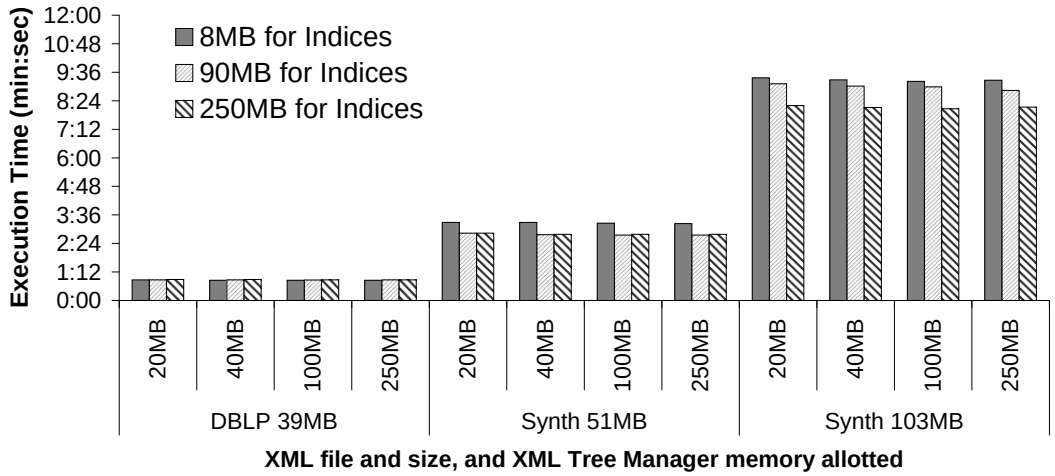


Figure 6.5: Performance of two different queries run with limited memory under traditional overflow resolution and convergent query processing.

cleanup, Tukwila partitioned the query plan into multiple pipelines when necessary to fit any particular join's data structures in memory.[7]

We present the results on two queries. Query A is a modified version of TPC-H Q10, where we removed all selection predicates and returned a number of additional attributes (preventing the system from projecting them away). Both traditional and CQP approaches were provided with source statistics. We executed on a 1GHz machine. Here we find that CQP is slightly faster, but because most of the join work has been completed prior to overflow, both overflow strategies work reasonably well.

Query B is a variation on TPC-H Q5 with several selection predicates removed. We provided the traditional method with cardinalities, but CQP was given no statistics. We ran on an 866MHz machine. In this case, we see that convergent query processing does slightly better with 12MB of memory and slightly worse with 20MB. At 20MB, the traditional incremental, lazy-flushing strategy works better than paging out all data and starting a new phase, but note that CQP was handicapped with no cardinality information.

[7]Note that our cleanup join implementation itself does not yet support partial paging of some of its results to disk — we intend to add this extension in the future, since it would allow us to work in extremely small memory spaces.

6.4.5 *CQP* vs. *Mid-Query Re-Optimization*

We also conducted a preliminary comparison between convergent query processing and the expected performance from Kabra and DeWitt's mid-query re-optimization approach [KD98], which breaks a plan into separate pipelines and re-optimizes at each boundary. Using TPC-H Query 5, we simulated the effects of their approach by taking the output from our optimizer and inserting a materialization point after the first three joins had completed. Note that this neglects mid-query re-optimization's overhead associated with statistics monitoring.

For this query, we found that convergent query processing completed the entire query in the same amount of time, 62 seconds, that it took for that first pipeline to complete — this is because the optimizer has a difficult time determining which sources should be joined in the first pipeline. In contrast, CQP can recover from mistakes without waiting for the pipeline to complete. An additional point worth noting is that convergent query processing had produced a first answer in only 5 seconds, whereas the Kabra and DeWitt approach will not return any answers until it reaches the final pipeline.

6.4.6 *Additional Experimental Conclusions*

For the experimental results presented in this section, the settings of the convergent query processor were kept fixed. However, for completeness we also tried to examine the performance effects of the various parameters. I briefly discuss some of our conclusions here.

Originally, we had expected to implement a number of strategies to reduce optimization and re-optimization times, under the assumption that full bushy re-optimization would be too expensive. Somewhat surprisingly, the overhead of a full bushy re-optimization, when run in a separate thread, was minuscule in these cases. For small queries such as TPC-H Query 3, there was no measurable performance degradation between running the re-optimizer alongside the query, and simply executing the query alone. TPC-H Query 5 is somewhat more complex, with a 6-table join, but still took only $\frac{1}{2}$ second to optimize. Considering that re-optimization has the potential to produce significant savings, we believe that an overhead of a few percent is acceptable in many cases. For queries with many relations, a transformational optimizer may be more appropriate than a bottom-up one, as we discuss in the next section.

Two other parameters that affect performance were the re-optimization interval and the threshold for switching plans. Here, we found that Tukwila and its cost model were surprisingly robust to parameter variations. Individually changing the interval to 2 seconds (from 10) or the threshold to 16% (from 50%) had no effect on the number of phases for most of the experimental queries, and on some runs added an additional phase to Query 3. Changing both in conjunction added at most one phase to Query 3 and two phases to Query 5 in a few runs; but typically, it again had minimal effect. Interestingly, at these levels, some of the running times were slightly better, presumably because the system switched to a better plan at an earlier point.

Our general observation is that the Tukwila optimizer truly *does* converge on a plan, perhaps after a few phase changes. This seems to be fairly stable even with variations in the parameters, even with different sort orderings between tables, and furthermore, the optimizer's cost model does seem to correlate well with real-world performance.

6.5 Conclusion

In this chapter, I have proposed, implemented, and experimentally validated a new model for *convergent query processing*, based on a *phased execution model* that allows for arbitrary transforms to be applied in mid-pipeline to a relatively broad class of queries. Specifically, I have made the following contributions:

- A novel means of executing a query as a sequence of phases, where each phase gets a different partition of the source data and a new phase may be initiated at any point.

- A set of algebraic transformations for a wide variety of operators, which enable these operators to be used in phased query execution, by guaranteeing that the union of the phases' results produces correct results.

- A complete system architecture for supporting convergent query processing, with an implementation in the Tukwila data integration system.

- Experiments demonstrating that convergent query processing provides improved performance in many cases, that it can scale to low-memory situations, that it is low-overhead, and that it does tend to stabilize on a good query plan.

The implementation discussed in this chapter showed significant benefits for querying across a network; however, there is no reason to believe that convergent query processing's benefits are limited to this context. In the near future, I hope to extend the basic work in this chapter to the context of local databases. As in the data integration context, the query optimizer may have inaccurate or insufficient statistics even in the local case, and may choose a poor query plan as a result. A key difference between this context and the data integration one is that query processing in a local database has fewer "idle cycles" in which the CPU can be trying to find a better plan — hence, it becomes increasingly important to have inexpensive re-optimization operations, and to be very selective in performing re-optimizations. However, in this context, there are also a richer set of resources from which the query processor can get statistical information: for instance, histograms, indices, and random access to the data are all available here.

The System-R paradigm of optimize-then-execute, based on statistical information, has achieved great success in the database world. However, it has limitations, such as reliance on potentially out-of-date statistics, exponential increase in error as queries get more complex, and the expectation that costs remain fixed over time. I believe that convergent query processing finally allows database systems to overcome these limitations, and thus to broaden their applicability to new areas, without incurring significant overhead.

Chapter 7

TUKWILA APPLICATIONS AND EXTENSIONS

One of the benefits of developing a working data integration system is that we have been able to deploy it in real-world applications and to use it at the core of other research projects. Most of the current applications are still at a fledgling state, so a detailed case study is not possible at this time — however, in this chapter I briefly discuss some of the applications and how the Tukwila adaptive XML query processor is being utilized. I begin with a discussion of data management for ubiquitous computing and how the Tukwila engine was useful in this context. Section 7.2 describes peer data management systems and the Piazza project, and how we are leveraging Tukwila's basic engine in this system. Section 7.3 describes Tukwila's use in a real-world, biomedical informatics application called GeneSeek, and I summarize the lessons learned from these efforts in Section 7.4.

7.1 Data Management for Ubiquitous Computing

One of the major projects at the University of Washington has been an investigation into ubiquitous computing, the Portolano [EHAB99] project. Ubiquitous computing is based on the premise that an increasing number of devices are being controlled by microprocessors, and that ultimately these devices will be able to communicate via standardized wireless technologies (e.g., 802.11, Bluetooth) or even wired home networks. Once this occurs, it will be possible for different devices to work together to manage daily tasks, hopefully with little or no obtrusive interaction with the user.

Clearly, a major aspect of these interactions is a standardized way of classifying and sharing data. To address this need, we proposed the Sagres[1] data management system for ubiquitous computing [ILM+00]. Sagres provides a virtual, graph-structured view of all data within a ubiquitous computing environment, and allows devices to query and update this data as a means of controlling behaviors. It is based on XML stan-

[1]Sagres was the home of the Portuguese school of navigation, and hence a likely repository of the "portolano" charts used by navigators to explore the world.

dards (so while support has not yet been implemented, it could easily support UDDI[2] and uPNP[3]). A predefined but extensible ontology is used to classify data within the system, and devices associate their data with items in the ontology. Finally, an event-condition-action rule-based system controls interactions between the devices.

Our initial Sagres demonstration showed an example ubiquitous computing environment: we illustrated a scenario in which a business traveler could have a significant part of her scheduling work performed invisibly. The actual system implementation was fully operational except that we simulated the physical devices; our implementation added a simple graph-based storage subsystem and a rule engine to the basic Tukwila query processing architecture. (A real implementation would have needed device support for both queries and updates, but would have otherwise worked similarly.) The primary contribution of Sagres was to show that a slight generalization of data integration could be a very useful paradigm for controlling device interactions — in particular, the uniform interface over all data makes it easy to extend the environment.

In building Sagres, we came to realize that most of the database-related research challenges are not unique to the ubiquitous computing domain. Clearly, at the systems level, there are some unique requirements (e.g., low-power protocols, the need to interact with extremely primitive sensors and other devices), but at the data level, we see three important issues: (1) different devices have different *capabilities* and performance, (2) the different schemas and ontologies of the member devices must be mapped into a common terminology, hopefully in an automated way, and (3) data caching and replication are extremely important. The capability problem already exists in conventional data integration — for instance, some web-based data sources have very limited query capabilities. Similarly, problem (2) is shared with any data integration system, as well as the emerging Semantic Web [BLHL01], which seeks to extend the web with pages that are marked up with terms from various ontologies. The third problem, that of caching and replication (or, more generally, of *data placement and utilization*) is common to distributed databases and to peer-to-peer environments. As a result, we changed our research focus from ubiquitous computing to a more general peer-to-peer data management environment, leading to the Piazza peer data management system,

[2] See `www.uddi.org`

[3] See `www.upnp.org`

which I discuss next.

7.2 Peer Data Management

One of the major new points of focus in the UW Database Group is the area of "peer data management" [GHI+01]. We have proposed the peer data management system (PDMS) as an attempt to provide data integration services in a more decentralized and ad hoc fashion. The current data integration model requires a single mediated schema, which must be defined by the system administrator. As new data sources are added or users need access to new concepts, the schema must evolve — this evolution is much simpler than that for a data warehouse, but it still requires considerable input from a centralized system administrator.

Our view is that this central point of control is not likely to be able to support a series of different schemas that are customized for different applications; that schema modification in a data integration system is too "heavyweight" for simple, ad-hoc collaborations. Hence, we have defined a peer data management system as a query answering and semantic integration system where there is no central schema — instead, each peer in the system can define its own virtual schema as well as provide its own data sources, and it can provide semantic relationships between its concepts and/or those of other peers across the network. A user can select any peer's schema as his or her point of reference and pose a query over this schema; the system will provide all available answers by using the transitive closure of the semantic relationships and their underlying data sources. An example of this is shown in Figure 7.1, where various emergency services integrate their systems, and upon an earthquake a new set of schemas and data sources can instantly join the system and share data.

The PDMS concept is a generalization of data integration that removes the central schema; hence, we build on the basic data integration formalisms for expressing semantic relationships between schemas. Our formal language for semantics subsumes both the local-as-view [DL97, DG97, LRO96, PL00] and global-as-view [Hal01, ASD+91] formalisms of data integration: in essence, it allows us to define the relationship between two schemas by equating a pair of queries, one over each source (or by asserting that a query over one schema is contained within a query over a second schema). Not surprisingly, this formalism as described is too expressive to be decidable: we must restrict our language in a number of ways, such as removing cycles

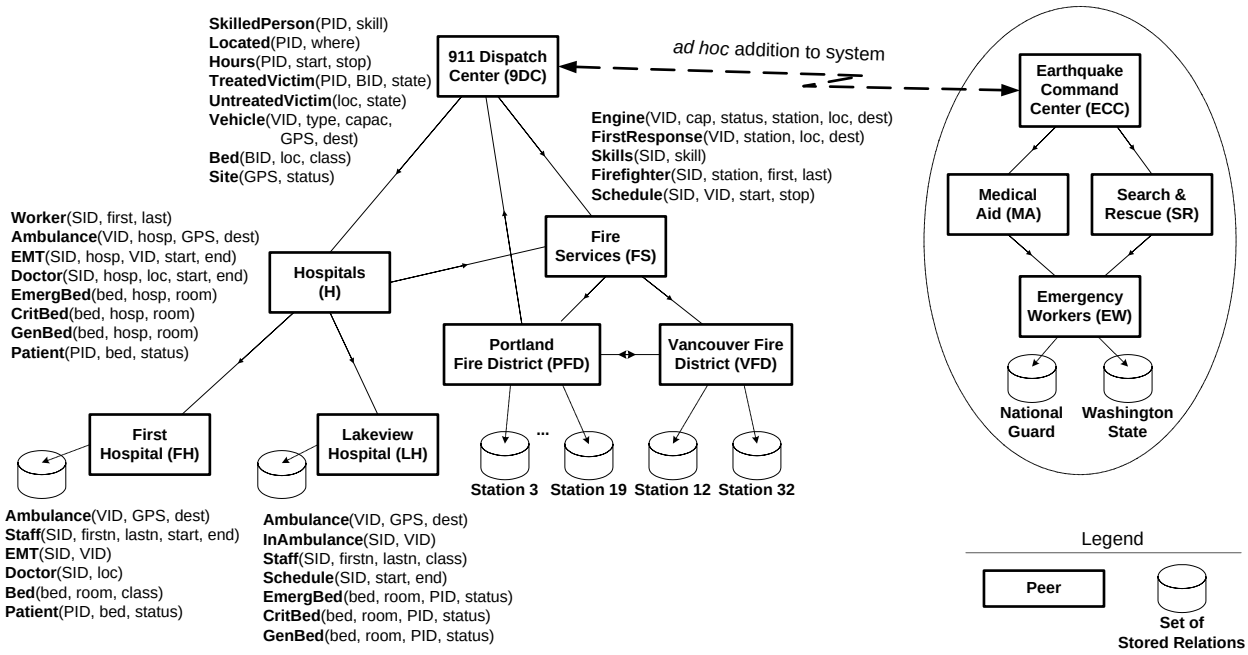
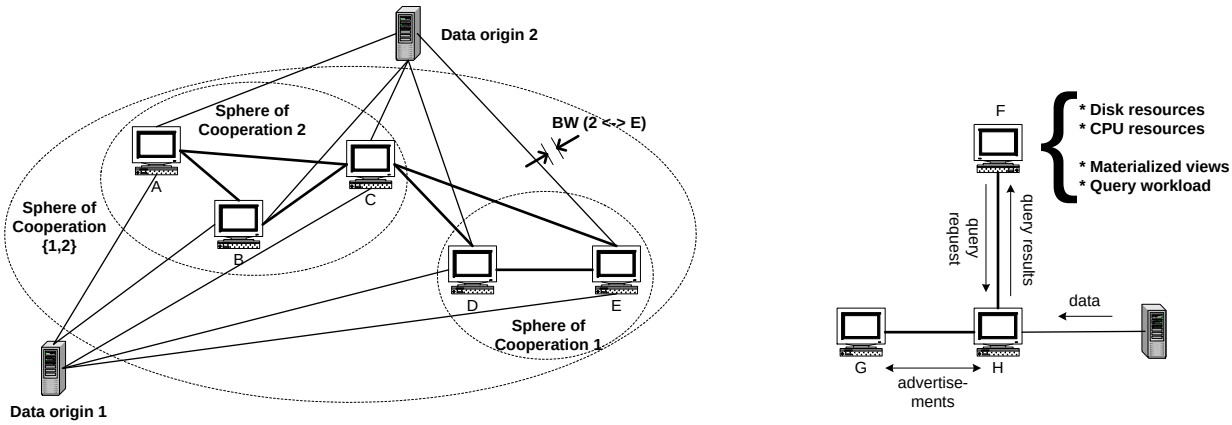


Figure 7.1: Example of schema mediation in a PDMS for coordinating emergency response in the Portland and Vancouver areas. Arrows is the figure indicate that there is (at least a partial) mapping between the relations of the peers. Stored relations are located at various fire stations and hospitals. The hospitals and fire districts run peers within the PDMS, publishing the stored relations for system use. Next, the Hospitals and Fire Services peers mediate between the incompatible schemas at the layer below. Finally, a 911 Dispatch Center provides a global view of all emergency services. In the event of an earthquake, a new Command Center and new relief workers can be added on an ad hoc basis, and they will be immediately integrated with existing services.



(a) System Diagram

(b) Detailed View

Figure 7.2: Piazza system architecture. Data origins serve original content; peer nodes (A-E) cooperate to store materialized views and answer queries, but have limited disk and CPU resources. Nodes are connected by bandwidth-constrained links, and advertise their materialized views. Nodes belong to *spheres of cooperation* with which they share resources; these spheres may be nested within successively larger spheres.

within mapping definitions. For more details and an experimental evaluation of an algorithm for query reformulation in a PDMS, see [HIST02].

In terms of integration, peer data management poses a number of challenges that exceed those of standard data integration. First, decentralization and scalability are important considerations in a true peer-to-peer environment. Decentralization is important because it removes the possibility of a single point of failure, and also a single machine can no longer be a scalability bottleneck on the entire system. However, it has been shown that decentralization often results in a *less* scalable system than centralization: some decentralized systems, such as Gnutella, are severely limited in their scale because they require too many "ping" and query messages for coordination and operation [SGG02, DSS00].

Current research in the Piazza PDMS project (see Figure 7.2) attempts to define a scalable, decentralized set of protocols and algorithms for coordinating between and optimizing across peers. The current architecture assumes a simplified model in which most decisions are made at the scope of a single peer or cluster of peers ("sphere of cooperation") — avoiding the need for complex distributed query optimization and wide-

area distributed decision making — but we initially assume the presence of global knowledge about peer resources and network connectivity. Our next step will be to migrate to an architecture that assumes only partial knowledge of global state.

Our current focus is on defining a mapping language for XML-based peers (extending the work of [HIST02] from the relational context), query rewriting and reformulation algorithms for this language, and the data placement and query optimization techniques that will allow our Piazza PDMS to replicate data throughout the system and use it efficiently. The core query processor in Piazza is the Tukwila XML processing engine, and the new query optimizer will leverage the techniques presented in this thesis.

Anther PDMS topic of immediate interest is understanding the relationship between a P2P query answering system and the description-logics-based "semantic web" proposed by Berners-Lee and others [BLHL01]. Our formalism, which is based on the datalog language, has a different means of expressing concepts from description logics (e.g., containment in datalog corresponds to subsumption in description logics). It is generally more tractable to answer queries in a datalog context, and we believe that database-style operations are likely to be the highest level of functionality that is important in many applications.

7.3 Integration for Medicine: GeneSeek

The GeneSeek project [MHTH01] is a data integration system designed to integrate biomedical databases (in this case, databases related to genetics). GeneSeek queries phenotype, gene, locus, and protein information from numerous public web sources such as OMIM[4] and LocusLink[5]. The basic approach of GeneSeek is to provide a global, graph-structured mediated schema incorporating all of these data sources, and to accept queries in a high-level declarative language called PQL [MSHTH02].

GeneSeek models individual entities (e.g., genes or proteins) as nodes and relationships as edges. In contrast to conventional tree-structured XML, the GeneSeek model is truly a graph, and edges express a type (e.g., consists-of or codes-for) as well as a relationship. Since these logical concepts can be physically encoded as XML (e.g., edges can be modeled as XLinks or `IDREF`s), the authors do make use of XML as the physical

[4] http://www.ncbi.nlm.nih.gov/

[5] http://healthlinks.washington.edu/basic_sciences/molbio/

data representation, but the user perceives the higher-level logical encoding.

All queries over the graph model are expressed in PQL, a semistructured query language with many similarities to StruQL [FFK+98]. A key difference between PQL and StruQL is that GeneSeek supports additional metadata about particular sources and relationships (e.g., its level of curation), and PQL queries can be restricted based on this metadata. At times, this capability greatly improves the relevance of the answers.

A PQL query over the mediated schema is converted into an XQuery over the data sources (or, more precisely, over the XML from their wrappers). This query is fed into the Tukwila query processor, which fetches data from the wrappers and combines it to produce the answers.

Since the GeneSeek data sources are extremely large and the wrappers do not warehouse the XML contents of the sources, GeneSeek relies heavily on pushing selection predicates into the wrappers and on Tukwila's *web-join* operator — which allows the system to fetch a data item from one source and use it to build a query over a different source. Without the web-join, the amount of data being transferred across the network would be prohibitively large, but with it, wrappers typically only return the ten or so answers that are relevant to the query.

While GeneSeek has not yet been heavily deployed in practice, it has shown itself to be useful in the biomedical domain, and it demonstrates the utility of the Tukwila system. Furthermore, it illustrates the need for an operator like the web-join, which can compose dynamic queries based on the contents of a particular data source.

7.4 Summary

As this chapter demonstrates, the Tukwila query processor has evolved from simply a research platform for investigating adaptive query processing to a component of a number of applications and new research systems. Many data management tools and applications need a query answering engine, and our core not only provides good performance for network-bound data, but it is highly flexible, modular, and extensible, and hence an ideal tool for distributed data management.

Several aspects of the Tukwila architecture served us well in our applications. First, the core engine was designed in an extensible way, which made it easy to add features such as active rules. Second, the core modules provided standards-based interfaces for easy coupling with other components — the execution engine has a SOAP-

based API for accepting query plans from an external optimizer, or it can do its own optimization of abstract syntax trees created from XQueries. Data sources can be files or HTTP servers. Any XML-like query language and reformulation algorithm can be layered over the query optimizer, so long as it can be translated into XQueries over XML sources. Third, the level of performance provided by the Tukwila infrastructure was sufficient to solve the needs of applications in these domains.

Ultimately, I believe that a natural progression would lead to an even more flexible version of Tukwila. Tukwila is an adaptive query processing engine for network-bound data — it has no storage subsystem, nor support for indices. I believe that it would be very useful to take this core engine and re-factor it in a way that will allow it to be combined with storage systems (e.g., Shore [CDF+94]) to build a full adaptive data management system — I believe that the adaptive features of Tukwila can be of use in the local context, and extended to concepts such as physical storage and data layout, and that many interesting applications can be built from this.

Chapter 8

RELATED WORK

I have discussed a number of related works earlier in this dissertation, but in this chapter I present a more global view of the topics covered by my thesis and the related work in each area. I begin in the first section with a discussion of data integration, then move on to discuss XML query processing in Section 8.2, and I conclude with a discussion and comparison of adaptive query processing techniques in Section 8.3.

8.1 Data Integration (Chapter 2)

Data integration has been a topic of study that actually dates back to the MULTIBASE system [SBD+81] of the early 1980s. Autonomy and schema heterogeneity are often characteristics associated with federated databases, as well (e.g., the wide-area Mariposa system [SAL+96], or the federated capabilities now built into Microsoft SQL Server). However, key differentiating characteristics of data integration systems include the ability to work with non-database data sources (e.g., web sources) as well as data sources not built to work together: data integration is typically retrofitted over existing sources that were never designed to interchange data with other sources in different schemas.

The advent of the World Wide Web and the Internet were responsible for popularizing the data integration concept during the mid-1990s. In fact, data integration was a point of interest not only to database researchers, but also the AI community: it was a logical progression of the "softbot" information-gathering agent efforts pioneered by Etzioni and Weld at the University of Washington [EW94], as well as a subject of interest to many other groups. Thus, projects such as Occam [KW96], Razor [FW97], SIMS [AK93], and Ariadne [AAB+98] all arose from the AI community. These efforts generally focused on the planning aspects of data integration, i.e., on finding legal query plans (given restrictions on the input required by data sources), and on optimizing these plans. SIMS and Ariadne mapped sources to a description logics terminology for schema mediation, and they used inference and planning techniques to do query

translation.

Another topic of focus, particularly in the AI community, was the development of wrappers, which convert data and queries between the formats of the data sources and data integration system. Techniques have been developed for automated learning of wrappers [KDW97, AK97, BFG01] from user-provided examples, and various toolkits have been proposed to speed up developer programming of wrappers [SA99]. Note, however, that the need for wrappers today has diminished because of XML's advent — wrappers are now primarily needed only for legacy applications.

The database community had been working on efforts to distribute and federate databases for many years even before the Web phenomenon. Early efforts included distributed databases such as System-R* [ML86] and Distributed INGRES [ESW78], which extended the basic techniques of System-R and INGRES for databases distributed across LANs[1]. Mariposa was a descendent of INGRES that attempted to truly distribute to the wide-area, and it made use of an economic model of bidding for services as its basic infrastructure; unfortunately, the model was not tremendously successful, and this was one of the few Stonebraker projects whose technology was never adopted in industry. Instead, the database industry developed simpler distributed versions of Oracle, DB2, and Informix more like the System-R* model. Additionally, standards such as ODBC [ODB97] and OLE-DB [Bla96] were proposed for allowing for limited interoperability between databases and certain types of data exchange with applications.

A few early data integration systems, such as Pegasus [ASD+91], appeared in the very early 90s, but as with the AI community, it was the advent of the Web that triggered significant interest in integrating data. Significant projects included the Stanford TSIMMIS [GMPQ+97] project, which was one of the first systems to propose a semistructured data model as a means of schema mediation; Hermes [ACPS96], IRO-DB [GST96], and Disco [TRV98], which generally focused on learning expected performance levels from sources and on handling source failures; and Carnot and Infosleuth [WBJ+95], which were data integration systems for deductive databases. All of these efforts used the global-as-view approach to schema mediation: the mediated schema is defined as a view over the data sources. This has the benefit of allowing sim-

[1]Note that a distributed database is neither heterogeneous nor autonomous: the database administrator partitions data across a set of servers that all run the same database software and contain portions of the same database.

ple "unfolding" of the views (view definitions can be macro-expanded into the query), but it has a trade-off that the mediated schema may have to be revised anytime a new source is added, and it also has difficulty representing incomplete data sources.

The Information Manifold [LRO96] was significant because it popularized the local-as-view formalism, which gave significantly greater flexibility (addition of data sources generally does not require revisions to the mediated schema, and incomplete data sources are easy to model) at the cost of a new query reformulation step. The authors of the system proposed the bucket algorithm as a way of reformulating mediated queries to refer to the underlying sources. (Note that Razor, which appeared slightly later, was the corresponding system from the AI community that made use of local-as-view. It used the inverse rules algorithm of [DL97] to reformulate queries.) The Tukwila system uses the same basic approach as these systems, but focuses on the problem of providing their capabilities at high performance and at scale.

In the commercial world, integration of web sources has been successful in a few isolated cases (e.g., comparison shopping services, such as those provided by `shopper.com` and formerly provided by companies such as Jango and Junglee) — but most of these systems relied on hand-coding rather than true data integration systems (other than Jango, which did make use of a global-as-view integration system). Full data integration systems have been much more successful in the enterprise, in e-commerce infrastructure, and in efforts to manage life-science data: for instance, IBM Almaden's Garlic data integration technology [HKWY97] has been incorporated into DataJoiner and later DB2; companies such as Nimble Technology and Enosys Markets are attempting to market their data integration tools to large organizations. Most CTOs of large firms seem to agree that there is a need for a data integration infrastructure, but to this point few have committed the financial resources to it. This movement seems inevitable, however.

Continuous Queries and Information Dissemination

Concepts that are vaguely related to data integration, but which typically have no concept of schema mediation, include continuous queries [LPBZ95, CDTW00, Aea01, MSHR02] and information dissemination systems [FZ96, AF00, GMOS02].

A *continuous* or *continual* query is a view defined over an external data source. Whenever the external data source changes, the results should propagate to any user

who subscribes to that view. This concept was first proposed by Liu and Pu in [LPBZ95], and has been adopted by the NiagaraCQ [CDTW00], Xyleme [Aea01, MAM01], and Telegraph [MSHR02] projects. Challenges include handling large numbers of continuous queries simultaneously (since many users may be posing very similar continuous queries) and performing the minimum amount of result computation necessary (perhaps using differential evaluation instead of complete query re-evaluation, or employing adaptive techniques like eddies).

Information dissemination systems are similar in concept but are not restricted to a single data source. Here, users register *profiles* with a central server — a profile generally contains one or more queries expressing data of interest to the user. The information dissemination system is constantly receiving new data and documents. Each new document gets matched against the profiles of every user, and the document gets disseminated to those users to whom it may be of interest. As with continuous queries, the challenges mostly lie in executing huge numbers of queries in parallel and in routing the data to the appropriate users. However, information dissemination systems typically do matching and routing rather than full query processing — hence, queries in this environment consist of selection-like operations and little query computation is required.

8.2 XML Processing (Chapter 4)

Although XML query processing is currently a hot topic in both research and the commercial world, it is a young field and no one really knows how pervasive XML will ultimately be — nor how it will be primarily used. XML is both a document representation format and a data representation format. It now has a data model, so it can be the default representation of data; but perhaps it will simply be a "wire format" for data sharing. If it is a wire format, what will the producers and consumers be — databases, web browsers, web servers, etc.?

At one point, XML was envisioned as the successor to HTML: all web sites would serve semantically richer content, and all web browsers would combine XML with XSLT [XSL99] to render the content appropriately. Due to a combination of factors, this transition appears to have lost momentum. First, until very recently, the Netscape family of browsers did not support XML processing — so a significant part of the web client population could not render XML. Second, there is a large legacy document base

in HTML, and in fact documents are much easier to create in HTML (one need only consider presentation, rather than semantics, in building a document). Third, many commercial web sites have a proprietary interest in their data, and would prefer to hide rather than reveal its semantics.

Anecdotally, XML seems to be achieving significant success in two general areas: as a replacement for SGML in document databases and as a data interchange "wire format". However, development efforts seem to be occurring in all aspects of XML usage, and it still remains to be determined which of these will have the most impact.

The document or text database community has generally fallen outside the mainstream SIGMOD audience, largely because they have a different set of requirements than typical relational-model databases. Rather than joining tables, most document database queries perform approximate keyword matches within a particular context, or other similar operations. Products such as Documentum[2] and the DOCS family of products[3] are examples of document databases that are generally used by libraries, organizations with large numbers of reports (e.g., law firms), and news agencies. Web search engines such as Google[4] have many similarities to document databases (especially since they often cache and retrieve documents). Several commercial and research projects focus on the problem of querying across large numbers of text-oriented XML documents (either stored in a local repository or distributed across the web but indexed locally), including Xyleme [MAM01] and its storage system, Natix [KM00], Niagara [NDM+00], and eXcelon [XLN]. Interestingly, while many of these systems attempt to offer text document query capabilities, these are minor extensions to XPath or XQuery (perhaps with exact-match keyword search): there is no provision for the approximate-match querying supplied by document databases. In many ways, they are not full document databases, and they are better suited to querying more traditional semi-structured data.

The problem of querying across large numbers of XML *data* fragments, typically using XQuery, has been a popular one. Examples of these systems includePoet software's XML repository[5], as well as Niagara, eXcelon, and Xyleme (which also query

[2] http://www.documentum.com

[3] http://www.hummingbird.com

[4] http://www.google.com

[5] http://www.poet.com

documents, as described above). Their emphasis is on scalability to many small XML fragments, rather than large data volumes. Queries can generally be posed across numerous documents, and indexes are used to find the relevant documents. Niagara and Xyleme allow triggers to be set over external documents, so actions can be initiated if those documents are changed (forming continuous queries).

Moving up to larger volumes of XML, there are a number of efforts to construct scalable XML databases, including Timber [AKJK+02], Toxin [BBM+01], later versions of Lore [GMW99], and numerous efforts to automatically store and retrieve XML data with relational systems [CFI+00, SGT+99, FK99a, DFS99]. Also, several projects that focus on small XML documents, such as Xyleme, also scale to large data volumes. Most of these systems are adaptations of object-oriented or semi-structured storage systems, and make use of path-oriented indices such as join indices [Val87], access support relations [KM90], DataGuides [GW97], and t-indices [MS99].

Interestingly, the problem of querying XML data as it is streaming "over the wire" has been a relatively neglected problem. Some limited query capabilities have been provided by XSLT processors such as James Clark's XT[6], Microsoft's MSXML[7], and the Apache Xalan system[8]: these systems can manipulate in-memory representations of a single document. Several data integration systems also address unique problems in this domain. Niagara can compute partial answer sets for a given query, which may be desirable for interactive query scenarios when a user is not interested in the full answer set. MIX [BGL+99] supports demand-driven evaluation of XML content — only requesting the data from a wrapper if it is to appear in the query. The Agora system [MFK+00] can convert from XML structure directly into "edge relations" to perform query processing, but then it must re-construct the XML at the output. Tukwila supports incremental, pipelined evaluation of queries and dynamic requests for content using its x-scan and web-join operators, and it does not spend time de-constructing and re-constructing XML hierarchy.

Finally, there is considerable interest in converting data from traditional systems into exportable XML views, since the majority of queryable data lies in relational databases: SilkRoute [FTS99] and XPERANTO [CFI+00] support creation of XML

[6] http://www.blnz.com/xt/

[7] http://www.microsoft.com/xml

[8] http://xml.apache.org/xalan-c

queries and views over relational systems, and IBM [Tre99], Oracle [BKKM00], and Microsoft [Rys01] all support some XML query and export features in their products. These systems are very useful for exporting data into XML to facilitate data integration; they assume all data starts off in relations, and users construct (and then query over) progressively more hierarchical XML views of that data. At some point, all relational database systems are expected to directly support an XQuery interface in parallel with the existing SQL interface, and both front-ends will feed directly into the same query optimizer and execution engine. For now, the research and development efforts revolve around supporting XQuery through middleware: a translation layer focuses on efficiently rewriting XQueries posed over XQuery view definitions into SQL statements over relational tables.

I personally believe that the last two problems — that of querying XML "over the wire" and of exporting data into XML form — are likely to be the ones with the most impact, at least in the short term. These are the solutions that will enable B2B e-commerce, collaborations between groups, and so forth. However, for such data interchange to really succeed, we will need significant standardization of schemas (as organizations such as OASIS are trying to promote) or, even better, development of sophisticated semi-automated schema mapping and management tools (e.g., [DDH01, BHP00]).

Work Related to X-scan

The use of finite-state machines in matching regular-expression-like patterns is not unique to x-scan. For instance, the Knuth-Morris-Pratt substring-matching algorithm uses finite state machines to perform matches. Likewise, the XFilter operator [AF00], developed in parallel with our algorithm, also uses a similar technique for filtering XML documents according to an XPath expression, returning a boolean value rather than a tuple stream, and providing the basis for an efficient document dissemination system. Later work has been done by Suciu et al. on extending the basic ideas of XFilter with "lazy" conversion of nondeterministic automata to deterministic ones [GMOS02].

The novelty of x-scan lies in two features. First, so far as we are aware, our algorithm is the first to use state machines as the basis for incrementally extracting data for pipelining complex queries — x-scan returns a tuple stream, not a true/false flag. This makes a tremendous difference in enabling incremental query processing, and it

also introduces a number of challenges. In particular, our algorithm must support hierarchies of bindings and path expressions, and in graph-structured mode, it must also traverse references and avoid cycles. Furthermore, the use of inlining and sargable predicates can make a significant difference in overall performance.

It is worth noting that an unpublished work by Cluet and Moerkotte from several years ago proposed a logical *scan* operator, which had similar characteristics to x-scan, as the basis for tree-structured querying. We can now complement that work with a specific algorithm, support for graph-structured data, and an experimental evaluation.

Infinite Streams

A topic of recent popular interest is that of "data streams" [BW01, DGGR02b, MSHR02, GGR02, SV02]: usually, a data stream is expected to be a data set that is infinitely large (e.g., data from sensors), and the query processor is expected to have fixed storage. Some recent work at the University of California-San Diego [SV02] has been investigating the use of x-scan like finite automata for processing streams; in particular, they hope to specify the maximum level of expressiveness that can be implemented in an automaton-based query algorithm that has fixed state.

8.3 Adaptive Query Processing (Chapters 5, 6)

The use of adaptive capabilities in query processing actually dates back to the original relational systems (and predates the use of cost-based query optimization). The INGRES query processing algorithm originally interleaved steps of constructing a query execution plan and executing it [SWKH76]. Its approach was to decompose a query into a subquery with a single variable, and then execute this portion; later portions of the query can be executed using substitution of results from the first step, and so on. This approach was quickly eclipsed by the less flexible but more elegant, cost-based System-R style optimizers. System-R introduced the concepts of precomputing statistics offline and using dynamic programming and cardinality estimation to find a promising query plan. It also introduced commonly used heuristics such as supporting only left-linear trees to reduce the search space and postponing cross-products to the end of execution[9].

[9]Note that neither of these heuristics is something that should be used in all common query processing scenarios. Research in [HKWY97, FLMS99] has shown that exhaustive enumeration of plans may be

Unfortunately, it has been shown that the static optimization approach has a number of weaknesses, mostly due to insufficient statistics about the underlying data. For instance, histograms on tables are typically one-dimensional only, which misses correlations between attributes — hence, the optimizer must make an independence assumption when joining tables. A number of works have shown that the error in an optimizer's cardinality estimates grows exponentially with the number of joins [IC91], and that predicates with conjunctive or disjunctive expressions also introduce further error [AZ96]. In response to these problems, a number of solutions have been proposed. The first of these is to adapt and refine statistics based on the results of past queries: this approach is adopted by work by Chen and Roussopoulos [CR94], who focus on improving selectivity estimates; IBM's DB2 LEO [SLMK00], which infers "adjustment factors" for joins to estimate the correlation between predicates; and Microsoft's SQL Server AutoAdmin [BC02], which uses statistics on subquery expressions to better model larger queries' costs.

Another approach is to interleave query plan selection and execution: this approach was first implemented by Graefe and Ward [GW89, CG94] with *choose nodes*, where a query optimizer could create several "contingent plans" and the execution engine could select one at runtime, based on pre-execution conditions. A "lazy evaluation" version of this is the dynamic re-optimization technique of [KD98], which instruments a query plan with statistics collectors and, if the optimizer estimates are sufficiently bad, triggers a plan re-optimization at materialization points during execution. Kabra and DeWitt retrofitted this interleaving capability into the existing Paradise system, so they added a separate component between the optimizer and execution system to determine what adaptive features should be added to the plan. Our initial implementation of the Tukwila system [IFF+99] also supported a similar capability, but we built the adaptivity logic into the query optimizer because the optimizer had more information that it could use to reason about good re-optimization points. However, we found that this approach posed a significant challenge: it was necessary to add a materialization point so a re-optimization could be initiated if necessary, but this could detract from performance if the query plan did not actually need to be re-optimized at runtime. Furthermore, this sort of plan partitioning requires good knowledge about the

---

required in data integration, and star-schema decision-support queries are often more efficient if the cross-products are executed first.

underlying data sources so the initial stage of execution can be well-chosen, and our optimizer did not always have such knowledge.

A third approach to "dynamic optimization" was described by Antoshenkov [AZ96] and implemented in DEC (later Oracle) Rdb: running several alternative query plan subtrees in parallel competition. The system would keep the plan that appeared to be progressing the fastest, and would terminate the others. This approach works well if the system can determine what the "best few" plans are, but it assumes consistent performance throughout execution and cannot adapt to mid-execution changes.

The techniques cited above mostly concentrate on improving query optimization for local data. Another focus has been on processing remote data: early works such as System-R* assumed local-area networks and consistent performance, but many of today's query processing projects focus on querying across the wide area, which introduces new problems such as variable transfer rates, intermittent connectivity, and lack of data source statistics due to autonomy.

Works such as [GST96, ACPS96, ZRZB01, BRZZ99] attempt to learn performance patterns for different data sources; they make the assumption that performance is regular, at least for a given time of day or day of the week, and they attempt to model these patterns. Several pieces of work have also focused on the problem of handling source failures, e.g., [TRV98, BT98], and in general the approach is to present partial answers if necessary. Many data integration systems encounter similar problems with source availability, but simply switch to alternate web sources as necessary. This is one of the functions of Tukwila's collector operator.

Less severe than source failure, but still likely to impact query performance, are source delays and source burstiness. Here, we would like to schedule any other available work while waiting for a data source. One solution to this problem is query scrambling [UFA98], which attempts to execute alternate portions of the query plan upon an initial source delay. A second approach is the adaptive scheduling technique proposed by Bouganim et al. [BKV98], which schedules the work of a separate query process during a source delay — if a large number of concurrent queries is being executed, this improves overall throughput, although it does not improve the response time for the delayed query[10]. The third approach is to use a variant of the pipelined hash

[10]We observe that this has parallels in the computer architecture community: most of the adaptive scheduling techniques discussed in this paragraph are analogous to out-of-order execution, but the Bouganim approach is similar to simultaneous multithreading, which executes a context switch on a

join [RS86, WA91, HS93], which allows both of its child subtrees to execute independently (and thus in parallel, so one subtree's execution thread can be doing work as the other is delayed). Numerous extensions and adaptations to this algorithm have been proposed [UF00, HH99, IFF+99], mostly focusing on prioritizing the number of CPU cycles given to each thread or providing support for dynamic destaging of results to disk on a memory overflow. Our implementation of the double pipelined join was one of these extensions.

The ripple join of [HH99] looks at prioritizing reads from one portion of a query plan versus another to improve estimates for aggregate values. Along a similar vein, Urhan and Franklin propose dynamic pipeline scheduling in [UF01] for prioritizing answers in query processing. Both of these works are useful when CPU resources are sufficiently constrained that not all data can be processed at the rate it comes in (and this is typically the case except for simple queries with slow data sources).

A third class of adaptive approaches allows for dynamic re-optimization of queries in mid-execution (rather than at pipeline boundaries). The first such implementation is the eddy [AH00]. Eddies are a novel technique based on *flow* of tuples through operators, rather than on more traditional query optimization cost metrics. An eddy consists of a number of select and join operators; each tuple is routed through all of the operators in a sequence governed by the flow rates through the operators. In effect, the eddy sends each tuple through a different query plan formed by commuting its constituent operators. The eddy approach has recently been further extended with STeMs [RH01], which expose the individual data structures of individual operators within the eddy. With STeMs, each join operator within the eddy is effectively "split" into two mostly-independent data structures (one for each input source). Each of these data structures has a single input and a single output, and the eddy's dataflow logic can make more appropriate decisions than it could with the binary join operator. Furthermore, multiple alternative STeMs, each implemented with a different algorithm, can be executed in parallel on a single join operator — the eddy has a choice of which STeM to use, which is equivalent to choosing between different join algorithms.

Unfortunately, even with STeMs, eddies appear to be limited in several key ways. First, all decision-making within eddies are done based on flow, which is highly local in its scope and limited in its ability to deal with delays. Currently, decision-making

---

delay.

must be done in a per-tuple basis, although the Berkeley database group is changing to a more "block-oriented" approach. Second, the current state-of-the-art eddy implementations are far less efficient than standard query execution trees — so if the eddy's adaptivity is not needed, it will be adding significant overhead. Third, eddies are currently only able to deal with select-project-join queries.

My interest has been in providing a solution that, like eddies, can adapt a plan in mid-execution — but that can make decisions at any level of granularity (from per-tuple up to per-pipeline or even per-query); that can be expanded to include more sophisticated transformations, such as push-down of grouping operations; that can make use of any existing knowledge in choosing its initial plan; and that can rival the performance of standard query processing when adaptivity is not needed. I believe the convergent query processing framework gives us this flexibility, and that my initial experiments have demonstrated that the basic techniques work well even with standard System-R-like query optimization and standard iterator-driven query execution.

Categorizing Adaptive Techniques

To summarize the various adaptive query processing techniques at a higher level, we can examine them along a number of dimensions:

- **Frequency of adaptivity**: how frequently does the technique modify query execution?

- **Power of adaptivity**: does the technique support only a change in scheduling (i.e., suspension of one part of the query plan so another part can be run), commutativity of operators in the query plan, or complete plan transformations (including operators, implementations, and scheduling)?

- **Scope of decision-making**: is the decision made only on an intra-operator basis, a subplan basis, or along the whole query?

- **Overhead of adaptivity**: how expensive is it to adapt execution?

- **Cost of decision-making**: how expensive is it to determine when to adapt?

Table 8.1: A comparison of the common adaptive query processing techniques. "Adapt statistics" includes the statistics-adapting techniques of Chen and Roussopoulos, DB2, and SQL Server. Costs listed are general qualitative assessments and difficult to measure quantitatively.

Technique	**Freq.**	**Power of Adaptivity**	**Scope of Decisions**	**Overhead of Change**	**Cost of Deciding**
INGRES	Operator	Commute some ops	Subtree	Medium	Medium
Adapt statistics	Query	Revise cost model	Query plan	Low	Low
Choose nodes	Query	Choose subplan	Optimizer-limited	Low	Minimal
Rdb Dynamic Optimization (Antoshenkov)	Query	Choose plan	Query plan	High	Low
Dynamic re-optimization (Kabra/DeWitt)	Pipeline	Re-optimize unexecuted	Unexecuted pipelines	High	High
Query scrambling	Delay	Reschedule/ add join	Pipeline schedule	Medium	Low
Pipelined hash join	Tuple	Schedule child	Input. schedule	Minimal	Minimal
Dynamic rescheduling	Tuple	Prioritize	Pipeline schedule	Minimal	Minimal
Ripple join	Tuple	Prioritize	Pipeline schedule	Minimal	Minimal
Eddies + STeMs	Tuple	Change SPJ plan	Internal schedule	Low	Low
Convergent query proc.	Varies	Change SPJGU plan	Varies	Medium	Varies

Table 8.1 compares a number of adaptive query processing techniques along these dimensions. I believe that convergent query processing subsumes aspects of many of the previous techniques: the implementation and experimental evaluation of Chapter 6 only used specifiable time intervals and global re-optimization in its decision-making, but the basic framework is easily scalable to more frequent re-optimizations, as well as more local transformations and more global optimizations. I believe there is enormous potential for using adaptive re-optimization on queries, and I am already in the process of evaluating some of these points in the spectrum of adaptive query processing, as I discuss in the next chapter.

Chapter 9

CONCLUSIONS AND FUTURE DIRECTIONS

Over the past twenty years, the database community has largely relied on the same architecture for query processing: offline statistics gathering, static query optimization based on these statistics, and execution of the query plan using deterministic, iterator-based algorithms. This architecture has been quite successful in many situations, and the result has been a very large commercial market for database systems.

However, today many query processing experts (e.g., [Win02]) agree that standard query optimization has reached its limits and needs to be re-thought: many sophisticated query optimization techniques must be enabled or disabled by hand, because standard query optimizers do not have sufficient statistics to determine whether to use them; and in general, query optimizer cost models are extremely inaccurate for complex queries [IC91, AZ96, CR94, SLMK00, BC02, KD98, AH00]. Furthermore, deterministic, iterator-based scheduling of query operators is also too inflexible for network-based data, as demonstrated by [UFA98, UF00, UF01, AH00]. Finally, more adaptive scheduling possibilities and better interactive response are enabled by pipelined query execution. Thus, in this dissertation, I have proposed a set of adaptive techniques for query processing: adaptive re-optimization using convergent query processing, adaptive scheduling using the double pipelined join and collector, and pipelined execution for XML using the x-scan-based Tukwila execution architecture. As a part of my thesis work, I have implemented these techniques in the Tukwila data integration system. From this, we have learned a number of lessons.

Pipelined processing of streaming data is beneficial for performance and flexibility. In developing the x-scan algorithm and our pipelined XML query processing architecture, we expected better time to first answers, due to the approach's incremental nature. However, we found that our algorithm was sufficiently efficient to give better *complete* answers as well. Furthermore, pipelined execution allows for more flexible scheduling in query processing.

Adaptive scheduling and overlapping I/O with computation are necessary for wide-area data. Algorithms such as the double pipelined join produce better performance because they can use I/O delays to perform computation in other parts of the query plan. Furthermore, the double pipelined join outputs tuples much earlier, which means more data can be fully processed early in query plan execution.

Adaptive re-optimization can be done frequently without wasting significant work. Our convergent query processing formalism exploits the distributivity of the union operator to split query execution into phases at virtually any point. By reusing intermediate results, in most cases we can avoid wasting work.

An architecture that supports re-optimization can be efficient. In contrast to an approach like the eddy, which makes heavy use of multiple threads and queues to support re-optimization, our approach preserves the efficient aspects of iterator-based query execution: re-use of memory space between operators, support for deterministic or non-deterministic scheduling of operators, and pipelining.

Adaptive features provide performance benefits. We have experimentally demonstrated that the adaptive features of Tukwila provide superior performance to similar non-adaptive implementations, especially for the data integration context.

Tukwila is a flexible and useful software artifact. Tukwila has proven to be robust and flexible enough to be used in numerous applications: it has also been deployed in the GeneSeek project, a real-world medical informatics application, and it has been used as a core component in the Sagres data management system for ubiquitous computing and the Piazza peer data management system.

9.1 Future Work in Adaptive Query Processing

I believe that my work has made significant advances towards solving the problem of processing queries in zero-knowledge or little-knowledge environments. However, there are two very important directions in which the convergent query processing techniques should be further extended.

9.1.1 More Sophisticated Transformations

A number of complex transformations have been defined for query optimization, including query decorrelation [SPL96] via magic sets, pushdown of grouping operations, partitioning of data sets into different regions for parallel execution on different nodes, and merging of common subexpressions across queries (i.e., multi-query optimization). Many of these operations are triggered simply by heuristics, and as a result they are often under-employed or not properly calibrated. I have already demonstrated how convergent query processing can adaptively push grouping operations to good effect — I hope to extend the basic model to incorporate more complex rewrites as well.

The ability to have multiple query plans in simultaneous execution, and to route data to them according to any policy, also presents a number of interesting new possible execution techniques. Not only can we adaptively route data to different parallel nodes with identical query plans, but we can route certain data items to one plan and other data items to a different plan — depending, for instance, on their value or on system load. We can also employ *speculative optimizations*: for instance, if we observe that a given table seems to be sorted or mostly sorted, we can exploit the order in a query plan; data that comes "out of order" can be routed to a different, parallel plan. We can prioritize certain tuples based on time or data value and route them to one plan, and send the remaining tuples to an alternate plan or even temporarily to disk.

9.1.2 Dynamic Alterations to Queries

Ultimately, there are even possibilities for allowing the user to dynamically alter the executing query on-the-fly, as proposed in [RH02]. As a query changes, intermediate results from the previous query can be re-used in the new one.

Moreover, it should be possible to provide a query environment in which certain query subexpressions may have predefined, simpler-to-compute "preview" expressions: if the system needs to meet certain quality-of-service or performance constraints, it can dynamically switch from a query expression to its preview to maintain the required performance level. In the end, the actual query expression will be computed and the preview answers will be replaced.

9.1.3 Combining with Data Placement and Query Reformulation

In a large-scale distributed environment with many data sources (particularly a peer-to-peer environment such as a peer data management system), caching and replication of data (materialization of views) becomes increasingly useful: many in the client population may be asking similar queries, and thus caching certain results in optimal places (data placement) becomes important. Once a large number of views is available, query reformulation can produce huge numbers of possible rewritings, because it can use not only underlying data sources, but also cached views. Now an important challenge is how to prioritize the rewritings such that the user gets numerous good answers in little time. This topic is the focus of current research with one of my thesis advisors and several colleagues.

9.1.4 Understanding Optimality in a Dynamic Environment

Another topic of immense interest is that of developing an analytical model of adaptive query processing, so we can truly understand what policy is optimal in an environment where data distributions and arrival rates may be dynamic. There are many issues here — two of the most important how often to adjust a query plan (given that a plan that is optimal for a local region may not be optimal for global execution) and how to balance between "exploring" alternate query plans and "exploiting" the current plan to produce more answers. It is worth noting that that one may need to know on the order of n^2 selectivities in trying to determine an optimal plan, but these selectivities may change frequently and dynamically, and plan performance is more greatly impacted by some selectivities than others. This is another topic of current research with my advisors.

9.2 Envisioning a Universal Data Management Interface

The data integration field is an important one, but ultimately I believe that the long-term challenges lie in expanding the scope and breadth of managing heterogeneous data: the vast majority of the world's data is stored not using data management tools, but in flat files, spreadsheets, documents, and custom file formats. Sometimes this is because the particular application that created the data has particular performance needs, data needs to be easily shared (e.g., on the web), or database techniques were simply too "heavyweight" to be used.

Interestingly, a recent trend among most applications is standardization on XML-based data formats. Not only do all of the major database systems support XML import and export of their data, but web services are using XML as the format for their RPC protocol, major business applications such as Microsoft Office and StarOffice are using XML as a storage format for all documents, and many SGML documents are being converted to XML. This presents many opportunities for data integration – but I believe that it also presents a unique opportunity to build a unified, distributed storage and retrieval system for all applications. Such a system would be a repository for structured documents (expressed in XML) as well as binary-formatted objects, and would subsume the filesystem and databases for storage and the Web for data distribution. The benefit would be a uniform, application-neutral interface to all accessible data, which maintained the schema and semantics. Such a system would have many more opportunities to tune itself than a standard filesystem (e.g., it could change the internal storage representation of an XML document, migrate data to archival storage, or replicate the data as needed).

Development of such a system poses many new challenges, and I highlight a few below.

Range of quality-of-service needs. One significant obstacle in building a "universal" data management tool is that different applications have dramatically different performance needs and goals. Some applications may need transactional semantics or concurrency control, but may have simple query needs (e.g., an e-mail system); others may support complex queries but may have no need for concurrency; still others may simply need to swap data to disk temporarily, with no need for concurrency or querying. Furthermore, some applications need to be optimized for overall throughput, and others need to return only a few answers very quickly. An important topic of study here is how to build modular architectures that support different quality-of-service needs, how applications should specify those needs, and how a query processor can optimize its output under different performance constraints. We recently proposed a concept called a "query preview" that allows a user to express easier-to-compute approximations to certain query expressions: the query processor can use the preview expression instead of the query to produce approximate answers more quickly, and it can later refine these answers with the real data. I believe that preview queries are a promising first step towards being able to provide quality-of-service guarantees about sustained

performance.

Data sharing with rich semantics. We have recently begun studying the problem of sharing semantically rich data in a decentralized fashion in the Piazza project [GHI+01], a *peer data management system*. In contrast to data integration, peer data management systems (PDMSes) have no central mediated schema and no central schema administrator. Instead, every participant in the system can define his or her own schema as a new frame of reference; each participant can add mappings to describe relationships between any existing schemas; each participant can add new data sources and map them into existing schemas. The peer data management system allows a user to pose a query from the perspective of any peer's schema, and it will exploit all relevant data in answering the query. Other issues under study in the Piazza project include data placement (replicating the answers to certain queries at the most appropriate location so they can be used to answer future queries), propagation of updates to the cached or replicated data, and appropriate indexing of content to make sure only the relevant peers are involved in answering queries.

The ability to update data. Updates should be possible to express both from an application and from outside the application. In both cases, not only the original data, but also all replicas, must be updated. I recently co-authored a paper on expressing updates over XML data [TIHW01], and others in the Piazza research group have described how source updates can be propagated to replicas [MGH02].

Annotations and curated views. In an environment that supports many users and applications, a feature that will undoubtedly be of great importance is the ability to publish annotated and curated views of source data. Rather than directly manipulating the source data, a data curator may define a view that inserts, deletes, or changes the underlying source data in key ways, and which may also add annotations to this data. This is different from recent work by Buneman et al. [BKT02], which studies the problem of pushing annotations from mediated views directly down to the underlying data sources and to other mediated views — I believe that we need to encapsulate the annotations and modifications within the view, so they are maintained even when the data source changes. Furthermore, I believe that the annotations may overlay schema information as well as make modifications to the data — for instance, a string

that contains address information might be annotated so each item in the address gets separately tagged.

Text querying. Querying of document data, which often consists of large paragraphs of free-form text, will be important. This capability should be defined in a way that allows a user to annotate freeform text with additional semantic tags (as described above). To support these types of operations, a data management system must support rich, information retrieval-style queries with approximate matching of keywords. Currently, XQuery has very limited support for such concepts, and this will need to be addressed. There may also be need for new index types that support querying of both text and structure.

Identifying matching concepts and entities. Clearly, as more data is shared, it becomes increasingly important to be able to map from one schema to another, but also to map from one data instance to another. The schema matching problem has been the focus of much work at the University of Washington [DDH01, MBR01] and Microsoft Research [BHP00], and identity matching has been investigated in works such as [Coh98, PR01]. I am hopeful that these techniques will continue to develop and that they can be used in a universal data management system.

Self-tuning storage. Clearly, for any data management tool to be pervasive, it cannot require expert administration. Instead, it must be self-tuning, in terms of its query processing, its data layout, and its data placement. I have worked in the areas of adaptive query processing and in data placement, and hope to investigate data layout next. I believe that techniques like convergent query processing, which allow query execution to be partitioned into separate phases, may provide a low-overhead way of executing queries (and comparing performance) over many different data layout strategies in parallel. Self-tuning has recently become the focus of significant industry efforts at IBM and Microsoft, but I believe there are many opportunities for academic research in the area as well.

Support for arbitrary code. Finally, since data of all types will need to be managed, it seems likely that a universal data management system will need to provide

support for arbitrary user defined code. User-defined functions have always been difficult to support in object-oriented and object-relational systems, primarily because their costs were unpredictable. I am hopeful that adaptive query processing can address some of these problems by monitoring the costs of each function call and re-optimizing as necessary.

While the vision of a universal data management interface is quite ambitious and requires significant research efforts in many areas, I believe that over time it will be realized. I am hopeful that many of the lessons I have learned from adaptive query processing will translate to this new realm, and that much of the work of others will also be useful. It would be a significant achievement if the data management community really could manage most of the world's data, and if users and application programmers could store and query their data in a transparent, intuitive way that preserves semantics.

BIBLIOGRAPHY

[AAB+98] José Luis Ambite, Naveen Ashish, Greg Barish, Craig A. Knoblock, Steven Minton, Pragnesh J. Modi, Ion Muslea, Andrew Philpot, and Sheila Tejada. Ariadne: A system for constructing mediators for internet sources. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 561–563, 1998.

[ACPS96] Siobel Adali, K. Selcuk Candan, Yannis Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, 1996.

[Aea01] Serge Abiteboul and et al. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, June 2001.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, 2000.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, 2000.

[AK93] Yigal Arens and Craig A. Knoblock. SIMS: Retrieving and integrating information from multiple sources. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 562–563, 1993.

[AK97] Naveen Ashish and Craig A. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems, Kiawah Island, South Carolina, USA, June 24-27, 1997, Sponsored by IFCIS, The Intn'l Foundation on Cooperative Information Systems*, pages 160–169, 1997.

[AKJK+02] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA USA*, 2002.

[Ant93] Gennady Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 538–547, 1993.

[ASD+91] Fafi Ahmed, Philippe De Smedt, Weimin Du, William Kent, Mohammad A. Ketabchi, Witold Litwin, Abbas Rafii, and Ming-Chien Shan. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12), 1991.

[AZ96] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optmization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[BBM+01] Denilson Barbosa, Attila Barta, Alberto Mendelzon, George Mihaila, Flavio Rizzolo, and P. Rodriguez-Gianolli. ToX – the Toronto XML engine. In *International Workshop on Information Integration on the Web*, Rio de Janeiro, 2001.

[BC02] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, 2002.

[BCF+02] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simeon, and Mugur Stefanescu. XQuery 1.0:

An XML query language. http://www.w3.org/TR/xquery/, 30 April 2002. W3C working draft.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A multidimensional workload-aware histogram. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, 2001.

[BFG01] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with (lixto). In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, 2001.

[BFRW01] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL — a model for W3C XML schema. In *Proceedings of the Tenth International World Wide Web Conference, Hong Kong, China, May 1-5, 2001*, volume 10, 2001.

[BGL+99] Chaitanya K. Baru, Amarnath Gupta, Bertram Ludäscher, Richard Marciano, Yannis Papakonstantinou, Pavel Velikhov, and Vincent Chu. XML-based information mediation with MIX. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 597–599, 1999.

[BHP00] Philip A. Bernstein, Alon Y. Halevy, and Rachel Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, December 2000.

[BKKM00] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, and Ravi Murthy. Oracle8i - the XML enabled data management system. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, CA USA*, pages 561–568, 2000.

[BKT02] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *Proceedings of the*

Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 3-5, 2002, Madison, Wisconsin USA, pages 150–158, 2002.

[BKV98] Luc Bouganim, Olga Kapitskaia, and Patrick Valduriez. Memory-adaptive scheduling for large query execution. In *CIKM '98, Proceedings of the Seventh International Conference on Information and Knowledge Management, November 1998, Bethesda, Maryland, USA*, November 1998.

[Bla96] José A. Blakeley. Data access for the masses through OLE DB. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 161–172, 1996.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[BRZZ99] Laura Bright, Louiqa Raschid, Vladimir Zadorozhny, and Tao Zhan. Learning response times for websources: A comparison of a web prediction tool (WebPT) and a neural network. In *CoopIS 1999*, 1999.

[BT98] Phillipe Bonnet and Anthony Tomasic. Partial answers from unavailable sources. In *Proceedings of the International Conferece on Flexible Query Answering Systems*, pages 43–54, May 1998.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 303(3):109–120, September 2001.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, 1994.

[CDTW00] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, 2000.

[CFI+00] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene Shekita, and Subbu Subramanian. XPERANTO: Publishing object-relational data as XML. In *ACM SIGMOD Workshop on the Web (WebDB), Dallas, TX*, 2000.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 150–160, 1994.

[CGM99] Chen-Chuan K. Chang and Hector Garcia-Molina. Mind your vocabulary: Query mapping across heterogeneous information sources. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 335–346, 1999.

[Coh98] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 201–212, 1998.

[Col89] Latha S. Colby. A recursive algebra and query optimization for nested relations. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 273–283, 1989.

[CR94] Chung-Min Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD Interna-*

tional Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994, pages 161–172, 1994.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 354–366, 1994.

[DDH01] AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, 2001.

[DFF+99] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proceedings of the Eighth International Word Wide Web Conference, Toronto, CA, 1999*, 1999.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 431–442, 1999.

[DG97] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA*, pages 109–116, 1997.

[DGGR02a] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, 2002.

[DGGR02b] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, 2002.

[DL97] Oliver Duschka and Alon Levy. Recursive plans for information gathering. In *ICJCAI '97*, 1997.

[DSS00] Clip2 DSS. Gnutella: To the bandwidth barrier and beyond. World Wide Web: www.clip2.com/gnutella.html, November 2000.

[EHAB99] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. Next century challenges: Data-centric networking for invisible computing: The Portolano project at the University of Washington. In *Proceedings of MOBICOM-99, Seattle, WA*, August 1999.

[ESW78] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 31 - June 2, 1978*, pages 169–180, 1978.

[EW94] Oren Etzioni and Daniel Weld. A softbot-based interface to the Internet. *Communications of the ACM*, pages 72–76, July 1994.

[FFK+98] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. Catching the boat with strudel: Experiences with a website management system. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 414–425, 1998.

[FK99a] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, March 1999.

[FK99b] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[FKL97] Daniela Florescu, Daphne Koller, and Alon Levy. Using probabilistic information in data integration. In *VLDB'97, Proceedings of 23rd Interna-*

tional Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, pages 216–225, 1997.

[FLM99] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *Proceedings of the AAAI Sixteenth National Conference on Artificial Intelligence*, pages 67–73, 1999.

[FLMS99] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 311–322, 1999.

[FMK00] Daniela Florescu, Ioana Manolescu, and Donald Kossman. Integrating keyword search into XML query processing. In *Proceedings of the Ninth International World Wide Web Conference, Amsterdam, NL, 2000*, May 2000.

[FMN02a] Mary Fernandez, Jonathan Marsh, and Marton Nagy. XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/TR/query-datamodel/, 30 April 2002. W3C working draft.

[FMN02b] Mary Fernandez, Jonathan Marsh, and Marton Nagy. XQuery 1.0 and XPath 2.0 data model. http://www.w3.org/TR/query-datamodel/, 30 April 2002. W3C working draft.

[FMS01a] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middle-ware queries. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, May 2001.

[FMS01b] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middle-ware queries. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, 2001.

[Fri99] Marc T. Friedman. *Representation and Optimization for Data Integration*. PhD thesis, University of Washington, 1999.

[FTS99] Mary Fernandez, Weng-Chiew Tan, and Dan Suciu. SilkRoute: Trading between relations and XML. In *Proceedings of the Ninth International World Wide Web Conference, Amsterdam, NL, 2000*, November 1999.

[FW97] Marc Friedman and Daniel S. Weld. Efficiently executing information-gathering plans. In *ICJCAI '97*, pages 785–791, 1997.

[FZ96] Michael J. Franklin and Stanley B. Zdonik. Dissemination-based information systems. *Data Engineering Bulletin*, 19(3):20–30, 1996.

[GGR02] Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Querying and mining data streams: You only get one look. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, 2002. Invited tutorial.

[GHI+01] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can databases do for peer-to-peer? In *WebDB Workshop on Databases and the Web*, June 2001.

[GMOS02] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. Available from www.cs.washington.edu/homes/suciu/files/paper.ps, February 2002.

[GMPQ+97] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD Workshop on the Web (WebDB), Philadelphia, PA*, pages 25–30, 1999.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gro01] The Meta Group. www.metagroup.com, 2001.

[GST96] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. Calibrating the query optimizer cost model of iro-db, an object-oriented federated database system. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 378–389, 1996.

[GTK01] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, 2001.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 358–366, 1989.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445, 1997.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 287–298, 1999.

[HIST02] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. Submitted for publication, 2002.

[HKWY97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 276–285, 1997.

[HMH01] Mauricio A. Hernandez, Renee J. Miller, and Laura M. Haas. Clio: A semi-automatic tool for schema mapping. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, 2001.

[HS93] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[HSR91] Tina M. Harvey, Craig W. Schnepf, and Mark A. Roth. The design of the Triton nested relational database system. *SIGMOD Record*, 20(3):62–72, 1991.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 268–277, 1991.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 299–310, 1999.

[IHW] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Convergent query processing. Submitted for publication.

[IHW01] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Integrating network-bound XML data. *IEEE Data Engineering Bulletin Special Issue on XML*, 24(2), June 2001.

[ILM+00] Zachary G. Ives, Alon Y. Levy, Jayant Madhavan, Rachel Pottinger, Stefan Saroiu, Igor Tatarinov, Shiori Betzler, Qiong Chen, Ewa Jaslikowska, Jing Su, and Wai Tak Theodora Yeung. Self-organizing data sharing communities with SAGRES. In *SIGMOD 2000, Proceedings ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, page 582, 2000.

[ILW+00] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive query processing for internet applications. *IEEE Data Engineering Bulletin Special Issue on Adaptive Query Processing*, 23(2), June 2000.

[JXT01] Project JXTA: Protocol specification revision 1.1.1. platform.jxta.org/spec/v1.0/JXTAProtocols.pdf, 12 June 2001.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 106–117, 1998.

[KDW97] N. Kushmerick, R. Doorenbos, and D. Weld. Wrapper induction for information extraction. In *ICJCAI '97*, 1997.

[KM90] Alfons Kemper and Guido Moerkotte. Access support in object bases. In *SIGMOD 1990, Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 364–374, 1990.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, CA USA*, page 198, 2000.

[KW96] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *AAAI '96*, pages 32–39, August 1996.

[LPBZ95] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou. Differential evaluation of continual queries. Technical Report TR95-17, University of Alberta, June 1995.

[LPH00] Ling Liu, Calton Pu, and Wei Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, CA USA*, pages 611–621, 2000.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262, 1996.

[LYV+98] Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey D. Ullman, and Murty Valiveti. Capability based mediation in tsimmis. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 564–566, 1998.

[MAM01] Amelie Marian, Serge Abiteboul, and Laurent Mignent. Change-centric management of versions. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, 2001.

[MBR01] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with Cupid. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, 2001.

[MFK+00] Ioana Manolescu, Daniela Florescu, Donald Kossman, Florian Xhumari, and Don Olteanu. XML and relational: How to live with both. In *VLDB*

2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt, September 2000.

[MGH02] Peter Mork, Steve Gribble, and Alon Halevy. Propagating updates in a peer data management system. Unpublished, February 2002.

[MHTH01] Peter Mork, Alon Halevy, and Pter Tarczy-Hornoch. A model for data integration systems of biomedical data applied to online genetic databases. In *Proceedings of the American Medical Informatics Association (AMIA) Annual Symposium, American Medical Informatics Association, Washington DC, November 2001*, 2001.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB'86, Proceedings of 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan*, pages 149–159, 1986.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 103–114. ACM Press, 1994.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Database Theory — ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, pages 277–295, 1999.

[MSHR02] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, 2002.

[MSHTH02] Peter Mork, Ron Shaker, Alon Halevy, and Peter Tarczy-Hornoch. PQL: A declarative query language over dynamic biological schemata. In *Proceedings of the American Medical Informatics Association (AMIA) Annual Symposium, San Antonio, TX*. American Medical Informatics Association, November 2002.

[NDM+00] Jeffrey Naughton, David DeWitt, David Maier, Jianjun Chen, Leonidas Galanis, Kristin Tufte, Jaewoo Kang, Qiong Luo, Naveen Prakash, Feng Tian, Jayavel Shanmugasundaram, Chun Zhang, Ravishankar Ramamurthy, Bruce Jackson, Yuan Wang, Anurag Gupta, and Rushan Chen. The Niagara internet query system. Available from www.cs.wisc.edu/niagara/papers/NIAGRAVLDB00.v4.pdf, 2000.

[NDM+01] Jeffrey Naughton, David DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayvel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, June 2001.

[NET01] What are XML web services? www.microsoft.com/net/xmlservices.asp, May 2001.

[ODB97] Open database connectivity documentation. www.microsoft.com/data/odbc/default.htm, 1997.

[PL00] Rachel Pottinger and Alon Levy. A scalable algorithm for answering queries using views. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, 2000.

[PR01] Hanna Pasula and Stuart J. Russell. Approximate inference for first-order probabilistic languages. In *ICJCAI '01*, pages 741–748, 2001.

[Qia96] Xiaolei Qian. Query folding. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 48–55, 1996.

[RH01] Vijayshankar Raman and Joseph M. Hellerstein. Using state modules for adaptive query processing. Available from www.cs.berkeley.edu/∼rshankar/nsqp3.pdf, 2001.

[RH02] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, 2002.

[ROH99] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland*, pages 599–610, 1999.

[RS86] Louiqa Raschid and Stanley Y. W. Su. A parallel processing strategy for evaluating recursive queries. In *VLDB'86, Proceedings of 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan*, pages 412–419, 1986.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 105–112, 1995.

[Rys01] Michael Rys. Bringing the internet to your database: Using SQLServer 2000 and XML to build loosely-coupled systems. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 465–472, 2001.

[SA99] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland*, pages 738–741, 1999.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD*

International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1, pages 23–34, 1979.

[SAL+96] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[SBD+81] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W.T. Lin, and Eugene Wong. MULTIBASE – integrating heterogeneous distributed database systems. In *Proceedings of 1981 National Computer Conference*, pages 487–499. AFIPS Press, 1981.

[SGG02] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN) 2002, San Jose, CA*, January 2002.

[SGT+99] Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland*, pages 302–304, 1999.

[SKS+01] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying XML views of relational data. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 261–270, 2001.

[SLMK00] Michael Stillger, Guy Lohman, Volker Markl, and Mokhtar Kandil. LEO — DB2's LEearning Optimizer. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 19–28, 2000.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In *Proceedings of the Twelfth International Conference on*

Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, pages 450–458, 1996.

[SSB+00] Jayavel Shanmugasundaram, Eugene Shekita, Rimon Barr, Michael Carey, Berthold Reinwald, Bruce Lindsay, and Hamid Pirahesh. Efficiently publishing relational data as XML documents. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, 2000.

[SV02] Luc Segoufin and Victor Vianu. Validating streaming xml documents. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 3-5, 2002, Madison, Wisconsin USA*, pages 53–64, 2002.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China*, 2002.

[SWKH76] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The design and implementation of INGRES. *TODS*, 1(3):189–222, 1976.

[Tam] Tamino: Technical description. www.softwareag.com/tamino/details.htm.

[TIHW01] Igor Tatarinov, Zachary Ives, Alon Halevy, and Daniel Weld. Updating XML. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, June 2001.

[Tre99] Harold Treat. Plugging in to XML. *DB2 Magazine*, Winter 1999. Also available at http://www.db2mag.com/winter99/treat.shtml.

[TRV98] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling access to distributed heterogeneous data sources with DISCO. *IEEE Transactions On Knowledge and Data Engineering*, 1998.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, 2002.

[UF99] Tolga Urhan and Michael J. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, College Park, February 1999.

[UF00] Tolga Urhan and Michael J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[UF01] Tolga Urhan and Michael J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, September 2001.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 130–141, 1998.

[Val87] Patrick Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.

[WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. First International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, December 1991.

[WBJ+95] Darrell Woelk, Bill Bohrer, Nigel Jacobs, K. Ong, Christine Tomlinson, and C. Unnikrishnan. Carnot and InfoSleuth: Database technology and the world wide web. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 443–444, San Jose, CA, 1995.

[Win02] Marianne Winslett. Distinguished database profiles: David DeWitt speaks out. *SIGMOD Record*, 31(2):50–62, June 2002.

[XLN] eXcelon Corporation: Platform. www.exceloncorp.com/platform/index.shtml.

[XSc99] XML Schema part 1: Structures. www.w3.org/TR/1999/WD-xmlschema-1-19991217/, 17 December 1999. W3C Working Draft.

[XSL99] XSL Transformations (XSLT), version 1.0. www.w3.org/TR/xslt, 16 November 1999. W3C recommendation.

[YPAGM98] Ramana Yerneni, Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Fusion queries over internet databases. In *Advances in Database Technology - EDBT 1998, 5th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, pages 57–71, Valencia, Spain, 1998.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, 2001.

[ZRZB01] Vladimir Zadorozhny, Louiqa Raschid, Tao Zhan, and Laura Bright. Validating an access cost model for wide area applications. In *CoopIS 2001*, 2001.

VITA

Zachary Ives will be joining the Computer and Information Sciences Department at the University of Pennsylvania as an Assistant Professor in January 2003. His research interests include query processing for distributed and heterogeneous data, peer-to-peer and distributed systems, XML and semistructured data, and distributed data management.

He will be graduating in July 2002 with a Ph.D. in Computer Science from the University of Washington. He received his Master of Science degree in Computer Science from the University of Washington in 1999, his Bachelor of Science from Sonoma State University in 1997, and Associate of Sciences degrees in Computer and Information Sciences and Electrical and Electronic Technology from Mendocino Community College in 1995.