STORAGE AND QUERY PROCESSING OPTIMIZATIONS FOR HIERARCHICALLY-ORGANIZED DATA

by

Alan Dale Halverson

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2006

To my wife, Kris—my love, best friend, and confidant.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Jeff Naughton. His guidance, inspiration, and patience with me during the five years I have been a graduate student have been an amazing gift. He encouraged me to keep going when times were tough, but also provided honest and tough feedback when necessary. He likes to give his students the latitude to make mistakes and learn from them. I appreciate all of these things—they have made me stronger.

Many other faculty and industry professionals have provided guidance for me during graduate school. In particular, David DeWitt has been a mentor and friend to me during my time at Wisconsin-Madison—thanks. Others at Wisconsin include Raghu Ramakrishnan and Remzi Arpaci-Dusseau.

From industry, I want to thank Guy Lohman in particular. My experience under his supervision at IBM Almaden Research Center opened my ideas to the possibility of a second career in industry. His passion for technology and ideas is amazing. Thanks, Guy.

The work presented in Chapter 2 was jointly developed with Guy Lohman, Vanja Josifovski, Hamid Pirahesh, and Matthias Moerschel at IBM Almaden Research Center. Some content in Chapter 3 benefitted from discussions with Jennifer L. Beckmann and David J. DeWitt.

I would next like to thank my professors at Luther College, my undergraduate alma mater—Walt Will, Alan MacDonald, and Steve Hubbard. Their passion for teaching and one-on-one interaction with students provided me the inspiration to come back to graduate school. Perhaps I will teach at Luther one day—nothing would make me happier.

My family deserves special recognition. My wife Kris, and my two sons Benjamin and Joshua. All of them have made sacrifices during my time in graudate school so that I might succeed and finish my degree. My boys probably don't even remember I time I wasn't in graduate school! They keep asking me to pretend I'm sick so I can play with them. Soon guys, soon – I promise. Thanks for being patient.

My extended family has also been very supportive of my time in graduate school. My parents Dale and Becky. My brother John and his wife Kristie, their children Zachary and Rachel. My sister Joy and her husband Kevin. My wife's parents Mick and Elaine. My brother-in-law Tim, his wife Allisa, and their daughter Ahna. Thanks to all of you for your love and support.

I would like to thank NCR and the family of Anthony C. Klug for their generous support of my graduate school research through the fellowship endowed in Anthony's memory. I have been fortunate enough to hold this fellowship for three years.

Finally, I would like to thank God for the strength to complete the task I started so many years ago. It has not been easy, but things worth doing rarely are.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

STORAGE AND QUERY PROCESSING OPTIMIZATIONS FOR HIERARCHICALLY-ORGANIZED DATA

Alan Dale Halverson

Under the supervision of Professor Jeffrey F. Naughton

At the University of Wisconsin-Madison

Hierarchical data can be found anywhere multiple pieces of information are connected by a relationship. The first chapter of my thesis deals with processing relational queries in the context of a native XML storage system. We take advantage of hierarchical XML to equate the nested structure of the XML documents with the key relationship between two data items. A join query can then be rewritten as a storage scan, allowing acceleration of the query. The second chapter provides a comparison of row and column oriented storage optimizations given an assumption of read-mostly data access. We focus on two, “super tuples” and “column abstraction”, to elucidate the difference between row and column storage layouts. Column abstraction allows optimization of hierarchically organized data by storing repeating values only once. We extend the read-optimized relational store in the third chapter to evaluate two predicate evaluation strategies—a scan-based solution over both standard super tuples and the PAX layout, and an index-based strategy with and without a slot array in super tuples.

Jeffrey F. Naughton

ABSTRACT

Hierarchical data can be found anywhere multiple pieces of information are connected by a relationship. The first chapter of my thesis deals with processing relational queries in the context of a native XML storage system. We take advantage of hierarchical XML to equate the nested structure of the XML documents with the key relationship between two data items. A join query can then be rewritten as a storage scan, allowing acceleration of the query. The second chapter provides a comparison of row and column oriented storage optimizations given an assumption of read-mostly data access. We focus on two, “super tuples” and “column abstraction”, to elucidate the difference between row and column storage layouts. Column abstraction allows optimization of hierarchically organized data by storing repeating values only once. We extend the read-optimized relational store in the third chapter to evaluate two predicate evaluation strategies—a scan-based solution over both standard super tuples and the PAX layout, and an index-based strategy with and without a slot array in super tuples.

Chapter 1

Introduction

We live in an increasingly connected world. Many people have high speed, always-on Internet access at their homes, and Google is a verb in nearly every language on the planet. Our access to multiple streams of information in digital formats has the potential to keep us in touch with world events and far-away loved ones. We have access to this information on multiple devices, such as wired desktop computers at home and work, and wireless laptop computers, PDAs and cell phones. The promise of these connectivity options is simple enough—to enable access to information when and where we need it. While this may be true, I see two additional trends developing. First, high-speed access to data and information has not reduced how much time people spend using the Internet, but rather has allowed them to view more information in the same amount of time. Second, tools for helping users manage interconnected streams of information have lagged behind access to the information itself. A modern web browser provides a history of sites visited, but the list is flat and provides only a day-by-day grouping of sites. Missing is the hierarchical discovery of the information and the information itself.

The Internet is a hierarchy of links and information, and our minds work in a similar fashion. While Google is a good model for cutting across hierarchy to find information quickly, the flat result list model does not match how we discover and remember information. In fact, hierachical data objects are pervasive in information management – a customer and their orders, a grouping of sales figures by continent, country, and region, a computer filesystem, and so on. Developing techniques for efficiently managing information hierarchies is critical to bridging this gap.

My research interests concern optimizations for storage and query processing of hierarchically organized data. In this thesis, I present three projects developed in the context of these interests: a system for processing relational queries over XML [13] data, a comparison of storage optimization techniques for relational schemas in the context of row-oriented and column-oriented storage for a read-mostly query workload, and an exploration of predicate evaluation strategies for a read-optimized relational store. The next three sections briefly describe each project.

1.1 Relational Over XML

In the first chapter of my thesis, I develop a method for evaluating relational queries over XML data stored natively. I called this method ROX, or Relational Over XML. In some circumstances, XML documents will conform to a regular, repeating schema—in fact, the data may have been stored in a traditional relational database and published to conform to the desired XML schema. In this case, defining a virtual table definition, or “nickname”,

of the data in terms of path expressions over the XML schema is straightforward. ROX works by translating references to these nicknames into XPath [10] expressions, evaluating each expression over the XML data, and translating the XML result back into a relational rowset.

One key observation of this work is that XML allows related data to be stored together in a "de-normalized" fashion. For example, a Customer element may be the parent of several Orders elements. When referential integrity guarantees this one-to-many relationship, we can create a relationship between the nicknames that define the relational view for each Customer and Order based on the structural relationship between the two. In this case, a relational join between Customer and Orders can become a simple, navigation-based scan of the XML document. We evaluated this idea in the XML prototype system using several variations of the TPC-H [18] benchmark schema and discovered that obviating such joins allowed the translated queries to run within $1/2$ order of magnitude of the optimized standard relational execution time, despite obvious inefficiencies in the XML prototype.

1.2 A Comparison of C-Store and Row-Store in a Common Framework

Relational data management systems store data as complete rows of information. Such a design provides good performance for query workloads which feature a mixture of reads and writes to the database. Recently, Stonebraker et al. proposed a column-oriented system called C-Store that is optimized for a "read-mostly" query workload. Their evaluation shows incredible performance benefits when compared to a traditional relational DBMS.

Motivated by their work, I set out to design a set of storage and query optimizations for a row-oriented system. I designed two storage improvements—"super tuples" and "column abstraction". Super tuples simply pack many relational rows into a single storage block the size of a disk page, and later using a secondary lightweight iterator to extract each logical tuple. Column abstraction utilizes data ordering techniques—both for arbitrary sort columns as well as ordering by the columns in the one side of a one-to-many join in a materialized view—to store repeating data only once to save disk space. I implemented a prototype system that applied these optimizations to both the row- and column-oriented architectures and found that row storage is performance competitive with column storage for most sequential scans. I also developed a cost model which breaks down the total scan cost into disk I/O, iteration cost, and local tuple reconstruction cost. The cost model correctly identifies the trends and relative performance of each storage optimization and storage choice.

1.3 Predicate Evaluation Strategies for an Read-Optimized Row Store

In the third chapter of my thesis, I extend my development and analysis of the read-optimized relational store to include predicate evaluation strategies. The materialized views defined and stored for matching queries in a read-mostly context may match more than one query in a given query workload when those queries differ only by a selection predicate. I evaluate scan-based and index-based predicate evaluation strategies over super tuples, and explore alternative storage layouts that can accelerate evaluation for each – the PAX [9]

layout for warm scans, and a lightweight slot array for indexes. I develop a detailed cost model to estimate the relative costs of each strategy and layout, and extend my prototype read-optimized relational store to provide experimental validation of the strategies and cost models.

Chapter 2

ROX: Relational Over XML

An increasing percentage of the data needed by business applications is being generated in XML format. Storing the XML in its native format will facilitate new applications that exchange business objects in XML format and query portions of XML documents using XQuery. This chapter explores the feasibility of accessing natively-stored XML data through traditional SQL interfaces, called Relational Over XML (ROX), in order to avoid the costly conversion of legacy applications to XQuery. It describes the forces that are driving the industry to evolve toward the ROX scenario as well as some of the issues raised by ROX. The impact of denormalization of data in XML documents is discussed both from a semantic and performance perspective. We also weigh the implications of ROX for manageability and query optimization. We experimentally compared the performance of a prototype of the ROX scenario to today's SQL engines, and found that good performance can be achieved through a combination of utilizing XML's hierarchical storage to store relations "pre-joined" as well as creating indices over the remaining join columns. We have developed an experimental framework using DB2 8.1 for Linux, Unix and Windows, and have gathered initial performance results that validate this approach.

2.1 Introduction

After two decades of commercially-available products, relational database systems (RDBMSs) supporting the SQL query language standard are an unqualified commercial success, with a huge industry-wide investment in applications such as Enterprise Resource Planning (ERP) [3, 4, 5] and Customer Relationship Management [3, 6] that query an RDBMS with SQL. As the acceptance and sources of XML documents have proliferated, many commercial relational database systems have adapted by developing techniques for storing XML documents in relational systems by shredding documents into relations [20, 21, 32] and/or by storing each document as an unstructured, large object (LOB) [21]. However, shredding and recomposing all documents, many of which will never be retrieved, is unduly expensive. Alternatively, searching XML documents stored as LOBs is prohibitively slow. As more enterprises exchange business objects, such as purchase orders, in XML format, applications will increasingly need to efficiently query portions of XML documents via the emerging XQuery standard [11]. This will lead to storing the data in some native XML format that efficiently supports XQuery.

Legacy relational interfaces and native XML storage appear to be on a collision course that raises many interesting questions. Can the relational and XML data be treated separately, storing each in the appropriate type of repository? In other words, will data from relational sources be queried exclusively by SQL, and XML data exclusively by XQuery? Or will databases of the future have to be hybrids, storing both relational and XML? Or will

we just convert relations into XML objects and store everything in XML format? Regardless, what is to become of the "legacy" applications written in "good old" SQL that need access to data that increasingly originates as XML data? Do they need to be re-written, or can XML repositories support both XQuery and SQL? Will there be evolution, or a revolution?

We are convinced that XML adoption must necessarily be an evolution—that existing relational applications are too big and complicated to convert them all rapidly or inexpensively from SQL to XQuery. We also project that the data accessed by these SQL applications will increasingly come from XML sources and need to also be accessible via XQuery, and hence will be stored in native XML format.

This chapter therefore explores how to efficiently support Relational Over XML (ROX), i.e., the existing SQL interface to a native XML store. We postulate a database containing a blend of both tables and XML documents, with an increasing percentage of XML documents over time. The ROX scenario limits our consideration to SQL queries as input that return rows as output, in order to support legacy applications, even though the system is very likely to also support XQuery interfaces to the same database.

The ROX scenario alone raises many important issues. Perhaps the most important is whether ROX can perform as well as today's SQL engines. What is the impact of the obvious expansion of data caused by tags and other structuring information? How much should XML documents be normalized, and does the denormalization supported by XML help or hinder performance? Or is normalization of data obsolete with the advent of XML? The

remainder of this chapter is organized as follows. The next section summarizes the evolution of XML data management. Section 2.3 discusses issues involving query semantics of SQL and XQuery, tradeoffs for selecting an appropriate XML schema, and performance concerns. We present our ROX experimental design in Section 2.4, and the results of those experiments in Section 2.5. Our conclusions and directions for future research comprise the last section.

2.2 The Evolution of XML Data Management Systems

Storing and processing XML data have been a focus of the database research community for much of the last decade. Several XML data management systems have been proposed, most based on various degrees of adaptation and reuse of relational [7] technology. There are two main reasons for reuse of relational technology. First, adaptation is presumably less expensive and allows faster time to market than development from scratch. The other reason is that such hybrid systems are capable of storing both relational (structured) and XML (semi-structured) data. As most applications are likely to operate over both types of data, the new generation of databases will need to support both allowing the application to access a single data repository.

Several different architectures have been proposed for building a hybrid XML-relational database, as illustrated in Figure 2.1. Chronologically, the first attempts were based on reusing the whole RDBMS stack when processing XQuery queries: from the SQL query language to the relational data storage. In this XML-Over-Relational (XOR) approach, the

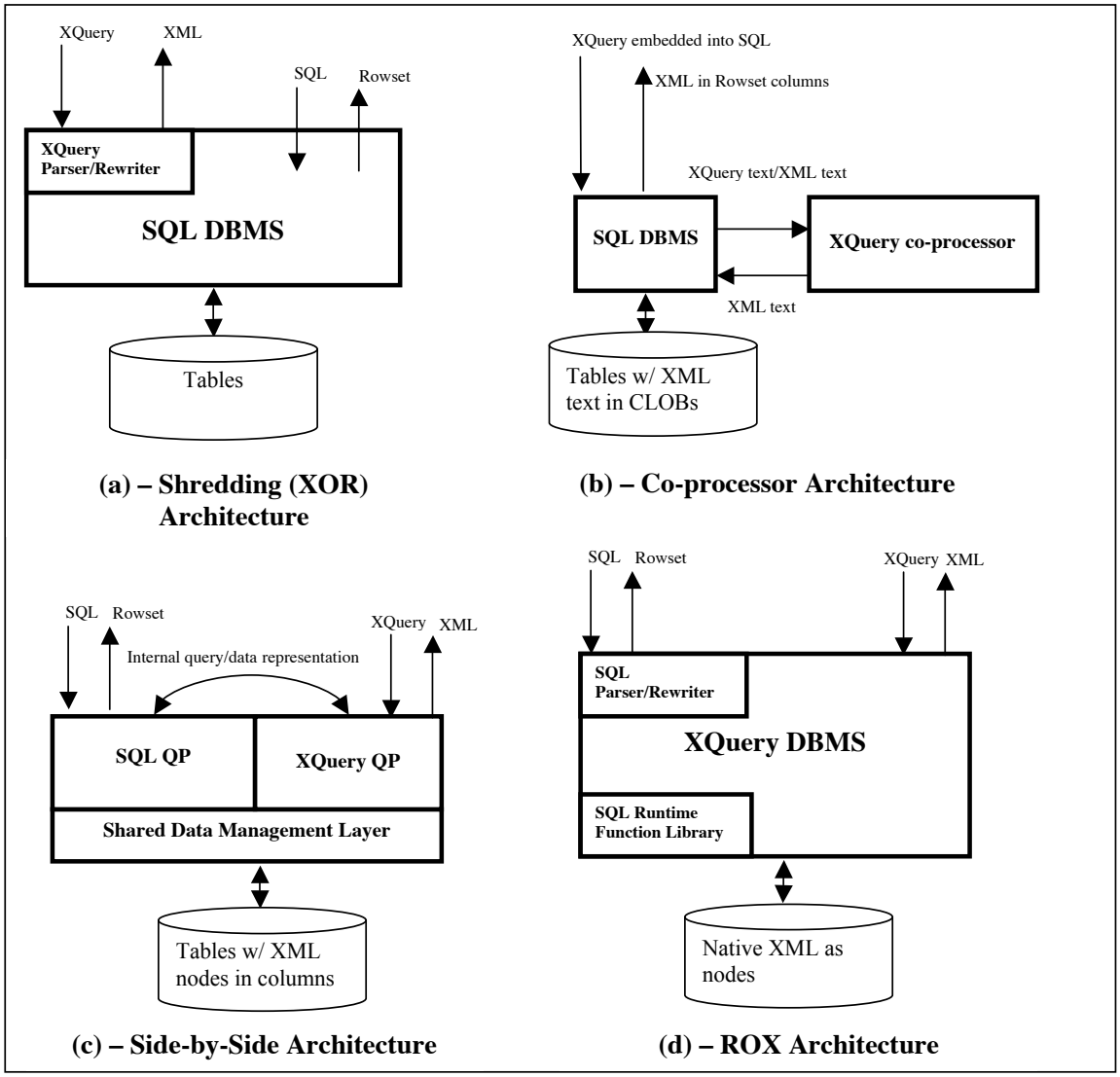


Figure 2.1 **Architecture choices for mixed SQL and XQuery systems**

XML documents are shredded into atomic values that are then stored in relational tables. XQuery queries are translated into SQL queries to be evaluated by the existing query processor. Several research prototypes have explored this scheme, such as LegoDB [12], and XPeranto [35], and several products offer different shredding and XPath querying capabilities based on this approach [2, 21]. The advantage of this architecture is that it requires almost no modification of existing database engines. As such, an XQuery implementation can easily be adapted to several different DBMS systems. However, as the XQuery language has evolved into an elaborate and complex standard, it has become clear that translating XQuery queries into SQL queries is a daunting task. While shred-and-query systems claim compatibility with a subset of the language, none has managed to produce a fully compliant XQuery implementation.

Next in the timeline were systems that are on the other side of the architectural spectrum, named Co-processor Architecture in Figure 2.1(b). Here, XML is stored as unparsed text in VARCHAR or LOB columns of relational tables. The XML data is opaque to the RDBMS and only the storage layer is re-used. The XML data is queried using an XQuery processor that is external to the database and invoked much like a user-defined function. The communication between the two processors is using textual or equivalent format. The SQL and the XQuery processors can be developed separately and interchanged. This solution is attractive for its relative simplicity and modularity.

Most of today's commercial systems support this type of XML manipulation using stored procedures to invoke an external XQuery processor [2, 21], in conjunction with

XOR support. However, due to the loose coupling of the query processors, usually the entire XML document is brought into memory before processing, severely limiting the size of the data and optimization possibilities.

Several systems have been reported that support only XQuery. Systems such as Niagara [28] and Timber [25] break the XML document into nodes and store the node information in a B+-tree, with all document nodes stored in order at the leaf level. This allows for efficient document or sub-tree reconstruction by a simple scan of the leaf pages of the tree. In Niagara, additional inverted list indexes are created to enable efficient structural join algorithms for ancestor/descendant paths. However, these systems do not support SQL or relational storage.

More recently more native storage of XML documents has been proposed in [26] and [40]. In our work we take a similar approach where the XML data is stored as in a native tree format in which document nodes are in most cases clustered together on a page. Bulk processing is performed using indexes, while the storage is optimized for fast navigation to evaluate the non-index portions of the query. Parent-child traversal does not require a join between different tables. Since most XPath expressions require parent-child traversal, this scheme allows for efficient access to the data. We use this model as an example to explore the consequences of representing relational data with hierarchical trees.

At this point we consider where will the system architecture move beyond today's state of the art? Can we project the direction of the path based on the evolution so far? Is the

current situation similar to the introduction of the relational database systems compared to IMS? We try to analyze the issues from several angles and answer to these questions.

One probable direction in the short term is the side-by-side architecture, as shown in Figure 2.1(c). In this architecture there is a tighter coupling between the query processors than in the architecture based on shredding. Query fragments can be translated from one language to the other and exchanged using internal data structures that may not adhere to the language semantics. Such a mechanism improves the efficiency of the translation and allows more degrees of freedom in the evaluation. For example, when returning values from XQuery to SQL, as required when evaluating SQL/XML, queries might require that an element is constructed by an embedded XQuery query and then shredded by an SQL table function. Instead, the optimizer can re-write such queries so that rows of values are returned from the XQuery processor directly into the SQL processor, although rows are not part of the Query Data Model [22].

While more efficient than the first two architectures, the side-by-side architecture introduces many complexities. It requires that various system components have compatible definitions on both sides of the system. For example, the catalogue description of internal objects such as indexes and materialized view definitions need to be matched to both the SQL and XQuery queries. While these issues pose interesting research challenges, we view this architecture only as a partial solution that will be simplified and eventually morph into the Relational over XML architecture shown in Figure 2.1(d) where the primary processing is performed by an XQuery engine with native XML storage model. In its engine

all transformations are governed by the XQuery language specification and the Query Data Model. The SQL support is divided between a thin parse-and-rewrite layer and a library for support of the SQL functions and operators that cannot be mapped to the XQuery functions and operators.

The ROX architecture is at the opposite extreme of the solution space when compared to the first XQuery query processor designs, in which the XQuery processor was a thin layer over SQL database systems. The obvious question is what makes this architecture viable if the opposite solution has not been implemented in any of the major database products? Furthermore, in terms of development cost, this architecture requires a complete XQuery engine that is adapted to run SQL queries, seemingly a much more demanding path than the opposite route.

As XQuery and the Query Data Model conceptually subsume the SQL language and the relational model, implementing SQL on top of an XQuery engine poses significantly lesser challenge than the opposite. We also believe that this architecture will not be achieved by developing an XQuery engine from scratch. Existing relational engines will be morphed into this architecture possibly through the intermediate stages represented by the other architectures depicted in Figure 2.1. It seems to us that, beyond the initial releases of the commercial database products for XML data management, the main forces in the database engine evolution will be to increase the performance and reduce the complexity of the relational-XML engines. These two forces will be the major factors in the appearance of the ROX architecture, shown in Figure 2.1(d).

2.3 ROX Model Issues

While unable to implement a complete ROX architecture, we single out three issues that are crucial to demonstrate the viability of the infrastructure and explore each in more detail. We first overview the language semantics issues and how the semantics differences between SQL and XQuery impact the ROX architecture. Then we turn our attention to the data layout and normalization related issues as posed by the nested XML data model. Finally we consider the performance impact of an XML native format, query optimization issues, and XML data manageability.

2.3.1 Language Semantics

The main difficulty in running SQL queries over an XQuery implementation is providing semantically correct answers to the queries. Although SQL and XQuery have similarities, there is an abundance of differences. First of all, the languages are defined over different data models. The SQL language is defined over the relational model [7], while the XQuery language is based on the Query Data Model [22] that represents XML data as typed trees. SQL queries operate over column values, while XQuery manipulates ordered, heterogeneous sequences of values and node references. While a detailed description of the differences is beyond the scope of this discussion, in general, the Query Data Model is much more elaborate than the relational data model. This is the core reason why XQuery-to-SQL translation is unsuitable as a basis for a fully functional XQuery system.

The languages also differ in their operational semantics. The most quoted difference is the document order preservation of XQuery vs. unordered semantics of SQL. Furthermore, each language standard contains precise descriptions of the language operators. These specifications seldom match. For example, the comparison operators in SQL use 3-value logic, operating over Boolean operands and returning true, false or NULL. The same XQuery operators (general comparison) operate over sequences of nodes or values and return true and false. There is no NULL value defined in the XQuery data model. Another discrepancy stems from the different definitions of the basic data types as decimals and datetime. As many of the built-in functions operate over such values, they might potentially return different results.

Despite these differences, XQuery is designed to be able to manipulate structured data along with unstructured [16]. Therefore, there is an overlap in the functionality of SQL and XQuery. While different in data model and semantics, when constrained over structured data, many XQuery operations have semantics close to that of SQL, and under certain cardinality constraints match the SQL semantics. For example both XQuery and SQL numeric operations are based on the IEEE standard and seem to be reconcilable. Furthermore the XQuery arithmetic operators treat empty sequences in the same manner as SQL operators treat the NULL values. This is also true for the XQuery value comparison operators (eq, gt, neq, etc.) which have 3-value logic (returning empty sequence if any of the operators is an empty sequence) as the comparison operators of SQL. Translating the SQL comparison operators into XQuery value comparison operators, we can achieve the same semantics as

in SQL. This allows pushdown of simple arithmetic and comparison predicates from SQL to XQuery. While the XQuery Boolean operators operate using 2-value logic, it is simple to implement 3-valued Boolean operators in XQuery that have same semantics as the SQL operators. With such a small implementation effort, a large class of SQL predicates over numeric types and strings can be translated into equivalent XQuery predicates. However, based on the current standards proposal for XQuery, it seems that it will not be possible to translate all SQL functions to existing XQuery functions. We envision this necessary for SQL datatypes that are not subsumed into XQuery datatypes, such as types representing date-time and timestamps [20].

2.3.2 Normalization

One of the key benefits of a native XML store is not having to normalize the elements that make up a business object by shredding them into tables. For example, consider a common business object—the purchase order shown in Figure 2.2, which might contain some customer elements and one or more line items describing each object being purchased. Since the purchase order arrives in XML format, it is tempting to store the entire document as it comes into the system, to minimize any processing. But is that the right thing to do? Does the nesting of XML documents make normalization of objects in databases obsolete? And if not, what elements should be normalized and which should not?

The answer is that normalization is still needed in XML databases, for the same reason it was needed in relational databases: redundancy of data and update anomalies [19]. In the above example, the line items nested within an order are wholly owned by that order, so

```
<order>
  <date>12 July 2003</date>
  <customer>
   <ID>43839</ID>
   <name>Slaghorn Bolts</name>
   <contact>Joyce Smith</contact>
   <address>
    53495 N. First St.
    Cleveland, OH 45678
   </address>
   <order_discount>
    0.10
   </order_discount>
  </customer>
  <line_item>
   <part_ID>RYZ04856-8945</part_ID>
   <quantity>33</quantity>
   <discount>0.12</discount>
  </line_item>
  <line_item>
   <part_ID>KFE389745-2248</part_ID>
   <quantity>15</quantity>
   <discount>0.05</discount>
  </line_item>
  <line_item>
   <part_ID>OI230988-2833</part_ID>
      <quantity>100</quantity>
   <discount>0.21</discount>
  </line_item>
</order>
```

Figure 2.2 **Purchase order in XML format**

they cannot suffer the update anomalies of shared subobjects. However, the customer information is a bit more subtle. It is likely to be shared by many other orders, so keeping it with each purchase order would both be unnecessarily redundant and risk update anomalies. For example, the customer address is on the purchase order, but it's probably the same address as on hundreds of other orders from the same customer. But if it suddenly changed, this order might be sent to the address that was in effect at the time the order was made, rather than the address in effect when the order is shipped. Other attributes of the customer suffer from a similar problem. So pretty clearly the customer information should be normalized. However, some elements of the customer are really elements of the interaction of the customer and this order. For example, the order_discount element might depend upon the size of the overall order and how valued this customer is. Hence it cannot be normalized out of the purchase order.

The good news is that native storage of XML documents permits denormalization when it makes sense semantically (sub-objects are not shared), while still retaining the option to normalize data, i.e., when subobjects may be shared and hence risk update anomalies. The database designer is thus free to do what best models the data, rather than forcing the design into a large number of overly-normalized, homogeneous tables. And, as we shall see later, denormalization can also have performance benefits by obviating the need for some joins when the data is queried.

Today's relational engines use data redundancy in form of pre-joined relations to speed up evaluation of queries [24]. Materialized view techniques will also be important in XML

databases in general and with the ROX model in particular. In Section 2.5.3, we show that data nesting that matches the query structure allow for much better evaluation times. As opposed to relational systems in which the materialized, pre-joined views are flat tables, an XML engine allows these to be in non-first normal form, similar to the proposal of [33].

2.3.3 Performance

To achieve good query performance for the ROX model, we must consider several issues. The storage of the XML tree could be sorted in either depth-first (document) or breadth-first order. The depth-first order is advantageous when the goal is efficient reconstruction of the XML. However, if our XML documents have several levels of hierarchy, queries referencing only data at the top levels of the document will suffer.

Another concern is the overhead of storing the structure of the document inline with the data being represented. Although it is possible that the stored XML document conformed to a stated XML schema, in general our storage format must allow for XML documents which lack a predefined schema. This storage overhead forces a native XML store to consume more space for representing a certain dataset than the relational storage. The absence of an XML schema also forces data in the document to be stored as text, which also adds storage overhead. To facilitate efficient selection and value-based joins between XML documents, an XML index is required. As with relational systems, such an index will allow us to find documents which contain a certain value in a certain location. Many indexing strategies for XML have been proposed, such as inverted lists of elements for structural joins and path indexes.

2.3.4 Optimization

Compiling SQL queries on XML documents presents new challenges for query optimization. Although denormalized data in the form of materialized views and join indexes is already widely exploited by relational query optimizers, both the query and the denormalized data are defined in relational terms, usually SQL. In ROX, the optimizer must now match joins and predicates in the SQL query to XPath expressions that define the schema of XML documents (presumably the XML documents manipulated by ROX will have a schema with sufficient homogeneity to permit a tabular view of them). Join predicates between documents must also be folded into predicates at various points of an XPath expression, depending upon the join order. In our experiments, discussed below, we performed this mapping manually to avoid the challenge of automating it. Having documents with various schemas—or even no schema at all!—mixed together in the same repository, called "schema chaos", negates the homogeneity that simplified the cost model and the database statistics on which relational optimization depended. And though the denormalization of XML documents reveals correlations among objects, it is not at all clear what database statistics are needed to summarize those correlations and how those statistics can be exploited to accurately estimate the number of documents satisfying a particular SQL query. And this doesn't even consider the considerable challenges of optimizing XQuery queries!

2.3.5 Manageability

Will a database of XML documents be easier or more difficult to manage than relational tables?

Some would argue that management of XML repositories should be child's play. Since real-world objects no longer need to be normalized into homogeneous collections of rows (tables), the XML repository can be reduced to a single, virtualized heap of heterogeneous objects (documents), creating the relational equivalent of the Universal Relation [27]. In lieu of perhaps tens of thousands of normalized tables, there would be only one collection of documents to configure, backup, recover, reorganize, collect statistics on, etc. Database design would be trivial, normalization would be unnecessary, and one index over this entire collection would suffice to find any object in the database—the "Google model" applied to databases!

On the other hand, management of modern databases entails far more than just deciding what tables and indexes to create. As argued in Section 2.3.2 above, some normalization will still be required to avoid update anomalies, so logical database design may be less constrained but certainly not obviated. Eventually, XML systems will permit the definition of the XML equivalent of materialized views, and deciding which to create will surely be no easier than it is now for relational systems. Even if all documents are in one monolithic collection, administrators will probably have to define arbitrary boundaries within that collection for administration purposes, so that pieces can be maintained while the rest of the database is available for querying and updating, much as the rows of large tables

are usually divided into ranges for administrative purposes [8]. And given the increased challenges posed by optimizing queries against these heterogeneous collections (see previous section), it is likely that the database statistics required for optimization will be far more extensive than for relational systems. For performance reasons, we might still want to cluster related documents together to exploit the larger pre-fetching chunks that relatively slower disk arms necessitate, or possibly de-cluster them to spread access among multiple arms for greater I/O parallelism, rather than simply append each new document to the end of the heap.

2.4 Design for Experimentation

In Section 2.2, we described a number of architectural alternatives for a mixed SQL and XQuery system, including the ROX model. We now provide a description of the implementation we chose for our experimental framework.

A full implementation of the ROX architecture would require a fully-functional XQuery DBMS, upon which a thin SQL-to-XQuery translation layer would sit. However, building a system like this would take a significant number of person-years to implement. Instead, we took advantage of a prototype XML store available to us and implemented a much simpler mapping layer. This experimental architecture is described in detail in the following sections.

```
CREATE NICKNAME REGION(
  R_REGIONKEY int
    OPTIONS(XPATH 'R_REGIONKEY/text()'),
  R_NAME char(25)
    OPTIONS(XPATH 'R_NAME/text()'),
  R_COMMENT varchar(152)
    OPTIONS(XPATH 'R_COMMENT/text()'))
FOR SERVER xml_server
OPTIONS(XPATH '/REGION');
```

Figure 2.3 **Nickname definition**

```
<REGION>
  <R_REGIONKEY>2</R_REGIONKEY>
  <R_NAME>ASIA</R_NAME>
  <R_COMMENT>sladfkj weoiu sdflkj
  </R_COMMENT>
</REGION>
```

Figure 2.4 **Sample XML document**

2.4.1 SQL to XQuery Translation using the XML Wrapper

For our experiments, we modified an existing product called the XML Wrapper, which is part of the IBM DB2 Information Integrator, version 8.1. The unmodified XML Wrapper provides a mechanism for presenting relational views of XML data stored as text files on disk. Each relational view defined over an XML document is called a nickname, and utilizes syntax similar to the CREATE TABLE statement.

In Figure 2.3 we show a possible CREATE NICKNAME statement that DB2 would use in conjunction with the XML Wrapper to query the XML document shown in Figure 2.4. The product version of the XML Wrapper uses the Xerces [30] XML parser and the Xalan [29] XPath evaluator to find data in the XML document(s). Both Xerces and Xalan are subprojects of the Apache XML project [1]. The wrapper queries the XML and creates relational rows conforming to the CREATE NICKNAME statement to hand back to the database engine.

Because XML allows hierarchical nesting of elements, entities may be stored physically together in the same document. For example, you might store a Customer with all of the

Orders he has placed as child elements of the Customer. To exploit this, the XML wrapper allows special columns to be specified as the PRIMARY_KEY or FOREIGN_KEY for a nickname. For example, a column ‘fk’ in a nickname for ORDERS may be defined as the FOREIGN_KEY of the PRIMARY_KEY column ‘pk’ in a nickname for CUSTOMER. When a SQL query references these two nicknames with an equality join predicate between fk and pk, the wrapper knows that any Order information returned will be found as sub-elements of the Customer in the XML document. Any paths specified by the ORDERS nickname must be relative to the XPATH specified for the CUSTOMER nickname. The value for the pk column is simply a serialization of a Xerces element reference.

For our experiments, we modified the existing XML Wrapper to be an interface to a prototype XML store. Since our data had been previously parsed and stored in this native XML store, we removed the Xerces code. Also, the Xalan XPath evaluator could not be used, since it operates over in-memory DOM trees only. In its place, we used a custom XPath evaluation engine that evaluates paths over the prototype XML store. To implement the PRIMARY_KEY column option, we used an internally generated XML node identifier. We present an overview of our experimental architecture in Figure 2.5.

One of the primary advantages of a native XML store is that we have an opportunity to create one or more indices over the loaded data. The prototype XML store contains a path-based XML indexing module, but lacks automatic XML index selection in the query optimizer. To enable using each XML index created, we wrote custom parameterized table functions that take a key value as input and return relational rows that are the inner join

result for that key value. This works because the XML index stores the same XML node reference value that the XPath evaluator uses. This idea also allows us to hand-optimize the join order for queries that refer to more than two nicknames by using a column from the result of one table function call as the input for another. For example, if we want to force a scan of the CUSTOMER nickname to be the outer entity in an index nested-loops join with ORDERS, we would write the following SQL query:

```
SELECT O.O_ORDERDATE
FROM CUSTOMER C, tfORDERS(C.C_CUSTKEY) O;
```

In this example, `tfORDERS()` is a user-defined table function that takes as input a customer key and returns columns from the ORDERS nickname from rows that contain a matching `O_CUSTKEY` value. The XML documents that match are found by performing a lookup in the XML index to find all documents which contain the path /ORDERS/O_CUSTKEY/text() = [C_CUSTKEY], where C_CUSTKEY is the value passed to the table function.

2.4.2 Prototype Walkthrough

To illustrate the prototype ROX architecture, we will describe how the following SQL query is executed:

```
SELECT r_name,
  COUNT(n_nationkey) AS n_count
FROM region, nation
WHERE r_regionkey = n_regionkey
GROUP BY r_name;
```

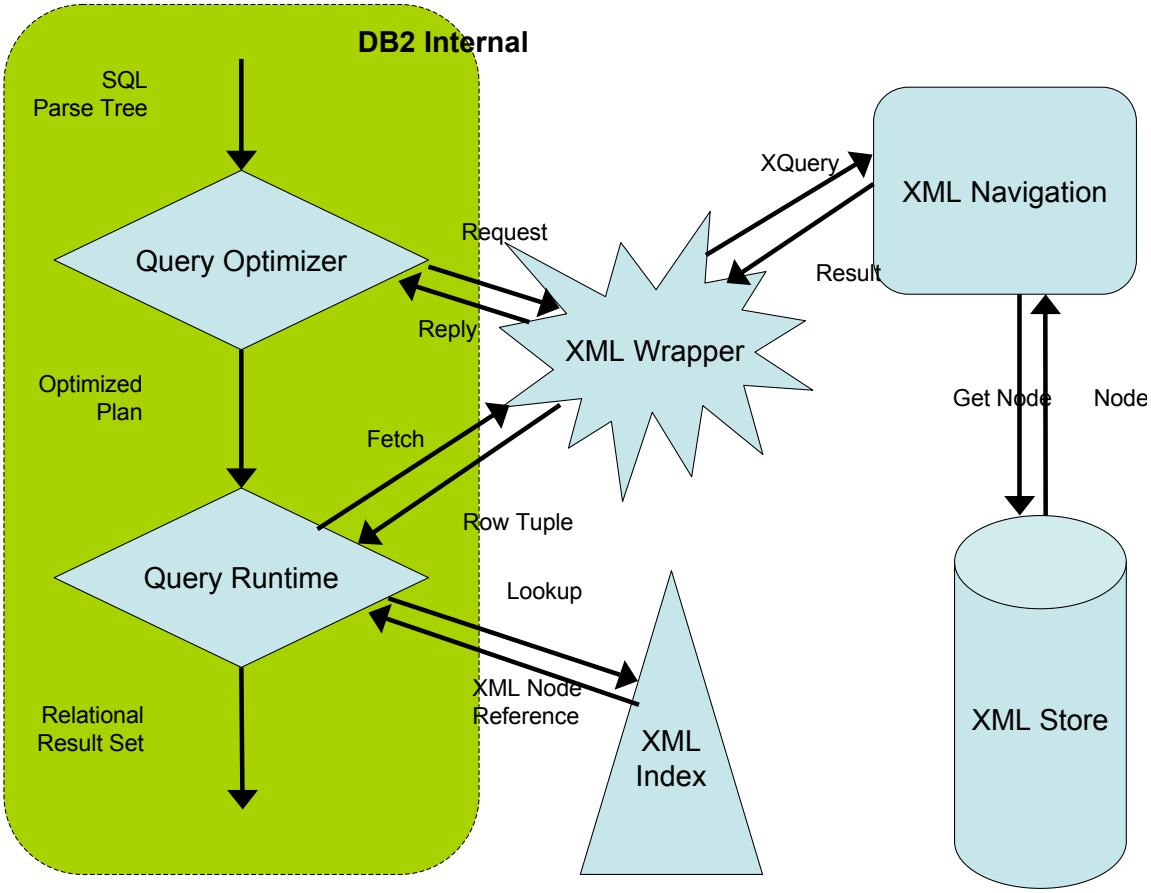


Figure 2.5 **Experimental Architecture**

Logically, the input to the query optimizer is a SQL parse tree. For our example query, it will contain references to our nickname definitions for REGION and NATION, as well as the R_REGIONKEY = N_REGIONKEY predicate. Since DB2 only knows about the definition, it must consult the server specified by the CREATE NICKNAME statement, and therefore our modified XML wrapper, to create alternate execution plans and cost estimates. Plans enumerated include plans for: REGION only, NATION only, a plan that pushes the equality predicate into the wrapper and returns rows containing both REGION and NATION columns, and a plan for NATION which takes as input a context R_REGIONKEY and returns rows with an equal N_REGIONKEY column. We would accept this last plan only if R_REGIONKEY was defined with the PRIMARY_KEY option in the REGION nickname, and N_REGIONKEY defined with the FOREIGN_KEY option and referencing the REGION nickname. With a full cost model in the wrapper, we could tell the optimizer that one or more of these plans would provide the best performance. For each plan the wrapper can accept, we create a structure containing everything necessary to execute the plan later at runtime, and return control back to the optimizer. In the prototype, the structure would contain an XQuery to be executed at runtime. For example, we would create the following XQuery when asked to scan the REGION nickname:

```
for $a in /REGION, $b in $a/R_NAME
return $a, $b;
```

Once the optimizer chooses a final query plan, the query runtime takes control and begins to execute the plan. Any operator in the plan containing a packed structure created by the wrapper during optimization now calls back into the wrapper requesting to open a

cursor based on the information contained in that structure. For our example query, the first request might be to do a table scan on the REGION nickname.

For each row returned from the wrapper for that scan, a second cursor would be opened over the NATION nickname, with an additional parameter containing the value of the R_REGIONKEY column for the current REGION row. Recall that the value of the R_REGIONKEY column would be the internal XML node identifier for the REGION element parenting the NATION information to return. The XML navigation would begin with the node identifier passed in, rather than from the document root. If the XML nodes are stored on disk in document order, we likely have the relevant NATION elements already in memory. The prototype expects DB2 to perform the calculation of the N_COUNT output column and to handle the GROUP BY R_NAME clause. Note that better performance could be achieved for this query by pushing the aggregate down into the XML Wrapper, but the prototype did not do so.

2.4.3 Experimental Dataset

We chose the TPC-H [18] dataset for our experiments. This dataset is well known throughout both the industrial and academic research communities, and is representative of a normalized relational schema that can be adapted to the ROX model. The schema consists of eight entities, namely REGION, NATION, SUPPLIER, PART, PARTSUPP, CUSTOMER, ORDERS, and LINEITEM. The PARTSUPP entity exists to allow a many-to-many relationship between PART and SUPPLIER.

For the corresponding XML schema of this dataset, we have quite a few choices. As with the relational schema, we discard any choice which results in data duplication. Please refer to Section 2.3.2 for our discussion of data normalization. We compare three XML schemas for our experiments, named Unnest, Nest2, and Nest3. Our Unnest schema consists of one XML document per relational row per relational table. The root element of each document is the name of the relation from which it came, each sub-element the name of a column from that relation, and the text contained in each sub-element is a value from the row that we used to generate the document. Figure 2.4 shows an example XML document created from one row of the REGION table. Our Nest2 schema stores LINEITEM elements nested within the correct ORDERS element, and PARTSUPP within PART, but leaves the remaining data as in the Unnest schema. Finally, the Nest3 schema stores LINEITEM elements within ORDERS elements, which in turn are nested within the correct CUSTOMER element, with all other data as in the Unnest schema. With the TPC-H schema, it is not possible to create a semantically meaningful, properly normalized document with four levels of nesting.

2.5 Experiments and Results

This section presents the experimental results we gathered to validate the feasibility and performance of the ROX model.

All experiments were executed on a quad processor PowerPC-based machine running AIX 5.1, equipped with 16GB of main memory and SCSI disks. Data and indices were

loaded into separate DB2-managed tablespaces striped across 22 5GB SCSI disks. All timings reported in this section are an average of 5 runs. We calculated that all timings for each average are within 1% of the average value with 95% confidence.

All experiments are run using data generated at TPCH Scale Factor 0.1. This means our largest entity, LINEITEM, has approximately 600,000 rows. The raw data is nearly 100 MB on disk.

2.5.1 Storage Comparison

In this section, we examine the storage requirements of both the relational and XML versions of the TPC-H data set. The number of disk pages required to store the data has a direct impact on the cost of any sequential scan. For this experiment, we loaded several of the TPC-H relations into both standard DB2 tables and our native XML storage engine, and present the disk storage requirements in Table 2.1.

Relation(s)	Relational	XML
CUSTOMER	2656	13312
ORDERS-LINEITEM	100960	888832
PART-PARTSUPP	15904	66560

Table 2.1 **Relational and XML Storage Requirements for selected TPC-H relations, in KB**

It is clear that a generic XML store generates significant storage overhead when compared to the same data stored relationally. These overheads are due mostly to three factors. All text data is stored in Unicode format in the prototype XML store. Although DB2 allows tables to store Unicode data, it does not do so unless explicitly asked to by the user. This

is a factor of two size increase for any text data in the TPC-H tables. Secondly, a generic XML store must duplicate the document structure for every relational record converted to XML format. For XML documents with a high structure-to-data ratio, this overhead is high. Finally, our XML storage engine currently stores all XML data in text format. For any numeric data, this adds significant storage overhead. Storage for the element tags does not require significant overhead, however. Each unique element and attribute name is entered into a mapping table, which allows us to store an integer tag ID for each document node.

2.5.2 Bufferpool Effects

As discussed in Section 2.5.1, the storage required for the XML schemas under test is significantly more than for the relational version of the data into DB2. It therefore makes sense to consider the effects of varying the size of DB2's bufferpool on query performance. We chose to run each series of queries using four different bufferpool sizes. The sizes were chosen to be 10%, 25%, 50%, and 100% of the total storage (data and indices) required for the schema. Using this definition implies that the 10% case for the relational schema is a much smaller number of pages than for the 10% XML schema case. Please refer to Table 2.2 for the specific bufferpool sizes tested. It may seem unfair to use different bufferpool sizes for the tests. After all, when 100% of the relational data and indices fit in memory, only 11% of the XML data and indices fit. Further, given a specific memory budget, the relational data has a size advantage and should benefit from it. However, if the

scale factor of the data increased, we would not have a choice but to choose a bufferpool size $<$10% in both cases.

Size	Relational	XML
10%	450	4000
25%	1125	10000
50%	2250	20000
100%	4500	40000

Table 2.2 **Tested bufferpool sizes (in number of 32K pages)**

We present a graph in Figure 2.6 which illustrates the effects that the bufferpool size has on the performance of TPC-H query 10. The results presented are normalized to the execution time when the 100% bufferpool size is used. For the relational schema, additional memory directly contributes to decreased query execution time. However, additional memory does not give an advantage for queries executed using the XML schemas. The additional CPU cost of navigating through the XML schema and converting the retrieved text to the correct column datatype may be to blame. Similar results were obtained for the other queries we tested.

When the number of bufferpool pages are roughly the same for the relational and XML schemas, the relational schema appears to be the clear winner. In Figure 2.7, we present the results of two queries executed over all schemas. The relational schema was tested at the 100% bufferpool level of 4500 pages, while the XML schemas used their 10% level of 4000 pages. For Q10, the best XML schema is still a factor of about 19x slower than the relational schema. For Q22, we see that the Unnest and Nest2 schemas are about a factor of 5 slower.

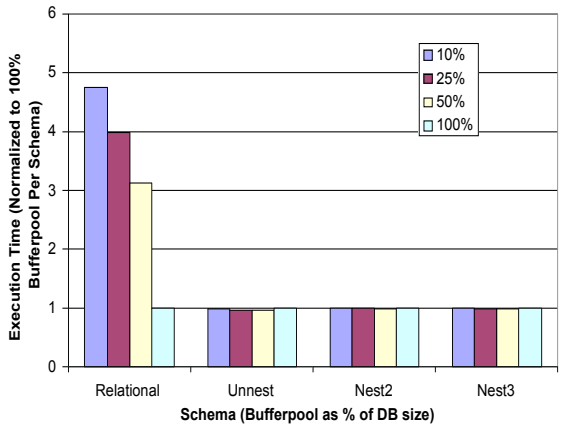


Figure 2.6 **TPC-H Q10 bufferpool effects. For each schema, the execution times are normalized to the 100% bufferpool execution time. The percentages listed are relative to the total size of the data and indicies being tested.**

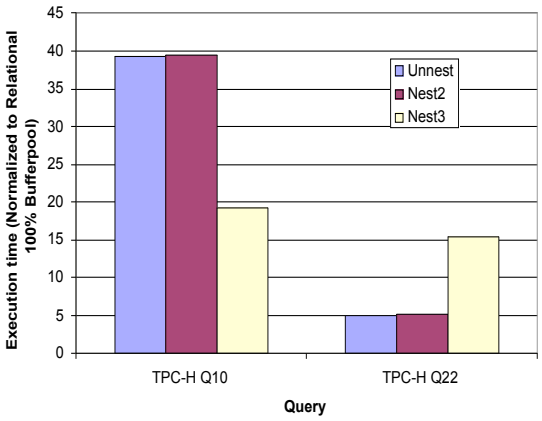


Figure 2.7 **Bufferpool effects when all queries are executed with approximately the same number of bufferpool pages**

2.5.3 Schema Variations

In this section, we discuss the measured effects of varying the nesting of the XML schema. All experiments discussed in this section assume a 10% bufferpool size. As the level of nesting in each XML document is increased, we encounter mixed performance results. Consider the graph in Figure 2.8, which shows the normalized execution times for two TPC-H queries for our three XML schemas. TPC-H Q10 is called the Returned Item Reporting Query. This query is basically a four-way join between NATION, CUSTOMER, ORDERS, and LINEITEM. This query fits our Nest3 schema extremely well, and the results show that this query is about twice as fast with Nest3 as with either the Nest2 or Unnest schemas. One obvious question, though, is why we don't see any benefit from the Nest2 schema, which has LINEITEM nested in ORDERS. The answer lies in the fact that

we are utilizing the XML index to join CUSTOMER to ORDERS in both cases, and also for ORDERS to LINEITEM in the Unnest case. As we will see in the next section, the XML index performs very well and brings Unnest's performance in line with Nest2. Although Nest3 was the clear favorite for Q10, it suffers for other queries such as Q22. This query scans CUSTOMER looking for customers in specific countries who have never placed an order but have a good account balance. The country selection predicate is reasonably selective, and so the join to ORDERS can be avoided for most customers. The Nest3 schema performs very poorly for this query due to the storage of each CUSTOMER's ORDERS and LINEITEM information—the very attribute that made it much better for Q10. Since this query does not use the ORDERS information very often and never uses the LINEITEM information, we needlessly pay to load them from disk. CUSTOMER information packs much better in the Unnest and Nest2 schemas, as both utilize the CUSTOMER-only XML document format. These results suggest that the XML schema chosen should factor in the expected query workload, if known.

2.5.4 XML Index Exploitation

We now consider the benefits of utilizing the XML index to aid in joining across documents. In all of our chosen XML schemas, we maintain some normalization. For example, all of our schemas keep PART and LINEITEM unnested. To find the name of a part given a specific lineitem, we must do a standard join. In this section, we compare two types of joins—index nested loops join using the XML index, and DB2's hash join. In Figure 2.9, we show results for TPC-H query 5—the Local Supplier Volume Query. This query joins

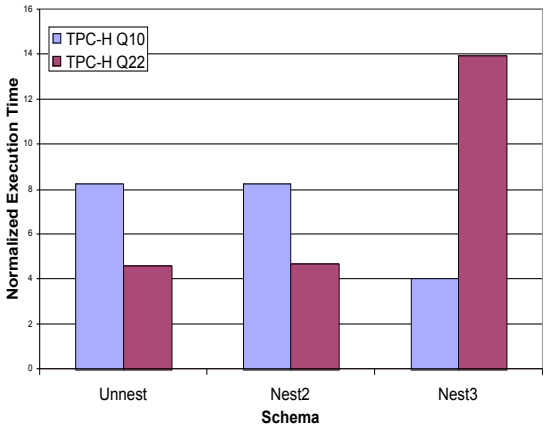


Figure 2.8 **Schema Effects for two TPC-H queries, normalized to the Relational time for each query**

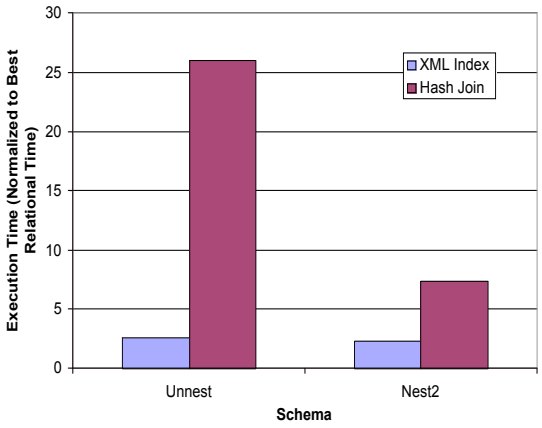


Figure 2.9 **TPC-H Q5: Local Supplier Volume Query**

six of the eight tables using a total of six equijoin predicates. The extra join ensures that the supplier and customer are from the same nation. Here, we have normalized the results to the execution time for the Relational schema. For this query, utilizing the XML index provides a very tangible benefit. The Unnest schema shows a much larger improvement over the hash join plan than the Nest2 schema, with both schemas at about 3 times the relational query time when the XML index is used. In the Hash Join case, the Unnest2 performs better because the plan executed still takes advantage of the nesting of LINEITEM in ORDERS, thereby obviating a very expensive join.

These results show that our immature ROX prototype can achieve performance within a factor of three of a mature relational database system for an important category of queries. With proper tuning and optimization of the storage format, XML navigation, and XML index utilization, even better results could be obtained.

2.6 Conclusions/Future Work

In this chapter, we have discussed an alternate solution to the problem of integrating relational and XML data sources to support both new XQuery and legacy SQL queries. We called this solution the ROX model, and described the architectural tradeoffs involved. Our solution allows existing SQL applications to continue to run unmodified, and allows a gradual transition of some or all data to the XML storage format. We created a prototype to compare the performance of the ROX model to the standard relational model, and found that it can compete for an important class of queries. We found that the choice of the XML schema to represent the relational data can have a profound impact on performance. Also, by utilizing an index over the XML storage, we could achieve performance within a factor of three of a mature relational DBMS for queries with many joins.

Many research questions remain open for future work. Updates in a native XML storage system can pose problems such as document order anomalies and subtree locking issues. Further, an XML update standard (or even a candidate specification) does not yet exist. It would be interesting to consider the ROX model as the XML update standard is created.

Storage overheads associated with general native XML stores are a significant source of performance problems when using the ROX model to perform sequential scans. Identifying ways to store XML compactly and exploring tree storage alternatives based on document access patterns are interesting areas for future research.

For a given query workload and XML schema, we could utilize the query optimizer to create alternate plans that would be possible if a different XML schema was available, and

use this information to automatically suggest a better XML schema for that query workload, much as was done in DB2's Index Advisor [38] and Design Advisor [41].

Resolving these questions will bring us closer to the time when XQuery and SQL queries can both be processed efficiently against both structured and semistructured databases.

Chapter 3

A Comparison of C-Store and Row-Store in a Common Framework

Recently, a “column store” system called C-Store has shown significant performance benefits by utilizing storage optimizations for a read-mostly query workload. The authors of the C-Store paper compared their optimized column store to a commercial row store RDBMS that is optimized for a mixture of reads and writes, which obscures the relative benefits of row and column stores. In this chapter, we describe two storage optimizations for a row store architecture given a read-mostly query workload—“super tuples” and “column abstraction.” We implement both our optimized row store and C-Store in a common framework in order to perform an “apples-to-apples” comparison of the optimizations in isolation and combination. We also develop a detailed cost model for sequential scans to break down time spent into three categories—disk I/O, iteration cost, and local tuple reconstruction cost. We conclude that, while the C-Store system offers tremendous performance benefits for scanning a small fraction of columns from a table, our optimized row store provides disk storage savings, reduced sequential scan times, and low additional CPU overheads while requiring only evolutionary changes to a standard row store.

3.1 Introduction

Recently, a column-oriented storage system called C-Store [36] has shown provocative performance results when compared to a commercial row-oriented DBMS. Their comparison of the read-optimized C-Store ideas to a write-optimized commercial DBMS obscures the relative benefits of row and column storage for read-mostly workloads. For example, one sequential scan query in the C-Store evaluation takes 2.54 seconds for C-Store while the DBMS takes 18.47 seconds—even with a materialized view that directly answers the query. In this chapter, we show that the row store can also be optimized for a read-mostly query workload, and the query above can be run in as little as 1.42 seconds with our optimized row store. In an attempt to shed light on the comparison between the two, we implement both a read-optimized row store and the C-Store system in a common framework.

The C-Store architecture uses several main techniques to improve performance when compared to current commercial relational systems. First, C-Store stores each column of a relation separately on disk. In this way, scanning a small fraction of the columns from a relation with many columns saves disk I/O. Second, it carefully packs column values into large page-sized blocks to avoid per-value overheads. Third, C-Store uses a combination of sorting and value compression techniques to further reduce disk storage requirements. Both the page packing and sorting/compression techniques are an attempt to trade decreased I/O for increased CPU utilization.

The performance evaluation presented in the C-Store paper uses a modified TPC-H [18] schema and query workload to measure the combined effects of their performance improvement techniques. The reported results are very impressive—the C-Store system provides a significant performance improvement compared with a commercial row store. Although the commercial row store compares poorly, an optimized row store can benefit from most of the same performance techinques proposed for the C-Store system. Specifically, our optimized row store uses both careful page packing, which we call "super tuples," and sorting to enable compression, which we call "column abstraction." The remaining technique—column storage—is the primary difference between row- and column-oriented storage. Careful page packing is particularly low-hanging fruit for a row store. Enforced sorting of the relation and storing repeating values only once to save space is slightly more effort, as it breaks the one-to-one mapping of the logical relational schema to the physical tuple on disk.

The main contributions of this chapter are as follows:

- We provide descriptions of the "super tuple" and "column abstraction" performance techniques to optimize for a read-mostly query workload.

- We build a software artifact to evaluate these performance improvements for both the row and column stores in isolation and in combination, using a common storage manager. Our experiments vary tuple width, number of rows, level of sorting and column abstraction, and number of columns scanned to identify performance trends.

- We propose and validate a formal cost model for sequential scan for both row and column storage. The model takes into account the storage improvements and their effects on performance by identifying three factors which contribute to overall scan time. We compare the model predictions with our experimental results. We also use the model to forecast the behavior of systems and scenarios to gain further insight into performance trends.

The rest of the chapter proceeds as follows. In Section 3.2, we present the storage optimizations and implementation details. Section 3.3 describes our experimental prototype and evaluates the storage optimizations in isolation and combination to discover performance trends. We then develop our formal cost model, with validation and forecasting, in Section 3.4. We describe related work in Section 3.5, and offer conclusions and future directions for research in Section 3.6.

3.2 Storage Optimizations

In this section, we describe the "super tuple" and "column abstraction" optimizations for the row store architecture. To illustrate the effects of each storage option, we will use an instance of a materialized view defined in the C-Store [36] paper. The view is based on a simplified version of the schema from the TPC-H benchmark [18], and is defined using SQL as follows:

```
CREATE VIEW D4 AS
SELECT L_RETURNFLAG, C_NATIONKEY, L_EXTENDEDPRICE
FROM   Customer, Orders, Lineitem
WHERE   C_CUSTID = O_CUSTID
```

L_RETURNFLAG	C_NATIONKEY	L_EXTENDEDPRICE
A	3	23
A	3	34
A	9	64
N	3	88
N	14	49
R	9	16
R	9	53
R	9	7
R	11	63
R	21	72
R	21	72

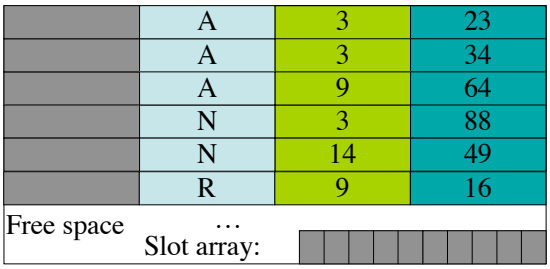
Table 3.1 **Example instance of materialized view D4**

```
 AND O_ORDERID = L_ORDERID
ORDER BY   L_RETURNFLAG, C_NATIONKEY;
```

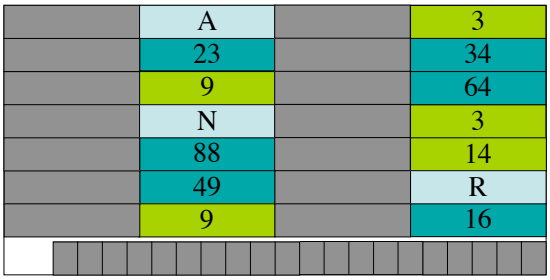
The definition is identical to the view called D4 in the C-Store paper with the exception of the secondary `ORDER BY` on the `C_NATIONKEY` column. Table 3.1 contains an instance of the D4 view that we use in all examples for this section. Figure 3.1(a) shows how a standard row store would layout the first few rows of the D4 view on a disk page.

3.2.1 Super Tuples

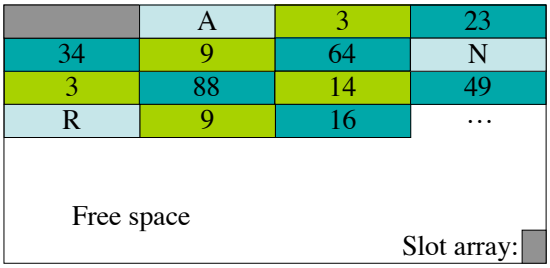
All of the major DBMS products use a variant of the slotted page for storage of tuples in a table. Slotted pages use an array of slots that point to the actual tuples within the page. Typically each tuple is prefaced by a header that provides metadata about the tuple. For example, metadata in the Shore storage manager [15] includes the type of tuple (small or large), the size of the user-specified record header, and the total size of the record if it is larger than one page and split across disk pages. The tuple header is implementation specific, but typically is 8-16 bytes in addition to the tuple's slot entry.



(a) Standard Row Store



(b) Column Abstraction



(c) Super Tuples and Column Abstraction

Figure 3.1 **Row layout on storage pages for view D4 for (a) the standard row store, (b) row store with column abstraction, and (c) row store with super tuples and column abstraction**

While the slotted page design provides a generic platform for a wide range of data storage needs, these per-tuple overheads can be problematic. Even for an 80 byte tuple, a 16 byte overhead is 20%. We reduce per-tuple overhead by packing many tuples into page-sized "super tuples." For fixed-length tuples, the super tuple is an array of tuple-sized entries which can be indexed directly. For variable length tuples, the tuple length must be stored. The super tuple design uses a nested iteration model, which ultimately reduces CPU overhead and disk I/O.

An important side effect of using super tuples is that external addressability of individual tuples is more difficult. Both the C-Store design and our optimized row store trade the storage benefits derived from tight packing of values for additional overhead associated with utilizing and maintaining value indexes. We present an analysis of the effects of super tuples on index-based predicate evaluation in Chapter 4.

3.2.2 Column Abstraction

Sorting provides an opportunity for disk storage savings. If the database can guarantee that tuples are retrieved from storage according to the sort order, we can store each unique value in the sort column once and then store the remaining unsorted attributes separately, according to the specific storage architecture. In this chapter, we use the term "column abstraction" to describe the process of storing repeating values once. Disk space savings are higher when the number of unique values in the sorted column is smaller.

The columns in a materialized view may come from different tables and be related to each other by one or more join keys. For example, consider the one-to-many relationship

L_RETURNFLAG	C_NATIONKEY	L_EXTENDEDPRICE
A	***3***	***23***
A	3	***34***
A	***9***	***64***
N	***3***	***88***
N	***14***	***49***
R	***9***	***16***
R	9	***53***
R	9	***7***
R	***11***	***63***
R	***21***	***72***
R	21	***72***

Table 3.2 **Column abstraction encoding of data from Table 3.1. Only need to store values in boxes – other values are implicit.**

between the C_NATIONKEY and L_EXTENDEDPRICE columns in our example D4 materialized view. Even when D4 is sorted by L_RETURNFLAG first, we can save space on disk by storing C_NATIONKEY once for each related L_EXTENDEDPRICE. We show in Table 3.2 how the sort column(s) for view D4 can be used to more efficiently encode the same data. We show which values must be stored on disk by drawing boxes around them. L_RETURNFLAG and C_NATIONKEY are sort attributes for D4 which allows us to store repeating values for each attribute only once. Note that C_NATIONKEY is sorted only within each unique L_RETURNFLAG value, so we must store values such as 3 and 9 more than once. Figure 3.1(b) shows how we layout pages in our optimized row store using column abstraction for view D4. Note that storage needs have increased due to additional row headers for the abstracted columns. In Figure 3.1(c), we show that combining super tuples with column abstraction creates a more efficient disk page layout.

In general, a view may not specify an explicit sort. However, referential integrity constraints may specify an enforced one-to-many relationship between two or more tables in the view definition. We use the referential integrity information to insert an implicit sort on the columns from the one side of the one-to-many join(s). It is sufficent to sort on the primary key of the "one" side of the join. If the view does not project the key, we sort by all columns mentioned on the one side of the join. As an example, consider our instance of view D4. With enforced one-to-many relationships for Customer to Orders and Orders to Lineitem, we add a secondary sort on the C_NATIONKEY column when the view is not already sorted by that column. This implicit sort opens up another opportunity to store the

repeating column(s) only once to save space. Sorting must be performed only once during population of the materialized view. At query runtime, scanning the view produces tuples in the correct sort order without additional sorting.

3.2.3 Updates and Indexing

Both C-Store and our optimized row store pose problems for updates and indexing. This is a result of a deliberate decision to optimize for scan-mostly read-mostly workloads. Our goal in this section is not to prove that our optimized row store can be efficiently updated, but rather, to mention that data in row-stores with our optimizations can be updated and indexed, although the performance of these operations will not match their counterparts in a standard row-store.

The super tuple and column abstraction optimizations create additional inconvenience in processing updates for both C-Store and row stores. Inserting rows may force super tuples to be rebuilt or split across two pages. Updates to existing rows may force several rows in the table to be deleted and reinserted elsewhere. C-Store takes a “snapshot isolation” approach to handling updates in batch, and a similar technique can be used in our optimized row store.

Indexing columns in tables optimized for read-mostly also presents implementation challenges. C-Store only allows indexes on the primary sort column of each “projection”. Their design allows updates to the index to be bounded to a specific range of values in the index, as the values and pages containing those values are correlated by the sort. Indexes on other columns of the table are possible for both C-Store and the optimized row store,

but maintenance is expensive when table records move within a super tuple or are split to a new page due to inserts.

3.3 Evaluation

To evaluate the performance benefits of specific storage improvements, we created an experimental prototype. The prototype is designed to allow each storage optimization introduced in Section 3.2 to be applied in isolation and in combination for both the row and column stores. We report results for the column store only with the "super tuple" optimization, since the per-value overheads are several times larger than the data itself without super tuples.

We first provide a detailed description of the prototype. To calibrate the performance of our C-Store implementation, we then compare our implementation to the C-Store system [36] using query Q7 from the C-Store paper. Later in this section, we evaluate the benefits of the "super tuple" optimization for the row store, sorting and run-length encoding benefits for both the row and column stores, and finally the effects of combining the optimizations. We focus on identifying performance trends that emerge rather than trying to choose the "best" combination.

3.3.1 Experiments Description

We implemented the row store and column store architectures in a single prototype using Shore [15] as the storage manager. We implemented a sequential scan operator for the

row and column stores that can operate over the super tuple and column abstraction optimizations. We ran the experiments on a dual processor Pentium 4 2.4GHz Xeon machine with 1GB of main memory running Fedora Core 3 with a stock 2.6.13 kernel. We created a hardware RAID-0 volume using six 250GB disk drives to contain the data volumes. A separate 250GB disk stored the system catalog information. Shore was configured to use a 32KB page size and a 512MB buffer pool. All reported results are the average of five runs.

By implementing all storage architectures and optimizations in a single prototype, our goal is to hold performance variables constant while changing only the variable of interest. Our prototype avoids memory copies from the buffer pool whenever possible. Shore offers direct read-only access to data which allows us to minimize expensive copy-out operations.

In our C-Store implementation, we allocate 256MB in main memory to be divided equally among the columns scanned for sequential prefetching of pages. For example, when scanning 8 columns, we sequentially read 32MB from each column during the scan. Without prefetching, random I/O can easily dominate scan times for a column store when reading a large number of columns. The necessity for page prefetching in a column store is further motivated in [31].

We turned off locking and logging to match the settings used in the C-Store evaluation [36]. We believe this is fair since an underlying assumption of both papers is a read-mostly query workload and all queries being evaluated are read-only. We gathered results for a cold Shore buffer pool and file system cache. We ran our experiments with warm buffers as well, but do not report these results since the contribution of disk I/O to the total

a1	a2	a3	a4
1	***1***	***1***	***1***
1	1	***2***	***2***
1	1	***3***	***3***
1	***2***	***1***	***1***
1	2	***2***	***2***
1	2	***3***	***3***
1	***3***	***1***	***1***
	...		
1	4	***3***	***3***
2	***1***	***1***	***1***
2	1	***2***	***2***
	...		

Table 3.3 **Instance of Gen_2_4_3 table with boxes around actual values stored. Sorted by column a1 and a2**

scan times does not change our analysis of performance trends. To eliminate file system caching effects, we unmounted and remounted the data volume just before each cold run.

All data sets consist of rows of 4, 8, 16, and 32 integer columns with a varying number of rows per data set. We synthetically generated the data to enable exploration of various column abstraction choices. The data for each column is a simple sequence of integers, starting at 1. When a new level of column abstraction starts, the column values at each lower level of abstraction reset and begin counting from 1 again. See Table 3.3 for an example. The frequency of each value within a column is important for column abstraction, but the exact values do not matter.

To evaluate the effects of sorting and encoding techniques on sequential scan performance, we generated data sets which provide encoding opportunities. Consider the 4-column data set in Table 3.3. The rows are sorted first by column a1 and then by column a2.

We call this data set Gen_2_4_3, and it contains 24 rows in total. Recall from Section 3.2.2 that column abstraction is the process of storing repeating values from sort column only once to save disk space. For the data set in Table 3.3, we have 2 unique values in column a1 and 4 unique values in column a2. For each unique a2 value, we have 3 unique values for columns a3 and a4. We have drawn boxes around the values in the data set that must be stored when using column abstraction. We use the name of the relation to describe the number of unique values at each level of column abstraction. In this case, the name Gen_2_4_3 specifies three levels, and specifies $2 * 4 * 3 = 24$ tuples. Our experimental data sets follow the same naming convention. The chosen data sets allowed us to measure the effects of both constant rows and constant total data size for all tuple widths.

3.3.2 C-Store Query 7

To ensure that our implementation of C-Store had performance representative of the system presented in [36], we acquired their code [37] and compared the performance of their implementation of a column store with ours on our hardware. The result was that our implementation of a column store was comparable to theirs. We present one representative query as an example of the comparison. We ran query Q7 from their evaluation on our benchmark hardware to establish a baseline. We also implemented query Q7 in our Shore-based prototype, which is represented in SQL:

```
SELECT c_nationkey, sum(l_extendedprice)
FROM lineitem, orders, customer
WHERE l_orderkey=o_orderkey AND
 o_custkey=c_custkey AND
 l_returnflag='R'
```

```
GROUP BY c_nationkey;
```

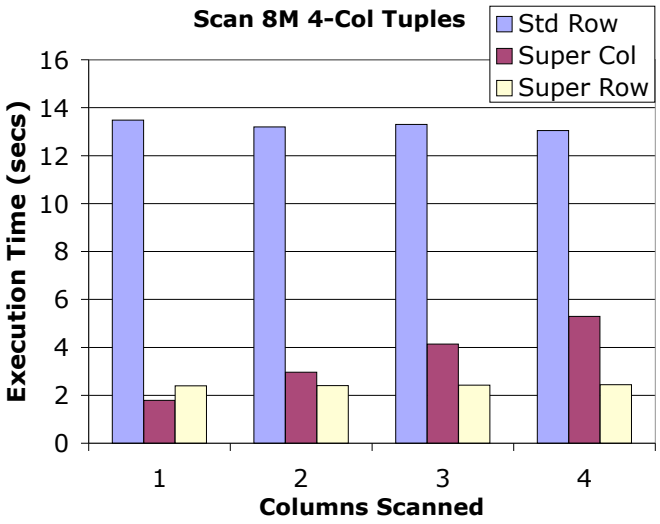
We loaded their D4 projection (materialized view) and implemented the query plan according to the method used by the C-Store system. We executed the query in our system and theirs using our hardware. The hardcoded query plan for Q7 in the C-Store prototype system assumes that the view is sorted by the L_RETURNFLAG column, and that the L_RETURNFLAG column is run-length encoded. We ran the query in the C-Store prototype on our benchmark hardware, and it took 4.67s. By contract, our Shore-based C-Store implementation took 3.95s for the same query plan, which provides evidence that our C-Store implementation does not introduce overheads that would render the rest of our experiments suspect. For comparison, the C-Store paper [36] reported a time of 2.54s for their system for query Q7 on their 3.0 GHz benchmark machine.

3.3.3 Super Tuple Effects

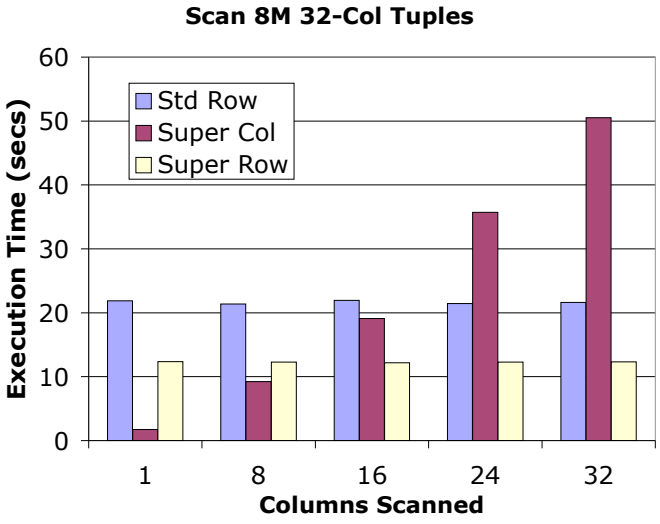
To show the benefits of the super tuple storage optimization, we performed two experiments. First, we measured the effects of varying the number of columns per tuple scanned when combined with the super tuple optimization. We then compared standard and super tuple row storage by holding rows scanned and fields scanned constant.

3.3.3.1 Vary Columns Scanned

The primary benefit of the column store design is its ability to read only the data for columns requested by a query. We show the effects of varying the number of scanned columns in Figure 3.2. For both graphs, we scanned 8 million rows.



(a) 4-Column Tuples



(b) 32-Column Tuples

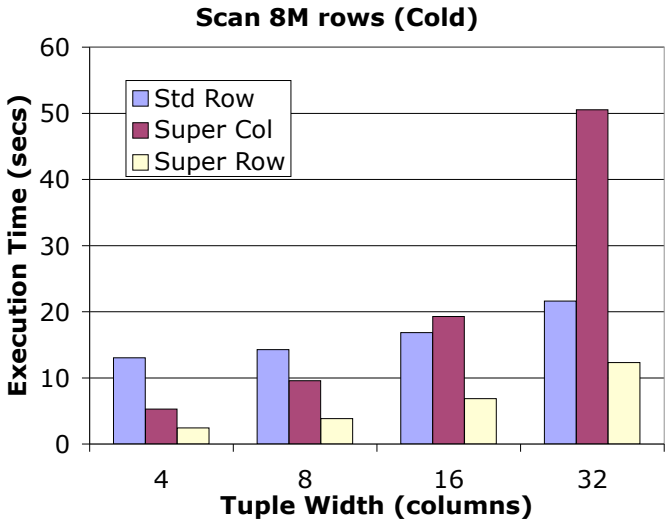
Figure 3.2 **Execution times for varying number of columns scanned for an 8M row table without abstractions for (a) 4-Column and (b) 32-Column tuples.**

In Figure 3.2(a), we used 4-column tuples and varied the number of columns scanned. The standard row store takes more than twice the time of the super tuple row and column stores. When scanning one column, we see the column store is faster than the super tuple row store, but is slower for all other cases. Turning to Figure 3.2(b), we used a 32-column tuple and scanned 1, 8, 16, 24, and 32 columns. In this case, the column store enjoys a sizable performance edge over both the standard and super tuple row stores.

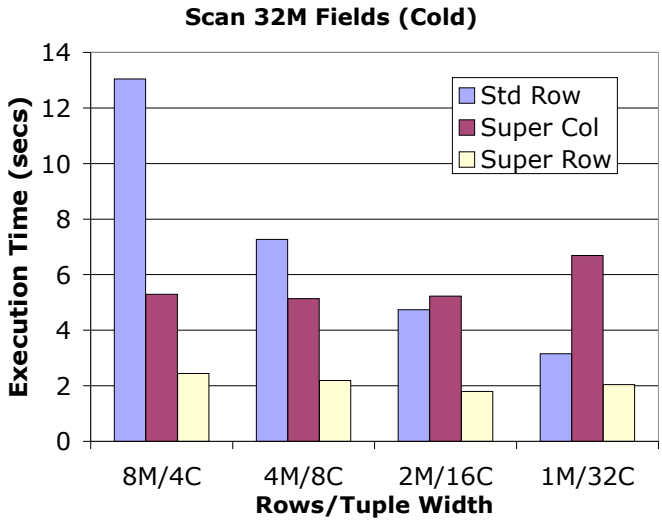
It is clear that C-Store performs extremely well when it scans a small fraction of the total number of columns in the table. This result puts us in a quandry as to how to show results for the remainder of the chapter; scanning a small fraction of the columns will show the column store as relatively better performing for all cases, while scanning all columns will show the row store in a more favorable light. We have opted to keep the optimizations separate and focus on performance trends for each storage choice individually. We therefore will scan all columns for each tuple width in all remaining graphs. The intent is to focus on the performance within each storage choice for a given storage optimization, rather than the relative performance of row and column stores.

3.3.3.2 Constant Rows and Constant Fields

To demonstrate the benefit of using “super tuples” for a row store, we present two graphs in Figure 3.3. We varied the number of columns per tuple in both graphs, but held the number of rows constant in Figure 3.3(a) and the total number of fields constant in Figure 3.3(b).



(a) Constant Row Cardinality



(b) Constant Data Size

Figure 3.3 **Super tuple effects when holding (a) rows and (b) fields constant. For the standard row store, small tuple sizes in both cases hurt performance.**

When the number of rows is held constant, as in Figure 3.3(a), the amount of data being scanned doubles as the tuple width doubles. We see that the scan times for all storage choices are increasing as the tuple width increases. Interestingly, the standard row store takes 13 seconds to scan 8 million 4-column tuples, but only 21.6 seconds to scan 8 million 32-column tuples. Although we have increase the amount of data by a factor of eight, the scan time has not even doubled. Part of the reason the scan time does not increase as expected is that disk requirements are augmented by per-tuple overheads. Shore's per-tuple overhead is 16 bytes, which is 100% of our 4-column tuple and 25% of the 32-column tuple. Disk I/O costs alone are not enough to explain this behavior, however. We will revisit this issue later in the chapter.

Figure 3.3(b) deals with varying the tuple width while the number of total fields remains constant. Holding the number of fields constant as the tuple width increases implies that the number of rows must decrease. We scanned 8 million 4-column tuples, but only 1 million 32-column tuples. We held the total number of fields $(rows*columns)$ scanned constant at 32 million. Again we saw that the super tuple row store is the fastest in all cases. In fairness to the column store, these experiments were the worst case for that storage choice. We expected that scan times for all storage choices would stay roughly constant for a constant data size. While we saw constant scan times for the column store and the super tuple row store, the scan times for the standard row store dramatically decreased as the tuple width increased. Again, disk I/O is part of the story due to the elimination of 7 million per-tuple overheads. We saw a crossover point between the standard row store and the column store

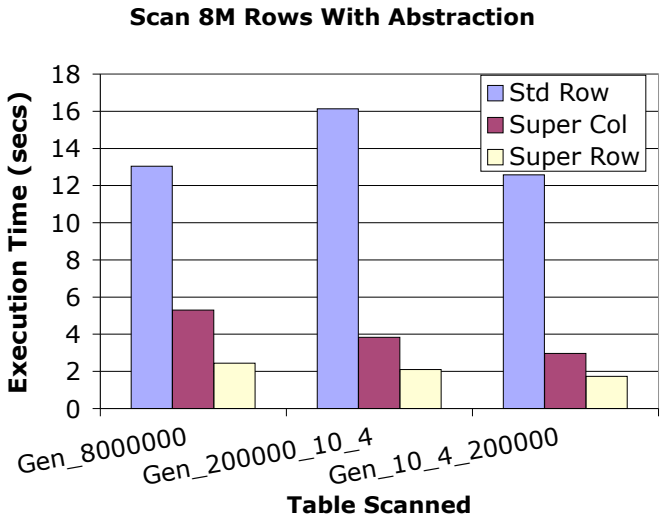
just below the 16-column tuple mark due to the marked decrease in standard row store scan costs.

For all of these experiments, we see that adding super tuples to standard row storage makes a significant difference in execution time for sequential scan.

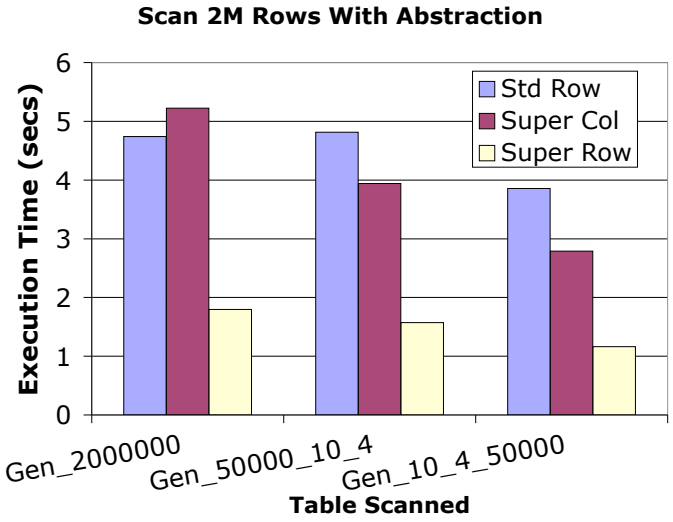
3.3.4 Column Abstraction Effects

We now turn our attention to the effects of column abstraction. We generated synthetic data sets specifically to demonstrate how varying the amount of repeating data affects scan performance. We expect scan times to decrease as we increase the number of columns and the amount of data to be stored by using the column abstraction technique. To verify this hypothesis, we present two graphs in Figure 3.4. We hold the number of fields scanned constant at 32 million in both graphs.

Figure 3.4(a) shows three column abstraction choices for an 8 million row table with 4-column tuples. Gen_8000000 uses no abstraction to provide a baseline for comparison. Gen_200000_10_4 stores three abstraction levels with one column in the first level with 200000 unique values, one column in the second level with 10 values per first level tuple, and two columns in the leaf level with 4 values per second level tuple. This table is similar to the join cardinalities of Customer, Orders, and Lineitem from the TPC-H schema, respectively. Gen_10_4_200000 also has three abstraction levels, but has ten unique values at the first level, 4 second level tuples per first level, and 200000 leaf tuples per second level tuple. This table is more like the D4 view we used in Section 3.2 as an example, with L_RETURNFLAG at the first level, C_NATIONKEY at then second



(a) 8M 4-Column Tuples



(b) 2M 16-Column Tuples

Figure 3.4 **Execution times for varying column abstractions for 32M fields using (a) 8M 4-Column and (b) 2M 16-Column tuples.**

level, and L_EXTENDEDPRICE at the third level. As the amount of abstracted data increases, we see a general trend for the scan times of the column store and the super tuple row store to decrease. Interestingly, the scan time for the standard row store increases from Gen_8000000 to Gen_200000_10_4. We recall that column abstraction increases the total number of physical tuples for a row store. When combined with per-tuple storage overhead in the standard row store, it becomes clear why scan time might increase for certain data sets and abstraction levels.

The benefit of column abstraction with a standard row store depends on the number of additional tuples created by the process more than the savings in disk I/O. If disk I/O is the primary bottleneck, the standard row store should always be faster with column abstraction, not slower in some cases as seen in Figure 3.4. We break down the total scan time in Section 3.4 to identify the contributing factors.

3.4 Cost Model and Analysis

In Section 3.2, we presented the basic storage optimizations along with implementation-specific details for row and column stores. In Section 3.3, we identified several performance trends for the storage optimizations in isolation and combination. In this section, we develop a cost model for sequential scans for several reasons. First, it will verify our understanding of the costs that determine the relative performance of a standard row store and the super tuple row and column stores. Second, having an accurate cost model allows us

to vary system parameters and/or properties of test data to forecast relative performance without actually building additional systems or loading the data.

At the most basic level, sequential scan is the most important factor in determining query performance. This is especially true when considering materialized views that have been created to exactly match the needs of a given query.

3.4.1 Cost Model Details

Our cost formulae depend on several variables, which we present in Table 3.4. The units for $SEQIO$, $RDMIO$, FC, and IC are "cost" units, which provide a basis for comparing scan costs relative to one another.

Figure 3.5 details the cost model for sequential scan of the traditional row store. We break each model down into three major contributing factors—disk I/O, iteration cost for the storage manager, and local per-tuple reconstruction cost. Tuple reconstruction, when necessary, consists of copying either a reference to the field value or the field value itself if it is small. We scale disk I/O costs by the fraction of pages expected to be in the DBMS buffer pool already. At the extremes, $F = 1$ when all pages must be read from disk and $F = 0$ when all pages can be found in the buffer pool. A traditional row store must make a call to the storage manager layer for each row in the table. If the per-iteration overhead is high, these costs may even be significant when the buffer pool is cold.

Although column abstraction reduces or eliminates data duplication, the abstract columns must be stored. For example, if we are storing columns from Customers and Orders using column abstraction, we need to store a tuple for each Customer in addition to

Var	Description
$SEQIO$	Cost of a single sequential I/O
$RDMIO$	Cost of a single random I/O
$\|R\|$	Size of storage (pages)
$\|P\|$	Size of "super tuple" storage (pages)
$\|\|R\|\|$	Cardinality of table (tuples)
C	Width of row (columns)
F	Fraction of cold pages
S	Number of columns being retrieved
FC	Cost of function call
IC	Cost of storage manager iteration
n	Abstraction levels (1 means all cols in leaf)
$C(n)$	Columns in abstraction level n
$\|\|L(n)\|\|$	Average cardinality of abstraction level n (tuples)
$\|BP\|$	Size of buffer pool (pages)
$PGSZ$	Usable size of disk page (bytes)
CSZ	Column size (bytes)
OH	Tuple overhead (bytes)

Table 3.4 **Cost Model Variables**

the tuple for each Order. However, using column abstraction may reduce the total number of disk pages ($|R|$), which will reduce disk I/O costs. With no column abstraction, we will have $n = 1$, $C(1) = C$ and $||L(1)|| = ||R||$, which simplifies the iteration cost to $||R|| * IC$.

$$SeqScan(StdRowStore) = |R| * SEQIO * F \tag{3.1}$$

$$+ \left(\sum_{i=1}^{n} \prod_{j=1}^{i} ||L(j)|| \right) * IC \tag{3.2}$$

$$+ ||R|| * FC \tag{3.3}$$

Figure 3.5 **Cost of sequential scan for standard relational storage with contributions from (3.1) Disk I/O (3.2) Storage manager calls, and (3.3) Local per-tuple overhead**

We provide a cost model for the "super tuple" row store in Figure 3.6. We base disk I/O and storage manager calls on the number of packed pages. The improvement in storage manager calls is the primary benefit of the super tuple row store, especially for small tuples.

Finally, we provide the cost model for our "super tuple" column store in Figure 3.7. We make several assumptions in this cost model. First, we assume that disk storage is uniformly distributed among the columns, which is certainly not true when a column is run-length encoded. We also assume a uniform distribution of per-column contribution to the cost of local tuple reconstruction. Finally, we model prefetching of column data pages in accordance with our prototype implementation, as described in Section 3.3.1.

In Figure 3.8, we present a model for estimating the number of pages required to store a table based on the number of rows, columns, and average column size. These formulae

$$SeqScan(SuperRowStore) = |P| * SEQIO * F \tag{3.4}$$

$$+ |P| * IC \tag{3.5}$$

$$+ ||R|| * FC \tag{3.6}$$

Figure 3.6 **Cost of sequential scan for "super tuple" relational storage with contributions from (3.4) Disk I/O, (3.5) Storage manager calls, and (3.6) Local per-tuple overhead**

$$|PS| = \frac{S}{C} * |P| \tag{3.7}$$

$$|PC| = \frac{|BP|/2}{S} \tag{3.8}$$

$$|RP| = \frac{|PS|}{|PC|} \tag{3.9}$$

$$SeqScan(SuperColumnStore) = (|RP| * RDMIO + (|PS| - |RP|) * SEQIO) * F \tag{3.10}$$

$$+ |PS| * IC \tag{3.11}$$

$$+ \frac{S}{C} * \sum_{i=1}^{n} \left(C(i) * \prod_{j=1}^{i} ||L(j)|| \right) * FC \tag{3.12}$$

Figure 3.7 **Cost of sequential scan for "super tuple" column storage with contributions from (3.7) Actual pages to scan, (3.8) Prefetch size per column, (3.9) Total random I/Os, (3.10) Disk I/O, (3.11) Storage manager calls, and (3.12) Local per-tuple overhead**

could easily be inverted to estimate row cardinality based on a measured (or sampled) count of storage pages. $ABSAV$ is a calculation of the reduction in size given information about column abstraction. Note that the sum is from 1 to $n-1$, so $ABSAV$ is zero without at least one level of column abstraction.

$$ABSAV = \sum_{i=1}^{n-1} \left(C(i) * CSZ * \left(||R|| - \prod_{j=1}^{i} ||L(j)|| \right) \right) \tag{3.13}$$

$$|R| = \frac{||R|| * (OH + C * CSZ) - ABSAV}{PGSZ} \tag{3.14}$$

$$|P| = \frac{||R|| * C * CSZ - ABSAV}{PGSZ} \tag{3.15}$$

Figure 3.8 **Calculations of (3.13) expected reduction in storage pages from abstraction, and resulting storage requirements for (3.14) regular and (3.15) "super tuple" storage.**

3.4.2 Model Validation and Prototype Performance Analysis

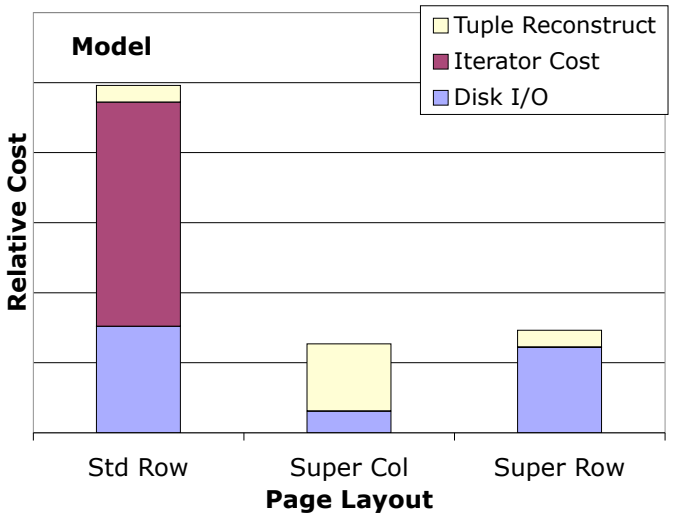
Our cost models attempt to capture performance trends as any set of variables change given constant values for the remaining variables. Before we begin the validation of our models, we must determine constant values for our prototype. Table 3.5 shows the values we hold constant and the measured values we use for $SEQIO$, $RDMIO$, FC, and IC. Their relative values were calculated from measurements taken during a scan using the prototype system on our test hardware. The values would change given other hardware - for example, $SEQIO$ would increase relative to IC and FC if we had a single disk spindle instead of the large RAID-0 array.

Var	Value
$SEQIO$	15000
$RDMIO$	450000
FC	6
IC	80
$\|BP\|$	16384 pages
$PGSZ$	32000 bytes
CSZ	4 bytes
OH	16 bytes

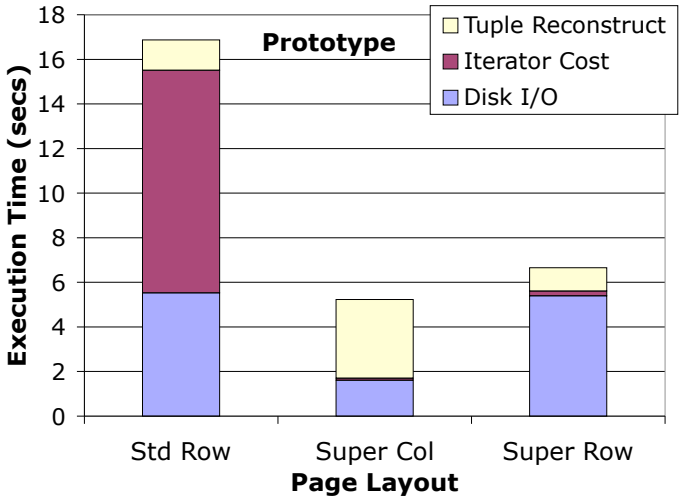
Table 3.5 **Prototype constant values for cost model variables**

In Figures 3.9 and 3.10, we show the predicted relative and actual prototype performance of scanning 4 and 16 columns, respectively, of the Gen_8000000 relation for our three page layouts. We see that the column store time increases as the number of column being scanned goes up. The increase is due mostly to the per-tuple local reconstruction cost. We also note that the cost of disk I/O decreases as the number of scanned columns decreases, as expected. Finally, we note the extremely high cost of tuple iteration for the standard row store. In contrast, tuple iteration is less than 1% of the total running time for both the column store and the "super tuple" row store. The cost model seems to track the three parts of total cost for both scans.

Figures 3.11 and 3.12 show the model prediction for scanning 32 million fields of data stored as 8 million 4-column rows and 1 million 32-column rows, respectively. In Figure 3.11 we again see the high iteration cost for the standard row store. In addition, the cost for disk I/O is very high for the standard row store compared to the "super tuple" column and row stores. Figure 3.12 tells a much different story. The model predicts that disk I/O is

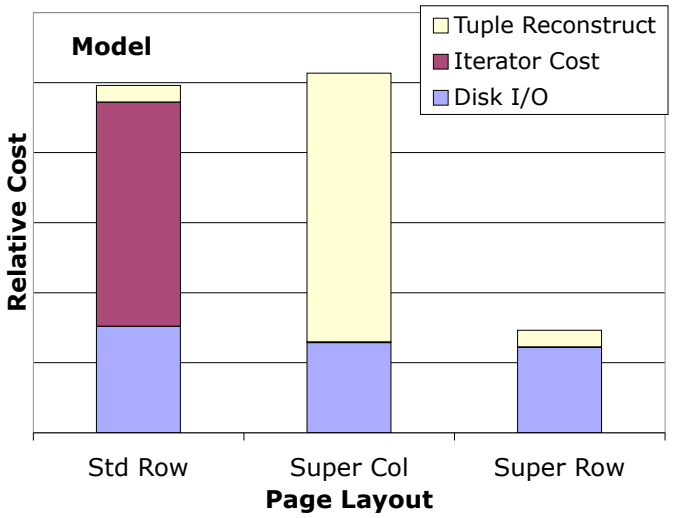


(a) Cost Model

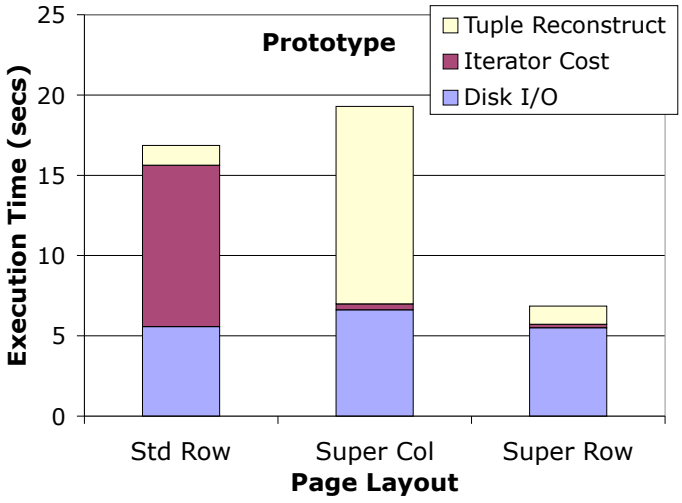


(b) Prototype

Figure 3.9 **Comparison of scanning an 8M row, 16 column table without abstractions scanning 4 columns using (a) Cost model and (b) Prototype.**



(a) Cost Model



(b) Prototype

Figure 3.10 **Comparison of scanning an 8M row, 16 column table without abstractions scanning 16 columns using (a) Cost model and (b) Prototype.**

now roughly the same for each of the page layout choices. Iteration costs for the standard row store are much lower, while tuple reconstruction has increased for the column store.

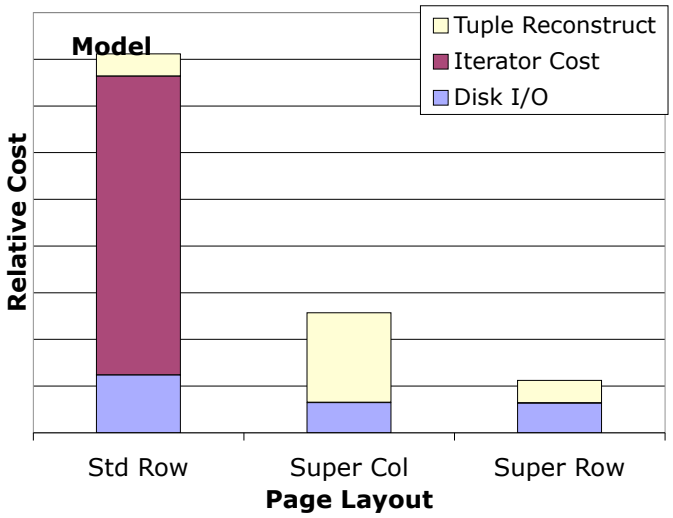
3.4.3 Model Forecasting

In Section 3.4.2, we validated our cost model against the Shore-based prototype system we created for experimental evaluation. In this section, we will change variables in the cost model to predict how systems with other characteristics would perform sequential scans.

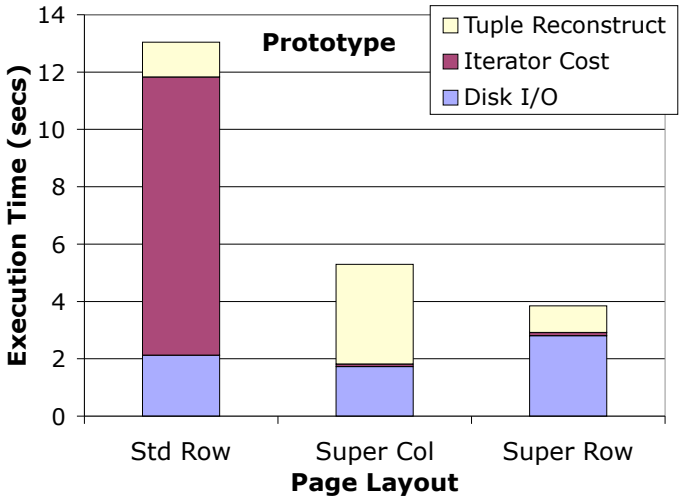
3.4.3.1 Sensitivity to Iteration Cost

Our experimental evaluation and cost model analysis demonstrates that using the Shore tuple iterator to scan a standard row store is CPU-bound. In fact, for our benchmark machine, iterating 1000 tuples on a page takes 5 times as long as reading the page from disk into the buffer pool! If possible, reducing per-tuple iteration cost for read-mostly workloads would provide a significant benefit even if no actual storage improvements are made.

Figure 3.13 shows the time for scanning 8 million 4-column tuples when the IC variable is 8 instead of the Shore value of 80. The cost model predicts that a sequential scan of all columns for the standard row store is now less than the column store scan time. Compare this graph to Figure 3.11(a) to see how dramatic the difference is. Reducing the iteration cost does not provide much performance improvement for the super tuple column and row stores—their iterations occur only once per disk page, not once per tuple. In fact, choosing the super tuple layout is a superior solution to reducing per-row iterator costs, since the iteration cost is paid once per page regardless of the number of tuples on the page.

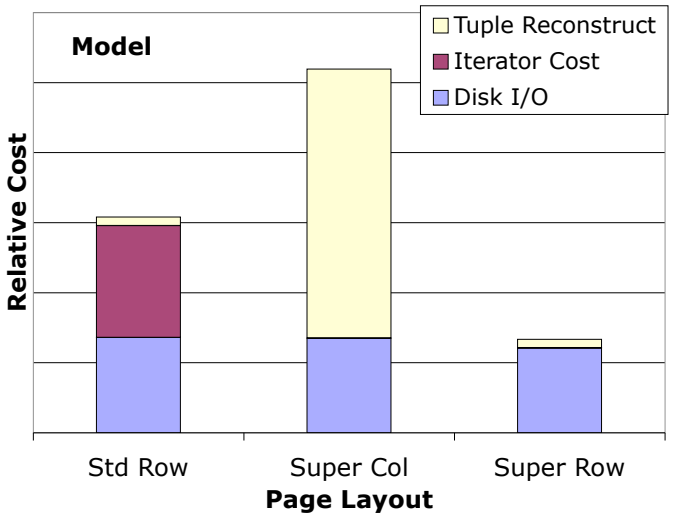


(a) Cost Model

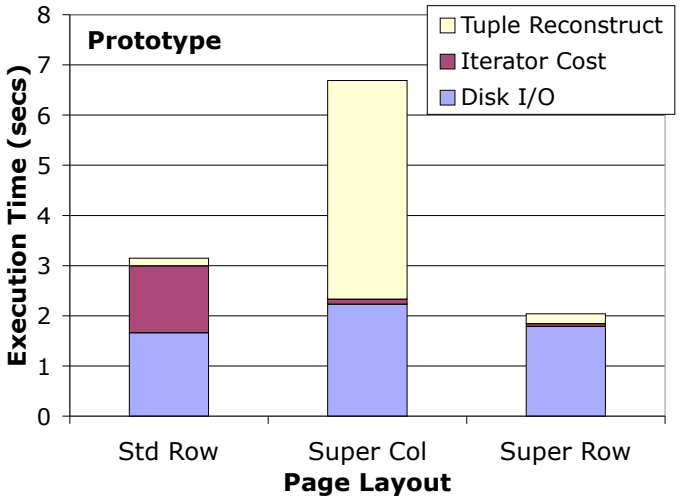


(b) Prototype

Figure 3.11 **Comparison of scanning all columns of an 8M row, 4 column table without abstractions using (a) Cost model and (b) Prototype.**



(a) Cost Model



(b) Prototype

Figure 3.12 **Comparison of scanning all columns of an 1M row, 32 column table without abstractions using (a) Cost model and (b) Prototype.**

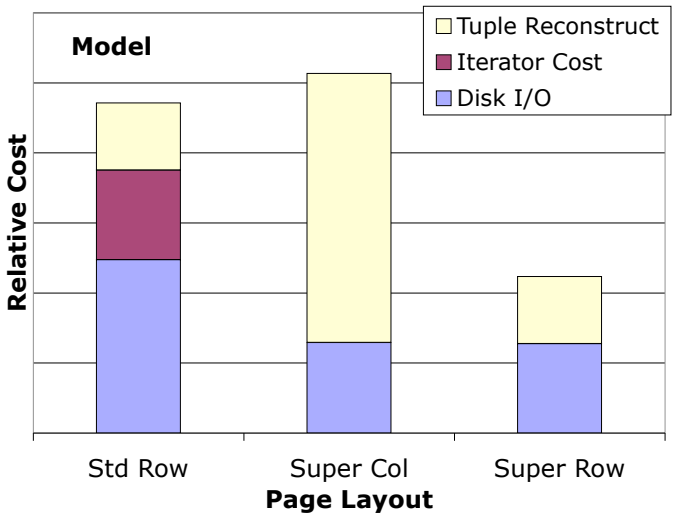


Figure 3.13 **Forecasted relative performance of scanning all columns of an 8M row, 4 column table without abstractions with $IC = 8$.**

3.4.3.2 Sensitivity to Tuple Width

Our experiments vary tuple width from 4 to 32 columns. Using the model, we can forecast relative performance for the three storage formats for wider tuples. Figure 3.14 shows the cost model forecast for scanning 25% of the columns in 8 million tuples for tuple widths of 64, 128, 256, and 512 columns. Our model predicts that the overhead of tuple reconstruction for the column store increases until it is less expensive to scan using the standard row store with no improvements somewhere between 256 and 512 columns. As the tuple width increases, the number of tuples per page decreases and asymptotically approaches 1.

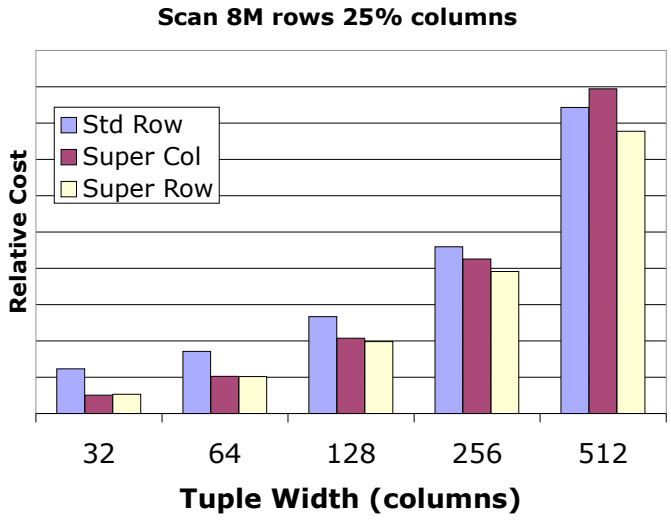


Figure 3.14 **Forecasted relative performance of scanning 25% of the columns of an 8M row table without abstractions as tuple width varies from 64 to 512 columns.**

3.5 Related Work

Optimizing storage of one-to-many joins to avoid redundancy has been explored in the context of Non-First Normal Form databases. NFNF architectures allow nesting relations by permitting relation attributes to be defined as a set of tuples conforming to an arbitrary schema. In [33], Scholl et al. proposed a method for providing a logical relational view of data to the user while transparently storing a hierarchical clustering of related tuples as nested relations using a subset of the NFNF model for query optimization. Their proposal achieves a result similar to column abstraction and super tuples. However, their proposal is for base-table storage and not optimizing storage of materialized views. Further, their evaluation does not provide a direct comparison to an optimized column store system.

In [9], Ailamaki et al. evaluate CPU and cache-related overheads of various data page layouts, including row- and column-oriented choices. Their main contribution is a third choice called PAX, which combines the two by storing each column of a relation on a "minipage" within each physical disk page. PAX is effectively a column store within a row store. We evaluate the effects of choosing PAX to aid in predicate evaluation over super tuples in Chapter 4 of this thesis.

Fractured mirrors [31] store two copies of relations—one row-oriented and one column-oriented—to provide better query performance than either storage choice can provide independently. The mirroring also provides protection against data loss in the event of disk failure. The evaluation of the fractured mirrors work does not consider the column abstraction or super tuple optimizations of either the row or column stores.

The Bubba system [17] used a novel combination of inverted files and a “remainder” relation comprised of non-inverted attributes to store a relation. The inverted files are used as a data compression technique for attributes which contain redundant values. The inverted files are similar to a true column-oriented storage system, and capture the benefits of reducing disk I/O to improve sequential scan time. This work provides early motivation for the C-Store system for both column-at-a-time storage and data compression.

3.6 Conclusion

While prior work on column storage has clearly demonstrated the performance improvements it can deliver over row stores, the relative benefits of column stores and row stores have been obscured because there was no comparison in a common implementation framework. Further, several of the optimizations exploited by the C-Store proposal have analogues in row stores, but these row store optimizations were not considered. In this chapter, we have attempted to shed light on the comparison between the two by implementing both in the same code base, and by defining and implementing the “super tuple” and “column abstraction” optimizations in the row store.

We noted several performance trends in our experimental evaluation. First, we verified the tremendous advantages of a column store system over a row store for workloads that access only a fraction of the columns of a table. Second, the “super tuple” optimization for the row store architecture appears to provide a significant performance benefit. Third, column abstraction can be used effectively to reduce storage needs for all storage choices,

although its benefit is limited for a row store when used in isolation without super tuples. Finally, we showed that the contribution of CPU cost to total scan time can be a sizable component for scans of tables in a standard row store given a reasonably balanced hardware configuration with good sequential disk I/O performance, and that the super tuple optimization reduces CPU utilization in this case. We used our cost model to forecast the performance with a lightweight iterator and found that the row store architecture could be improved significantly without any changes to the underlying storage.

Many areas for future research are apparent. The crossovers in scan performance between super tuple-based row and column stores suggests that automatic storage selection for a given query workload would be beneficial for a system optimized for read-mostly query workloads. The cost model we devleoped in this chapter can provide the basis for creating a storage selection "wizard." Note that selecting which views to materialize is an orthogonal issue—once the correct set of views is selected, one must still decide among the physical storage options.

We also note that column abstraction of one-to-many joins combined with super tuple-based row storage seems an ideal solution for efficient reconstruction of shredded XML documents or other complex entities. For normalized schemas which must frequently be re-joined but do not change frequently, choosing a super tuple-based materialized view as the primary storage for several tables in the schema may provide better performance.

Chapter 4

Predicate Evaluation Strategies for an Read-Optimized Row Store

In the previous chapter, we described the changes necessary to create a read-optimized variant of the traditional relational row store. Specifically, the optimized row store uses large disk pages, super-tuples to tightly pack tuples onto those pages and avoid per-row overheads, and column abstraction to avoid duplication of data. We defined materialized views to exactly answer the query workload to take full advantage of sequential I/O. While it is always possible to have one materialized view per query, we may be able to save significant disk space by noticing that several queries differ only by a WHERE predicate. In fact, these queries may have a significant overlap in the rows required to produce the answer. In this chapter, we evaluate several standard predicate evaluation techniques in the context of a read-optimized row store. For highly selective predicates over a single column, we explore using a value index strategy to evaluate equality and range predicates. In this context, we add a lightweight slot array to each super-tuple to reduce CPU overheads and compare with standard super-tuples. For less selective predicates, we compare super-tuple storage with the PAX storage layout.

4.1 Introduction

The main contributions of this chapter are as follows:

- We describe the scan and index-based predicate evaluation strategies for super tuple storage, and storage layout alternatives such as PAX [9] and a lightweight slot array that can accelerate predicate evaluation.
- We extend our software artifact to include support for PAX, super tuples with slot arrays, and value indexes over super tuple columns. Our experiments vary predicate selectivity, tuple width, and column abstraction properties to identify performance trends.
- We propose and validate a formal cost model for sequential scan and index-based predicate evaluation over super tuples, with and without slot arrays, and PAX storage. We compare the model predictions with our experimental results.

We are able to draw several conclusions from the experimental evaluation:

- Sequential scan-based predicate evaluation over super tuple storage is cost-competitive with both PAX and index-based strategies.
- Index-based evaluation strategies are effective only for very highly selective predicates—i.e., less than 1% selective.
- As column abstraction increases, index-based plans become less attractive due to decreasing scan-based evaluation costs.

- PAX storage layouts provide significant cache locality benefits for predicate evaluation, but require expensive tuple reconstruction and hurt performance for full scans.

The rest of this chapter proceeds as follows. In Section 4.2 and 4.3, we describe predicate evaluation for super tuple storage that utilize scan- and index-based strategies, respectively. We develop the cost model in Section 4.4. We show experimental results in Section 4.5, and we conclude in Section 4.6.

4.2 Scans with Selection Predicates

In this section, we describe scan-based predicate evaluation for super tuples, and issues that affect evaluation performance. We also discuss an alternative page layout called PAX, as defined by Ailamaki et al. in [9]. Finally, we consider the combination of PAX and column abstraction.

4.2.1 Scanning Super Tuples

The design of super tuples for row-based storage allows us to maximize the benefits of sequential scan of disk resident pages through the use of a large disk page size, tight packing of values on the page, and column abstraction when possible.

4.2.1.1 Design Issues

Tight value packing can save several bytes per tuple. For example, consider a tuple containing a 1 char field, a 4-byte int, another 1 char field, and a 2-byte short int. If we use traditional word alignment techniques to ensure aligned access to these attributes later,

we would waste a total of 4 bytes for each tuple. The wasted space would require 50% more disk pages for storage of a relation stored in this manner. With tight value packing, no space is wasted at the expense of misaligned memory reads.

The cost of misaligned reads is small compared with the cost of additional disk I/O. However, as our disk needs decrease, it stands to reason that disk pages are more likely to be in a warm memory cache. When the pages are memory resident, the misalignment penalty looks much worse as a percentage of the total scan time.

When super tuples are combined with column abstraction, misalignment can occur even when tuples contain only 4-byte integers. We store a 1-byte "chunk ID" to identify what columns follow. The "chunk ID" is stored inline with the super tuple storage, and thus causes misalignment due to tight packing.

4.2.2 PAX

An alternative storage layout called PAX [9] addresses these alignment concerns. PAX is a "column store within a row store" design. All columns for a set of tuples are stored on a single disk page, but the values within each column are stored together. The name "PAX" stands for Partition Attributes Across.

The PAX design provides two benefits and one detriment. The first benefit is de facto value alignment. Since we are grouping like-type values together, we get automatic alignment as long as the first value in each column is aligned correctly. Note that alignment is only beneficial when the column datatype is a multiple of the word size for the CPU. The second benefit, however, is the real design win for PAX. When evaluating a selective

predicate over a PAX column, many values in that column fit in a single L1 cache line. The processor can access memory in the L1 cache much faster than L2 or main memory. Thus, a highly selective predicate can be evaluated very efficiently, and cache misses due to columns not needed for predicate evaluation can be minimized.

The downside to the PAX design is one of tuple reconstruction cost. Since the tuples have been physically partitioned across the disk page, a new page of tuples satisfying the predicate must be created. Even if we copy out tuples from normal super tuple storage, the PAX tuples are more expensive to rebuild for the same reason that a one column predicate can be evaluated more quickly—cache misses.

4.2.2.1 PAX with column abstraction

PAX is easily extended to utilize column abstraction. Page layout proceeds as described in Section 3.2.2, with all columns being partitioned per the PAX design. The only exception is that we must encode the run-length for each level of column abstraction. We store the run-length and "chunk ID" information in a special meta-column on the page. For example, consider column abstraction for a materialized view containing Customer, Orders, and Lineitem. We may have a Customer that has 5 Orders, and the orders have 2, 4, 5, 7, and 3 Lineitems, respectively. We would store C5O2O4O5O7O3 to encode the run-lengths. Note that Lineitem is not represented in the run-length encoding, since leaf chunks do not repeat. We can infer that the page contains $2 + 4 + 5 + 7 + 3 = 21$ tuples.

Predicate evaluation over an abstracted PAX column should provide additional runtime savings, as many tuples can be eliminated (or qualified) with a single comparison.

4.3 Indexing Strategies

In Section 4.2, we described layout choices for scan-based predicate evaluation. In this section, we describe using a non-clustered value index as a predicate evaluation alternative for super tuple storage.

4.3.1 Super Tuple Layout

A typical value index contains pairs of attribute values and a record ID identifying the originating tuple. For highly selective predicates, using a value index to identify and retrieve satisfying records can be a cost effective alternative to a scan-based predicate evaluation strategy. Many factors determine when the index plan is better, such as clustering, buffer cache "warmth", and relative costs of sequential and random I/O. Previous heuristics suggest that the index plan will be beneficial only below 10% selectivity [34], and in practice the number is less than 4%.

Our read-mostly optimized storage layout reduces the benefits of index-based predicate evaluation even further. Larger disk pages force us to load even more irrelevant data for "rifle-shot" style or unclustered range index lookups. As mentioned in Section 3.2.1, one side effect of the super tuple layout is external addressability of tuples on the page is difficult. At best, we can construct a record ID consisting of a page number and a "scan ordinal" for use in an index.

4.3.2 Super Tuple Slot Array

We eliminated the slot array for our read-mostly storage optimizations as a space savings measure. Our justification is that read-mostly implies scan-mostly, and the slot array is unnecessary when each page is scanned completely most of the time.

In reality, the space of layout choices between write-optimized and read-optimized storage is a continuum. Adding a lightweight slot array to super tuple storage would aid index lookups at the expense of slightly longer scan times and increased disk usage. Our slot array would be three bytes per tuple—two byte integer offset for the start of the record, and one byte to hold the "chunk ID". We move the chunk ID to the slot array to allow more efficient discovery of the chunks necessary to reconstruct a tuple. Remember that our storage is ordered within each page. If we index a column from a Lineitem chunk, we must find the Orders and Customer chunks to complete the tuple during an index lookup. The record ID from the Lineitem attribute value index would tell us how to find the Lineitem chunk. We can simply scan the slot array backward to find the Orders and Customer chunks that precede the indexed Lineitem chunk in scan order. Note that this strategy only works when records are fully contained within a single page.

4.4 Cost Model

In this section, we present a detailed cost model for both scan-based and index-based predicate evaluation. Our cost formulae depend on several variables, which we present in Table 4.1. The units for $SEQIO$, $RDMIO$, FC, and IC are "cost" units, which provide a

basis for comparing scan costs relative to one another. New variables include f to represent the selectivity of the predicate being evaluated, and $L1miss$ to represent the cost of a memory reference that misses in the L1 data cache.

4.4.1 Scan-based evaluation

We first develop cost models for scan-based predicate evaluation. Figure 4.1 shows the model details for standard super tuple storage. The contributing costs are similar to the straight sequential scan costs presented in Figure 3.6. However, we have an additional predicate evaluation cost on line 4.3, and per-tuple overhead is reduced by the selectivity of the predicate on line 4.4.

$$SeqScanPred(SuperTuple) = |P| * SEQIO * F \quad (4.1)$$

$$+ |P| * IC \quad (4.2)$$

$$+ ||R|| * (comp + L1miss) \quad (4.3)$$

$$+ f * ||R|| * FC \quad (4.4)$$

Figure 4.1 **Cost of scan-based predicate evaluation for "super tuple" relational storage with contributions from (4.1) Disk I/O, (4.2) Storage manager calls, (4.3) Predicate evaluation, and (4.4) Local per-tuple overhead**

Figure 4.2 presents the cost model for PAX storage. To make the comparison to super tuple storage more clear, we present this model as a delta of the standard super tuple cost model, as shown on line 4.5. On line 4.6, we show the reduction in cost due to L1 cache locality of the values necessary for predicate evaluation. Note that the constant 0.875 reflects

Var	Description
$SEQIO$	Cost of a single sequential I/O
$RDMIO$	Cost of a single random I/O
$\|R\|$	Size of storage (pages)
$\|P\|$	Size of "super tuple" storage (pages)
$\|Q\|$	Size of slot array storage (pages)
$\|\|R\|\|$	Cardinality of table (tuples)
C	Width of row (columns)
F	Fraction of cold pages
f	Selectivity of predicate
$L1miss$	Cost of L1 miss to L2 cache
S	Number of columns being retrieved
FC	Cost of function call
IC	Cost of storage manager iteration
n	Abstraction levels (1 means all cols in leaf)
$C(n)$	Columns in abstraction level n
$\|\|L(n)\|\|$	Average cardinality of abstraction level n (tuples)
$\|BP\|$	Size of buffer pool (pages)
$PGSZ$	Usable size of disk page (bytes)
CSZ	Column size (bytes)
OH	Tuple overhead (bytes)

Table 4.1 **Cost Model Variables**

our assumption that eight attribute values fit in each L1 cache line. In a more general cost model, this constant would be calculated based on the size of the datatype of the column. Finally, on line 4.7 we have the increase in tuple reconstruction cost due to the partitioning of attributes across the page. In Section 4.4.3.1, we will show graphs that predict the behavior of scan-based predicate evaluation for super tuples and PAX.

$$SeqScanPred(Pax) = SeqScanPred(SuperTuple) \tag{4.5}$$

$$- 0.875 * ||R|| * L1miss \tag{4.6}$$

$$+ f * ||R|| * S * L1miss \tag{4.7}$$

Figure 4.2 **Cost of scan-based predicate evaluation for PAX storage with contributions from (4.5) Base super tuple evaluation cost, (4.6) Reduced evaluation cost, and (4.7) Increased tuple reconstruction cost**

4.4.2 Index-based Cost Model

We now turn our attention to a cost model for index-based predicate evaluation over super tuple storage. In Figure 4.3, we introduce several variables that will be used in the cost models. First is a variable k to represent the number of rows to be retrieved from storage. The second variable is $|Q|$, to represent the number of pages required for the super tuple storage with a slot array. As in Figure 3.8, we use $ABSAV$ to represent the reduction in size afforded by column abstraction. The third calculated variable is $|RP|$, to represent an estimation of the number of disk pages from the relation to be retrieved via random I/O. We use the Cardenas estimate [14] for this purpose. Yao [39] developed a more refined

estimator for page retrieval in this context, but acknowledges that the error in the Cardenas estimate is "practically negligible" for large blocking factors. Super tuples are an attempt to maximize the blocking factor $\frac{||R||}{|P|}$, so we use the slightly simpler Cardenas estimator. Finally, $|RQ|$ is the Cardenas estimator for retrieval of slot array pages.

$$k = \lceil f * ||R|| \rceil \tag{4.8}$$

$$|Q| = \frac{||R|| * (2 + C * CSZ) - ABSAV}{PGSZ} \tag{4.9}$$

$$|RP| = |P| * \left(1 - \left(1 - \frac{1}{|P|}\right)^k\right) \tag{4.10}$$

$$|RQ| = |Q| * \left(1 - \left(1 - \frac{1}{|Q|}\right)^k\right) \tag{4.11}$$

Figure 4.3 **Common factors for index-based predicate evaluation, including (4.8) Number of rows to be retrieved from storage, (4.9) Number of pages for slot array storage, (4.10) Cardenas estimate of super tuple pages to be retrieved, and (4.11) Cardenas estimate of slot array pages to be retrieved**

The cost model for index-based predicate evaluation for standard super tuples is presented in Figure 4.4. On line 4.13, we detail the penalty associated with not having a slot array on the page. The value $\frac{||R||}{2*|P|} * (comp + L1miss)$ represents the average case cost of scanning the page to find the tuple referenced in the index leaf entry record ID. Even if $k = 1$, the average scan cost is a reasonable guess for any page selected from an unclustered index.

The cost model for index-based predicate evaluation for super tuple pages with slot arrays is presented in Figure 4.5. Although the slot array allows us to avoid the scan penalty,

$$IndexPred(SuperTuple) = F * |RP| * RDMIO \tag{4.12}$$

$$+ k * \left(IC + \frac{||R||}{2 * |P|} * (comp + L1miss) \right) \tag{4.13}$$

$$+ k * FC \tag{4.14}$$

Figure 4.4 **Cost model for index-based predicate evaluation for standard super tuple storage, with contribututions from (4.12) Disk I/O, (4.13) Storage manager calls and page scans to find referenced tuples, and (4.14) Local per-tuple overhead**

the predicate may require more random I/Os due to the increase in pages required to store the relation with the slot array.

4.4.3 Cost Model Graphs

In this section, we present and analyze several graphs based on the cost model for predicate evaluation over super tuples. Our constant values representing measurements from our prototype are presented in Table 4.2. We present graphs for scan-based evaluations in Section 4.4.3.1, followed by index-based evaluations in Section 4.4.3.2. All graphs have arbitrary "cost units" on the Y axis. The cost model does not output time-based values. We label the Y axis with Relative Cost to underscore that the costs can be compared to each other for the same set of input variables, but not across graphs or directly to measured execution times.

$$IndexPred(SuperTupleWithSlots) = F * |RQ| * RDMIO \quad (4.15)$$

$$+ k * IC \quad (4.16)$$

$$+ k * FC \quad (4.17)$$

Figure 4.5 **Cost model for index-based predicate evaluation for super tuple storage with a slot array, with contribututions from (4.15) Disk I/O, (4.16) Storage manager calls, and (4.17) Local per-tuple overhead**

Var	Value
$SEQIO$	15000
$RDMIO$	450000
FC	6
IC	80
$\|BP\|$	16384 pages
$PGSZ$	32000 bytes
CSZ	4 bytes
OH	16 bytes
$comp$	0.1
$L1miss$	0.5

Table 4.2 **Prototype constant values for cost model variables**

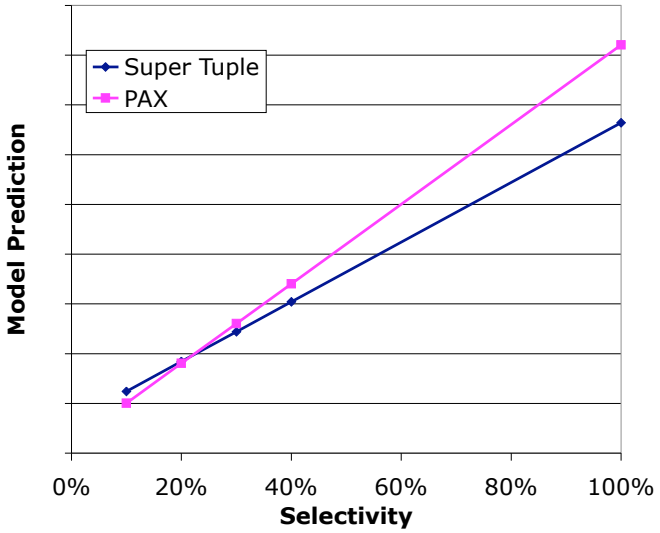
4.4.3.1 Scan-based Estimates

In Figure 4.6, we present the cost model estimate for scan-based predicate evaluation of 4-column and 32-column tuples for a variety of selectivity values. The estimate is based on a completely warm buffer pool to highlight the effects of the PAX layout. We notice that PAX is estimated to be the best layout for both cases when the percentage of tuples selected by the predicate is low. As the percentage increases, standard super tuples become the best choice. The main difference between the two graphs is where the crossover point occurs. We estimate the crossover at about 20% selectivity for 4-column tuples, while it occurs at less than 5% selectivity for 32-column tuples. It seems that PAX incurs more of a penalty for wider tuples.

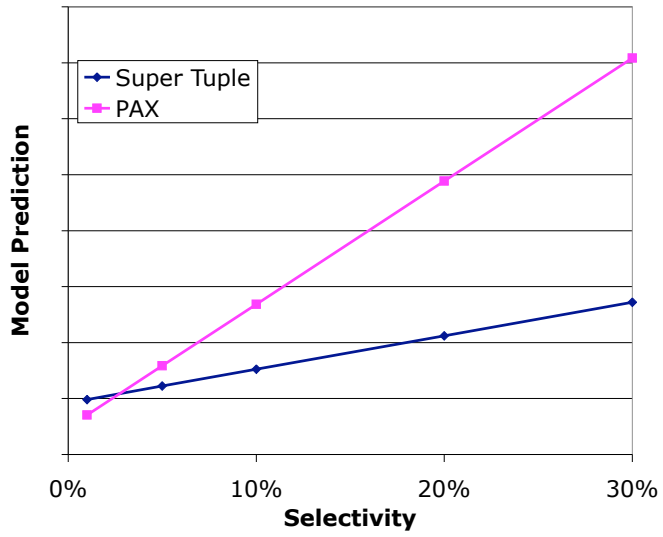
4.4.3.2 Index-based Estimates

In this section, we present estimates for index-based evaluation over super tuple storage, with and without a slot array on the page. Figure 4.7 shows the cost model predictions for a cold buffer pool, while Figure 4.8 presents the same pair of graphs for a warm buffer pool. We show results for both 4-column and 32-column tuples again for varying selectivity.

When we have a cold buffer pool, as in Figure 4.7, we see that disk I/O costs are the dominant factor. In fact, for the 32-column case, our estimate is that the costs are the same with or without the slot array. However, we estimate that the layout without a slot array will perform better for 4-column tuples. This may seem surprising at first blush, considering the increased scan time to find the actual tuple referenced without a slot array. However,



(a) 4-Column Tuples



(b) 32-Column Tuples

Figure 4.6 **Cost model prediction for varying selectivity of Super Tuple and PAX storage using using (a) 4-Column and (b) 32-Column tuples.**

for one million 4-column tuples, we have $|RP| = 532$ and $|RQ| = 594$, which gives the storage without a slot array a 10% advantage in disk I/O.

The warm buffers case, as shown in Figure 4.8, is another matter entirely. In both cases, storing the slot array is a clear win for both 4-column and 32-column tuples up to about 1% selectivity. After that, it is better to scan the relation and evaluate the predicate on the fly.

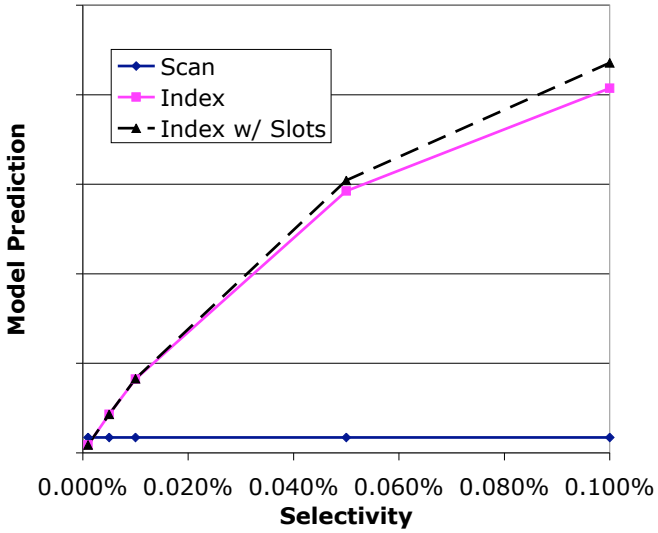
4.5 Experiments

In this section, we discuss our experimental evaluation of scan-based and index-based predicate evaluation strategies for super tuples. We also compare our cost model predictions from Section 4.4 with the measured results.

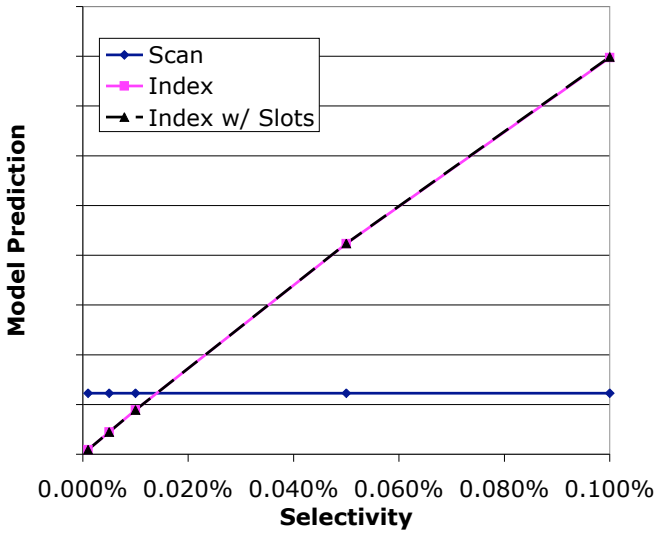
4.5.1 Experiments Description

The experimental prototype used in this section is esentially identical to the system described in Section 3.3.1. Rather than repeat the description, we simply describe the changes necessary to perform our predicate evaluation experiments.

We utilize a synthetic dataset for all experiments, using a modified version of Gray's data generator as presented in [23]. We chose his generator to utilize the random distribution of unique keys. Subsequently building an index on the random unique key column allows us to use an index range scan of a subset of the keys to achieve random I/O behavior for the index-based plans. All experiments use a relation of one million tuples.

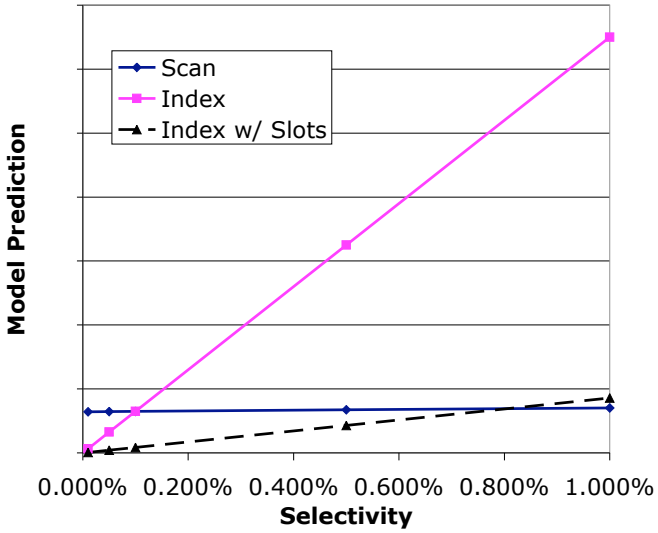


(a) 4-Column Tuples

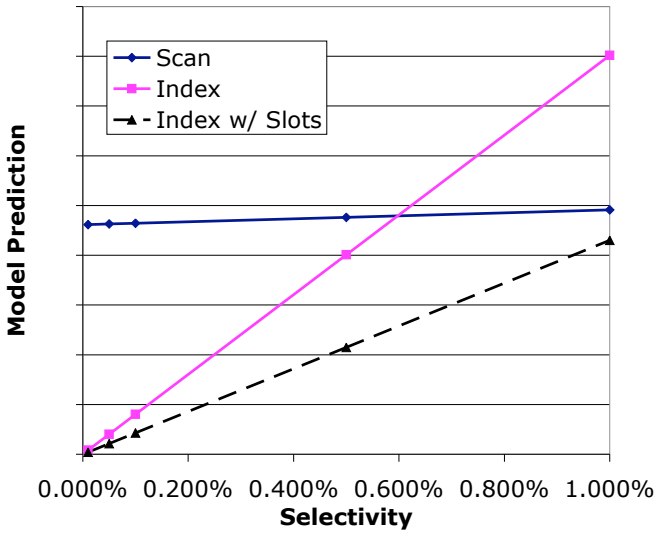


(b) 32-Column Tuples

Figure 4.7 **Cost model prediction for varying selectivity of index-based evaluation for super tuple storage with a cold buffer pool using using (a) 4-Column and (b) 32-Column tuples. Scan of super tuple storage provided as baseline.**



(a) 4-Column Tuples



(b) 32-Column Tuples

Figure 4.8 **Cost model prediction for varying selectivity of index-based evaluation for super tuple storage with a warm buffer pool using using (a) 4-Column and (b) 32-Column tuples. Scan of super tuple storage provided as baseline.**

We implemented PAX as described by Ailamaki et al. in [9]. In our prototype, however, we use a lightweight tuple iterator for PAX pages instead of a Shore-based PAX iterator. Avoiding per-tuple calls into Shore eliminates overheads that would cloud the comparison.

We added the lightweight slot array as an option for super tuple storage to the prototype. When the slot array is present, the “chunk ID” byte is also stored in the slot array instead of inline like normal super tuples. Indexes are provided by Shore, and are a standard B+-tree variant with key/RID pairs at the leaf level.

4.5.2 Scan-based predicates

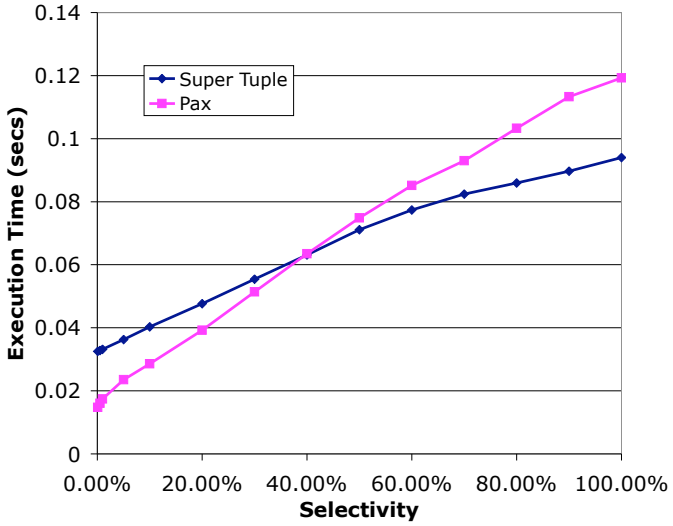
In Figure 4.9, we show how scan-based predicate evaluation execution time varies with selectivity for both normal super tuples and the alternate PAX layout. We show the results for a warm buffer pool only. When the buffer pool is cold, the cost of disk I/O dominates both plans and the difference between the two layouts is negligible. PAX was designed to take advantage of memory cache locality [9], and should be evaluated when data pages are already in memory. Figure 4.9(a) shows the results for 4-column tuples. We note that the PAX layout is the dominant plan for selectivities below approximately 40%, after which the super tuple layout is the better choice. PAX requires tuple reconstruction for tuples that satisfy the predicate, which explains why execution time increases faster for PAX than it does for super tuples. Contrasting with the 32-column relation results shown in Figure 4.9(b), we see that the crossover point occurs much earlier, before 10% selectivity. As tuple width increases, PAX incurs a larger cost for tuple reconstruction. If a subset of

columns are being projected in combination with a selective predicate, the PAX benefits will increase.

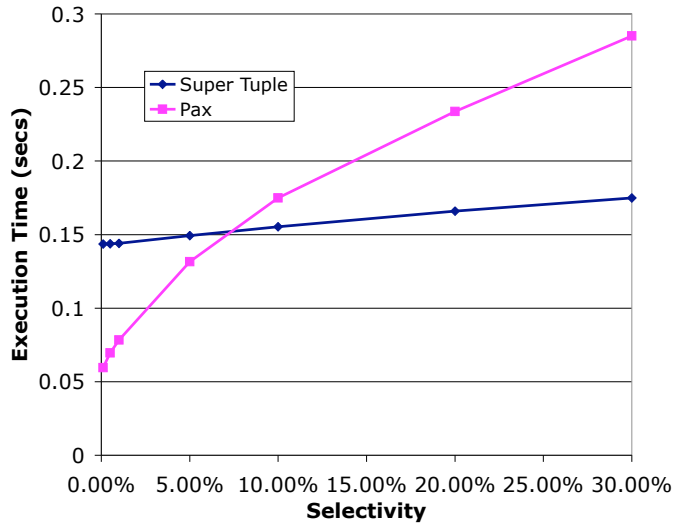
We note that the general trends shown in Figure 4.9 are correctly captured by our cost model, as shown in Figure 4.6. Our model is slightly agressive with where the crossover points occur, due to a slight overestimation in the cost of tuple reconstruction. As selectivity increases, the PAX minipages required to reconstruct each tuple are more likely to be found in the L2 cache. Our simple cost model does not reflect this probability.

For highly selective predicates over a relation in main memory, PAX provides a substantial benefit for overall execution time without requiring additional on-disk storage. Due to increased tuple reconstruction costs, however, standard super tuples are a reasonable alternative. Further, super tuples represent a more "stable" scan time and may be preferable when selectivity of predicates in the query workload is not known up front.

We also evaluated the effects of using column abstraction for both super tuples and PAX. The results are presented in Figures 4.10 for abstraction typical of a materialized view involving Customer, Orders, and Lineitem relations from TPC-H. The storage contains $25,000$ unique Customers, 10 Orders per Customer, and 4 Lineitems per Order. The results for a predicate on the Customer column are somewhat surprising, as PAX is slower than super tuples for all selectivities. Column abstraction is designed to reduce disk storage costs, and therefore primarily benefits cold buffers. It does require additional CPU costs for keeping track of the current tuple chunks within the page. Disqualifying a tuple at the Customer level of abstraction does not remove the need to advance the currency pointers



(a) 4-Column Tuples



(b) 32-Column Tuples

Figure 4.9 **Execution times for varying selectivity of Super Tuple and PAX storage using using (a) 4-Column and (b) 32-Column tuples.**

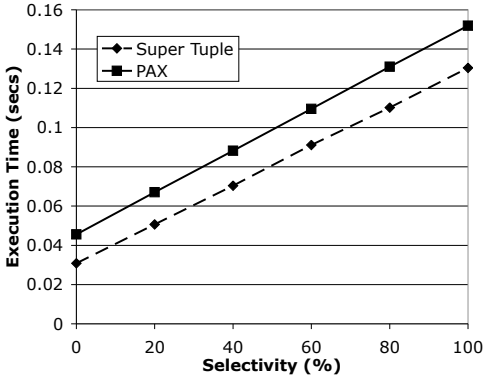
into each minipage for PAX, and doing so requires reading the run-length and chunk ID information. We have 10 Orders per Customer, and each Order has Lineitems that must be skipped. Column abstraction adds runtime overhead for PAX, and when the amount of abstraction is small the primary benefit of PAX is removed.

A similar set of graphs in Figure 4.11 show results for a materialized view of columns from Region, Nation, and Customer. More abstraction is possible for this dataset since it contains 10 unique Regions, 4 Nations per Region, and $25,000$ Customers per Nation. In this case, column abstraction provides a significant benefit to PAX by reducing the number of comparisons required to disqualify tuples, requiring minimal overhead to keep track of minipage positions, and reducing tuple reconstruction costs due to reusing columns copied from the Region and Nation abstraction levels. Figure 4.11(a) shows that PAX is better than super tuples until around 80% selectivity for a predicate on the Region column. Crossover points occur earlier for predicate on Nation and Customer columns.

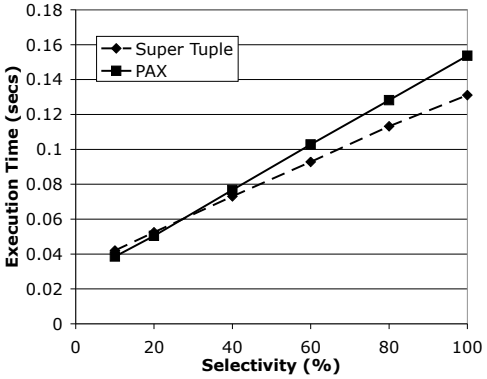
4.5.3 Index-based Evaluation

We next present experimental results for index-based predicate evaluation over super tuples. We vary selectivity for both 4-column and 32-column tuple widths. Results for cold buffers are found in Figure 4.12, while Figure 4.13 details the results for warm buffers.

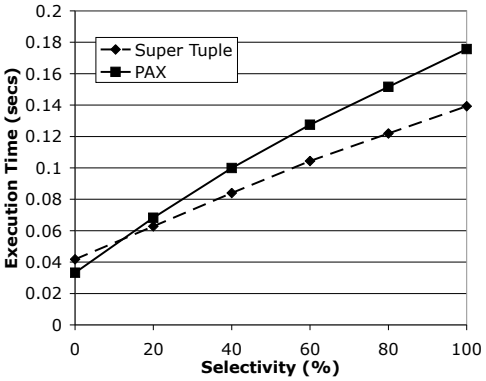
First, we will discuss the cold buffers case in Figure 4.12. As with our model predictions from Figure 4.7, we see the standard super tuples (without a slot array) give superior performance for 4-column tuples. Having a slot array does provide a benefit for the 32-column tuple width, however. As with the model prediction, we see that the scan-based



(a) Customer column

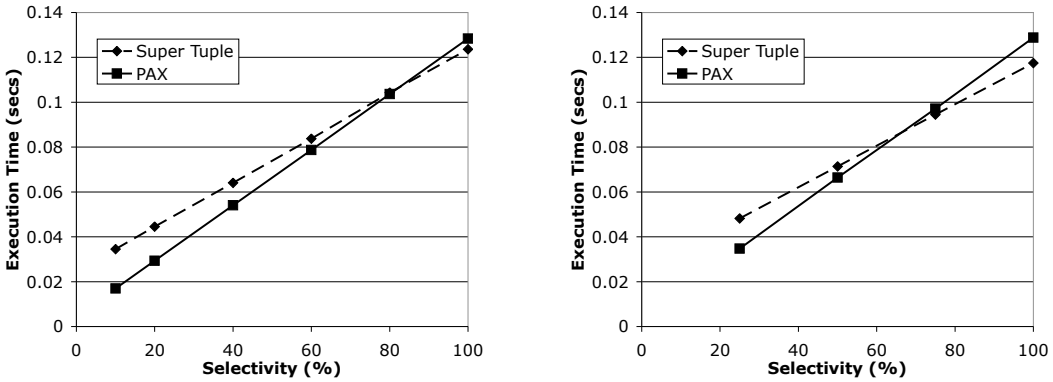


(b) Orders column



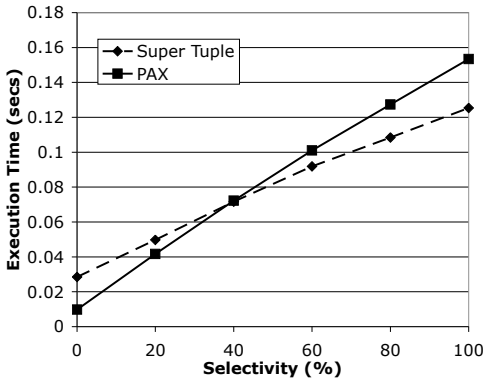
(c) Lineitem column

Figure 4.10 **Average execution times for storage of Customer, Orders, and Lineitem columns in a materialized view using column abstration for PAX and Super Tuple layouts with a predicate on a column from (a) Customer, (b) Orders, and (c) Lineitem.**



(a) Region column

(b) Nation column



(c) Customer column

Figure 4.11 **Average execution times for storage of Region, Nation, and Customer columns in a materialized view using column abstration for PAX and Super Tuple layouts with a predicate on a column from (a) Region, (b) Nation, and (c) Customer.**

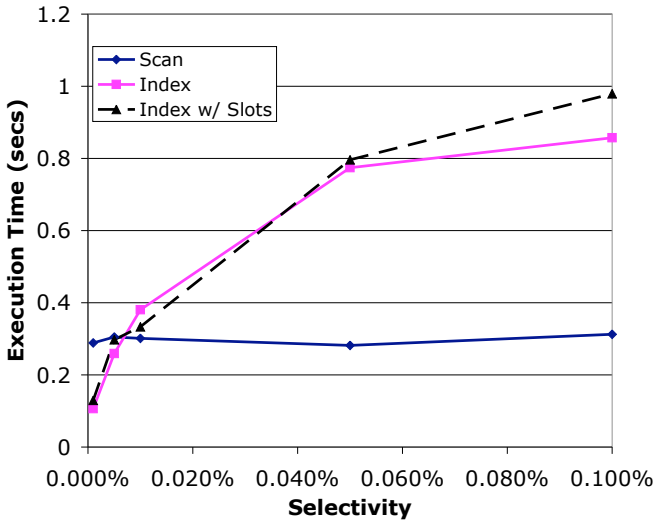
approach is cost-competitive even for very highly selective predicates. Maintaining an index would be beneficial for "rifle-shot" queries such as looking up one Orders record.

Turning to the warm buffers case shown in Figure 4.13, we note that having a slot array seems to always provide benefit. This result is in agreement with our model predictions from Figure 4.8. The crossover point for the standard super tuple index plan seems to agree with the model, but it would appear that our model underestimates the cost of the index plan with a slot array.

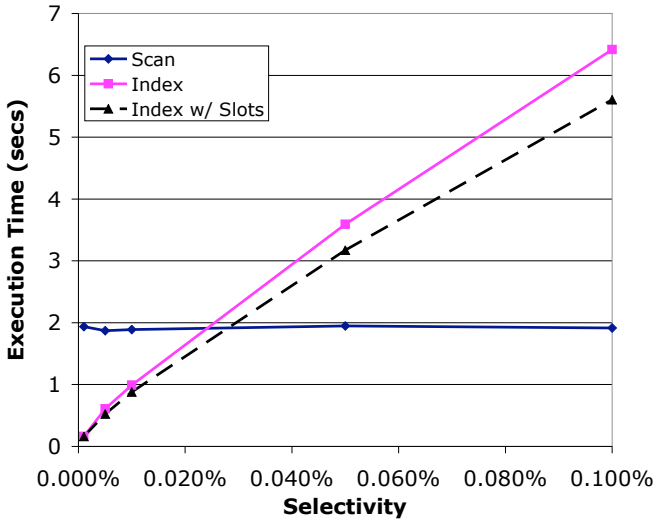
Although index-based plans are rarely beneficial for super tuples above 1% selectivity, they still provide a tremendous benefit for individual tuple lookups. In Figure 4.14, we show average execution times for individual tuple lookups for both cold and warm buffer pools. Each bar on the graph is an average of 30 lookups randomly distributed throughout the relation. We omit the scan-based alternative from these graphs due to the large difference in run times—please refer to Figures 4.12 and 4.13 to see the scan times. In the warm bufferpool case, the large error bars for the Index cases show the downside to not having a slot array. We also see that lookups with a larger tuple width are faster due to decreased scan costs once the correct page from the relation is retrieved. In both cases, having a slot array on the page provides superior performance.

4.6 Conclusion

In this chapter, we have evaluated two strategies for predicate evaluation of super tuples. We first presented a scan-based approach, and presented the PAX storage layout as

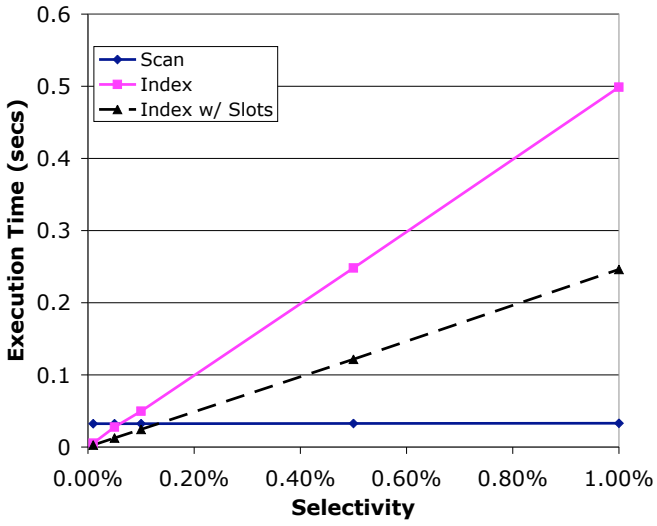


(a) 4-Column Tuples

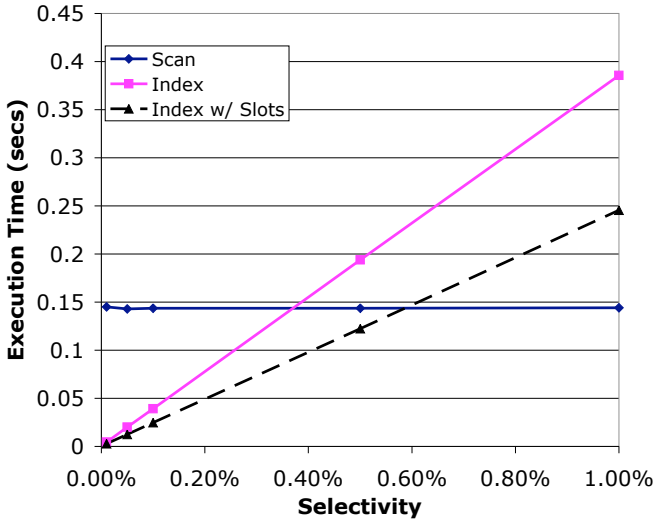


(b) 32-Column Tuples

Figure 4.12 **Execution times for varying selectivity of index-based evaluation for super tuple storage with a cold buffer pool using using (a) 4-Column and (b) 32-Column tuples. Scan of super tuple storage provided as baseline.**

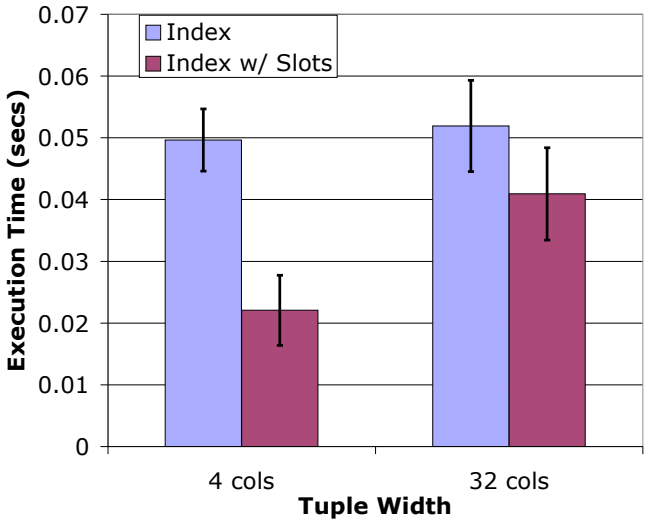


(a) 4-Column Tuples

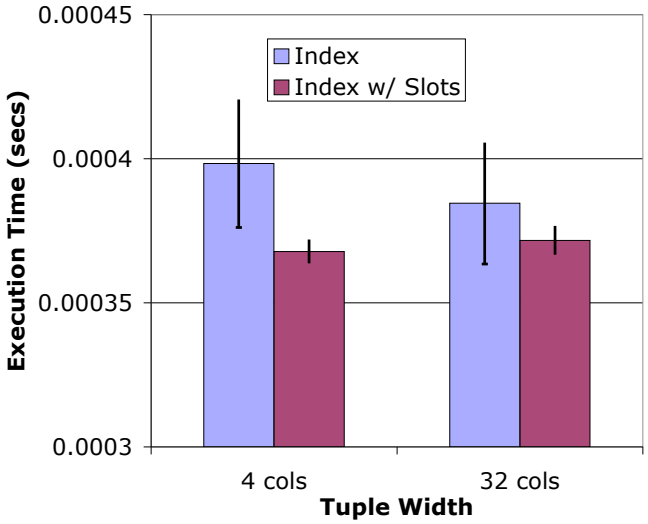


(b) 32-Column Tuples

Figure 4.13 **Execution times for varying selectivity of index-based evaluation for super tuple storage with a warm buffer pool using using (a) 4-Column and (b) 32-Column tuples. Scan of super tuple storage provided as baseline.**



(a) Cold



(b) Warm

Figure 4.14 **Average execution times for individual index tuple lookups for (a) Cold and (b) Warm buffer pools.**

an alternative. We also detailed an index-based approach, and discussed the tradeoffs associated with introducing a lightweight slot array to accelerate index-based lookups into super tuples. We developed and analyzed formal cost models for each evaluation approach, and performed an experimental evaluation. We discovered that both PAX and the index-based approaches can provide a runtime benefit for highly selective predicates. However, standard super tuples provide reasonable and stable performance when used as a scan-only storage format for predicate evaluation.

Chapter 5

Conclusion

In this thesis, I have presented storage and query processing optimizations for hierarchically-organized data. In this chapter, we discuss contributions made by this thesis, potential applications of this work, and share some final words.

5.1 Contributions

The primary contributions of this thesis involve the motivation, implementation, and evaluation of alternative system designs for storage and query processing of hierarchical data. For the read-optimized relational store, we also provide a detailed and accurate cost model to further aid in evaluation and cost forecasting.

The ROX prototype showed that relational query processing of XML data can be a reasonable alternative to standard relational storage, especially when the data is stored using a schema that matches the primary query workload access patterns.

The read-optimized relational store evaluation showed that significant storage and query processing optimizations are possible in exchange for additional insert and update costs. The row-oriented architecture is cost competitive with the column oriented design for most queries, only bested when a small fraction of the columns are projected. We also discovered

that scan-based predicate evaluation is cost competitive with index-based evaluation for range predicates in a read-optimized relational store. For individual tuple lookups, having a value index available provides the best performance.

5.2 Potential Applications

The storage optimizations proposed in my most recent work clearly provide benefits for read-mostly query workloads. A natural next step is to design and build a storage selection "wizard". Given an input query workload and a logical relational schema, two approaches are possible. First, hold the choice of storage (row- or column-oriented) and optimizations constant and choose a set of materialized views which cover the query workload while providing the best performance. Second, hold the set of materialized views constant and choose the best combination of storage and optimizations. Note that the best storage choice may be a combination of row- andcolumn-oriented structures. Adding a small update workload into the equation would likely change the decisions made by the wizard.

Hierarchically-organized data is pervasive in data and information flow. I would like to investigate techniques for harnessing the flow of news and information on the Web and providing both a personal and scalable lineage and provenance record for near and long term recall purposes. For example, RSS feeds have become a widely used method for aggregating news and personal interest information. Unfortunately, these views are extremely transient—all of the story links may be gone within a day. Some limited historical archiving is possible with current RSS aggregators, but they cannot answer contextual questions.

For example, which sites did I visit as a result of following an RSS link? Which unrelated subjects were interesting to me in the same time frame? At the Internet scale, can we trace the spread of news through RSS by following links back to the source? The provenance of news can provide us with a metric for establishing a "seed-rank" to establish each feeds reputation for either breaking news or simply linking to others. The graph created may also discover cycles which expose sites that cite each other as references to establish credibility. Although developing techniques and systems to answer these questions may be years away, I believe that my techniques for efficiently storing and querying hierarchically-organized data will play an important role.

5.3 Final Words

Mountains of data are being generated every day, and our ability to manage this data continues to be a challenge. My work focusing on the management of hierarchically organized data can help with a subset of the problem. I would like to continue developing techniques for managing data and systems to prove that these techniques are viable.

LIST OF REFERENCES

[1] The apache xml project. http://xml.apache.org/.

[2] Microsoft sql server 2000 sdk documentation. http://www.microsoft.com/.

[3] Oracle. http://www.oracle.com/solutions/.

[4] Peoplesoft. http://www.peoplesoft.com/corp/en/products/ent/index.jsp.

[5] Sap. http://www.sap.com/solutions/erp/.

[6] Siebel. http://siebel.com/products/index.shtm.

[7] Database language sql – part 2: Foundations (sql/foundations), iso final draft international standard, 1998.

[8] Db2 for z/os and os/390 version 7 using the utilities suite. http://www.redbooks.ibm.com/redbooks/pdfs/sg246289.pdf, 2001.

[9] Anastassia Ailamaki et al. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3), 2002.

[10] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. Xml path language (xpath) 2.0, June 2006.

[11] Scott Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery, November 2005.

[12] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon. Legodb: Customizing relational storage for xml documents, 2002.

[13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maier, and François Yergeau. Extensible markup language (xml) 1.0 (third edition). 2004.

[14] Alfonso F. Cardenas. Analysis and performance of inverted data base structures. *Commun. ACM*, 18(5):253–263, 1975.

[15] Michael J. Carey et al. Shoring up persistent applications. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD Conference*. ACM Press, 1994.

[16] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. Xml query use cases. `http://www.w3.org/TR/xquery-use-cases/`, September 2005.

[17] George Copeland et al. Data placement in Bubba. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, June 1988. ACM Press.

[18] Transaction Processing Performance Council. TPC Benchmark H (Decision Support). `http://www.tpc.org/tpch/default.asp`, August 2003.

[19] C. J. Date. *An Introduction to Database Systems, 8th Edition.* Addison-Wesley, 2003.

[20] Andrew Eisenberg and Jim Melton. Sql/xml is making good progress. *SIGMOD Record*, 31(2):101–108, 2002.

[21] L. Ennser, C. Delporte, M. Oba, and K. Sunil. Integrating xml with db2 xml extender and db2 text extender. `http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246130.pdf`, 2001.

[22] M. Fernandez, A. Malhorta, J. Marsh, and M. Nagy. Xquery and xpath 2.0 data model. `http://www.w3.org/TR/xpath-datamodel`, November 2005.

[23] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. 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*, pages 243–252. ACM Press, 1994.

[24] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[25] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: A native xml database. *VLDB J.*, 11(4):274–291, 2002.

[26] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of xml data. In *ICDE*, page 198, 2000.

[27] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, 1984.

[28] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel 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 Eng. Bull.*, 24(2):27–33, 2001.

[29] The Apache XML Project. Xalan an xsl processor. `http://xml.apache.org/xalan-c/index.html`.

[30] The Apache XML Project. Xerces: a validating xml parser. `http://xml.apache.org/xercesc/index.html`.

[31] Ravishankar Ramamurthy et al. A case for fractured mirrors. In *Proceedings of the 28th Internation Conference on Very Large Data Bases, Hong Kong, China; August 20-23, 2002*.

[32] M. Scardinia and S. Banerjee. Xml support in oracle 9i, December 2000.

[33] Marc H. Scholl et al. Supporting flat relations by a nested relational kernel. In *Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*.

[34] 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 Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[35] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John E. 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.

[36] Michael Stonebraker et al. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*.

[37] Michael Stonebraker et al. C-Store System Source Code Version 0.1. `http://db.csail.mit.edu/projects/cstore/`, November 2005.

[38] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[39] S. B. Yao. Approximating block accesses in database organizations. *Commun. ACM*, 20(4):260–261, 1977.

[40] Ning Zhang, Varun Kacholia, and M. Tamer Özsu. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *ICDE*, pages 54–65, 2004.

[41] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending materialized views and indexes with ibm db2 design advisor. In *ICAC*, pages 180–188, 2004.