Query Execution in Column-Oriented Database Systems

by

Daniel J. Abadi
`dna@csail.mit.edu`

M.Phil. Computer Speech, Text, and Internet Technology,
Cambridge University, Cambridge, England (2003)
&
B.S. Computer Science and Neuroscience,
Brandeis University, Waltham, MA, USA (2002)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author . . . .
Department of Electrical Engineering and Computer Science
February 1st 2008

Certified by . . . .
Samuel Madden
Associate Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Query Execution in Column-Oriented Database Systems

by

Daniel J. Abadi

`dna@csail.mit.edu`

Abstract

There are two obvious ways to map a two-dimension relational database table onto a one-dimensional storage interface: store the table row-by-row, or store the table column-by-column. Historically, database system implementations and research have focused on the row-by row data layout, since it performs best on the most common application for database systems: business transactional data processing. However, there are a set of emerging applications for database systems for which the row-by-row layout performs poorly. These applications are more analytical in nature, whose goal is to read through the data to gain new insight and use it to drive decision making and planning.

In this dissertation, we study the problem of poor performance of row-by-row data layout for these emerging applications, and evaluate the column-by-column data layout opportunity as a solution to this problem. There have been a variety of proposals in the literature for how to build a database system on top of column-by-column layout. These proposals have different levels of implementation effort, and have different performance characteristics. If one wanted to build a new database system that utilizes the column-by-column data layout, it is unclear which proposal to follow. This dissertation provides (to the best of our knowledge) the only detailed study of multiple implementation approaches of such systems, categorizing the different approaches into three broad categories, and evaluating the tradeoffs between approaches. We conclude that building a query executer specifically designed for the column-by-column query layout is essential to achieve good performance.

Consequently, we describe the implementation of C-Store, a new database system with a storage layer and query executer built for column-by-column data layout. We introduce three new query execution techniques that significantly improve performance. First, we look at the problem of integrating compression and execution so that the query executer is capable of directly operating on compressed data. This improves performance by improving I/O (less data needs to be read off disk), and CPU (the data need not be decompressed). We describe our solution to the problem of executer extensibility – how can new compression techniques be added to the system without having to rewrite the operator code? Second, we analyze the problem of tuple construction (stitching together attributes from multiple columns into a row-oriented "tuple"). Tuple construction is required when operators need to access multiple attributes from the same tuple; however, if done at the wrong point in a query plan, a significant performance penalty is paid. We introduce an analytical model and some heuristics to use that help decide when in a query plan tuple construction should occur. Third, we introduce a new join technique, the "invisible join" that improves performance of a specific type of join that is common in the applications for which column-by-column data layout is a good idea.

Finally, we benchmark performance of the complete C-Store database system against other column-oriented database system implementation approaches, and against row-oriented databases. We benchmark two applications. The first application is a typical analytical application for which column-by-column data layout is known to outperform row-by-row data layout. The second application is another emerging application, the Semantic Web, for which column-oriented database systems are not currently used. We find that on the first application, the complete C-Store system performed 10 to 18 times faster than alternative column-store implementation approaches, and 6 to 12 times faster than a commercial database system that uses a row-by-row data layout. On the Semantic Web application, we find that C-Store outperforms other state-of-the-art data management techniques by an order of magnitude, and outperforms other common data management techniques by almost two orders of magnitude. Benchmark queries,

which used to take multiple minutes to execute, can now be answered in several seconds.

Thesis Supervisor: Samuel Madden
Title: Associate Professor of Computer Science and Electrical Engineering

To my parents, Harry and Rowena, and brother, Ben

Acknowledgments

I would like to thank all members of the C-Store team – at Brandeis University: Mitch Cherniack, Nga Tran, Adam Batkin, and Tien Hoang; at Brown University: Stan Zdonik, Alexander Rasin, and Tingjian Ge; at UMass Boston: Pat O'Neil, Betty O'Neil, and Xuedong Chen; at University of Wisconsin Madison: David DeWitt; at Vertica: Andy Palmer, Chuck Bear, Omer Trajman, Shilpa Lawande, Carty Castaldi, Nabil Hachem (and many others); and at MIT: Mike Stonebraker, Samuel Madden, Stavros Harizopoulos, Miguel Ferreira, Daniel Myers, Adam Marcus, Edmond Lau, Velen Liang, and Amersin Lin. C-Store has truly been an exciting and inspiring context in which to write a PhD thesis.

I would also like to thank other members of the MIT database group: Arvind Thiagarajan, Yang Zhang, George Huo, Thomer Gil, Alvin Cheung, Yuan Mei, Jakob Eriksson, Lewis Girod, Ryan Newton, David Karger, and Wolfgang Lindner; and members of the MIT Networks and Mobile Systems group: Hari Balakrishnan, John Guttag, Dina Katabi, Michel Goraczko, Dorothy Curtis, Vladimir Bychkovsky, Jennifer Carlisle, Hariharan Rahul, Bret Hull, Kyle Jamieson, Srikanth Kandula, Sachin Katti, Allen Miu, Asfandyar Qureshi, Stanislav Rost, Eugene Shih, Michael Walfish, Nick Feamster, Godfrey Tan, James Cowling, Ben Vandiver, and Dave Andersen with whom I've had many intellectually stimulating conversations over the last few years.

Thanks to Miguel Ferreira, who worked closely with me on the initial C-Store query execution engine prototype and on the compression subsystem (which became Chapter 4 of this dissertation); to Daniel Myers who helped code the different column-oriented materialization strategies (behind Chapter 5 of this thesis); and to Adam Marcus and Kate Hollenbach for their collaboration on the Semantic Web application for column-oriented databases (Chapter 8).

Thanks especially to Mitch Cherniack who introduced me to the field of database systems research, serving as my undergraduate research adviser; to Hari Balakrishnan who convinced me to come to MIT and took me as his student before Sam arrived; to Magdalena Balazinska who took me under her wing from the day I arrived at MIT, helping me to figure out how to survive graduate school, and serving as an amazing template for success; and to Frans Kaashoek for serving on my PhD committee.

Thanks to the National Science Foundation who funded the majority of my research; both directly through a Graduate Research Fellowship and more indirectly through funding the projects I've worked on.

My research style, philosophy, and career have been enormously influenced through close interactions and relationships with three people. First, I am fortunate that David DeWitt spent a sabbatical year at MIT while I was a student there. The joy he brings to his research helped me realize that I wanted to pursue an academic career. I am influenced by his passion and propensity to take risks.

Second, the C-Store project and this thesis would not have happened if it were not for Mike Stonebraker. From Aurora, to Borealis, to C-Store, to H-Store, collaboration on projects with him at the lead has been a true pleasure. I am influenced by his emphasis on attacking real-world practical problems and his ruthless disdain for the complex.

Third, and most importantly, I am indebted to my primary research adviser, Samuel Madden. For someone who must deal with the many stresses inherent in the tenure process at a major research institution, it is impossible to imagine someone being more selfless, having more of his students' interests in mind, or giving them more freedom. I am influenced by his energy, his interpersonal skills, and his dedication to his research. I hope to advise any future students who choose to study with me in my academic career in a very similar way.

Contents

List of Figures

13

List of Tables

Chapter 1

Introduction

The world of relational database systems is a two dimensional world. Data is stored in tabular data structures where rows correspond to distinct real-world entities or relationships, and columns are attributes of those entities. For example, a business might store information about its customers in a database table where each row contains information about a different customer and each column stores a particular customer attribute (name, address, e-mail, etc.).

There is, however, a distinction between the conceptual and physical properties of database tables. This aforementioned two dimensional property exists only at the conceptual level. At a physical level, database tables need to be mapped onto one dimensional structures before being stored. This is because common computer storage media (e.g. magnetic disks or RAM), despite ostensibly being multi-dimensional, provide only a one dimensional interface (read and write from a given linear offset).

1.1 Rows vs. Columns

There are two obvious ways to map database tables onto a one dimensional interface: store the table row-by-row or store the table column-by-column. The row-by-row approach keeps all information about an entity together. In the customer example above, it will store all information about the first customer, and then all information about the second customer, etc. The column-by-column approach keeps all attribute information together: all of the customer names will be stored consecutively, then all of the customer addresses, etc.

Both approaches are reasonable designs and typically a choice is made based on performance expectations. If the expected workload tends to access data on the granularity of an entity (e.g., find a customer, add a customer, delete a customer), then the row-by-row storage is preferable since all of the needed information will be stored together. On the other hand, if the expected workload tends to read per query only a few attributes from many records (e.g., a query that finds the most common e-mail address domain), then column-by-column storage is preferable since irrelevant attributes for a particular query do not have to be accessed (current storage devices cannot be read with fine enough granularity to read only one attribute from a row; this is explained further in Section 3.2).

The vast majority of commercial database systems, including the three most popular database software systems (Oracle, IBM DB2, and Microsoft SQL Server), choose the row-by-row storage layout. The design implemented by these products descended from research developed in the 1970s. The design was optimized for the most common database application at the time: business transactional data processing. The goal of these applications was to automate mission-critical business tasks. For example, a bank might want to use a database to store information about its branches and its customers and its accounts. Typical uses of this database might be to find the balance of a particular customer's account or to transfer $100 from customer A to customer B in one single atomic transaction. These queries commonly access data on the granularity an entity (find a customer, or an account, or branch information; add a new customer, account, or branch). Given this workload, the row-by-row storage layout was chosen for these systems.

Starting in around the 1990s, however, businesses started to use their databases to ask more detailed analytical queries. For example, the bank might want to analyze all of the data to find associations between customer attributes and heightened loan risks. Or they might want to search through the data to find customers who should receive VIP treatment. Thus, on top of using databases to automate their business processes, businesses started to want to use databases to help with some of the decision making and planning. However, these new uses for databases posed two problems. First, these analytical queries tended to be longer running queries, and the shorter transactional write queries would have to block until the analytical queries finished (to avoid different queries reading an inconsistent database state). Second, these analytical queries did not generally process the same data as the transactional queries, since both operational and historical data (from perhaps multiple applications within the enterprise) are relevant for decision making. Thus, businesses tended to create two databases (rather than a single one); the transactional queries would go to the transactional database and the analytical queries would go to what are now called data warehouses. This business practice of creating a separate data warehouse for analytical queries is becoming increasingly common; in fact today data warehouses comprise $3.98 billion [65] of the $14.6 billion database market [53] (27%) and is growing at a rate of 10.3% annually [65].

1.2   Properties of Analytic Applications

The nature of the queries to data warehouses are different from the queries to transactional databases. Queries tend to be:

- **Less Predictable**. In the transactional world, since databases are used to automate business tasks, queries tend to be initiated by a specific set of predefined actions. As a result, the basic structure of the queries used to implement these predefined actions are coded in advance, with variables filled in at run-time. In contrast, queries in the data warehouse tend to be more exploratory in nature. They can be initiated by analysts who create queries in an ad-hoc, iterative fashion.

- **Longer Lasting**. Transactional queries tend to be short, simple queries ("add a customer", "find a balance", "transfer $50 from account A to account B"). In contrast, data warehouse queries, since they are more analytical in nature, tend to have to read more data to yield information about data in aggregate rather than individual records. For example, a query that tries to find correlations between customer attributes and loan risks needs to search though many records of customer and loan history in order to produce meaningful correlations.

- **More Read-Oriented Than Write-Oriented**. Analysis is naturally a read-oriented endeavor. Typically data is written to the data warehouse in batches (for example, data collected during the day can be sent to the data warehouse from the enterprise transactional databases and batch-written over-night), followed by many read-only queries. Occasionally data will be temporarily written for "what-if" analyses, but on the whole, most queries will be read-only.

- **Attribute-Focused Rather Than Entity-Focused**. Data warehouse queries typically do not query individual entities; rather they tend to read multiple entities and summarize or aggregate them (for example, queries like "what is the average customer balance" are more common than "what is the balance of customer A's account"). Further, they tend to focus on only a few attributes at a time (in the previous example, the balance attribute) rather than all attributes.

1.3   Implications on Data Management

As a consequence of these query characteristics, storing data row-by-row is no longer the obvious choice; in fact, especially as a result of the latter two characteristics, the column-by-column storage layout can be better. The third query characteristic favors a column-oriented layout since it alleviates the oft-cited disadvantage of storing data in columns: poor write performance. In particular, individual write queries can perform poorly if data is

laid out column-by-column, since, for example, if a new record is inserted into the database, the new record must be partitioned into its component attributes and each attribute written independently. However, batch-writes do not perform as poorly since attributes from multiple records can be written together in a single action. On the other hand, read queries (especially attribute-focused queries from the fourth characteristic above) tend to favor the column-oriented layout since only those attributes accessed by a query need to be read, and thus this layout tends to be more I/O efficient. Thus, since data warehouses tend to have more read queries than write queries, the read queries are attribute focused, and the write queries can be done in batch, the column-oriented layout is favored.

Surprisingly, the major players in the data warehouse commercial arena (Oracle, DB2, SQL Server, and Teradata) store data row-by-row (in this dissertation, they will be referred to as "row-stores"). Although speculation as to why this is the case is beyond the scope of this dissertation, this is likely due to the fact that these databases have historically focused on the larger transactional database market and wish to maintain a single line of code for all of their database software [64]. Similarly, database research has tended to focus on the row-by-row data layout, again due to the field being historically transactionally focused. Consequently, relatively little research has been performed on the column-by-column storage layout ("column-stores").

1.4 Dissertation Goals, Challenges, and Contributions

The overarching goal of this dissertation is to further the research into column-oriented databases, tying together ideas from previous attempts to build column-stores, proposing new ideas for performance optimizations, and building an understanding of when they should be used. In essence, this dissertation serves as a guide for building a modern column-oriented database system. There are thus three sub-goals. First, we look at past approaches to building a column-store, examining the tradeoffs between approaches. Second, we propose techniques for improving the performance of column-stores. Third, we benchmark column-stores on applications both inside and outside their traditional sweet-spot. In this section, we describe each of these sub-goals in turn.

1.4.1 Exploring Column-Store Design Approaches

Due to the recent increase in the use of database technology for business analysis, planning, and intelligence, there has been some recent work that experimentally and analytically compares the performance of column-stores and row-stores [34, 42, 43, 64, 63, 28]. In general, this work validates the prediction that column-stores should outperform row-stores on data warehouse workloads. However, this body of work does not agree on the magnitude of relative performance. This magnitude ranges from only small differences in performance [42], to less than an order of magnitude difference [34, 43], to an order of a magnitude difference [63, 28], to, in one case, a factor of 120 performance difference [64].

One major reason for this disagreement in performance difference is that there are multiple approaches to building a column-store. One approach is to vertically partition a row-store database. Tables in the row-store are broken up into multiple two column tables consisting of (table key, attribute) pairs [49]. There is one two-column table for each attribute in the original table. When a query is issued, only those thin attribute-tables relevant for a particular query need to be accessed — the other tables can be ignored. These tables are joined on table key to create a projection of the original table containing only those columns necessary to answer a query, and then execution proceeds as normal. The smaller the percentage of columns from a table that need to be accessed to answer a query, the better the relative performance with a row-store will be (i.e., wide tables or narrow queries will have a larger performance difference). Note that for this approach, none of the DBMS code needs to be modified — the approach is a simple modification of the schema.

Another approach is to modify the storage layer of the DBMS to store data in columns rather than rows. At the logical level the schema looks no different; however, at the physical level, instead of storing a table row-by-row, the table is stored column-by-column. The key difference relative to the previous approach is that table keys need not be repeated with each attribute; the ith value in each column matches up with the ith value in all of the other columns (i.e., they belong to the same tuple). Similarly with the previous approach, only those columns that are relevant

for a particular query need to be accessed and merged together. Once this merging has taken place, the normal (row-store) query executor can process the query as normal. This is the approach taken in the studies performed by Harizopoulos et. al. [43] and Halverson et. al. [42]. This approach is particularly appealing for studies comparing row-store and column-store performance since it allows for the examination of the relative advantages of systems in isolation. They only vary whether data is stored by columns or rows on disk; data is converted to a common format for query processing and can be processed by an identical executor.

A third approach is to modify both the storage layer and the query executor of the DBMS [63, 64, 28]. Thus, not only is data stored in columns rather than rows, but the query executor has the option of keeping the data in columns for processing. This approach can lead to a variety of performance enhancements. For example, if a predicate is applied to a column, that column can be sent in isolation to the CPU for predicate application, alleviating the memory-CPU bandwidth bottleneck. Further, iterating through a fixed width column is generally faster than iterating through variable-width rows (and if any attribute in a row is variable-width, then the whole row becomes variable-width). Finally, selection and aggregation operations in a query plan might reduce the number of rows that need to be created (and output as a result of the query), reducing the cost of merging columns together. Consequently, keeping data in columns and waiting to the end of a query plan to create rows can reduce the row construction cost.

Thus, one goal of this dissertation is to explore multiple approaches to building a column-oriented database system, and to understand the performance differences between these approaches (and the reasons behind these differences).

Note that each of the three approaches described above is successively more difficult to implement (from a database system perspective). The first approach requires no modifications to the database and can be implemented in all current database systems. Of course, this approach does require changes at the application level since the logical schema must be modified to implement this approach. If this change is to be hidden from the application, an automatic query rewriter must be implemented that automatically converts queries over the initial schema to queries over the vertically partitioned schema. The second approach requires a modification to the database storage manager, but the query executor can be kept in tact. The third approach requires modifications to both the storage manager and the query executor.

Since the first approach can be implemented on currently available DBMSs without modification, if it performs well, then it would be the preferable solution for implementing column-stores. This would allow enterprises to use the same database management software for their transactional/operational databases and their data warehouses, reducing license, training, and management costs. Since this is the case, in Chapter 2, we explore this approach in detail, proposing multiple ways (in addition to the vertical partitioning technique) to build a column-store on top of a row-store, implementing each of these proposals, and experimenting with each implementation on a data warehousing benchmark. We find that while in some cases this approach outperforms raw row-store performance, in many cases it does not. In fact, in some cases, the approach performs worse than the row-store, even though column-stores are expected to outperform row-stores on data warehouse workloads. We then explore the reasons behind these observations. Finally, we implement the latter two approaches (a column-oriented storage manager under a row-oriented query executor and a column-oriented storage manager under a column-oriented query executor), and show experimentally why these approaches do not suffer from the same problems as the first approach.

To demonstrate the significant performance differences between these approaches, Figure 1-1 presents the performance results of a query from the Star Schema Benchmark [56, 57] (a benchmark designed to measure performance of data warehouse database software) on each implementation. These results are compared with the performance of the same query on a popular commercial row-oriented database system. The vertical partitioning approach performs only 15% faster than the row-store while the modified storage layer approach performs 24% faster. It is not until the third approach is used (where the storage layer **and** the execution engine are modified) that a factor of three performance improvement is observed (note, further performance improvements are possible, and this Figure will be revisited in Chapter 9 after the full column-store architecture is presented in detail).

The results of these experiments motivate our decision to abandon row-stores as a starting point to build a column-store and to start from scratch in building a column-store. In Chapter 3, we present the architecture and bottom-up implementation details of our implementation of a new column-store: C-Store, describing the storage

Sample SSBM Query (2.3)

	Row-Store	Vertical-Partitioning	Baseline Column-Store	Full Column-Store
■ Sample SSBM Query (2.3)	46.0	39.0	34.8	13.3
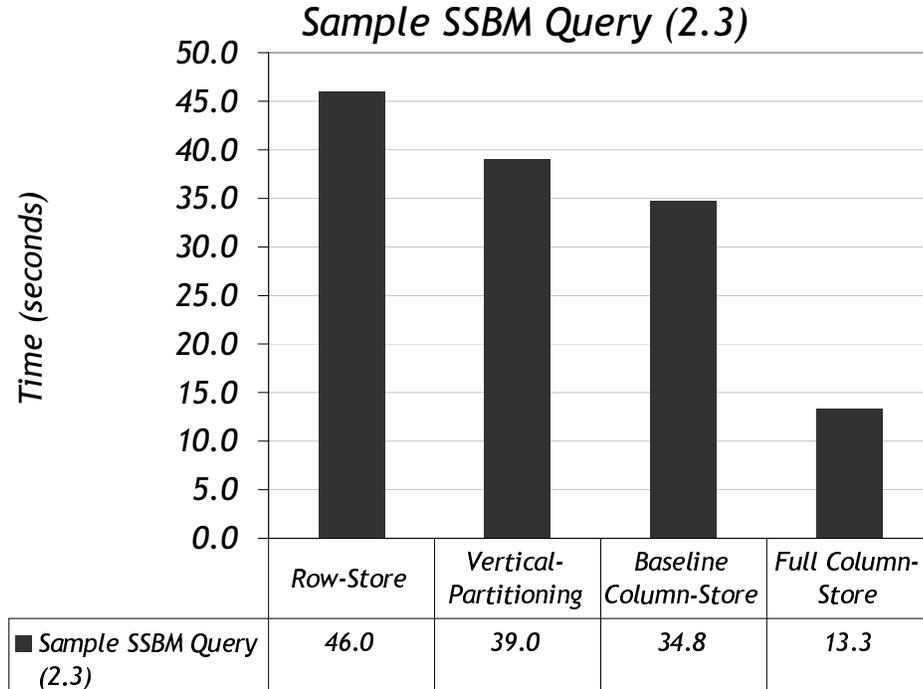
Figure 1-1: Performance varies dramatically across different column-oriented database implementations.

layer and query executor (including the data flow models, execution models, and the operator set). Since detailed design descriptions of database systems are rarely published (especially for commercial databases) this dissertation thus contains one of the few published blueprints for how to build a modern, disk-based column-store.

1.4.2   Improving Column-Store Performance

A second goal of this dissertation is to introduce novel ideas to further improve column-oriented database performance. We do this in three ways. First, we add a compression subsystem to our column-store prototype which improves performance by reducing the I/O that needs to be performed to read in data, and, in some cases, by allowing operators to process multiple column values in a single iteration. Second, we study problem of tuple construction cost (merging of columns into rows). This common operation in column-stores can dominate query time if done at the wrong point in a query plan. We introduce an analytical model and some heuristics to keep tuple construction cost low, which improves query performance. Finally, we introduce a new join technique, the "invisible join", that converts a join into a set of faster predicate applications on individual columns. We now describe each of these three contributions in turn.

Compression

In Chapter 4, we discuss compression as a performance optimization in a column-store. First, let us explain why compression (which is used in row-stores to improve performance as well) works differently in column-stores than it does for row-stores. Then we will describe a problem we must solve and summarize results.

Intuitively, data stored in columns is more compressible than data stored in rows. Compression algorithms perform better on data with low *information entropy* (high data value locality). Imagine a database table containing

information about customers (name, phone number, e-mail address, snail-mail address, etc.). Storing data in columns allows all of the names to be stored together, all of the phone numbers together, etc. Certainly phone numbers will be more similar to each other than surrounding text fields like e-mail addresses or names. Further, if the data is sorted by one of the columns, that column will be super-compressible (for example, runs of the same value can be run-length encoded).

But of course, the bottom line goal is performance, not compression ratio. Disk space is cheap, and is getting cheaper rapidly (of course, reducing the number of needed disks will reduce power consumption, a cost-factor that is becoming increasingly important). However, compression improves performance (in addition to reducing disk space) since if data is compressed, then less time must be spent in I/O as data is read from disk into memory (or from memory to CPU). Given that performance is what we are trying to optimize, this means that some of the "heavier-weight" compression schemes that optimize for compression ratio (such as Lempel-Ziv, Huffman, or arithmetic encoding), are less suitable than "lighter-weight" schemes that sacrifice compression ratio for decompression performance.

In Chapter 4, we evaluate the performance of a set of compression algorithms for use with a column-store. Some of these algorithms are sufficiently generic that they can be used in both row-stores and column-stores; however some are specific to column-stores since they allow compression symbols to span across values within the same column (this would be problematic in a row-store since these values are interspersed with the other attributes from the same tuple). We show that in many cases, these column-oriented compression algorithms (in addition to some of the row-oriented algorithms) can be operated on directly without decompression. This yields the ultimate performance boost, since the system saves I/O by reading in less data but does not have to pay the decompression cost.

However, operating directly on compressed data requires modifications to the query execution engine. Query operators must be aware of how data is compressed and adjust the way they process data accordingly. This can lead to highly nonextensible code (a typical operator might consist of a set of 'if statements' for each possible compression type). We propose a solution to this problem that abstracts the general properties of compression algorithms that facilitates their direct operation so that operators only have to be concerned with these properties. This allows new compression algorithms to be added to the system without adjustments to the query execution engine code.

Results from experiments show that compression not only saves space, but significantly improves performance. However, without operation on compressed data, it is rare to get more than a factor of 3 performance improvement, even if the compression ratio is more than a factor of 10. Once the query execution engine is extended with extensible compression-aware techniques, it is possible to obtain more than an order of magnitude performance improvement, especially on columns that are sorted or have some order to them.

Tuple Construction

Chapter 5 examines the problem of tuple construction (also called tuple materialization) in column-oriented databases. The challenge is as follows: In a column-store, information about a logical entity (e.g., a person) is stored in multiple locations on disk (e.g. name, e-mail address, phone number, etc. are all stored in separate columns). However, most queries access more than one attribute from a particular entity. Further, most database output standards (e.g., ODBC and JDBC) access database results entity-at-a-time (not column-at-a-time). Thus, at some point in most query plans, data from multiple columns must be combined together into 'rows' of information about an entity. Consequently, tuple construction is an extremely common operation in a column store and must perform well.

The process of tuple construction thus presents two challenges. **How** should it be done and **when** (at what point in a query plan) should it be done. The naive solution for **how** it should be done is the following: For each entity, i, that must be constructed, seek to the ith position in the first column and extract the corresponding value, seek to the ith position in the second column and extract the corresponding value, etc., for all columns that are relevant for a particular query. Clearly, however, this would lead to enormous query times as each seek costs around 10ms. Instead, extensive prefetching should be used, where many values from the first column are read into memory and held there while other columns are read into memory. When all relevant columns have been read in, tuples

are constructed in memory. In a paper by Harizopoulos et. al. [43], we show that the larger the prefetch buffer per column, the faster tuple materialization can occur (buffer sizes on the order of 18MB are ideal on a modern desktop-class machine). Since the "prefetching" answer is fairly straight-forward, this dissertation, in particular, Chapter 5 focuses on the second challenge: **when** should tuple construction occur. The question is: should tuples be constructed at the beginning of a query plan as soon as it is determined which columns are relevant for a query, or should tuple construction be delayed, and query operators operate on individual columns as long as possible?

Results from experiments show that for queries without joins, waiting as long as possible to construct tuples can improve performance by an order of magnitude. However, joins significantly complicate the materialization decision, and in many cases tuples should be materialized before a join operator.

Invisible Join

The final performance enhancement component of this dissertation is presented in Chapter 6. We introduce a new join operator, the "invisible join", designed to join tables that are organized in a star schema — the prevalent (accepted as best practice) schema design for data warehouses.

An example star schema is presented in Figure 1-2. The basic design principle is to center data around a fact table that contains many foreign keys to dimension tables that contain additional information. For example, in Figure 1-2, the central fact table contains an entry for each instance of a customer purchase (every time something gets bought, an entry is added to the fact table). This entry contains a foreign key to a customer dimension table which contains information (name, address, etc.) about that customer, a foreign key to a supplier table containing information about the supplier of the item that was purchased, etc. In addition, the fact table contains additional attributes about the purchase itself (like the quantity, price, and tax of the item purchased). In general, the fact table can grow to be quite large; however, the dimension tables tend to scale at a much slower rate (for example, the number of customers, stores, or products of a business scales at a much slower rate than the number of business transactions that occur).

The typical data warehouse query will perform some kind of aggregation on the fact table, grouping by attributes from different dimensions. For example, a business planner might be interested in how the cyclical nature of the holiday sales season varies by country. Thus, the planner might want to know the total revenue from sales, grouped by country and by the week number in the year. Further, assume that the planner is only interested in looking at numbers from the final three months of the year (October-December) and from countries in North America and Europe. In this case, there are two dimensions relevant to the query — the customer dimension and the date dimension. For the customer dimension, there is a selection predicate on region (region='North America' or region='Europe') and an aggregation group-by clause on country. For the date dimension, there is a selection predicate on month (between 'October' and 'December') and a group-by clause on week.

The straightforward algorithm for executing such a query would be to extract (and construct) the relevant tuples from the fact table and the two dimension tables before performing the join. For the example above, the customer key, date key, and revenue columns would be extracted from the fact table; the customer key, region, and country would be extracted from the customer dimension table; and the date key, month, and week would be extracted from the date dimension table. Once the relevant columns have been extracted, tuples are constructed, and a normal row-oriented join is performed (using any of the normal algorithms — nested loops, hash, sort-merge, etc.).

We introduce an improvement on this straightforward algorithm that employs an "invisible join". Here, the joins are rewritten into predicates on the foreign key columns in the fact table. These predicates can either be a hash lookup (in which case a hash join is simulated) or more advanced techniques of predicting whether a tuple is expected to join can be used. By rewriting the joins as selection predicates on fact table columns, they can be executed at the same time as other selection predicates that are being applied to the fact table, and any of the predicate application algorithms described in Chapter 5 can be used. For example, each predicate can be applied in parallel and the results merged together using fast bit-map operations. Alternatively, the results of a predicate application can be pipelined into another predicate application to reduce the number of times the second predicate must be applied. Once all predicates have been applied, the appropriate tuples can be extracted from the relevant dimensions (this can also be done in parallel).

23

Figure 1-2: Schema of the SSBM Benchmark

In Chapter 6, we will show that this alternate algorithm (the "invisible join") performs significantly faster than the straightforward algorithm (between a factor of 1.75 and a factor of 10 depending on how the straightforward algorithm is implemented). In fact, we will show that joining fact tables and dimension tables using the invisible join can in some circumstances outperform an identical query on a widened fact table where the join has been performed in advance!

1.4.3 Benchmarking Column-Store Performance

The experiments and models presented in the chapters dedicated to this second goal (Chapters 4, 5, and 6) do not compare directly against a row-store. Rather they compare the performance of a column-store that implements the new techniques with an equivalent column-store without these additional features.

Hence, the third goal of the dissertation is to take a step back and perform a higher level comparison of the performance of column-stores and row-stores in several application spaces. This comparison is performed both on the traditional sweet-spot (data warehousing), and also a novel application for column-stores: Semantic Web data management. The contributions of each benchmark are now described in turn.

Data Warehouse Application

This benchmark allows us to tie together all of the performance techniques introduced in this dissertation. It uses the same Star Schema Benchmark data and query sets investigated in Chapter 2 and presents new results on the complete

C-Store column-oriented database system that implements the optimizations presented in Chapters 4, 5, and 6. These numbers are compared with the three column-store approaches implemented in Chapter 2 and with a row-store. It then breaks down the reason for the performance differences, demonstrating the contributions to query speedup from compression, late materialization, and the invisible join. Thus, this chapter analyzes the contributions of the performance optimizations presented in this thesis in the context of a real, complete data warehousing benchmark.

Semantic Web Application

Historically column-oriented databases have been relegated to the niche (albeit rapidly growing) data warehouse market. Commercial column-stores (Sybase IQ, SenSage, ParAccel, and Vertica) all focus on warehouse applications and research column-store prototypes all are benchmarked against TPC-H (a decision support benchmark) [7] or the Star Schema Benchmark [57]. Indeed, all of the experiments run in the previous components of the dissertation are run on TPC-H, TPC-H-like data, or the Star Schema Benchmark. The reason for the data warehouse focus is, as described above, that their read-mostly, attribute focused workload is an ideal fit for the column-oriented tradeoffs. But if (especially after a high performance executor is built as described in this dissertation) a column-store can outperform a row-store by a large factor on its ideal application, it is reasonable to suspect it might also significantly outperform a row-store on an application which is not quite so ideal. In the final component of the dissertation, we examine one such "new" application for column-stores: the Semantic Web.

The Semantic Web is a vision by the W3C that views the Web as a massive data integration problem. The goal is to free data from the applications that control them, so that data can be easily described and exchanged. This is accomplished by supplementing natural language found on the Web with machine readable metadata in statement form (e.g., X is-a person, X has-name "Joe", X has-age "35") and enabling descriptions of data ontologies so that data from different applications can be integrated through ontology mapping or applications can conform to committee-standardized ontologies.

Implementing Semantic Web data management using a column-store is not straightforward. The statement format of Semantic Web data is called the Resource Description Framework (RDF). Data is stored in a series of "triples" containing a subject, property, and object (e.g., in the example above, X is a subject, has-name a property, and "Joe" an object). Semantic Web applications over RDF data tend to be read-mostly since statements about resources often do not need to be updated at a high frequency (and often these updates can occur in batch). However, applications in this space do not share the attribute oriented focus that data warehouses do; rather queries over RDF data tend to access many attributes and be rife with joins.

Simply storing these triples in a column-store with a three column table (one column for subject, one for property, and one for object) does not reduce the large amount of joins that are necessary in the execution of most queries. Thus, we propose a data reorganization in which the single three column table is reorganized into a wider, sparser table, containing one column for each unique property. This wide table is then vertically partitioned to reduce the sparsity. We run some experiments and find that both row-stores and column-stores benefit from this reorganization; however, column-stores improve by more than an order of magnitude more than row-stores (which already perform a factor of three faster than the equivalent system applied to the original data layout). Thus, in Chapter 8, we conclude that RDF data management is a new potential application for column-oriented database systems.

1.5   Summary and Dissertation Outline

There are six components of this dissertation. The first component is divided into two chapters. Chapter 2 presents multiple approaches to building a column-store and discusses the challenges of each approach. We use experiments to demonstrate the problems faced in implementing a column-store on top of a commercial row-store. We conclude that changes to the core database system are needed in order to see the benefits of a column-oriented data layout. This leads to the following chapter, where we present these changes in the context of describing the column-store we built to perform the experiments in this dissertation: "C-Store". In essence, Chapter 2 describes top-down implementations of column-stores, while Chapter 3 takes a more bottom-up approach.

The next three components, each presented in separate chapters, introduce performance enhancements to the column-store query executor: Chapter 4 discusses how compression can be built into column-stores both at the storage layer and query executor levels. Chapter 5 then looks at the problem of tuple construction in column-stores and Chapter 6 introduces the invisible join algorithm.

The final two components evaluate the performance of the fully implemented column-store described in this dissertation on complete benchmarks. First, in Chapter 7, we look at the most common application for column-stores — data warehousing. We examine the impact of the various performance optimizations proposed in this dissertation, directly comparing them with each other. The last component, presented in Chapter 8, applies column-oriented database technology to a Semantic Web benchmark. Chapter 9 concludes and presents some ideas for future work.

In summary, this dissertation examines multiple approaches to building a column-oriented database, describes how the performance of these databases can be improved though building a column-oriented query executor, experimentally evaluates the design decisions of this query executor, and demonstrates that column-stores can achieve superior performance to row-stores; not only in their traditional sweet-spot (data warehouses), but also in alternative application areas such as Semantic Web data management.

The main intellectual contributions of this dissertation are:

- A study of multiple implementation approaches of column-oriented database systems, along with a categorization the different approaches into three broad categories, and an evaluation of the tradeoffs between approaches that is (to the best of our knowledge) the only published study of this kind.

- A detailed bottom-up description of the implementation of a modern column-oriented database system.

- An evaluation of compression as a performance enhancement technique. We provide a solution to the problem of integrating compression with query execution, and in particular, the problem of execution engine extensibility. Experimental results demonstrate that a query executor that can operate directly on compressed data can improve query performance by an order of magnitude.

- An analysis of the problem of tuple materialization in column-stores. We provide both an analytical model and heuristics supported by experimental results that help a query planner decide when to construct tuples.

- A new column-oriented join algorithm designed for improving data warehouse join performance.

- A performance benchmark that demonstrates the contribution of various column-oriented performance optimizations (both those proposed in this thesis and also those proposed elsewhere) to overall query performance on a data warehouse application, and compares performance to other column-store implementation approaches and to a commercial row-store.

- A performance benchmark that demonstrates that column-oriented database technology can be successfully applied to Semantic Web data management.

Chapter 2

Column-Store Architecture Approaches and Challenges

2.1 Introduction

As described in Chapter 1, there are three approaches proposed in the literature to building a column-store. The first approach is to use a row-store to simulate a column-store (e.g., by vertically partitioning the data [49]), using a currently available DBMS with the storage manager and execution engines kept in tact; the second approach is to modify the storage manager to store tables column-by-column on disk, but to merge the columns on-the-fly at the beginning of query execution so the rest of the row-oriented query executor can be kept in tact [43, 42]; and the third approach requires modifications to both the storage manager and query execution engine [51, 28, 63]. Clearly, the first approach is easiest to implement since it requires no modifications to currently available DBMSs, the second approach is the next easiest to implement, and the third approach is the most difficult.

In this chapter, we investigate the challenges of building a column-oriented database system by exploring these three approaches in more detail. We implement each of these three approaches and examine their relative performance on a data warehousing benchmark. Clearly, the more one tailors a database system for a particular data layout, the better one would expect that system to perform. Thus, we expect the third approach to outperform the second approach and the second approach to outperform the first approach. For this reason, we are more interested in the magnitude of difference between the three approaches rather than just the relative ordering. For example, if the first approach only slightly underperforms the other two approaches, then it would be the desirable solution for building a column-store since it can be built using currently available database systems without modification.

Consequently, we carefully investigated the first approach. We experiment with multiple schemes for implementing a column-store on top of a row-store, including:

- Vertically partitioning the tables in the system into a collection of two-column tables consisting of (table key, attribute) pairs, so that only the necessary columns need to be read to answer a query;

- Using index-only plans; by creating a collection of indices that cover all of the columns used in a query, it is possible for the database system to answer a query without ever going to the underlying (row-oriented) tables;

- Using a collection of materialized views such that there is a view with exactly the columns needed to answer every query in the benchmark. Though this approach uses a lot of space, it is the 'best case' for a row-store, and provides a useful point of comparison to a column-store implementation.

We implement each of these schemes on top of a commercial row-store, and compare the schemes with baseline performance of the row-store. Overall, the results are surprisingly poor – in many cases the baseline row-store outperforms the column-store implementations. We analyze why this is the case, breaking down the fundamental from the implementation specific reasons for the poor performance.

We then implement the latter two approaches to building a column-store (the storage-layer and the storage-layer/query executor approaches) and compare results both with the above results. We find that the third approach significantly outperforms the other two cases (by a factor of 5.3 and a factor of 2.7). Further, the third approach contains more opportunities for further optimizations. We use this result to motivate our decision to design a new column-store (C-Store) using the third approach, which will be the building block for further experiments and optimizations presented in this dissertation.

2.2   Row-Oriented Execution

In this section, we discuss several different techniques that can be used to implement a column-database design in a commercial row-oriented DBMS (since we cannot name the system we used due to license restrictions, hereafter we will refer to it as System X). We look at three different classes of physical design: a fully vertically partitioned design, an "index only" design, and a materialized view design. In our evaluation, we also compare against a "standard" row-store design with one physical table per relation.

Vertical Partitioning: The most straightforward way to emulate a column-store approach in a row-store is to fully vertically partition each relation [49]. In a fully vertically partitioned approach, some mechanism is needed to connect fields from the same row together (column stores typically match up records implicitly by storing columns in the same order, but such optimizations are not available in a row store). To accomplish this, the simplest approach is to add an integer "position" column to every table – this is often preferable to using the primary key because primary keys can be large and are sometimes composite. This approach creates one physical table for each column in the logical schema, where the ith table has two columns, one with values from column i of the logical schema and one with the corresponding value in the position column. Queries are then rewritten to perform joins on the position attribute when fetching multiple columns from the same relation. In our implementation, by default, System X chose to use hash joins for this purpose, which proved to be expensive. For that reason, we experimented with adding clustered indices on the position column of every table, and forced System X to use index joins, but this did not improve performance – the additional I/Os incurred by index accesses made them slower than hash joins.

Index-only plans: The vertical partitioning approach has two problems. First, it requires the position attribute to be stored in every column, which wastes space and disk bandwidth. Second, most row-stores store a relatively large header on every tuple, which further wastes space (column stores typically – or perhaps even by definition – store headers in separate columns to avoid these overheads). To ameliorate these concerns, the second approach we consider uses *index-only plans*, where base relations are stored using a standard, row-oriented design, but an additional unclustered B+Tree index is added on every column of every table. Index-only plans – which require special support from the database, but are implemented by System X – work by building lists of (record-id,value) pairs that satisfy predicates on each table, and merging these rid-lists in memory when there are multiple predicates on the same table. When required fields have no predicates, a list of all (record-id,value) pairs from the column can be produced. Such plans never access the actual tuples on disk. Though indices still explicitly store rids, they do not store duplicate column values, and they typically have a lower per-tuple overhead than the headers in the vertical partitioning approach.

One problem with the index-only approach is that if a column has no predicate on it, the index-only approach requires the index to be scanned to extract the needed values, which can be slower than scanning a heap file (as would occur in the vertical partitioning approach.) Hence, an optimization to the index-only approach is to create indices with composite keys, where the secondary keys are from predicate-less columns. For example, consider the query `SELECT AVG(salary) FROM emp WHERE age>40` – if we have a composite index with an (age,salary) key, then we can answer this query directly from this index. If we have separate indices on (age) and (salary), an index only plan will have to find record-ids corresponding to records with satisfying ages and then merge this with the complete list of (record-id, salary) pairs extracted from the (salary) index, which will be much slower. We use this optimization in our implementation by storing the the primary key of each dimension table as a secondary sort attribute on the indices over the attributes of that dimension table. In this way, we can efficiently access the primary

key values of the dimension that need to be joined with the fact table.

Materialized Views: The third approach we consider uses materialized views. In this approach, we create an *optimal* set of materialized views for every query flight in the workload, where the optimal view for a given flight has only the columns needed to answer queries in that flight. We do not pre-join columns from different tables in these views. Our objective with this strategy is to allow System X to access just the data it needs from disk, avoiding the overheads of explicitly storing record-id or positions, and storing tuple headers just once per tuple. Hence, we expect it to perform better than the other two approaches, although it does require the query workload to be known in advance, making it practical only in limited situations.

2.3 Experiments

Now that we have described the techniques we used to implement a column-database design inside System X, we present our experimental results of the relative performance of these techniques. We first begin by describing the benchmark we used for these experiments, and then present the results.

All of our experiments were run on a 2.8 GHz single processor, dual core Pentium(R) D workstation with 3 GB of RAM running RedHat Enterprise Linux 5. The machine has a 4-disk array, managed as a single logical volume with files striped across it. Typical I/O throughput is 40 - 50 MB/sec/disk, or 160 - 200 MB/sec in aggregate for striped files. The numbers we report are the average of several runs, and are based on a "warm" buffer pool (in practice, we found that this yielded about a 30% performance increase for the systems we experiment with; the gain is not particularly dramatic because the amount of data read by each query exceeds the size of the buffer pool).

2.3.1 Star Schema Benchmark

For these experiments, we use the Star Schema Benchmark (SSBM) [56, 57] to compare the performance of of the various column-stores.

The SSBM is a data warehousing benchmark derived from TPC-H [7]. Unlike TPC-H, it is a pure, textbook star-schema (the "best practices" data organization for data warehouses). It also consists of fewer queries than TPC-H and has less stringent requirements on what forms of tuning are and are not allowed. We chose it because it is easier to implement than TPC-H and because we want to compare our results on the commercial row-store with our various hand-built column-stores which are unable to run the entire TPC-H benchmark.

Schema: The benchmark consists of a single fact table, the `LINEORDER` table, that combines the `LINEITEM` and `ORDERS` table of TPC-H. This is a 17 column table with information about individual orders, with a composite primary key consisting of the `ORDERKEY` and `LINENUMBER` attributes. Other attributes in the `LINEORDER` table include foreign key references to the `CUSTOMER`, `PART`, `SUPPLIER`, and `DATE` tables (for both the order date and commit date), as well as attributes of each order, including its priority, quantity, price, discount, and other attributes. The dimension tables contain information about their respective entities in the expected way. Figure 2-1 (adapted from Figure 2 of [57]) shows the schema of the tables.

As with TPC-H, there is a base "scale factor" which can be used to scale the size of the benchmark. The sizes of each of the tables are defined relative to this scale factor. In this paper, we use a scale factor of 10 (yielding a `LINEORDER` table with 60,000,000 tuples).

Queries: The SSBM consists of thirteen queries divided into four categories, or "flights". The full query set is presented in full in Appendix B, and the four query flights are summarized here:

1. Flight 1 contains 3 queries. Queries have a restriction on 1 dimension attribute, as well as the `DISCOUNT` and `QUANTITY` columns of the `LINEORDER` table. Queries measure the gain in revenue (the product of `EXTENDED-PRICE` and `DISCOUNT`) that would be achieved if various levels of discount were eliminated for various order quantities in a given year. The `LINEORDER` selectivities (percentage of tuples that pass all predicates) for the three queries are 1.9×10^{-2}, 6.5×10^{-4}, and 7.5×10^{-5}, respectively.
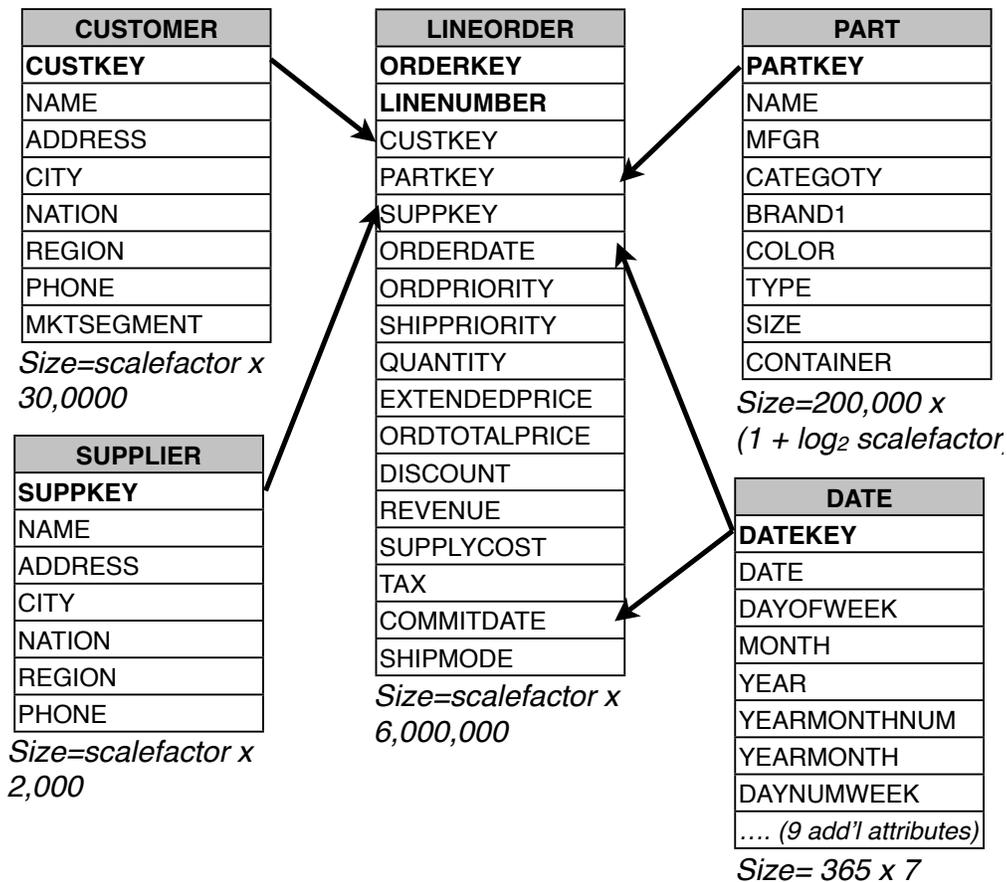
Figure 2-1: Schema of the SSBM Benchmark

2. Flight 2 contains 3 queries. Queries have a restriction on 2 dimension attributes and compute the revenue for particular product classes in particular regions, grouped by product class and year. The LINEORDER selectivities for the three queries are 8.0×10^{-3}, 1.6×10^{-3}, and 2.0×10^{-4}, respectively.

3. Flight 3 consists of 4 queries, with a restriction on 3 dimensions. Queries compute the revenue in a particular region over a time period, grouped by customer nation, supplier nation, and year. The LINEORDER selectivities for the four queries are 3.4×10^{-2}, 1.4×10^{-3}, 5.5×10^{-5}, and 7.6×10^{-7} respectively.

4. Flight 4 consists of three queries. Queries restrict on three dimension columns, and compute profit (REVENUE - SUPPLYCOST) grouped by year, nation, and category for query 1; and for queries 2 and 3, region and category. The LINEORDER selectivities for the three queries are 1.6×10^{-2}, 4.5×10^{-3}, and 9.1×10^{-5}, respectively.

2.3.2 Implementing a Column-Store in a Row-Store

We now describe the performance of the different configurations of System X on the SSBM. We configured System X to partition the lineorder table on orderdate by year (this means that a different physical partition is created for tuples from each year in the database). This partitioning substantially speeds up SSBM queries that involve a predicate on orderdate (queries 1.1, 1.2, 1.3, 3.4, 4.2, and 4.3 query just 1 year; queries 3.1, 3.2, and 3.3 include a substantially less selective query over half of years). Unfortunately, for the column-oriented representations, System X doesn't allow us to partition two-column vertical partitions on orderdate, which means that for those query flights that restrict on the orderdate column, the column-oriented approaches look particularly bad. Nevertheless,

we decided to use partitioning for the base case because it is in fact the strategy that a database administrator would use when trying to improve the performance of these queries on a row-store, so is important for providing a fair comparison between System X and other column-stores.

Other relevant configuration parameters for System X include: 32 KB disk pages, a 1.5 GB maximum memory for sorts, joins, intermediate results, and a 500 MB buffer pool. We enabled compression and sequential scan prefetching.

We experimented with six configurations of System X on SSBM:

1. A "traditional" row-oriented representation; here, we allow System X to use bitmaps if its optimizer determines they are beneficial.

2. A "traditional (bitmap)" approach, similar to traditional, but in this case, we biased plans to use bitmaps, sometimes causing them to produce inferior plans to the pure traditional approach.

3. A "vertical partitioning" approach, with each column in its own relation, along with the primary key of the original relation.

4. An "index-only" representation, using an unclustered B+tree on each column in the row-oriented approach, and then answering queries by reading values directly from the indexes.

5. A "materialized views" approach with the optimal collection of materialized views for every query (no prejoins were performed in these views).

The average results across all queries are shown in Figure 2-2, with detailed results broken down by flight in Figure 2-3. Materialized views perform best in all cases, because they read the minimal amount of data required to process a query. After materialized views, the traditional approach or the traditional approach with bitmap indexing, is usually the best choice (on average, the traditional approach is about three times better than the best of our attempts to emulate a column-oriented approach). This is particularly true of queries that can exploit partitioning on `orderdate`, as discussed above. For query flight 2 (which does not benefit from partitioning), the vertical partitioning approach is competitive with the traditional approach; the index-only approach performs poorly for reasons we discuss below. Before looking at the performance of individual queries in more detail, we summarize the two high level issues that limit the approach of the columnar approaches: tuple overheads, and inefficient column reconstruction:

Tuple overheads: As others have observed [49], one of the problems with a fully vertically partitioned approach in a row-store is that tuple overheads can be quite large. This is further aggravated by the requirement that the primary keys of each table be stored with each column to allow tuples to be reconstructed. We compared the sizes of column-tables in our vertical partitioning approach to the sizes of the traditional row store, and found that a single column-table from our SSBM scale 10 `lineorder` table (with 60 million tuples) requires between 0.7 and 1.1 GBytes of data after compression to store – this represents about 8 bytes of overhead per row, plus about 4 bytes each for the primary key and the column attribute, depending on the column and the extent to which compression is effective ($16\ bytes \times 6 \times 10^7\ tuples = 960\ MB$). In contrast, the entire 17 column `lineorder` table in the traditional approach occupies about 6 GBytes decompressed, or 4 GBytes compressed, meaning that scanning just four of the columns in the vertical partitioning approach will take as long as scanning the entire fact table in the traditional approach.

Column Joins: Merging two columns from the same table together requires a join operation. System X favors using hash-joins for these operations, which is quite slow. We experimented with forcing System X to use index nested loops and merge joins, but found that this did not improve performance because index accesses had high overhead and System X was unable to skip the sort preceding the merge join.

31
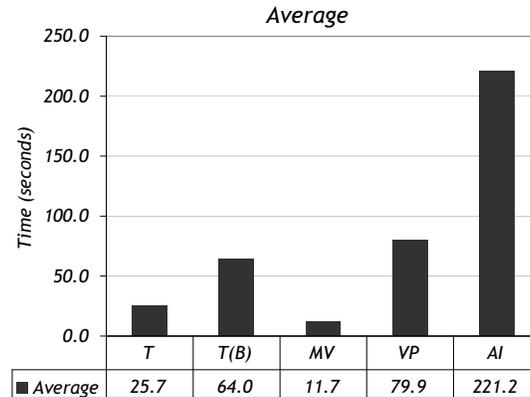
Figure 2-2: Average performance numbers across all queries in the SSBM for different variants of the row-store. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes.

Detailed Row-store Performance Breakdown

In this section, we look at the performance of the row-store approaches, using the plans generated by System X for query 2.1 from the SSBM as a guide (we chose this query because it is one of the few that does not benefit from `orderdate` partitioning, so provides a more equal comparison between the traditional and vertical partitioning approach.) Though we do not dissect plans for other queries as carefully, their basic structure is the same. The SQL for this query is:

```
SELECT sum(lo_revenue), d_year, p_brand1
FROM lineorder, dwdate, part, supplier
WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
  AND lo_suppkey = s_suppkey
  AND p_category = 'MFGR#12'
  AND s_region = 'AMERICA'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1
```

The selectivity of this query is 8.0×10^{-3}. Here, the vertical partitioning approach performs about as well as the traditional approach (65 seconds versus 43 seconds), but the index-only approach performs substantially worse (360 seconds). We look at the reasons for this below.

Traditional: For this query, the traditional approach scans the entire `lineorder` table, using four hash joins to join it with the `dwdate`, `part`, and `supplier` table (in that order). It then performs a sort-based aggregate to compute the final answer. The cost is dominated by the time to scan the `lineorder` table, which in our system requires about 40 seconds. For this query, bitmap indices do not help because when we force System X to use bitmaps it chooses to perform the bitmap merges before restricting on the `region` and `category` fields, which slows its performance considerably. Materialized views take just 15 seconds, because they have to read about 1/3rd of the data as the traditional approach.

Vertical partitioning: The vertical partitioning approach hash-joins the `partkey` column with the filtered `part` table, and the `suppkey` column with the filtered `supplier` table, and then hash-joins these two result sets. This yields tuples with the primary key of the fact table and the `p_brand1` attribute of the `part` table that satisfy the

Flight 1

	T	T(B)	MV	VP	AI
■ Q1.1	2.7	9.9	1.0	69.7	107.2
▨ Q1.2	2.0	11.0	2.0	36.0	50.8
□ Q1.3	1.5	1.5	2.0	36.0	48.5

Flight 2

	T	T(B)	MV	VP	AI
■ Q2.1	43.8	91.9	15.5	65.1	359.8
▨ Q2.2	44.1	78.4	13.5	48.8	46.4
□ Q2.3	46.0	304.1	11.8	39.0	43.9

Flight 3

	T	T(B)	MV	VP	AI
■ Q3.1	43.0	91.4	16.1	139.1	413.8
▨ Q3.2	42.8	65.3	6.9	63.9	40.7
□ Q3.3	31.2	31.2	17.9	48.2	531.4
▨ Q3.4	6.5	6.5	7.8	47.0	65.5

Flight 4

	T	T(B)	MV	VP	AI
■ Q4.1	44.4	94.4	29.2	208.6	623.9
▨ Q4.2	14.1	25.3	22.4	150.4	280.1
□ Q4.3	12.2	21.2	6.4	86.3	263.9
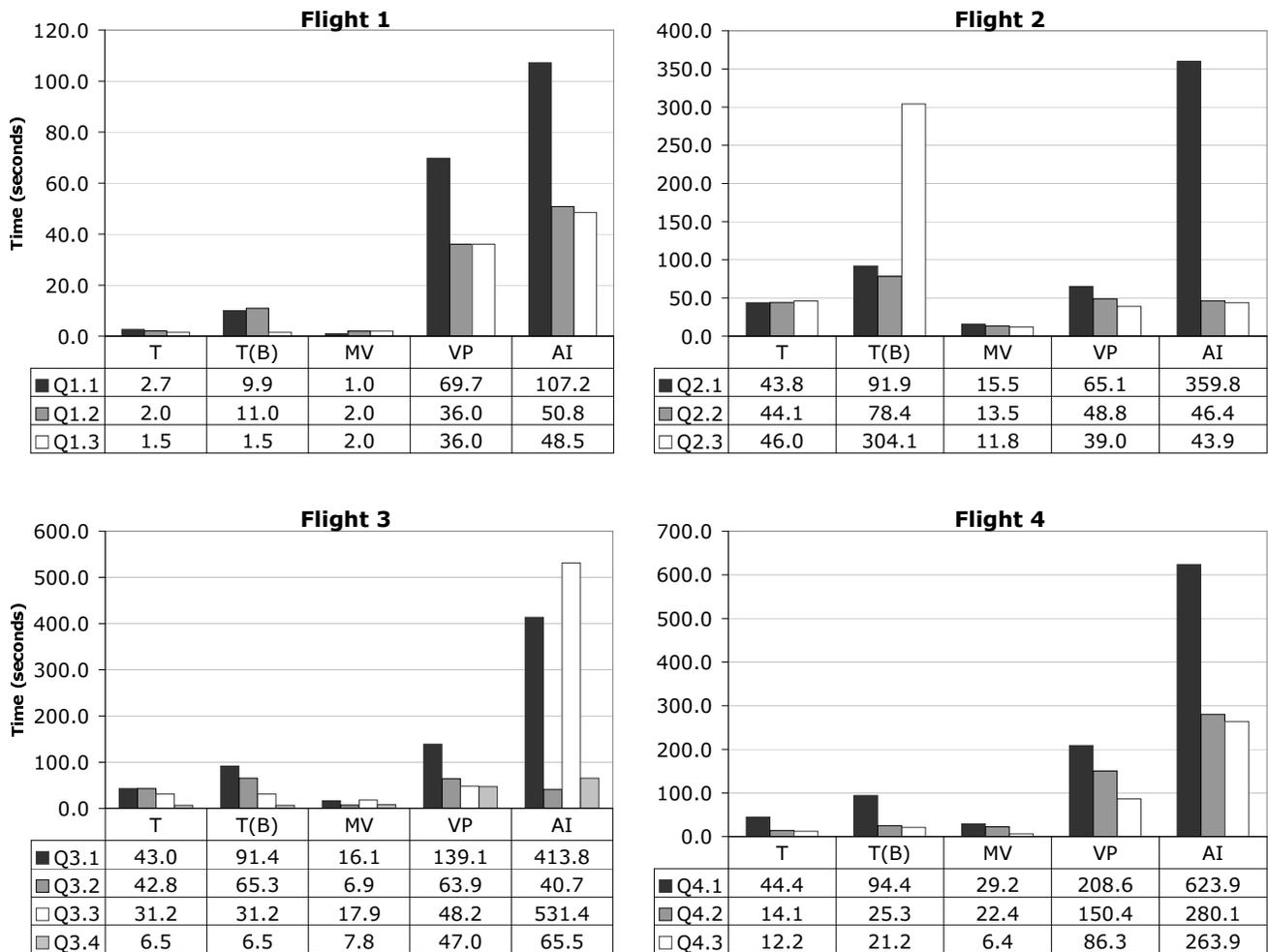
Figure 2-3: Performance numbers for different variants of the row-store by query flight. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes.

query. System X then hash joins this with the `dwdate` table to pick up `d_year`, and finally uses an additional hash join to pick up the `lo_revenue` column from its column table. This approach requires four columns of the `lineorder` table to be read in their entirety (sequentially), which, as we said above, requires about as many bytes to be read from disk as the traditional approach, and this scan cost dominates the runtime of this query, yielding comparable performance as compared to the traditional approach. Hash joins in this case slow down performance by about 25%; we experimented with eliminating the hash joins by adding clustered B+trees on the key columns in each vertical partition, but System X still chose to use hash joins in this case.

Index-only plans: Index-only plans access all columns through unclustered B+Tree indexes, joining columns from the same table on record-id (so they do not require explicitly storing primary keys in each index and never follow pointers back to the base relation). The plan for query 2.1 does a full index scan on the `suppkey`, `revenue`, `partkey`, and `orderdate` columns of the fact table, joining them in that order with hash joins. In this case, the index scans are relatively fast sequential scans of the entire index file, and do not require seeks between leaf pages. The hash joins, however, are quite slow, as they combine two 60 million tuple columns each of which occupies hundreds of megabytes of space. Note that hash join is probably the best option for these joins, as the output of the index scans is not sorted on record-id, and sorting record-id lists or performing index-nested loops is likely to be *much* slower. As we discuss below, we couldn't find a way to force System X to defer these joins until later in the plan, which would

have made the performance of this approach closer to vertical partitioning.

After joining the columns of the fact table, the plan uses an index range scan to extract the filtered `part.category` column and hash joins it with the `part.brand1` column and the `part.partkey` column (both accessed via full index scans). It then hash joins this result with the already joined columns of the fact table. Next, it hash joins `supplier.region` (filtered through an index range scan) and the `supplier.suppkey` columns (accessed via full index scan), and hash joins that with the fact table. Finally, it uses full index scans to access the `dwdate.datekey` and `dwdate.year` columns, joins them using hash join, and hash joins the result with the fact table.

Discussion

The previous results show that none of our attempts to emulate a column-store in a row-store are particularly effective. The vertical partitioning approach can provide performance that is competitive with or slightly better than a row-store when selecting just a few columns. When selecting more than about 1/4 of the columns, however, the wasted space due to tuple headers and redundant copies of the primary key yield inferior performance to the traditional approach. This approach also requires relatively expensive hash joins to combine columns from the fact table together. It is possible that System X could be tricked into storing the columns on disk in sorted order and then using a merge join (without a sort) to combine columns from the fact table but we were unable to coax this behavior from the system.

Index-only plans avoid redundantly storing the primary key, and have a lower per-record overhead, but introduce another problem – namely, the system is forced to join columns of the fact table together using expensive hash joins before filtering the fact table using dimension columns. It appears that System X is unable to defer these joins until later in the plan (as the vertical partitioning approach does) because it cannot retain record-ids from the fact table after it has joined with another table. These giant hash joins lead to extremely slow performance.

With respect to the traditional plans, materialized views are an obvious win as they allow System X to read just the subset of the fact table that is relevant, without merging columns together. Bitmap indices sometimes help – especially when the selectivity of queries is low – because they allow the system to skip over some pages of the fact table when scanning it. In other cases, they slow the system down as merging bitmaps adds some overhead to plan execution and bitmap scans can be slower than pure sequential scans. In any case, for the SSBM, their effect is relatively small, improving performance by at most about 25%.

As a final note, we observe that implementing these plans in System X was quite painful. We were required to rewrite all of our queries to use the vertical partitioning approaches, and had to make extensive use of optimizer hints and other trickery to coax the system into doing what we desired.

In the next section we study how column-stores designed using alternative approaches are able to circumvent these limitations.

2.4   Two Alternate Approaches to Building a Column-Store

Now that we have described in detail the first approach to building a column-store, we describe two alternative approaches: modifying the storage manager to store tables column-by-column on disk, but merging the columns on-the-fly at the beginning of query execution so the rest of the row-oriented query executor can be kept in tact; and modifying both the storage manager and query execution engine.

2.4.1   Approach 2: Modifying the Storage Layer

Unlike the first approach discussed in this chapter, this approach does not require any changes to the logical schema when converting from a row-store. All table definitions and SQL remain the same; the only change is the way tables are physically laid out on storage. Instead of mapping a two-dimensional table row-by-row onto storage, it is mapped column-by-column.

When storing a table on storage in this way, the tuple IDs (or primary keys) needed to join together columns from the same table are not explicitly stored. Rather, implicit column positions are used to reconstruct columns (the ith value from each column belong to the ith tuple in the table). Further, tuple headers are stored in their own separate columns and so they can be accessed separately from the actual column values. Consequently, a column stored using this approach contains just data from that column, unlike the vertical partitioning approach where a tuple header and tuple ID is stored along with column data. This solves one of the primary limitations of the previous approach. As a point of comparison, a single column of integers from the SSBM fact table stored using this approach takes just 240 MB ($4\ bytes \times 6 \times 10^7\ tuples = 240\ MB$). This is much smaller than the 960 MB needed to store a SSBM fact table integer column in the vertical partitioning approach above.

In this approach, it is necessary to perform the process of tuple reconstruction before query execution. For each query, typically only a subset of table columns need to be accessed (the set of accessed columns can be derived directly from inspection of the query). Only this subset of columns are read off storage and merged into rows. The merging algorithm is straightforward: the ith value from each column are copied into the ith tuple for that table, with the values from each component attribute stored consecutively. In order for the same query executor to be used as for row-oriented database systems, this merging process must be done before query execution.

Clearly, it is necessary to keep heap files stored in position order (the ith value is always after the $i-1$th value), otherwise it would not be possible to match up values across columns without a tuple ID. In contrast, in the storage manager of a typical row-store (which, in the first approach presented in this chapter, is used to implement a column-store), the order of heap files, even on a clustered attribute, is only guaranteed through an index. This makes a merge join (without a sort) the obvious choice for tuple reconstruction in a column-store. In a row-store, since iterating through a sorted file must be done indirectly through the index, which can result in extra seeks between index leaves, an index-based merge join is a slow way to reconstruct tuples. Hence, by modifying the storage layer to guarantee that heap files are in position order, a faster join algorithm can be used to join columns together, alleviating the other primary limitation of the row-store implementation of a column-store approach.

2.4.2   Approach 3: Modifying the Storage Layer and Query Execution Engine

The storage manager in this approach is identical to the storage manager in the previous approach. The key difference is that in the previous approach, columns would have to be merged at the beginning of the query plan so that no modifications would be necessary to the query execution engine, whereas in this approach, this merging process can be delayed and column-specific operations can be used in query execution.

To illustrate the difference between these approaches, take, for example, the query:

```
SELECT X
FROM TABLE
WHERE Y < CONST
```

In approach 2, columns X and Y would be read off storage, merged into 2-attribute tuples, and then sent to a row-oriented query execution engine which would apply the predicate on the Y attribute and extract the X attribute if the predicate passed. Intuitively, there is some wasted effort here – all tuples in X and Y are merged even though the predicate will cause some of these merged tuples to be immediately discarded. Further, the output of this query is a single column, so the executor will have to eventually unmerge ("project") the X attribute.

When modifications to the query executor are allowed, a different query plan can be used. The Y column is read off storage on its own, and the predicate is applied. The result of the predicate application is a set of positions of values (in the Y column) that passed the predicate. The X column is then scanned and values at this successful set of positions are extracted.

There are a variety of advantages of this query execution strategy. First, it clearly avoids the unnecessary tuple merging and unmerging costs. However, there are some additional less obvious performance benefits. First, less data is being moved around memory. In approach 2, entire tuples must be moved from memory to CPU for predicate application (this is because memory cannot be read with fine enough granularity to access only one attribute from a

tuple; this is explained further in Section 3.2). In contrast, in this approach, only the Y column needs to be sent to the CPU for predicate application.

Second, since heap files are stored in position order as described above, we can access the heap file directly to perform this predicate application. If the column is fixed-width, it can be iterated through as if it were an array, so calculations do not have to be performed to find the next value to apply the predicate to. However, once attributes have been merged into tuples, as soon as any attribute is not fixed width, the entire tuple is not fixed width, and the location of the next value to perform the predicate on is no longer at a constant offset.

2.5 Comparison of the Three Approaches

Now that we have described all three approaches, we can compare their performance. In Section 2.5.1 we give some context to help with the comparison of these approaches, and in Section 2.5.2 we perform the comparison.

2.5.1 Context for Performance Comparison

Directly comparing the performance of the three column-store approaches is not straightforward. Ideally, each approach would be implemented with as little variation in the code as possible – e.g., approach 1 and approach 2 would share a query executor and approach 2 and approach 3 would share a storage manager. However, since the major advantage of approach 1 is that it can be implemented using a currently available DBMS, it was important that we analyze performance on such a DBMS. Although open source database systems are starting to make inroads in the OTLP and Web data management markets, the data warehousing market is still dominated by large proprietary database software (Oracle, TeraData, IBM DB2, and Microsoft SQL Server) [65]. This is the reason why we choose one of these proprietary DBMSs for the experiments on implementation approach 1. However, given the proprietary nature of the code, extending it to implement the other 2 approaches was not an option.

Further, since none of the currently available column-stores (e.g. Sybase IQ or Monet) currently implement both approach 2 and approach 3, we chose to implement these approaches ourselves, from scratch (we will call our implementation "C-Store"). We implemented basic versions of these approaches. The storage manager was identical for each approach – columns from a table were stored in separate files with sparse indexes on position so that the block containing the attribute value for a specific tuple ID can be located quickly. For approach 2, basic row-store operators were used (select, project, join, aggregate) – we only implemented the necessary operators to run this benchmark. For approach 3, we allowed predicate application to be performed in the way we described in Section 2.4.2, where predicates are applied to a individual columns with positions being produced as a result. These position results are then sent to other columns for column extraction. Aside from predicate application, all other operators were normal row-store operators, with column merging occurring before these row-store operators. The nitty-gritty details of our DBMS implementation is given in Chapter 3.

Note that the code overlap between these latter two approaches is high (only differing in key differences between these approaches), but there is very little code overlap between these approaches and approach 1. Further, approach 1 was implemented by hundreds of people and consists of millions of lines of code, containing a variety of optimizations that would not be practical for a much smaller team of people to implement (including partitioning, compression, and multi-threading – though we will implement compression for experiments later in this dissertation). Thus, we expect the baseline performance of the commercial system to be significantly faster than the baseline performance of C-Store.

In order get a better sense of the difference in the baseline performance between the two systems, we compared the row-store query executor built for approach 2 with the query executor that came with the DBMS we used to implement approach 1. We compared these query executors for the materialized view case on the SSBM benchmark described above – materialized views of the fact table containing only those columns that are necessary to answer any particular query are used in query execution. Although approach 2 calls for all tables (including materialized views of the fact table) to be stored column-by-column, in order to compare baseline performance, for this experiment tables were stored row-by-row. Thus, both systems read the same data, and the column-store does not need to merge

Flight 1

	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■1.1	1.0	18.49	32.7	5.4
■1.2	2.0	9.18	27.3	2.2
□1.3	2.0	8.36	26.4	2.0

Flight 2

	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■2.1	15.5	33.29	40.8	16.7
■2.2	13.5	23.43	35.9	13.6
□2.3	11.8	22.06	34.8	13.3

Flight 3

	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■3.1	16.1	48.42	56.7	31.9
■3.2	6.9	22.12	33.5	15.7
□3.3	17.9	17.34	29.8	13.6
■3.4	7.8	17.54	29.8	13.4

Flight 4

	Row-Store MV	C-Store MV	C-Store Approach 2	C-Store Approach 3
■4.1	29.2	47.55	67.1	29.9
■4.2	22.4	39.51	60.0	20.3
□4.3	6.4	32.21	54.0	16.2

Figure 2-4: Performance numbers for column-store approach 2 and approach 3. These numbers are helped put in context by comparison to the baseline MV cases for the commercial row-store (presented above) and the newly built DBMS.

together relevant columns since they have already been pre-merged. Both systems are executing the same query on the same input data stored in the same way.

The results of this experiment are displayed as "Row-Store MV" and "C-Store MV" in Figure 2-4 for each of the 13 SSBM queries, and average performance difference is displayed in Figure 2-5. As expected, the commercial DBMS is consistently faster than the DBMS that we built. This performance difference is largest for query flight 1, where the commercial DBMS really benefits from being able to partition on date (while our DBMS cannot). Even on query flight 2, which does not benefit from partitioning (as described above), the commercial DBMS outperforms our DBMS; however in this case the difference is less than a factor of 2. We believe that the lack of compression and multi-threading in our implementation accounts for the bulk of this difference (we will show in later chapters that compression yields large performance gains).

2.5.2 Performance Comparison

Now that we have a better sense of the baseline performance differences across the DBMSs in which we perform our experiments, we now present performance results of the latter two approaches to building a column-store.
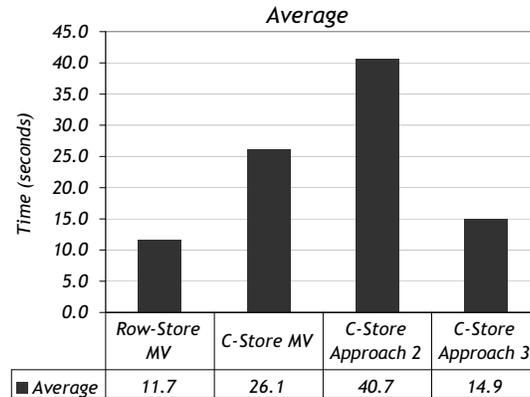
Figure 2-5: Average performance numbers across all 13 queries for column-store approach 2 and approach 3. These numbers are helped put in context by comparison to the baseline MV cases for the commercial row-store (presented above) and the newly built DBMS.

The results on these implementations are displayed as "C-Store Approach 2" and "C-Store Approach 3" in Figure 2-4 for each of the 13 SSBM queries, and average performance difference is displayed in Figure 2-5. As described in Section 2.5.1, comparing these results with the first approach is not straightforward. Nonetheless, given that the fastest performing version of approach 1 (vertical partitioning) went on average 79.9 seconds on this benchmark, and given that our baseline experiments showed that the DBMS used to implement the latter two approaches was inherently slower than the DBMS used to implement the first approach, one can conclude that these approaches significantly outperform the first approach, by at least a factor of 2.

Note that our implementation of approach 3 is very basic. Once one is willing to build an executor designed for the column-oriented layout, there are a variety of other optimizations that could be applied. Indeed we will show this in Chapters 4, 5, and 6 of this dissertation. Thus, the performance numbers for approach 3 is an upper bound for how well this approach can perform. We will revisit this issue in Chapter 7.

Nonetheless, approach 3 is already competitive with the materialized view approach. This is significant since the materialized view approach is the best-case scenario for a row-store, and is only useful in situations where a query workload is known in advance, so that the choice of columns to include in the views can be carefully selected.

2.6 Conclusion

In this chapter, we described three approaches to building a column-store. Each approach requires more modifications to the DBMS than the last. We implemented each approach and showed, through performance results, that these extra modifications to the DBMS result in significant performance gains. The third approach, which requires that the storage layer and the query execution engine be designed for the column-orientation of the data, performs almost a factor of 3 faster than the second approach (which requires only modification to the storage layer), and at least a factor of 5 faster than the first approach (which works on current DBMS systems without modification). Further, this approach opens up possibilities for further optimizations, that, as we will show in later chapters, result in significant performance gains. Thus, we recommend the third approach for building column-oriented database systems. Given that this is the case, we describe the details of how we built our column-store, "C-Store", using this approach in the next chapter, and describe optimizations to this approach in Chapters 4, 5, and 6.

Chapter 3

C-Store Architecture

In the previous chapter, we showed that building a database system with a storage layer and query executor designed for a column-oriented data layout ("approach 3") is the best performing approach to implementing a column-store. Consequently, we set out to build a complete column-store implementation using this approach (instead of the bare-bones version used for the previous chapter). We use this implementation, called C-Store, for most of the experiments presented in this dissertation, and extend its code-base for the performance optimizations presented in the next three chapters. In this chapter, we present a detailed, bottom-up description of the C-Store implementation.

At current time, approximately 90% of the code in C-Store was written to run experiments in the next few chapters (and approximately 85% of this code was written by the dissertation author). As a result, we focus in this chapter only on the architecture of the query execution engine and storage layers. Other parts of the system are mentioned in Section 3.7.

As a side note, although many of the ideas presented in this dissertation are currently being commercialized at Vertica Systems [10], Vertica and C-Store are two separate lines of code. The C-Store code line is open source [15], while the Vertica code-line is proprietary. C-Store's code-line was originally released in 2005; an update was released in 2006; and no more releases are planned.

3.1 Overview

C-Store provides a relational interface on top of data that is physically stored in columns. Logically, users interact with tables in SQL (though in reality, at present time, most query plans have to be hand-coded). Each table is physically represented as a collection of *projections*. A projection is a subset of the original table, consisting of all of the table's rows and subset of its columns. Each column is stored separately, but with a common sort order. Every column of each table is represented in at least one projection, and columns are allowed to be stored in multiple projections — this allows the query optimizer to choose from one of several available sort orders for a given column. Columns within a projection can be secondarily or tertiarily sorted; e.g., an example C-Store projection with four columns taken from TPC-H could be:

```
(shipdate, quantity, retflag, suppkey | shipdate, quantity, retflag)
```

indicating that the projection is sorted by shipdate, secondarily sorted by quantity, and tertiarily sorted by return flag in the example above. For example, the table: (1/1/07, 1, False, 12), (1/1/07, 1, True, 4), (1/1/07, 2, False, 19), (1/2/07, 1, True, 2) is secondarily sorted on quantity and tertiarily sorted on retflag. These secondary levels of sorting increase the locality of the data, improving the performance of most of the compression algorithms (for example, RLE compression can now be used on quantity and return flag; this will be discussed more in Chapter 4). Projections in C-Store often have few columns and multiple secondary sort orders, which allows most columns to compress quite well. Thus, with a given space budget, it is often possible to store the same column in multiple projections, each with a different sort order.

39

Projections in C-Store could in theory be related to each other via *join indices* [63], which are simply permutations that map tuples in one projection to the corresponding tuples in another projection from the same source relation. These join indices would be used to maintain the original relations between tuples when those tuples have been partitioned and sorted in different orders. To process some queries, C-Store would then use these join indices during query execution to construct intermediate results that contain the necessary columns. In practice however, the reconstruction of tuples containing columns from multiple projections using a join index is quite slow. Consequently, a projection containing all columns from a table is typically maintained, and if a query cannot be answered entirely from a different projection, this complete projection is used. Join indexes are not used in any experiments presented in this dissertation.

C-Store contains both a read-optimized store (RS) which contains the overwhelming majority of the data, along with an uncompressed write-optimized store (WS) which contain all recent inserts and updates. There is a background process called a Tuple Mover which periodically (on the order of once per day) moves data from the WS to the RS. All experiments presented in this dissertation focus on the RS since all queries experimented with are read-only. Load-time into a column-store is an important area of future work.

As will be described in Chapter 4, C-Store compresses each column using one of the methods described in Section 4.3. As the results presented in Chapter 4 will show, different types of data are best represented with different compressions schemes. For example, a column of sorted numerical data is likely best compressed with RLE compression, whereas a column of unsorted data from a smaller domain is likely best compressed using dictionary compression. Although not currently implemented, we envision (and this could be an interesting direction for future research) a set of tools that automatically select the best projections and compression schemes for a given logical table.

The rest of this chapter is outlined as follows. The next section gives some background on the way I/O works, both from disk to memory and from memory to CPU. This model is important take into consideration as one evaluates the fundamental differences between a row-by-row and column-by-column data layout. The remaining sections present specific components of the C-Store architecture. Section 3.3 presents the C-Store storage layer, and explains how it is influenced by the I/O model. Section 3.4 presents the C-Store execution model. Section 3.5 discusses the C-Store operators and Section 3.6 describes a common technique for improving column-oriented operation implemented in C-Store.

3.2 I/O Performance Characteristics

We now describe the assumptions this dissertation takes with regard to how data is read from disk to memory or from memory to CPU before being operated upon.

3.2.1 Disk

Modern disks are mechanical devices containing rotating platters with magnetic surfaces. Each platter is divided into concentric rings called tracks, and each track is divided into sectors. A sector is the smallest unit of data access on a disk, typically containing around 512 bytes of data. Consequently, even if one desires to read just one byte of data from a particular sector, at least 512 bytes are read (typically more since operating systems combine sectors into larger blocks).

When considering disk access performance, one must consider random access time (the time to physically move the read/write disk head to the right place, a "seek", and the time to get the addressed area of disk to a place where it can be accessed by the disk head, "rotational delay"), and transfer time (the time spent actually reading data). Random access time takes 4ms to 10ms on high-end modern disks and dominates access times for single accesses. Data stored consecutively on disk can be transferred at a rate of 40-110MB/s.

Since CPU can generally process data at a rate of more than 3GB a second, disk access time can easily dominate database workloads and so I/O efficiency is very important. There are two things to consider on this front. First, one wants to pack as much relevant data as possible inside disk sectors since disks cannot be read at a finer granularity.

Assuming multiple tuples fit inside a disk sector (under a row-by-row data layout), individual attributes from a tuple cannot be read (rather entire tuples must be read). Thus, if a query only accesses a small percentage of attributes from a tuple, the row-by-row layout is inefficient.

Second, since random access time can dominate individual access time, it is desirable to avoid individual accesses. Rather, it is desirable to read many consecutive sectors from a disk in a single I/O operation. This means that in a column-store, one must be cognizant of the cost of seeking back and forth between different columns on disk.

3.2.2 Memory

Memory has similar characteristics to disk but at a finer granularity. Data is read from memory to CPU as cache lines (typically 64 to 128 bytes). As with disks, multiple attributes can fit in a cache line, so the row-by-row data layout is inefficient. Although random access time is not as costly relative to sequential access time as for disks, cache prefetching in modern CPUs makes sequential access desirable.

3.3 Storage layer

C-Store stores each column from a table (or projection) in a separate file. Each column is divided into blocks with each block containing 64K of data. If data is compressed or variable width, it is possible for different blocks to contain different numbers of values for that column. Each block can be compressed using an algorithm best suited for that block, though in practice the same compression algorithm is used for all blocks in a column. Each block contains a small header at the beginning of the block, indicating the compression algorithm used, the number of values stored in the block, and the ordinal position (which serves as a tuple identifier) of the first value in the block. Some compression schemes require some additional information in the header containing the parameters used for those schemes. Thus, block headers contain approximately 10-20 bytes of data — negligible relative to the 64K of data stored in the block. In the future, it might be desirable to add pointers in the header to variable length values in a block to speed up random access, but so far we have not found this necessary to implement for the workloads we've been experimenting with.

These sequences of blocks can either be stored on the file system directly, or stored as 64K attributes inside Berkeley DB indexed files. The latter approach can be a little slower since reading a block requires an indirection through BerkeleyDB, but it allows for the use of the BerkeleyDB indexing code. At present time, if a column is stored on the file system directly, it cannot be indexed.

If a column is stored in BerkeleyDB, there are two types of indexes that can be used. First, if a column is sorted, then a sparse index on column value is created. In this case, the last value of each 64K block is stored in the index. This allows the block containing a specific value to be found rapidly (once the block is retrieved, a binary search must be used to find the actual value). Second, on all columns, a sparse index on position (which is also the tuple ID) is created. Again, the last position of each block is stored in the index. This is useful for operators that need to find a given value for a specific tuple ID (for example, if the executor needs to reconstruct a complete tuple, the position index for each attribute in the table can be accessed to find the block containing the attribute value for that tuple). In Section 3.4, we present a query plan (Figure 3-1) with another example where values need to be extracted at specific positions (the DS3 operator). Since each column is sorted in position order, it is usually best to scan the column to extract to values at multiple positions within the column; however, the index is used to jump to the first needed position.

3.3.1 Prefetching

As described above (Section 3.2), random access has a proportionally higher cost relative to sequential access, especially for disks (more than an order of magnitude). Although blocks from the same column are stored sequentially, different columns are stored separately. Consequently, if more than one column needs to be accessed for a particular

query, there is a danger of having to seek back and forth on disk as the two columns are read in parallel, which significantly slows query performance.

To alleviate this problem, C-Store by default (this is a parameter that can be modified) reads ("prefetches") 4MB (64 blocks) from a column at a time on any access. This data is held in memory as 4MB from other columns are read into memory. Once all columns are in memory, random access across columns is significantly faster (though as will be shown in Chapter 5, still can't be ignored). This pattern of access incurs a seek only once in every 64 blocks on a parallel column read, rather than on every block. However, the benefits of prefetching are workload dependent. For OLTP-style "needle-in-a-haystack" queries, reading 64 blocks when only 1 is needed can be wasteful of disk bandwidth. On the other hand, workloads in data warehouses tend to read a much higher percentage of data from a column as it is summarizing or aggregating it. Since C-Store is designed for data warehouse workloads, the default prefetch size is large; for other workloads this parameter should be reduced. In the future, this parameter might be able to self-tune.

3.4 Data Flow

In a row-store, a query plan is a tree of operators. Selections, projections, and aggregations have a single child while joins have multiple children. Since the operators are formed into a tree, all nodes have a single parent except the root node. The most common data flow model in row-stores (especially for single threaded databases) is the pull-based iterator model, where the root node calls "getNextTuple" on its children, who call "getNextTuple" on their children, all the way down the tree. Tuples are then processed and sent to parent nodes as the return value of this "getNextTuple" call. Since this requires one function call for each tuple processed per operator, some databases process and return multiple tuples at once rather an a single tuple, to amortize the cost of this function call. However, since the per tuple processing cost in row-stores are fairly high (additional function calls are required to extract attributes before they can be operated on) and many row-stores are I/O limited anyways, this optimization is not used in some row-stores.

For column-stores, this optimization is crucial. In order to reap the benefit of column-oriented vectorized operation (treating a column as an array and iterating through it directly, allowing compilers to make good loop pipelining optimizations, see Section 3.6), multiple values from a column must be present at once. Thus, blocks of data are passed between operators (or more accurately, as described below, iterators pointing to blocks), and the function call is "getNextBlock" rather than "getNextTuple". Nevertheless, the data flow model is still pull-based.

There is, however, an additional complication in row-stores not present in column-stores: operators in query plans no longer form trees. In fact, it is very common for an operator to have multiple parents. For example, Figure 3-1 shows a sample column-oriented query plan for a simple query that contains a selection clause on two columns:

```
SELECT shipdate, linenum FROM lineitem
WHERE  shipdate < CONST1 AND linenum < CONST2
```

The "DS" operators are short for "DataSource" and are described in detail in Chapter 5 (there are four types named DS1-DS4). In short, these operators are responsible for reading columns off disk and performing some kind of filtering. In this example, DS1 operators read a column off disk and apply a predicate to the column. The output of the operator are a list of positions of values that passed the predicate. These positional outputs are then intersected using an AND operator. Up until this point in the query plan, the operators have formed a tree. The problem is that we need to extract the shipdate and linenum values at positions that passed both predicates. The operator that does this is the "DS3" operator. The problem is that the output of the AND operator is required by both instances of the DS3 operator (for shipdate and for linenum). Thus, the AND operator has two parents and the operators no longer form a graph.

This lack of tree structure is problematic for a pull-based iterator model. If one parent consumes its input data at a faster rate than the other, data might be lost before the slower parent has an opportunity to process it.
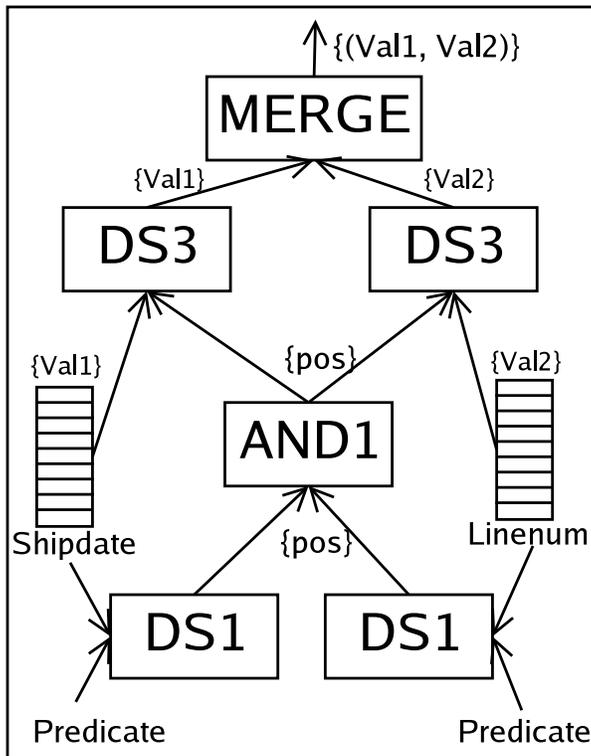
Figure 3-1: A column-oriented query plan

The problem is solved by ensuring that the graph is still rooted with a single node that has no parents. As long as this node requests data at the same rate from all of its children (and this property holds recursively for every other descendant node in the graph), then nodes with more than one parent are guaranteed not to have the problem of one parent requesting data at a faster rate than the other. Thus, C-Store code is written very carefully — when an operator's own "getNextBlock" is called, it will call "getNextBlock" on a child operator only if all data from the previous block have been processed and are guaranteed never to be needed again, and the last position processed from the previous block is equal to the last position processed from all of the other children blocks.

Note that this also simplifies memory management. Since "getNextBlock" is called on a child operator only if all data from the previous block have been processed and are guaranteed never to be needed again, all memory used to store the data from the previous block can be freed (or, more typically, repurposed to store the data for the next block).

3.4.1 Iterators

Most operators do not pass blocks of data between each other; rather they pass iterators (see Figure 3-2). Like a block, an iterator has a start position, an end position, and returns block values through getNext and other related methods. Unlike a block, it doesn't contain its own buffer of data; rather, it points to a buffer that exists externally to the iterator. This has the advantage that if the result of an operator must be sent to multiple parent operators (as described above), the entire output block does not need to be copied and sent to the parents; only iterators of these blocks need to be sent.

An iterator can point to an entire block, or to a ranged subset of a block. This is useful when an operator receives multiple inputs from different child operators and each input might cover a different position range. An array of iterators covering the position range of the intersection of the position ranges of each input can be easily created.
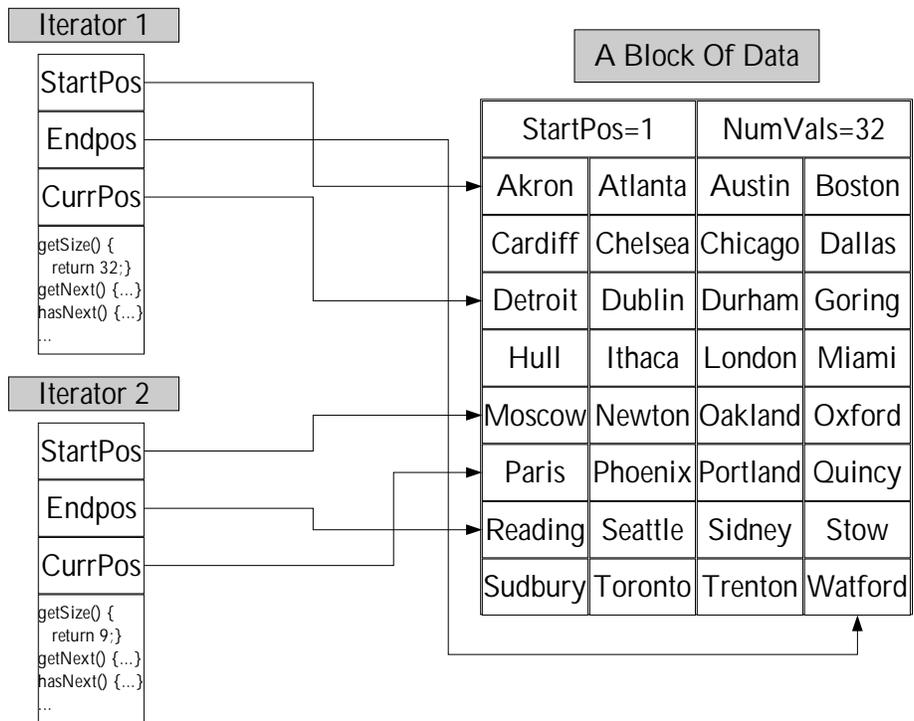
Figure 3-2: Multiple iterators can return data from a single underlying block

Take, for example, a binary operator that takes two inputs from two different columns — one is compressed and the other one is not compressed. Assume that on the first call, it receives an iterator pointing to a block with the first 30,000 values (positions 1-30,000) from the first input, and an iterator pointing to a block with 2,000 values (positions 1-2,000) from the second input. It can then create a new iterator covering the position range 1-2,000 from the first input so that it can perform its operation on the same tuple subset. On its next call it only needs to get the next input from the second child. Assuming that it gets another 2,000 values from this input, an iterator on the position range 2,001-4,000 can be created for the first input.

3.5 Operators

C-Store includes column-oriented versions of most of the familiar relational operators. The major differences between C-Store operators and relational operators are:

- Predicate evaluation produces bit-columns that can be efficiently combined. "DataSource" operators can be used to materialize a subset of values from a column and a bitmap.

- Projection is free since it requires no changes to the data, and two projections in the same order can be concatenated for free as well.

- Joins can produce *positions* rather than values. A discussion of this distinction is given in Sections 4.4.2 and 5.4.3.

There are a total of 19 operators implemented in C-Store. We describe each operator in detail in Appendix A.

3.6   Vectorized Operation

An oft-cited [19, 28, 29] advantage of column-oriented databases is vectorized operation. In order to process a series of tuples, row-stores first need iterate through each tuple, and then need to extract the needed attributes from these tuples through a tuple representation interface. This leads to tuple-at-a-time processing, where there are 1-2 function calls to extract needed data from a tuple for an operation (which, if the operation is a small expression or predicate evaluation, is low cost compared with the function calls). This leads to low IPC (instructions per cycle) efficiency and thus poor CPU performance.

Column-stores can take advantage of multiple attribute values (a "vector") being available at once to implement array iteration with good loop pipelining techniques so that operations can be performed more than once per function call and a higher IPC efficiency reached (this code also is able to take advantage of the super-scalar properties of modern CPUs [28]).

C-Store implements vectorized operation by exposing an "asArray" method from blocks (in addition to a "get-Next" method). Block.asArray() returns the entire contents of the block as a pointer to an array (in addition to the value size in bytes) so that operators can iterate through this array directly, rather than call getNext once for each value inside the block.

3.7   Future Work

Since this dissertation focuses on query execution, C-Store's query executor is reasonably complete. However, many of the other ideas proposed a 2005 vision paper [63] as planned parts of the system — a query parser, a plan generator, an optimizer, a database designer, and a tuple mover, are all either non-existent or bare-bones and rarely used. Since the commercial version of C-Store (Vertica) has implemented these system components, the plan is to only add them to C-Store if they present interesting research problems.

It is also important to note that C-Store runs in a single process, and is single-threaded. Consequently, it is not able to take advantage of the multiple cores present in modern servers. For an experimental system, this missed performance opportunity is not problematic; however production systems should be better able to take advantage of multiple cores (as Vertica is able to do), especially since the trend seems to be that the number of cores per machine are increasing.

3.8   Conclusion

In this chapter, we presented the architecture and implementation details of C-Store's query execution engine and storage layer. There were several components that look very different from corresponding components in standard row-oriented database systems. Obviously, at the storage layer level, storing each column in a separate file is unique to column-stores. However, the lack of dense indexes (C-Store only uses Berkeley DB sparse indexes to find the right 64K block) would be very unusual in a row-store. Further, prefetching takes increased importance in a column-store.

At the query execution engine level, C-Store's query plans look very different than row-oriented query plans. Not only is the operator set different (containing column-oriented versions of operators), but the structure of the plans is also different. In a row-store, every plan must form a tree, while in a column-store this requirement is not practical. This requires careful coding in operator implementation in the rate data is requested from children operators. Further, using vectorized operations is a key performance requirement (we will experimentally demonstrate this in Chapter 7).

Now that we have presented the basic execution engine architecture, in the next three chapters we describe three performance enhancing techniques that can be added to further improve performance. Chapter 4 discusses how compression can be built into the system both at the storage layer and query executor levels. Chapter 5 then looks at the problem of tuple construction, and Chapter 6 introduces the invisible join algorithm.

Chapter 4

Integrating Compression and Execution

4.1 Introduction

Compression in traditional database systems is known to improve performance significantly [39, 47, 59, 40, 48, 77]: it reduces the size of the data and improves I/O performance by reducing seek times (the data are stored nearer to each other), reducing transfer times (there is less data to transfer), and increasing buffer hit rate (a larger fraction of the DBMS fits in buffer pool). For queries that are I/O limited, the CPU overhead of decompression is often compensated for by the I/O improvements.

In this chapter, we revisit this literature on compression in the context of column-oriented database systems. Storing data in columns presents a number of opportunities for improved performance from compression algorithms when compared to row-oriented architectures. In a column-oriented database, compression schemes that encode multiple values at once are natural. In a row-oriented database, such schemes do not work as well because an attribute is stored as a part of an entire tuple, so combining the same attribute from different tuples together into one value would require some way to "mix" tuples.

Compression techniques for row-stores often employ dictionary schemes where a dictionary is used to code wide values in the attribute domain into smaller codes. For example, a simple dictionary for a string-typed column of colors might map "blue" to 0, "yellow" to 1, "green" to 2, and so on [39, 60, 32, 77]. Sometimes these schemes employ prefix-coding based on symbol frequencies (e.g., Huffman encoding [46]) or express values as small differences from some frame of reference and remove leading nulls from them (e.g., [67, 40, 60, 77]).

In addition to these traditional techniques, column-stores are also well-suited to compression schemes that compress values from more than one row at a time. This allows for a larger variety of viable compression algorithms. For example, run-length encoding (RLE), where repeats of the same element are expressed as (value, run-length) pairs, is an attractive approach for compressing sorted data in a column-store. Similarly, improvements to traditional compression algorithms that allow basic *symbols* to span more than one column entry are also possible in a column-store.

Compression ratios are also generally higher in column-stores because consecutive entries in a column are often quite similar to each other, whereas adjacent attributes in a tuple are not [51]. Further, the CPU overhead of iterating through a page of column values tends to be less than that of iterating through a page of tuples (especially when all values in a column are the same size), allowing for increased decompression speed by using vectorized code (i.e., operating directly on arrays, as was described in Section 3.6). Finally, column-stores can store different columns in different sort-orders [63], further increasing the potential for compression, since sorted data is usually quite compressible.

Column-oriented compression schemes also improve CPU performance by allowing database operators to operate directly on compressed data. This benefit is particularly realizable for compression schemes like run length encoding that refer to multiple entries with the same value in a single record. For example, if a run-length encoded column says the value "42" appears 1000 times consecutively in a particular column for which we are computing a SUM aggregate, the operator can simply take the product of the value and run-length as the SUM, without having to

decompress.

In this chapter, we study a number of alternative compression schemes that are especially well-suited to column stores, and show how these schemes can easily be integrated into C-Store.

The experiments on compression performance presented in this chapter yield several surprising results. For example, for some queries, heavy weight compression schemes (schemes that optimize for compression ratio and sacrifice compression and decompression speeds to do so, typically operating page-at-a-time, such as Lempel-Ziv) do not result in performance degradation. This is in contrast with recent papers published on database compression [40, 67, 32] that focus exclusively on light-weight schemes (e.g., dictionary coding) because they found that the decompression performance on heavy-weight schemes (using gzip) is so slow that the CPU cost of decompression outweighs the I/O savings of reading in fewer pages. In contrast, we find that efficient implementations of the Lempel-Ziv algorithm in C-Store on modern hardware (e.g., a 3 GHz Pentium IV with a 4-way striped RAID array) are, in fact, able to read and decompress data faster than pure uncompressed data can be read. Several researchers [59, 40] have noted other problems with heavy weight schemes: partial decompression is impossible (a page worth of data must be decompressed to access a single value), compressed pages have varying length, and compressed pages lead to poor utilization of the buffer pool due to having to decompress the data in memory (lighter weight schemes tend to operate at a finer granularity than a page and do not from these problems). Column-oriented architectures alleviate some of these concerns and we point out situations where heavy-weight schemes are well suited for compressing data. We also expect that with the increasing disparity in performance between CPU and memory/disk [27], heavy-weight schemes will become even more attractive.

In summary, in this chapter, we demonstrate several results related to compression in column-oriented database systems:

- We overview commonly used DBMS compression algorithms and show how they can be applied in column-store systems. We compare this traditional set of algorithms with compression algorithms especially suited for column-store systems.

- Through experiments, we explore the trade-offs between these algorithms, varying the characteristics of the data set and the query workload. We use results from these experiments to create a decision tree to aid the database designer to decide how to compress a particular column.

- We introduce an architecture for a query executor that allows for direct operation on compressed data while minimizing the complexity of adding new compression algorithms. We experimentally show the benefits of operating directly on compressed data.

It should be noted that the purpose of this work is not to propose fundamental new compression schemes. Many of the approaches that have been employed in this work have been investigated in isolation in the context of row-oriented databases, and all are known in the data compression literature. Only slight variations to these schemes are proposed. The purpose of this work is to explore the performance and architectural implications of integrating a wide range of compression schemes into a column-oriented database. Further, the focus is to use compression to maximize query performance, not to minimize storage sizes.

The chapter proceeds as follows. Section 4.2 surveys related work and contrasts our approach with other work on compression in database systems. Section 4.3 overviews and discusses the various column-oriented compression schemes. Section 4.4 discusses the architecture of our implemented query executor and how it is possible to reduce the complexity of adding new compression schemes to a column-oriented database system. Section 4.5 gives experimental results showing the improvements of these column-wise schemes over standard database compression schemes, and shows how data and query characteristics affect the choice of optimal scheme. We conclude in Section 4.6.

4.2   Related Work

While research in database compression has been around nearly as long as there has been research in databases [50, 61, 36], compression methods were not commonly used in DBMSs until the 1990s. This is perhaps because much of the early work concentrated on reducing the size of the stored data, and it was not until the 90s when researchers began to concentrate on how compression affects database performance [39, 47, 59, 40, 48]. This research observed that while compression does reduce I/O, if the CPU cost of compressing/decompressing the data outweighs this savings, then the overall performance of the database is reduced. As improvements in CPU speed continue to outpace improvements in memory and disk access [27], this trade-off becomes more favorable for compression. In order to keep CPU costs down, most papers focus on light-weight techniques (in the sense that they are not CPU-intensive) that result in sub-optimal compression but that have low CPU overhead so that performance is improved when taking into consideration all relevant costs.

One way that the CPU overhead of compression has been reduced over the years is by integrating knowledge about compression into the query executor and allowing some amount of operation directly on compressed data. In early databases, data would be compressed on disk and then eagerly decompressed upon being read into memory. This had the disadvantage that everything read into memory had to be decompressed whether or not it was actually used. Graefe and Shapiro [39] (and later Goldstein et. al. [40], and Westmann et. al [67], and in the context of column-oriented DBMSs, MonetDB/X100 [77]) cite the virtues of lazy decompression, where data is compressed on the attribute level and held compressed in memory, and data is decompressed only if needed to be operated on. This has the advantage that some operations such as a hybrid hash join see improved performance by being able to keep a higher percentage of the table in memory, reducing the number of spills to disk. Chen et. al. [32] note that some operators can decompress transiently, decompressing to perform operations such as applying a predicate, but keeping a copy of the compressed data and returning the compressed data if the predicate succeeds.

The idea of decreasing CPU costs by operating directly on compressed data was introduced by Graefe and Shapiro [39]. They pointed out that exact-match comparisons and natural joins can be performed directly on compressed data if the constant portion of the predicate is compressed in the same way as the data. Also exact-match index lookups are possible on compressed data if an order-preserving (consistent) compression scheme is used. Further, projection and duplicate elimination can be performed on compressed data. However, this is the extent of research on direct operation on compressed data. In particular, to the best of our knowledge, there has been no attempt to take advantage of some compression algorithms' ability to represent multiple values in a single field to simultaneously apply an operation on these many values at once. In essence, previous work has viewed each tuple as compressed or uncompressed, and when operations cannot simply compare compressed values, they must be performed on decompressed tuples. Our work shows that column-oriented compression schemes provide further opportunity for direct operation on compressed data.

Our work also introduces a novel architecture for passing compressed data between operators that minimizes operator code complexity while maximizing opportunities for direct operation on compressed data. Previous work [40, 67, 32] also stresses the importance of insulating the higher levels of the DBMS code from the details of the compression technique. In general, this is accomplished by decompressing the data before it reaches the operators (unless dictionary compression is used and the data can be processed directly). However, in some cases increased performance can be obtained in query processing if operators can operate directly on compressed data (beyond simple dictionary schemes) and our work is the first to propose a solution to profit from these potential optimizations while keeping the higher levels of the DBMS as insulated as possible.

In summary, in this dissertation (Chapter 4) we revisit much of this related work on compression in the context of column-oriented database systems and we differ from other work on compression in column-oriented DBMSs (Zukowski et. al [77] on MonetDB/X100) in that we focus on column-oriented compression algorithms and direct operation on compressed data (whereas [77] focuses on improving CPU/cache performance of standard row-based light-weight techniques).

4.3   Compression Schemes

In this section we briefly describe the compression schemes that we implemented and experimented with in C-Store. For each scheme, we first give a brief description of the traditional version of the scheme as previously used in row store systems (and cite papers that provide more detail when possible). We then describe how the algorithm is used in the context of column-oriented databases.

4.3.1   Null Suppression

There are many variations on the null compression technique (see [60, 67] for some examples), but the fundamental idea is that consecutive zeros or blanks in the data are deleted and replaced with a description of how many there were and where they existed. Generally, this technique performs well on data sets where zeros or blanks appear frequently. We chose to implement a column-oriented version of the scheme described in [67]. Specifically, we allow field sizes to be variable and encode the number of bytes needed to store each field in a field prefix. This allows us to omit leading nulls needed to pad the data to a fixed size. For example, for integer types, rather than using the full 4 bytes to store the integer, we encoded the exact number of bytes needed using two bits (1, 2, 3, or 4 bytes) and placed these two bits before the integer. To stay byte-aligned (see Section 4.3.2 for a discussion on why we do this), we combined these bits with the bits for three other integers (to make a full byte's worth of length information) and used a table to decode this length quickly as in [67].

4.3.2   Dictionary Encoding

Dictionary compression schemes are perhaps the most prevalent compression schemes found in databases today. These schemes replace frequent patterns with smaller codes for them. One example of such a scheme is the color-mapping given in the introduction. Other examples can be found in [39, 60, 32, 77].

   We implemented a column-optimized version of dictionary encoding. All of the row-oriented dictionary schemes cited above have the limitation that they can only map attribute values from a single tuple to dictionary entries. This is because row-stores fundamentally are incapable of mixing attributes from more than one tuple in a single entry if other attributes of the tuples are not also included in the same entry (by definition of "row-store" – this statement does not hold for PAX-like [21] techniques that columnize blocks).

   Our dictionary encoding algorithm first calculates the number of bits, X, needed to encode a single attribute of the column (which can be calculated directly from the number of unique values of the attribute). It then calculates how many of these X-bit encoded values can fit in 1, 2, 3, or 4 bytes. For example, if an attribute has 32 values, it can be encoded in 5 bits, so 1 of these values can fit in 1 byte, 3 in 2 bytes, 4 in 3 bytes, or 6 in 4 bytes. We choose one of these four options using the algorithm described in the next sub-section. Suppose that the 3-value/2-byte option was chosen. In that case, a mapping is created between every possible set of 3 5-bit values and the original 3 values. For example, if the value 1 is encoded by the 5 bits: 00000; the value 25 is encoded by the 5 bits: 00001; and the value 31 is encoded by the 5 bits 00010; then the dictionary would have the entry (read entries right-to-left)

```
X000000000100010 -> 31 25 1
```

where the X indicates an unused "wasted" bit. The decoding algorithm for this example is then straight-forward: read in 2-bytes and lookup entry in dictionary to get 3 values back at once. Our decision to keep data byte-aligned might be considered surprising in light of recent work that has shown that bit-shifting in the processor is relatively cheap. However our experiments show that column stores are so I/O efficient that even a small amount of compression is enough to make queries on that column become CPU-limited (Zukowski et. al observe a similar result [77]) so the I/O savings one obtains by not wasting the extra space are not important. Thus, we have found that it is worth byte-aligning dictionary entries to obtain even modest CPU savings.

Cache-Conscious Optimization

The decision as to whether values should be packed into 1, 2, 3, or 4 bytes is decided by requiring the dictionary to fit in the L2 cache. In the above example, we fit each entry into 2 bytes and the number of dictionary entries is $32^3 = 32768$. Therefore the size of the dictionary is 393216 bytes which is less than half of the L2 cache on our machine (1MB). Note that for cache sizes on current architectures, the 1 or 2 byte options will be used exclusively.

Parsing Into Single Values

Another convenient feature of this scheme is that it degrades gracefully into a single-entry per attribute scheme which is useful for operating directly on compressed data. For example, instead of decoding a 16-bit entry in the above example into the 3 original values, one could instead apply 3 masks (and corresponding bit-shifts) to get the three single attribute dictionary values. For example:

```
(X000000000100010 & 0000000000011111) >> 0  = 00010
(X000000000100010 & 0000001111100000) >> 5  = 00001
(X000000000100010 & 0111110000000000) >> 10 = 00000
```

These dictionary values in many cases can be operated on directly (as described in Section 4.4) and lazily decompressed at the top of the query-plan tree.

We chose not to use an order preserving dictionary encoding scheme such as ALM [24] or ZIL [73] since these schemes typically have variable-length dictionary entries and we prefer the performance advantages of having fixed length dictionary entries.

4.3.3   Run-length Encoding

Run-length encoding compresses runs of the same value in a column to a compact singular representation. Thus, it is well-suited for columns that are sorted or that have reasonable-sized runs of the same value. These runs are replaced with triples: (value, start position, run_length) where each element of the triple is given a fixed number of bits.

When used in row-oriented systems, RLE is only used for large string attributes that have many blanks or repeated characters. But RLE can be much more widely used in column-oriented systems where attributes are stored consecutively and runs of the same value are common (especially in columns that have few distinct values). As described in Chapter 3, the C-Store architecture results in a high percentage of columns being sorted (or secondarily sorted) and thus provides many opportunities for RLE-type encoding.

4.3.4   Bit-Vector Encoding

Bit-vector encoding is most useful when columns have a limited number of possible data values (such as states in the US, or flag columns). In this type of encoding, a bit-string is associated with each value with a '1' in the corresponding position if that value appeared at that position and a '0' otherwise. For example, the following data:

1 1 3 2 2 3 1

would be represented as three bit-strings:

bit-string for value 1: 1100001
bit-string for value 2: 0001100
bit-string for value 3: 0010010

Since an extended version of this scheme can be used to index row-stores (so-called bit-map indices [55]), there has been much work on further compressing these bit-maps and the implications of this further compression on

Properties	Iterator Access	Block Information
isOneValue()	getNext()	getSize()
isValueSorted()	asArray()	getStartValue()
isPosContig()		getEndPosition()

Table 4.1: Compressed Block API

query performance [52, 22, 48, 71, 70, 72, 23]; however, the most recent work in this area [71, 72] indicates that one needs the bit-maps to be fairly sparse (on the order of 1 bit in 1000) in order for query performance to not be hindered by this further compression, and since we only use this scheme when the column cardinality is low, our bit-maps are relatively dense and we choose not to perform further compression.

4.3.5 Heavyweight Compression Schemes

Lempel-Ziv Encoding. Lempel-Ziv ([74, 75]) compression is the most widely used technique for lossless file compression. This is the algorithm upon which the UNIX command gzip is based. Lempel-Ziv takes variable sized patterns and replaces them with fixed length codes. This is in contrast to Huffman encoding which produces variable sized codes. Lempel-Ziv encoding does not require knowledge about pattern frequencies in advance; it builds the pattern table dynamically as it encodes the data. The basic idea is to parse the input sequence into non-overlapping blocks of different lengths while constructing a dictionary of blocks seen thus far. Subsequent appearances of these blocks are replaced by a pointer to an earlier occurrence of the same block. We refer the reader to [74, 75] for more details.

For our experiments, we used a freely available version of the Lempel-Ziv algorithm [3] that is optimized for decompression performance (we found it to be much faster than UNIX gzip).

We experimented with several other heavyweight compression schemes, including Huffman and Arithmetic encoding, but found that their decompression costs were prohibitively expensive for use inside of a database system.

4.4 Compressed Query Execution

In this section we describe how we integrate the compression schemes discussed above into the C-Store query executor in a way that allows for direct operation on compressed data while minimizing the complexity of adding new compression algorithms to the system.

4.4.1 Query Executor Architecture

We extended C-Store to handle a variety of column compression techniques by adding two classes to the source code for each new compression technique. The first class encapsulates an intermediate representation for compressed data called a *compression block*. A compression block contains a buffer of the column data in compressed format and provides an API that allows the buffer to be accessed in several ways. Table 4.1 lists the salient methods of the compression block API. Note that compression blocks are not the same as the 64K storage blocks described in Chapter 4. A compression block can be quite small in its representation footprint (e.g., a single RLE triple), so a storage block can contain multiple compression blocks.

The methods listed in the properties column of Table 4.1 will be discussed in Section 4.4.2 and are a way for operators to facilitate operating directly on compressed data instead of having to decompress and iterate through it. For the cases where decompression cannot be avoided, there are two ways to iterate through block data. First is through repeated use of the getNext() method which will progress through the compressed buffer, transiently decompressing the next value and returning that value along with the position (a position is the ordinal offset of a value in a column) that the value was located at in the original column. Second is through the asArray() method which decompresses the entire buffer and returns a pointer to an array of data in the uncompressed column type.

The block information methods (see Table 4.1) return data that can be extracted from the compressed block without decompressing it. For example, for RLE, a block consists of a single RLE triple of the form (*value*, *start_pos*, *run_length*). getSize() returns *run_length*, getStartValue() returns *value*, and getEndPosition() returns (*start_pos* + *run_length* − 1). A more complex example is for bit-vector encoding: a block is a subset of the bit-vector associated with a single value. Thus, we call a bit-vector block a *non position-contiguous* block, since it contains a compressed representation of a set of (usually non-consecutive) positions for a single value. Here, getSize() returns the number of *on* bits in the bitstring, getStartValue() returns the value with which the bit-string is associated, and getEndPosition() returns the position of the last *on* bit in the bitstring.

The other class that is added to the source code for each new compression technique is a *DataSource* operator. A DataSource operator serves as the interface between the query plan and the storage manager and has compression specific knowledge about how pages for that compression technique are stored on disk and what indexes are available on that column. It thus is able to serve as a scan operator, reading in compressed pages from disk and converting them into the compressed blocks described above. For some heavy-weight compression schemes (e.g., LZ), the corresponding DataSource operator may simply decompress the data as it is read from disk, presenting uncompressed blocks to parent operators.

Selection predicates from the query can be pushed down into DataSources. For example, if an equality predicate is pushed down into a DataSource operator sitting on top of bit-vector encoded data, the operator performs a projection, returning only the bit-vector for the requested value. The selection thus becomes trivial. For an equality predicate on a dictionary encoded column, the DataSource converts the predicate value to its dictionary entry and does a direct comparison on dictionary data (without having to perform decompression). In other cases, selection simply evaluates the predicate as data is read from disk (avoiding decompression whenever possible).

4.4.2 Compression-Aware Optimizations

We will show in Section 4.5 that there are clear performance advantages to operating directly on compressed data, but these advantages come at a cost: query executor complexity. Every time a new compression scheme is added to the system, all operators that operate directly on this type of data have to be supplemented to handle the new scheme. Without careful engineering, there would end up being *n* versions of each operator – one for each type of compression scheme that can be input to the operator. Operators that take two inputs (like joins) would need n^2 versions. This clearly causes the code to become very complex very quickly.

To illustrate this, we study a nested loops join operator. We note that joins in column-oriented DBMSs can look different from joins in row-oriented DBMSs. In C-Store, if columns have already been stitched together into row-store tuples, joins work identically as in row-store systems. However, joins can alternatively receive as input only the columns needed to evaluate the join predicate. The output of the join is then set of pairs of positions in the input columns for which the predicate succeeded. For example, the figure below shows the results of a join of a column of size 5 with a column of size 3. The positions that are output can then be sent to other columns from the input relations (since only the columns in the join predicate were sent to the join) to extract the values at these positions.

$$\begin{array}{c}
\boxed{42} \\
\boxed{36} \\
\boxed{42} \\
\boxed{44} \\
\boxed{38}
\end{array}
\quad \bowtie \quad
\begin{array}{c}
\boxed{38} \\
\boxed{42} \\
\boxed{46}
\end{array}
\quad = \quad
\begin{array}{cc}
\boxed{1} & \boxed{2} \\
\boxed{3} & \boxed{2} \\
\boxed{5} & \boxed{1}
\end{array}$$
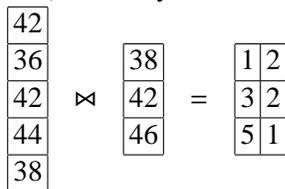
An outline for the code for this operator is shown Figure 4-1 (assume that the join predicate is an equality predicate on one attribute from each relation).

The pseudocode shows the join operator making some optimizations if the input columns are compressed. If one of the input columns is RLE and the other is uncompressed, the resulting position columns of the join can be expressed directly in RLE. This reduces the number of necessary operations by a factor of k, where k is the run-length of the RLE triple whose value matches a value from the uncompressed column. If one of the input columns is bit-vector encoded, then the resulting column of positions for the unencoded column can be represented using

```
NLJOIN(PREDICATE q, COLUMN c1, COLUMN c2)
  IF c1 IS NOT COMPRESSED AND c2 IS NOT COMPRESSED
    FOR EACH VALUE valc1 WITH POSITION i IN c1 DO
      FOR EACH VALUE valc2 WITH POSITION j IN c2 DO
        IF q(valc1,valc2) THEN OUTPUT-LEFT: (i), OUTPUT-RIGHT: (j)
      END
    END
  IF c1 IS NOT COMPRESSED AND c2 IS RLE COMPRESSED
    FOR EACH VALUE valc1 WITH POSITION i IN c1 DO
      FOR EACH TRIPLE t WITH VAL v,STARTPOS j AND RUNLEN k IN c2
        IF q(valc1,v) THEN:
          OUTPUT-LEFT: NEW RLE TRIPLE (NULL,i,k),
          OUTPUT-RIGHT: (j ... j+k-1)
      END
    END
  IF c1 IS NOT COMPRESSED AND c2 IS BIT-VECTOR COMPRESSED
    FOR EACH VALUE valc1 WITH POSITION i IN c1 DO
      FOR EACH VALUE valc2 WITH BITSTRING b IN c2 DO
        //ASSUME THAT THERE ARE num '1'S IN b
        IF q(valc1,valc2) THEN OUTPUT
          OUTPUT-LEFT: NEW RLE TRIPLE (NULL,i,num),
          OUTPUT-RIGHT: b
      END
    END
  ETC. ETC. FOR EVERY POSSIBLE COMBINATION OF ENCODING TYPES
```

Figure 4-1: Pseudocode for NLJoin

RLE encoding and the resulting column of positions for the bit-vector column can be copied from the appropriate bit-vector for the value that matched the predicate. Again, this reduces the number of necessary operations by a large factor.

So while many optimizations are possible if operators are allowed to work directly on compressed data, the example shows that the code becomes complex fairly quickly, since an if statement and an appropriate block of code is needed for each possible combination of compression types.

We alleviate this complexity by abstracting away the properties of compressed data that allow the operators to perform optimizations when processing. In the example above, the operator was able to optimize processing because the compression schemes encoded multiple positions for the same value (e.g., RLE indicated multiple consecutive positions for the same value and bit-vector encoding indicated multiple non-consecutive positions for the same value). This knowledge allowed the operator to directly output the join result for multiple tuples without having to actually perform the execution more than once. The operator simply forwarded on the positions for each copy of the joining values rather than dealing with each record independently.

Hence, we enhanced each compression block with methods that make it possible for operators to determine the properties of each block of data. The properties we have added thus far are shown in the Properties column of Table 4.1. isOneValue() returns whether or not the block contains just one value (and many positions for that value). isValueSorted() returns whether or not the block's values are sorted (blocks with one value are trivially sorted). isPosContig() returns whether the block contains a consecutive subset of a column (i.e. for a given position range within a column, the block contains all values located in that range). Properties are usually fixed for each compression scheme but could in principle be set on a per-block basis by the DataSource operator.

The table below gives the value of these properties for various encoding schemes. Note that there are many variations of each scheme. For example, we experimented with three versions of dictionary encoding before settling on the one described in this chapter; in one version there was a single dictionary entry per row value – i.e., a standard

54

row-based dictionary scheme; another version was a pure column-based scheme that did not gracefully degenerate into single values as in the current scheme. In most cases, each variation of the same scheme will have the same block properties in the table below. A no/yes entry in the table indicates that the compression scheme is agnostic to the property and the value is determined by the data.

Encoding Type	Sorted?	1 value?	Pos. contig.?
RLE	yes	yes	yes
Bit-string	yes	yes	no
Null Supp.	no/yes	no	yes
Lempel-Ziv	no/yes	no	yes
Dictionary	no/yes	no	yes
Uncompressed	no/yes	no	no/yes

When an operator cannot operate on compressed data (if, for example, it cannot make any optimizations based on the block properties), it repeatedly accesses the block through an iterator, as described in Section 4.4.1. If, however, the operator can operate on compressed data, it can use the block information methods described in Section 4.4.1 to take shortcuts in operation. For example, the pseudocode for a Count aggregator is shown in Figure 4-2. Here, the passed in column is used for grouping (e.g., in a query of the form SELECT c1, COUNT(*) FROM t GROUP BY c1). (Note: this code is simplified from the actual aggregation code for ease of exposition).

```
COUNT(COLUMN c1)
 b = GET NEXT COMPRESSED BLOCK FROM c1
 WHILE b IS NOT NULL
   IF b.ISONEVALUE()
     x = FETCH CURRENT COUNT FOR b.GETSTARTVAL()
     x = x + b.GETSIZE()
   ELSE
     a = b.ASARRAY()
     FOR EACH ELEMENT i IN a
       x = FETCH CURRENT COUNT FOR i
       x = x + 1
   b = GET NEXT COMPRESSED BLOCK FROM c1
```

Figure 4-2: Pseudocode for Simple Count Aggregation

Note that despite RLE and bit-vector encoding being very different compression techniques, the pseudocode in Figure 4-2 need not distinguish between them, pushing the complexity of calculating the block size into the compressed block code. In both cases, the size of the block can be calculated without block decompression.

Figure 4-3 gives some more examples of how join and generalized aggregation operators can take advantage of operating on compressed data given block properties.

In summary, by using compressed blocks as an intermediate representation of data, operators can operate directly on compressed data whenever possible, and can degenerate to a lazy decompression scheme when this is impossible (by iterating through block values). Further, by abstracting general properties about compression techniques and having operators check these properties rather than hardcoding knowledge of a specific compression algorithm, operators are shielded from needing knowledge about the way data is encoded. They simply have to condition for these basic properties of the blocks of data they receive as input. We have found that this architecture significantly reduces the query executor complexity while still allowing direct operation on compressed data whenever possible.

Property	Optimization
One value, Contiguous Positions	**Aggregation**: If both the group-by and aggregate input blocks are of this type, then the aggregate input block can be aggregated with one operation (e.g. if size was 8 and aggregation was sum, result is 8*value) **Join**: Perform optimization shown in the second if statement in Figure 4-1 (works in general, not just for RLE).
One value, Pos. Non-contiguous	**Join**: Perform optimization shown in the third if statement in Figure 4-1 (works in general, not just for bit-vector compression).
One value	**Aggregation Group-By clause**: The position list of the value can be used to probe the data source for the aggregate column so that only values relevant to the group by clause are read in
Sorted	**Max or Min Aggregation**: Finding the maximum or minimum value in a sorted block is a single operation **Join** Finding a value within a block can be done via binary search.

Figure 4-3: Optimizations on Compressed Data

4.5 Experimental Results

We ran experiments on our extended version of the C-Store system with two primary goals. First, we wanted to identify situations in which the encoding types described in Section 4.3 perform well. Second we wanted to demonstrate the benefits of operating directly on compressed data.

Our benchmarking system is a 3.0 GHz Pentium IV, running RedHat Linux, with 2 Gbytes of memory, 1MB L2 cache, and 750 Gbytes of disk. The disk can read cold data at 50-60MB/sec. We used a combination of synthetically generated and TPC-H data. For the experiments where we used TPC-H data, we used columns from the lineitem fact table at scale 10 which consists of just under 60,000,000 lineitems.

We begin by presenting results from a simple aggregation query on a single column of data encoded with each of the six encoding schemes described in Section 4.3. We used generated data so that we could carefully vary the data characteristics. We ran three variations of this experiment. In the first variation, we required the column to be decompressed as it was brought off disk. In the second variation, we lazily decompressed the data and allowed operators to apply optimizations to compressed data. In the third variation, queries ran with competition for CPU cycles. In these experiments, we observe that the number of distinct values and sorted run lengths are the primary determinant of query performance; we use these metrics to predict performance on TPC-H data.

We also present results from more complicated queries to further illustrate the benefits of different compression schemes and the interaction of these schemes with each other in multi-column queries. Section 4.6 summarizes our results.

4.5.1 Eager Decompression

In this experiment, we ran a simple aggregation on a single column of data encoded with each of the six encoding schemes described in Section 4.3. We ran on generated data and required that the column be decompressed as it was brought off disk. The query that we ran was simply:

```
SELECT SUM(C)
FROM TABLE
GROUP BY C
```

The column that we are aggregating has 100 million 32-bit integer values. Since most columns in C-Store projections have some kind of order (see Chapter 3), we assume sorted runs of size X (we vary X). For example, if column C is tertiarily sorted and the first column in the projection has 500 unique values and the second column in the projection has 1000 unique values then C will have average sorted runs of size 100000000/(500*1000)=200. If C itself has 10 unique values, then within each of these sorted runs, each value would appear 20 times. Since bit-vector compression is only designed to be able to run on columns with few distinct values, in our first set of experiments, we allowed
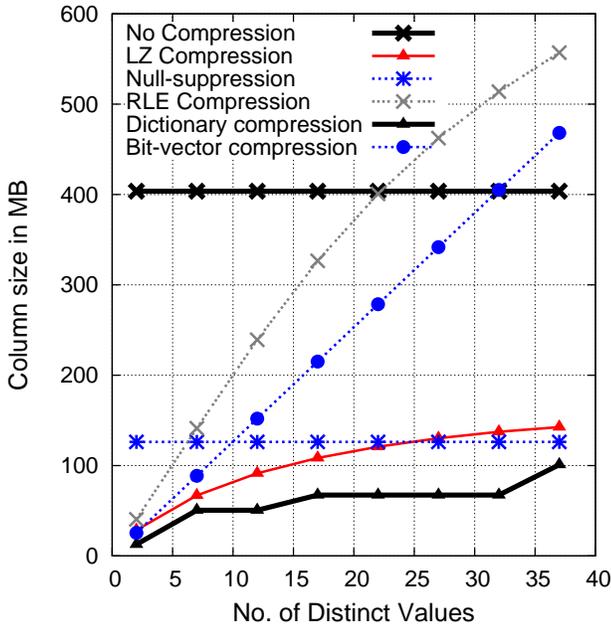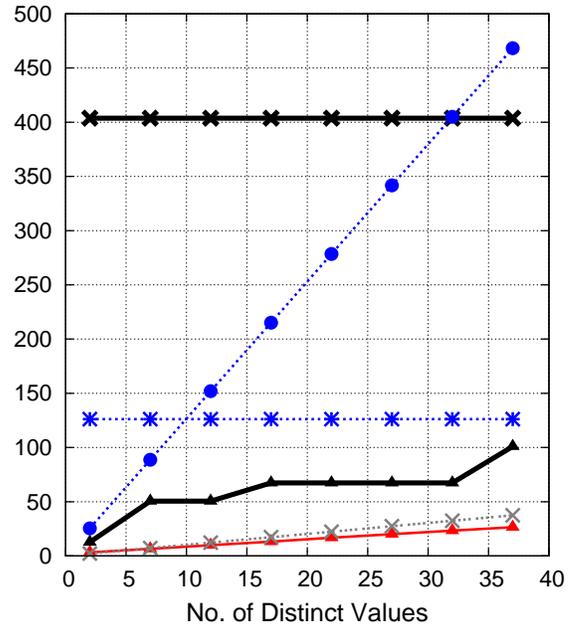
Figure 4-4: Compressed column sizes for varied compression schemes on column with sorted runs of size 50 (a) and 1000 (b)

the number of distinct values in C to vary between 2 and 40 (so that we could directly compare all the introduced compression techniques). Also, in most data-warehousing environments, there are a large number of columns with few distinct values; for example, in the TPC-H lineitem fact table, 25% of the columns have fewer than 50 distinct values. We experiment with columns with a higher number of distinct values in Section 4.5.3.

We experimented with four sorted run lengths in C: 50, 100, 500, and 1000. We compressed the data in each of the following six ways: Null suppression, Lempel-Ziv, RLE, bit-vector, dictionary, and no compression. The sizes of the compressed columns are shown in Figures 4-4(a) and 4-4(b) for different cardinalities of C (here, we use *cardinality* to mean the number of distinct values). We omit the plots for the 100 and 500 sorted runs cases as they follow the trends observed in Figure 4-4. In these experiments, dictionary and LZ compression consistently get the highest compression ratios. Dictionary does a slightly better job compressing the data than the heavy-weight LZ scheme at low column cardinalities since our implementation of LZ will occasionally leave some empty space at the end of a page if it gets compressed more than surrounding pages

RLE also performs well for low-cardinalities since it performs better with increasing length of runs of repeated values. The average run-length of a point on these graphs can be calculated directly by dividing the sorted run-length by the number of unique values; consequently RLE starts to take up more space than the uncompressed data as the number of distinct values approaches 25 since the average run-length approaches $(50/25 =)$ 2 and a single RLE triple takes up a little more space than two uncompressed values. The compression ratio for bit-vector is linear in the number of unique values in the column. Since we do not further compress the bit-vectors, as soon as the column cardinality is more than 32, type-2 compression is no longer more compressed than the original 32-bit data.

The performance of the aggregation query on these same compressed columns is shown in Figures 4-5(a) and 4-5(b) (again we do not show the plots for sorted runs of 100 and 500 since they follow the trends between these two graphs).
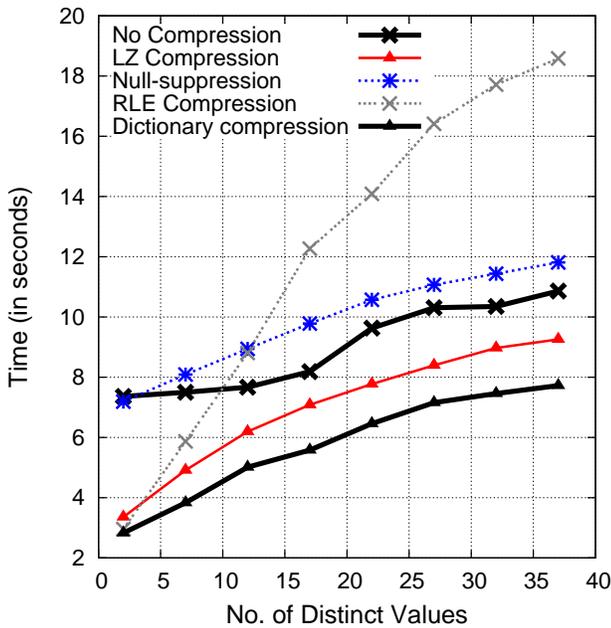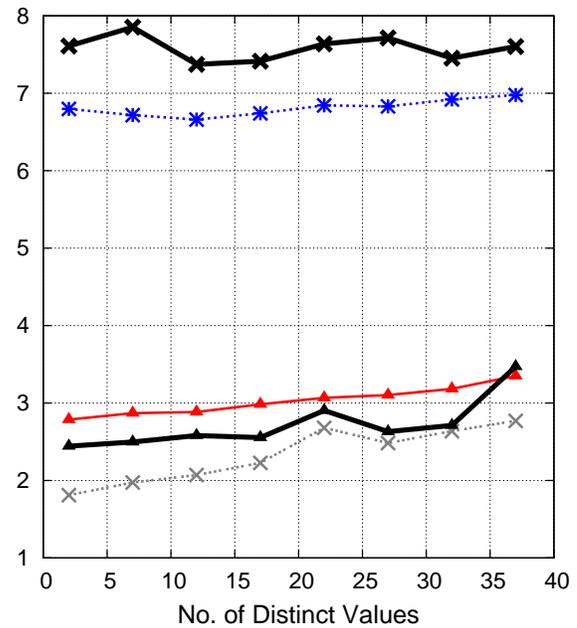
Figure 4-5: Query Performance With Eager Decompression on column with sorted runs of size 50 (a) and 1000 (b)

Not surprisingly, these results show that the size of the compressed column on disk is not a good indicator of query performance. This is most apparent for bit-vector compression which took from 35 to 120 seconds – an order of magnitude slower than the uncompressed line despite taking half the space on average – such that we could not show it on the same graph as the other schemes. Decompression costs are so significant because C-Store is not I/O bound on this query (since it does completely sequential I/O) so decompression costs dominate performance rather than (relatively) small differences in the compression ratio.

Bit-vector encoding was by far the slowest decompression scheme. To completely decompress a bit-vector encoded column, one must read in parallel and merge each bit-vector (one for each distinct value in the column). RLE and NS performed worse than dictionary and LZ (though RLE performed better as the average run-length of the column improved). This can be attributed to the fact that RLE and NS require if-then-else statements in the decompression code which makes loop pipelining difficult and results in code that does not take advantage of the super-scalar properties of modern CPUs (this was also observed in Monet DB/X100 [77]).

The uncompressed line in Figure 4-5(a) does not remain constant since an increased number of distinct values results in smaller runs of repeats of the same value, and since the aggregation code only has to do a hash look-up on the current value if the current value is different from the previous value, all compression schemes benefit from longer runs. Since CPU is not completely overlapped with I/O, this increased CPU cost is reflected in increased query time. However, the runs are sufficiently long in Figure 4-5(b) that this CPU effect is not observed as the query becomes I/O limited for the uncompressed data.

4.5.2 Operating Directly on Compressed Data

We ran the same experiments as in the previous section, without eager decompression. Operators were allowed to operate directly on compressed data. Since LZ and NS cannot operate on encoded data, their performance for these

experiments was identical (and we omit them from some of our graphs). However, since there are two alternative ways for operating directly on our dictionary compression scheme for this aggregation query, there are two lines on each graph corresponding to dictionary compression. The first approach, called *dictionary single-value*, simply extracts each individual dictionary symbol from a 32-bit dictionary-compressed record (as described in section 4.3.2), performs a count group-by aggregation on these symbols, then decompresses each symbol to its original value and multiplies this original value by the counts to get a sum. For example, if value 2 maps to symbol 000, value 4 maps to 001, and value 8 maps to 002 and the aggregator receives the following input:

`001, 001, 000, 001, 002, 002`

Then the aggregator would count the number of instances of each dictionary entry:

`001: 3; 000: 1; 002: 2`

and would then decode the symbols their original values and compute the sum to produce the output `12, 2, 16`.

The second approach, called *dictionary multi-value*, does the same thing except that it groups entire multi-value dictionary entries before decompressing them, combining counts for all entries containing a particular value, and multiplying these counts with each decompressed value in the dictionary entry. We separate these two schemes since *dictionary single-value* can be easily used for all aggregations but the *dictionary multi-value* shortcut can only be used well in group-by-self queries (where the group-by and aggregation clauses are on the same column, e.g. count(*)).

In addition to the dictionary optimizations, the aggregator also operates directly on RLE and bit-vector data as described in Section 4.4.2. The results are shown in Figures 4-6(a) and 4-6(b) and compared to the corresponding graphs in Figures 4-5(a) and 4-5(b) in Figure 4-6(c). We see that substantial performance gains are possible when data is not eagerly decompressed. The biggest gain is observed for bit-vector compression since it is so slow to decompress. RLE sees a bigger performance gain for longer runs since query time is directly a function of the number of triples in the column, and longer runs results in fewer triples. Dictionary encoding (without the multi-value optimization) does not see much benefit from operating directly on compressed data since dictionary decompression involves a single bit-shift and dictionary lookup per value, which, if it is results in a cache hit, is very fast. However, once the multi-value optimization is taken into account, performance improves significantly since, like RLE, multiple values are summarized in a single compression symbol, and the number of times the aggregation operator must be called is reduced.

Without operation on compressed data, compression becomes a tradeoff: the system must pay the an extra CPU cost to decompress data in exchange for the I/O savings of reading less data from storage. While in many cases this is a good tradeoff to make, in some situations the system might have a CPU intensive query running or might otherwise be CPU limited. In these situations, operating directly on compressed data becomes even more important, since it does not incur extra CPU cost (and in some cases reduces it). To demonstrate this point, we reran the same experiments with contention for CPU cycles (this was done by running C-Store at the same time as another process that infinitely accessed, processed, and wrote data to a large array). The bar graph in Figure 4-6(d) shows the average increase in query time caused by CPU contention compared with the results in Figures 4-6(a) and (b) for each compression technique. We reran the experiment with performance counters to find out whether the contention was for CPU cycles or for cache lines and found that the competition for cache lines accounted for less than 2% of the increase in query time. Thus contention for CPU cycles is the dominant reason for the increase in query time, and queries that were CPU limited take longer.

NS and LZ perform the worst (relative to their previous values) since the aggregator does not operate directly on this data. Similarly for RLE (for small average run lengths) and the value-at-a-time dictionary scheme (although the dictionary data does not need to be completely decompressed, the aggregator must still iterate through all values and dictionary entries must be bit-shifted into integers). However, for the schemes on which the aggregator can take short-cuts, the performance hit of CPU contention is significantly smaller. This is because the column-oriented nature of these schemes allow the aggregator to aggregate multiple values at once; the CPU cost of the aggregation is proportional to n, where n is *num_tuples* for the row-oriented schemes, but only *num_tuples/avg_run_len* for RLE,
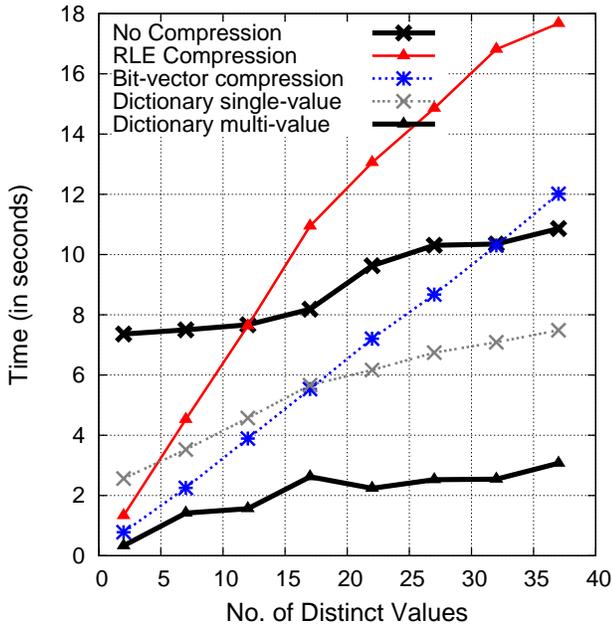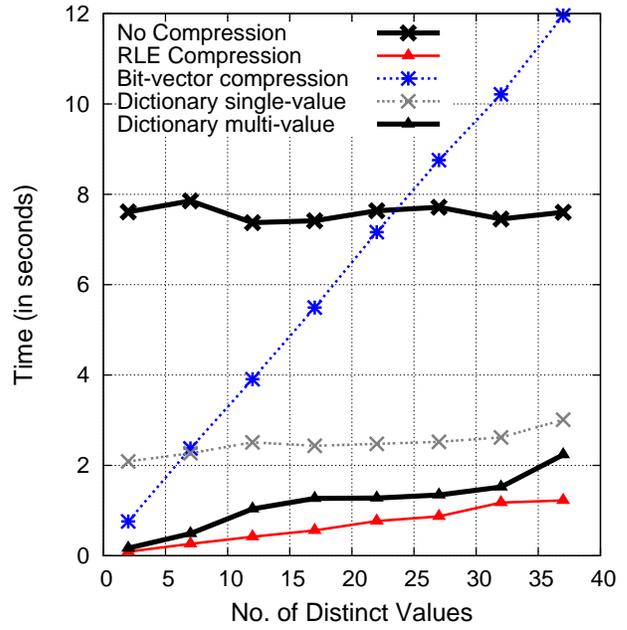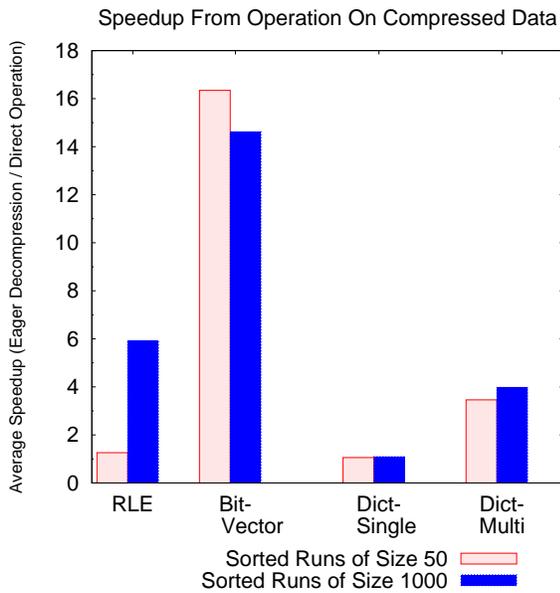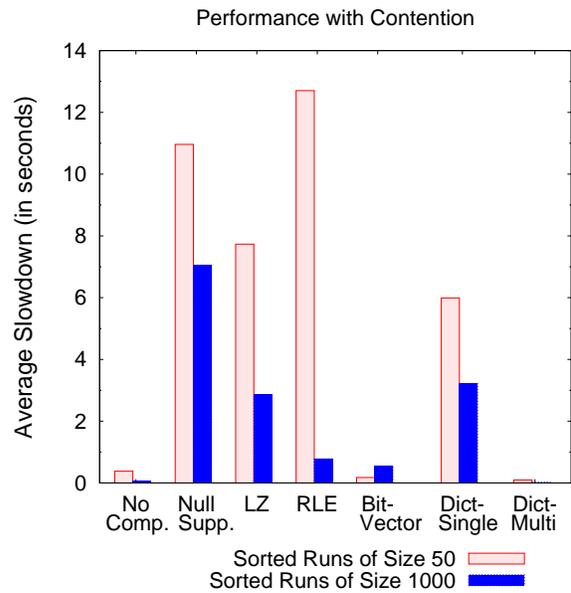
Figure 4-6: Query performance with direct operation on compressed data on column with sorted runs of size 50 (a) and 1000 (b). Figure (c) shows the average speedup of each line in the above graphs relative to the same line in the eager decompression graphs where direction operation on compressed data is not used. Figure (d) shows the average increase in query time relative to the query times in (a) and (b) when contention for CPU cycles is introduced.
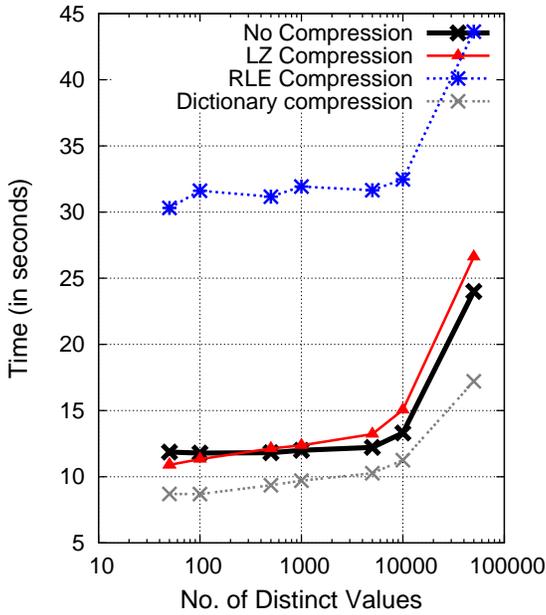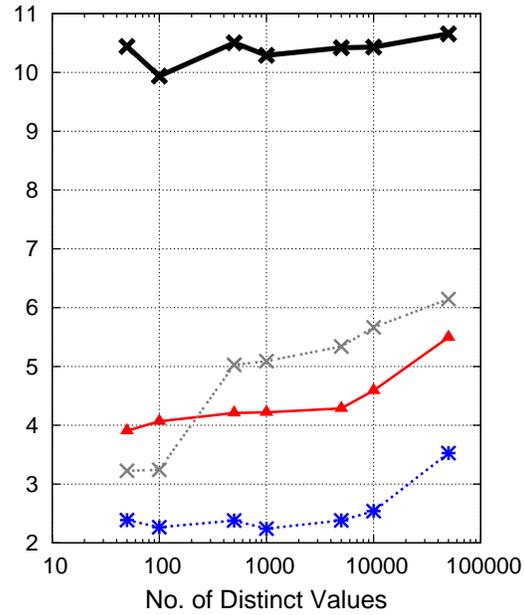
Figure 4-7: Aggregation Query on High Cardinality Data with Avg. Run Lengths of 1 (a) and 14 (b)

num_tuples/dict_entry_size for dictionary multi-value, and *num_distinct_values* for bit-vector encoding. Thus while normal compression simply trades "expensive" I/O time for "cheap" CPU, operating directly on compressed data reduces *both* I/O *and* CPU cycles. This suggests that even on a machine with a much faster I/O or a much slower CPU, compressing data and operating directly on it will be beneficial.

4.5.3 Higher column cardinalities

We now present some results for experiments with higher cardinality data. For these experiments we generated data from a uniform distribution (such that a value is equally likely to appear at any location independently of what values surround that tuple). We only experimented with RLE, LZ, dictionary, and no compression for these experiments since NS and bit-vector encoding perform poorly at higher cardinalities. Figure 4-7(a) shows the results of running the same aggregation query on this higher cardinality data, and Figure 4-7(b) shows the same experiment on the same distribution of data; however each tuple appears 14 times in a row (to increase the average run-length). Operators are allowed to operate directly on compressed data. Note that at high cardinalities (> 10000 values) the aggregation hash table no longer fits in cache, causing a discontinuous increase in query time.

These graphs show that schemes which take advantage of data locality (like RLE and LZ) perform poorly on random data but do well as soon as run-lengths are introduced, even with high data cardinalities. Dictionary encoding performs comparatively well on data with less locality.

The following table compares results from the previous experiments to summarize how data characteristics affect aggregate query performance on various compression types (times are in seconds). The best performing schemes are shown in bold. We show data with and without runs and with high and low cardinalities, since these properties appear to have the biggest effect on the performance of our compression schemes. For high and low cardinality rows, the number of distinct values was 10,000 and 37 respectively. For data with "Runs" we chose an average run-length of 14.

61

Data	RLE	LZ	Dictionary	Bit-Vector	No Comp.
No runs, low card.	17.67	9.30	**7.49**	12.02	10.86
Runs, low card.	**2.43**	3.93	3.29	9.83	7.59
No runs, high card.	32.48	15.05	**11.25**	N/A	13.31
Runs, high card.	**2.56**	4.48	4.56	N/A	9.52

This table shows that for RLE and LZ, run-length is a better indicator of performance than cardinality. As soon as the data has moderate sized runs, performance improves dramatically. This correlation between run-length and performance is less significant for the latter three techniques. As explained in Section 4.5.1, all techniques see some improvement with longer run-lengths.

4.5.4   Generated vs. TPC-H Data

To verify that our results on our generated data set match the results on more general data sets, we compared our query performance on our generated data to query performance on TPC-H data. For this set of experiments, we used the shipdate, supplier key, extended price, linenumber, quantity, extended price, and return flag columns from the TPC-H lineitem fact table and created the following projections:

```
(shipdate, retflag, quantity)  [314]
(price, retflag) [15]
(suppkey, linenumber)  [86]
(suppkey, retflag)  [200]
(shipdate, quantity)  [475]
```

Each projection was sorted from left to right (e.g., the first projection was primarily sorted on shipdate, secondarily sorted on retflag, and tertiarily sorted on quantity). This sorting resulted in varying average run-lengths of the right-most column (in brackets above). We then performed the same aggregation query as in the previous experiments on the final column of each of these six projections. Since the previous experiments showed that average run-length is a reasonable predictor of query performance for each compression scheme except bit-vector and dictionary, we took 10 columns from the previous set of experiments with similar run-lengths and compared query performance with the TPC-H columns (where average run-length is shown on the X axis). Since the scale 10 TPC-H data was 40% smaller than our generated data, we ran the query on the first 60% of the data in the generated data columns. The results are shown in Figure 4-8. As expected, run-length is a good predictor of query performance for the RLE, LZ, and null-suppression compression schemes.

4.5.5   Other Query Types

In this section we experiment with different types of queries to observe how compressing one column affects access to other columns in a query and also to observe further advantages of operating directly on compressed data. We focus on simple selection and join queries in this section; in Chapter 7 we will explore more complicated queries in the context of a complete data warehousing benchmark.

The first query we experimented with was a simple selection query (with an aggregation on top so that outputting query results wouldn't play a significant part in query time):

```
SELECT COL1, COUNT(*)
FROM CSTORE_PROJ1
WHERE PREDICATE(COL2)
GROUP BY COL1
```

Queries of this type are done in C-Store using *position filters* that work as follows. First, a predicate is applied to a column by sending it to the DataSource for that column. The DataSource produces a list of positions in that column
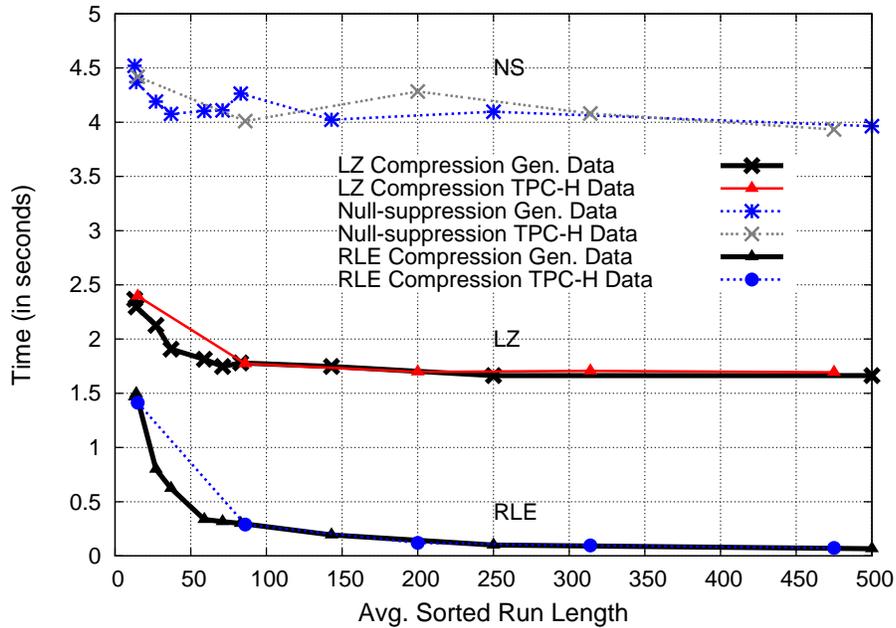
Figure 4-8: Comparison of query performance on TPC-H and generated data

for which that predicate succeeded. This list of positions can be represented as a compressed list or bit-string. This position list is then ANDed (or ORed) together with position lists from other applied predicates and the results are sent to the DataSources for all columns that are used by parent operators (e.g., all columns in the select clause of the query) to extract values. We refer to this action as *position filtering*. In the query above, the Count Aggregator consumes values from COL1 which are produced according to a position filter sent from COL2.

For this experiment, we used TPC-H data (scale 10 lineitem table). COL2 was the quantity column (the predicate was quantity == 1) and was compressed using RLE, bit-vector, dictionary compression, or with no compression. We experimented with COL1 being the suppkey, shipdate, linenumber, and returnflag columns from the same lineitem table. We use a projection that is sorted by COL1 and secondarily sorted by COL2. COL1 is therefore RLE compressed (this is usually the best option for sorted data). Figure 4-9(a) shows the results of running this query. The X axis represents the average run-length of the COL2 (l_quantity) column which varies according to the column we used for COL1.

Once again, operating directly on compressed data provides a substantial performance gain. Bit-vector encoding is very fast because it is already storing the result of the predicate as it already contains a *position list* for each unique value in the column. So applying the predicate amounts to simply producing the position list for the appropriate value. Additionally, the COL1 (RLE in this case) DataSource can take shortcuts based on the format of the position list that it receives. In this example, it is receiving a bit-vector (a non-position-contiguous list). Since COL1 contains a list of single-value, position contiguous triples, it is straightforward to take the intersection of these position contiguous triples with the non-position contiguous position blocks (by only looking at the start and end position of each triple and position block) and converting RLE blocks into bit-vector blocks. Most of the code for doing this is inside the bit-vector position block.

In the next experiment we ran the same query; however, we switched the role of the two columns in the query. So now the predicate is on COL1 and we position filter COL2 (which is again encoded using the same four compression techniques as in the previous query). The results of this experiment are shown in Figure 4-9(b). Bit-vector performs
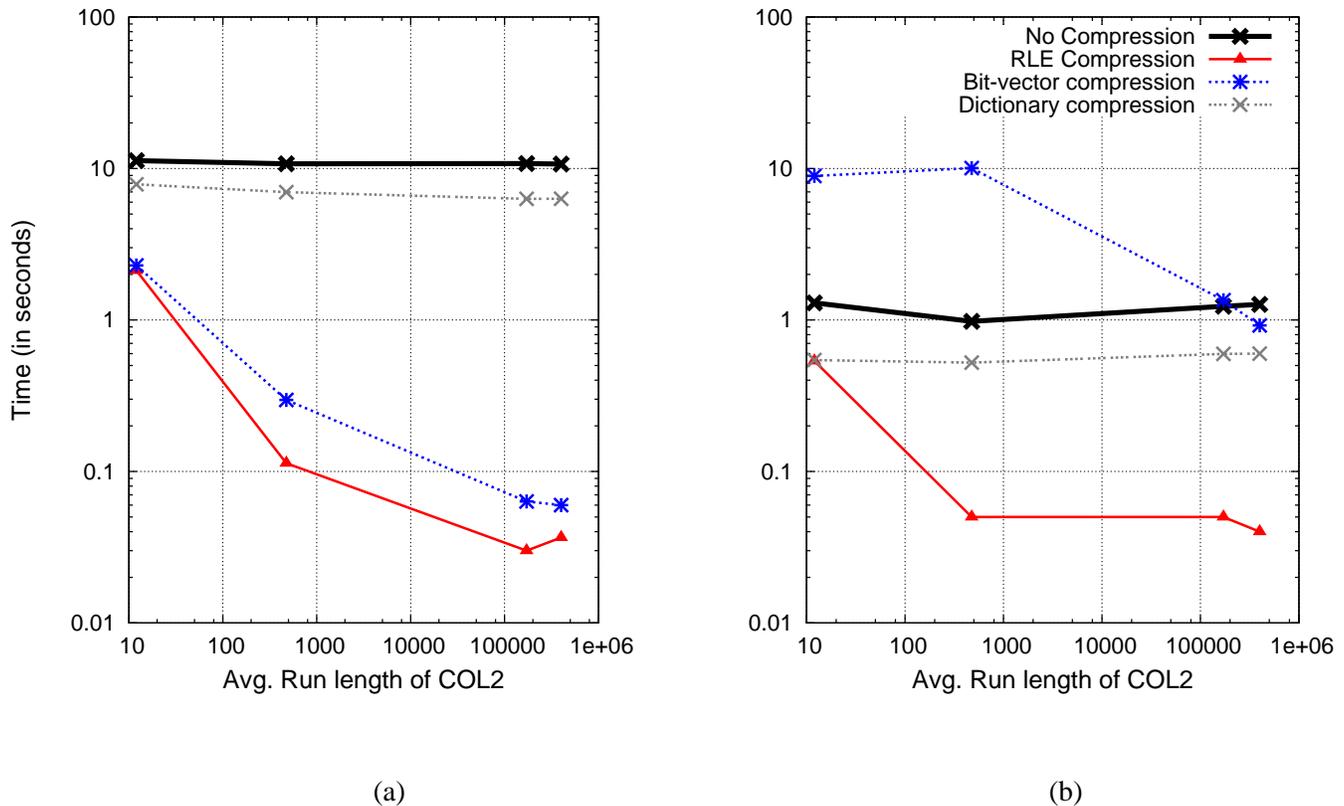
Figure 4-9: (a) Predicate on the variably compressed column, position filter on the RLE column and (b) Predicate on the RLE column, position filter on the variably compressed column. Note log-log scale.

much more poorly (note the log scale). This is because the query requires the values of the bit-vector column in position order which forces decompression which has already been shown to be slow (at very high run-lengths bit-vector encoding starts to see entire pages of '1's and '0's which causes it to optimize its operation, which is why it starts to perform well in the final two points in the graph). This difference in performance between Figures 4-9(a) and 4-9(b) illustrates that the proper choice of encoding type for a column depends not just on data characteristics, but also on the expected query workload. This observation supports a major future research goal of exploring the interaction between physical database design, optimization, and compression. It also indicates that redundantly storing the same column in the same sort order using different compression schemes might be a good idea.

The next query that we experimented with was a join query (again with an aggregation):

```
SELECT S.COL3, COUNT(*)
FROM CSTORE_P1 AS L, CSTORE_P2 AS S
WHERE PREDICATE(S.COL2) AND PREDICATE(L.COL1)
      AND L.COL2=S.COL1
GROUP BY S.COL3
```

The algorithm for performing joins in C-Store was described in Section 4.4.1. Assume for this query that CSTORE_P1 is a projection from the fact table and that CSTORE_P2 is a projection from a dimension table that contains its primary key (which is the common join case in star schema queries). Hence, L.COL2 is a foreign key into CSTORE_P2 (S.COL1 is the key). This query applies a predicate to each table before the join, does a foreign-primary key join, and then uses the position list result from the join to filter and aggregate a column from CSTORE_P2.

Again, we started with CSTORE_P1 being the lineitem fact table from TPC-H. The join attribute is the supplier
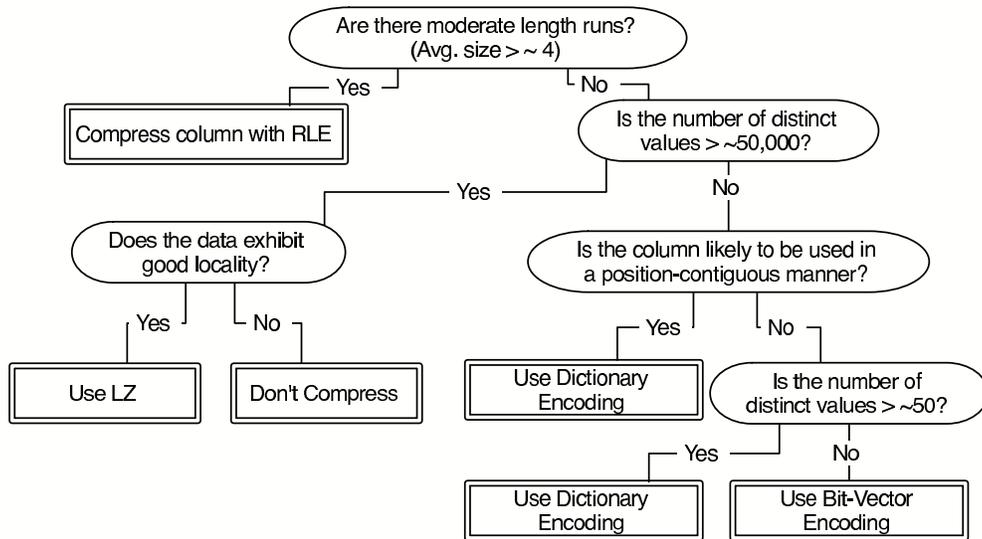
Figure 4-10: Decision tree summarizing our results regarding the proper selection of compression scheme.

foreign key. We assume the projections are sorted on S.COL2 and L.COL1 (this is the common case since the C-Store optimizer will have a choice as to what projections to use for a query and will choose projections that are sorted by predicated columns) and are therefore RLE encoded. We allowed L.COL2 (suppkey) to be secondarily sorted and encoded it with the same four coding algorithms as the previous (select) queries. In order to show results for the bit-vector case, we reduced the number of unique supplier keys in the fact table to just 50 values in one of our experiments (we allowed 50000 values in the other experiment). The results of performing this join are shown in the table below (times are in seconds).

Encoding Type	50 keys	50000 keys
RLE	0.06	0.07
Bit-vector	0.97	N/A
Dictionary	3.15	3.86
No Compression	4.08	4.3

The techniques for operating directly on RLE and bit-vector data have been discussed previously, for the join part of this query in Section 4.4.1 and for the resulting position filtering in the previous query in this section. To operate directly on dictionary data, the dimension table join column had to be recoded using the fact table's dictionary at the beginning of the query (this is included in the query time.)

4.6   Conclusion

The decision tree in Figure 4-10 summarizes our results and provides a heuristic for deciding which encoding scheme to use for a column.

In this tree, "exhibits good locality" means that the column is either one of the sort columns in the projection, is correlated with one of the sort columns in the projection, or otherwise contains repeated patterns of data. "Likely to be used in a position contiguous manner" means that that the column needs to be read in parallel with another column, so the column is not accessed out of order. For example, if the column is in the WHERE clause, accessing it in position contiguous fashion is not required, but if it is in the SELECT clause it is likely to be accessed via a sorted position list in a position contiguous manner.

65

In addition to the observations regarding when to use each of the various compression schemes, our results also illustrate the following important points:

- Physical database design should be aware of the compression subsystem. Performance is improved by compression schemes that take advantage of data locality. Queries on columns in projections with secondary and tertiary sort orders perform well, and it is generally beneficial to have low cardinality columns serve as the leftmost sort orders in the projection (to increase the average run-lengths of columns to the right). The more order and locality in a column, the better.

- It is a good idea to operate directly on compressed data. Sacrificing the compression ratio of heavy-weight schemes for the efficiency light-weight schemes in operating on compressed data is a good trade-off to make.

- The optimizer needs to be aware of the performance implications of operating directly on compressed data in its cost models. Further, cost models that only take into account I/O costs will likely perform poorly in the context of column-oriented systems since CPU cost is often the dominant factor.

In summary, this chapter shows that significant database performance gains can be had by implementing light-weight compression schemes and operators that work directly on compressed data. By classifying compression schemes according to a set of basic properties, we were able to extend C-Store to perform this direct operation without requiring new operator code for each compression scheme. Furthermore, our focus on column-oriented compression allowed us to demonstrate that the performance benefits of operating directly on compressed data in column-oriented schemes is much greater than the benefit in operating directly on row-oriented schemes.

Hence, we see this work as an important step in understanding the substantial performance benefits of column-oriented database designs. Although this chapter focused on fairly simple queries so as to carefully distill the performance characteristics of column-oriented compression, in Chapter 7 we will return to this subject, and evaluate how compression improves performance on a complete data warehousing benchmark. Before we do this however, we introduce two other performance optimizations in the next two chapters: late tuple materialization and the invisible join.

Chapter 5

Materialization Strategies

5.1   Introduction

Column-stores are essentially a modification only to the physical data structures of a database: at the logical and view level, a column-store looks identical to a row-store. For this reason, column-stores may choose to offer a standards-compliant relational database interface (e.g., ODBC, JDBC, etc). This has the advantage of decreasing the time to deployment of a column-store if it is replacing a row-store for a particular application. In order to implement these interfaces, separate columns must ultimately be stitched together into tuples of data to be output. Even if a column-store chooses not to offer standards-compliant interfaces, it might still have to stitch columns of data into rows to be output since applications generally want to process database output entity-at-a-time rather than attribute-at-a-time.

Determining when to do this stitching together in a query plan is the inverse of the problem of applying projections in a row-oriented database, since rather than deciding when to project an attribute out of an intermediate result flowing through the query plan, the system must decide when to add it in. Lessons from row-oriented databases (where projections are almost always performed as soon as an attribute is no longer needed) suggest a natural tuple construction policy: at each point at which a column is accessed, add the column to an intermediate tuple representation if that column is needed by some later operator or is included in the set of output columns. At the top of the query plan, these intermediate tuples can be directly output to the user. We call this process of adding columns to intermediate results *materialization* and call the simple scheme described above *early materialization*, since it seeks to form intermediate tuples as early as possible.

Surprisingly, experiments in this chapter will show that early materialization is not always the best strategy to employ in a column-store. Consider a simple example: suppose a query consists of three selection operators σ_1, σ_2, and σ_3 over three columns, $R.a$, $R.b$, and $R.c$ (all sorted in the same order and stored in separate files), where σ_1 is the most selective predicate and σ_3 is the least selective. An early materialization strategy could process this query as follows: read in a block of $R.a$, a block of $R.b$, and a block of $R.c$ from disk. Stitch them together into (likely more than one) block(s) of row-store style triples $(R.a, R.b, R.c)$. Apply σ_1, σ_2, and σ_3 in turn, allowing tuples that match the predicate to pass through. This strategy can be inefficient since if σ_1 was selective, many tuples were needlessly stitched to together, only to be immediately discarded by the first predicate.

There is another strategy that can be more efficient, however, on certain workloads. In this chapter, this second approach will be called *late materialization*, because it does not form tuples until after some part of the plan has been processed. It works as follows: first scan $R.a$ and output the positions (ordinal offsets of values within the column) in $R.a$ that satisfy σ_1 (these positions can take the form of ranges, lists, or a bitmap). Repeat with $R.b$ and $R.c$, outputting positions that satisfy σ_2 and σ_3 respectively. Next, use position-wise AND operations to intersect the position lists. Finally, re-access $R.a$, $R.b$, and $R.c$ and extract the values of the records that satisfy all predicates and stitch these values together into output tuples. This late materialization approach can potentially be more CPU efficient because it requires fewer intermediate tuples to be stitched together (which is a relatively expensive operation as it can be thought of as a join on position), and position lists are small, highly-compressible data structures that can be

operated on directly with little overhead. For example, 32 (or 64 depending on processor word size) positions can be intersected at once when ANDing together two position lists represented as bit-strings. Note, that a disadvantage of this late materialization approach is that it requires re-scanning the base columns to form tuples, which can be slow (though they are likely to still be in memory upon re-access if the query is properly pipelined).

The main contribution of this work is to systematically explore the trade-offs between different strategies and provide a foundation for choosing a strategy for a particular query. The focus is on standard warehouse-style queries: read-only workloads with selections, aggregations, and joins. We extended C-Store with a variety of materialization strategies, and experimentally evaluate the effects of varying selectivities, compression techniques, and query plans on these strategies. Further, we provide a model that can be used (for example) in a query optimizer to select a materialization strategy. Our results show that, on some workloads, late materialization can be an order of magnitude faster than early-materialization, while on other workloads, early materialization outperforms late materialization.

The remainder of this chapter is organized as follows. We illustrate the trade-offs between materialization strategies in Section 5.2 and then present both pseudocode and an analytical model for example query plans using each strategy in Section 5.3. We validate our models experimentally (using a version of C-Store we extended) in Section 5.4. Finally, we discuss related work in Section 5.5 and conclude in Section 5.6

5.2   Materialization Strategy Trade-offs

In this section we present some of the trade-offs that are made between materialization strategies. A materialization strategy needs to be in place whenever more than one attribute from any given relation is accessed (which is the case for most queries). Since a column-oriented DBMS stores each attribute independently, it must have some mechanism for stitching together multiple attributes from the same logical tuple into a physical tuple. Every proposed column-oriented architecture accomplishes this by attaching either physical or virtual *tuple identifiers* or *positions* to column values. To reconstruct a tuple from multiple columns of a relation, the DBMS simply needs to find matching positions. Modern column-oriented systems [29, 28, 63] store columns in position order; i.e., to reconstruct the i'th tuple, one uses the i'th value from each column. This accelerates the tuple reconstruction process.

As described in the introduction, tuple reconstruction can occur at different points in a query plan. *Early materialization* (EM) constructs tuples as soon as (or sometimes before) tuple values are needed in the query plan. *Late materialization* (LM) constructs tuples as late as possible, sometimes even at the query output. Each approach has a set of advantages.

5.2.1   Late Materialization Advantages

The primary advantages of late materialization are that it allows the executor to use high-performance operations on compressed, column-oriented data and to defer tuple construction to later in the query plan, possibly allowing it to construct fewer tuples.

Operating on Positions

Most queries contain one or more predicates in the WHERE clause that need to be applied. The result of predicate application is a subset of positions from a column whose values passed the predicate. If more than one predicate is applied, different subsets of positions will be produced from different columns. Tuple reconstruction thus requires a equi-join on position of multiple columns. However, since columns are sorted by position, this join can be performed with a relatively fast merge join.

Positions can be represented using a variety of compression techniques. *Runs* of consecutive positions can be represented using position ranges of the form [startpos, endpos]. Positions can also be represented as bit-maps using a single bit to represent every position in a column, with a '1' in the bit-map entry if the tuple at the position passed the predicate and a '0' otherwise. For example, for a position range of 11-20, a bit-vector of 0111010001 would indicate that positions 12, 13, 14, 16, and 20 contained values that passed the predicate.

These position representations can be operated on directly without using column values. For example, an AND operation of 3 single column predicates in the WHERE clause of an SQL query can be performed by applying each predicate separately on its respective column to produce 3 sets of positions for which the predicate matched. These 3 position lists can be intersected to create a new position list that contains a list of all positions of tuples that passed every predicate. This position list can then be sent to other columns in the same relation to retrieve additional column values from those logical tuples, which can then be sent to parent operators in the query plan for processing.

Position operations are highly efficient from a CPU perspective due to the highly compressible nature of position representations and the ease of operation on them. For example, intersecting two position lists represented using bit-strings is requires only n/32 (or n/64 depending on processor word size) instructions (if n is the number of positions being intersected) since 32 positions can be intersected in a single instruction. Intersecting a position range with a bit-string is even faster (requiring a constant number of instructions), as the result is equal to the subset of the same bit-string starting at the beginning of the position range and ending at the last position covered by the range.

Not only is the processing of positions fast, but their creation can also be fast since in many cases the positions of tuples that pass a predicate can be derived directly from an index on the column. For example, if there is a clustered index over a column and a predicate on a value range, the index can be accessed to find the start and end positions that match the value range, and these two positions can encode the entire set of positions in that column that match the predicate. Similarly, there might be a bit-map index on that column [55, 63, 18], in which case the positions matching a predicate can be derived by ORing together the appropriate bitmaps. In both cases, the original column values never have to be accessed.

Column-Oriented Data Structures

Another advantage of late materialization is that column values can be stored together contiguously in memory in column-oriented data structures. This has two performance advantages: First, the column can be kept compressed in memory using the same column-oriented compression techniques as were used to store the column on disk. Chapter 4 showed that techniques such as run length encoding (RLE) of column values and dictionary encoding are well suited for column stores and can easily be operated on directly. For example, an entire run of RLE-encoded values can be processed in a single operator loop. Tuple construction requires decompressing RLE data, since generally repeats are of values in a single column, not entire tuples.

Second, looping through values from a column-oriented data structure tends to be much faster than looping through values using a tuple iterator interface. First, entire cache lines are filled with values from the same column. This maximizes the efficiency of the memory bandwidth bottleneck [21] as the cache prefetcher only fetches relevant data. Second, high IPC (instructions-per-cycle) vector processing code can be written for column block access (see Section 3.6).

Construct Only Relevant Tuples

In many cases, a query outputs fewer tuples than are actually processed. Predicates usually reduce the number of tuples output, and aggregations combine tuples into summary tuples. Thus, if the executor waits long enough before constructing a tuple, it might be able to avoid constructing it altogether.

5.2.2 Early Materialization Advantages

The fundamental problem with delaying tuple construction is that in some cases columns may need to be accessed multiple times in the query plan. For example, suppose a column is accessed once to retrieve positions matching a predicate and a second time (later in the plan) to retrieve its values. If the matching positions cannot be determined from an index, then the column values will be accessed twice. Assuming proper query pipelining, the reaccess will incur no disk costs (the disk block will be in the buffer cache), but there will be a CPU cost in scanning the block to extract the values corresponding to the given positions. This cost can be substantial if the positions are not in sorted order (e.g., they were reordered by a join, as discussed in Section 5.4.3).

$|C_i|$	Number of disk blocks in Col_i
$\|C_i\|$	Number of "tuples" in Col_i
$\|POS LIST\|$	Number of positions in $POS LIST$
F	Fraction of pages of a column in buffer pool
SF	Selectivity factor of predicate
BIC	CPU time in ms of getNext() call in block iterator
TIC_{TUP}	CPU time for getNext() call in tuple iterator
TIC_{COL}	CPU time for getNext() call in column iterator
FC	Time for a function call
PF	Prefetch size (in number of disk blocks)
$SEEK$	Disk seek time
$READ$	Time to read a block from disks
RL	Avg. run-length in RLE encoded columns (RL_c) or position lists (RL_p) (equal to one if uncompressed)

Table 5.1: Notation used in analytical model

In early materialization, as soon as a column is accessed, its values are added to an intermediate-result tuple, eliminating the need for future reaccesses. Thus, the fundamental trade-off between early materialization and late materialization is the following: while late materialization enables several performance optimizations (operating directly on position data, constructing only relevant tuples, operating directly on column-oriented compressed data, and high value iteration speeds), if the column reaccess cost at tuple reconstruction time is high, a performance penalty is paid.

5.3 Query Processor Design

Having described the fundamental trade-offs between early and late materialization, we now present both pseudocode and an analytical model for component operators of each materialization strategy and give detailed examples of how these strategies are translated into query plans in a column-oriented system.

5.3.1 Operator Analysis

To better illustrate the trade-offs between early and late materialization, in this section we present an analytical model of the two strategies. The model is composed of three basic types of operators:

- Data source (DS) operators that read columns from disk, filtering on one or more single-column predicates or a position list as they go, and producing either vectors of positions or vectors of positions and values.

- AND operators that merge several position lists into a single position list in which positions are present only if they were present in all input position lists.

- Tuple construction operators that combine multiple narrow tuples of positions and values into wider tuples.

These operators are sufficient to express simple selection queries using each strategy. We use the notation in Table 5.1 to describe the costs of the different operators.

5.3.2 Data Sources

In this section, we consider the cost of accessing a column on disk via a data source operator. We consider four cases (the first and third used by LM strategies, the second and fourth used by EM):

Case 1: A column C_i of $|C_i|$ blocks is read from disk and a predicate with selectivity SF is applied to each tuple. The output is a column of positions. The pseudocode and cost analysis of this case is shown in Figure 5-1.

```
DS_Scan-Case1(Column C, Pred p)
 1. for each block b in C
 2.    read b from disk (if necessary)
 3.    for each tuple t in b (or RLE triple in b)
 4.       apply p to t
 5.       output positions from t
```

$$CPU =$$
$$|C_i| * BIC+ \tag{1}$$
$$\|C_i\| * (TIC_{COL} + FC)/RL_c+ \tag{3,4}$$
$$SF * \|C_i\| * FC \tag{5}$$
$$IO = (\frac{|C_i|}{PF} * SEEK + |C_i| * READ) * (1 - F) \tag{2}$$
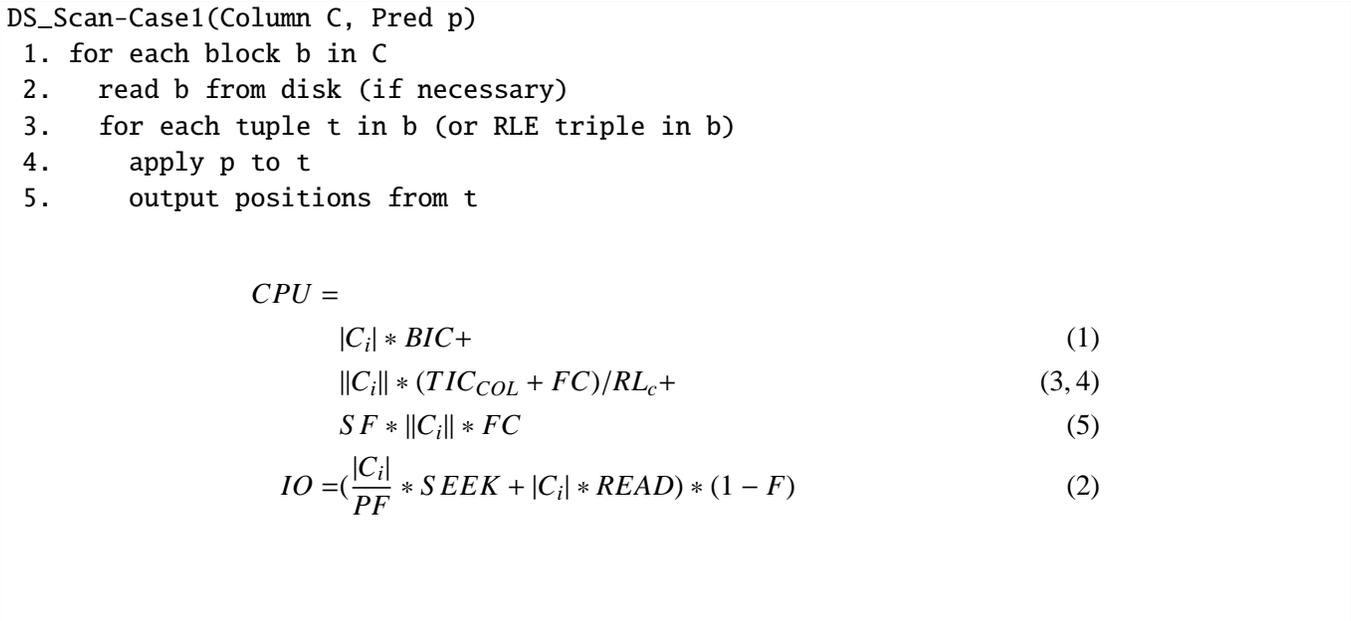
Figure 5-1: Pseudocode and cost formulas for data sources, Case 1. Numbers in parentheses in cost formula indicate corresponding steps in the pseudocode.

Case 2: A column C_i of $|C_i|$ blocks is read from disk and a predicate with selectivity SF is applied to each tuple. The output is a column of (position, value) pairs.

The cost of Case 2 is identical to Case 1 except for step (5) which becomes $SF * \|C_i\| * (TIC_{TUP} + FC)$. The slightly higher cost reflects the cost of gluing positions and values together for the output.

Case 3: A column C_i of $|C_i|$ blocks is read from disk or memory and filtered with a list of positions, $POSLIST$. The output is a column of the values corresponding to those positions. The pseudocode and cost analysis of this case is shown in Figure 5-2.

Case 4: A column C_i of $|C_i|$ blocks is read from disk and a set of tuples EM_i of the form $(pos, < a_1, \ldots, a_n >)$ is input to the operator. The operator jumps to position pos in the column and applies a predicate with selectivity SF. Tuples that satisfy the predicate are merged with EM_i to create tuples of the form $(pos, < a_1, \ldots, a_n, a_{n+1} >)$ that contain only the positions that were in EM_i and that satisfied the predicate over C_i. The pseudocode and cost analysis of this case is shown in Figure 5-3.

5.3.3 Multicolumn AND

The AND operator takes in k position lists, $inpos_1 \ldots inpos_k$ and produces a new list of positions representing the intersection of these input lists, $outpos$. Since operating directly on positions is fast, the cost of the AND operator in query plans tends to be insignificant relative to the other operators. Nonetheless, we present the analysis here for completeness.

This model examines two possible representations for a list of positions: a list of ranges of positions (e.g., 1-100, 200-250) and bit-strings with one bit per position. The AND operator is only used in the LM case, since EM always uses Data scan Case 4 above to construct tuples as they are read from disk. If the positional input to AND are all ranges, then it will output position ranges. Otherwise it will output positions in bit-string format.

We consider three cases:

Case 1: Range inputs, Range output

```
DS_Scan-Case3(Column C, POSLIST pl)
 1. for each block b in C
 2.  read b from disk (if necessary)
 3.  iterate through pl, for each pos. (range)
 4.   jump to pos (range) in b & output value(s)
```

$$CPU =$$

$$|C_i| * BIC+ \tag{1}$$

$$\|POSLIST\|/RL_p * (TIC_{COL})+ \tag{3}$$

$$\|POSLIST\|/RL_p * (TIC_{COL} + FC) \tag{4}$$

$$IO = (\frac{|C_i|}{PF} * SEEK + SF * |C_i| * READ) * (1 - F) \tag{2}$$

/* F=1 and IO → 0 if col already accessed */

/* $SF * |C_i|$ is a lower bound for the blocks needed to

be read in. For highly localized data (like the semi-sorted

data we will work with), this is a good approximation*/

Figure 5-2: Pseudocode and cost formulas DS-Case 3.

```
DS_Scan-Case4(Column C, Pred p, Table EM)
 1. for each block b in C
 2. read b from disk (if necessary)
 3. iterate through tuples e in EM, extract pos
 4. use pos to jump to correct tuple t in C and apply predicate
 5. if predicate succeeded, output <e, t>
```

$$CPU =$$

$$|C_i| * BIC+ \tag{1}$$

$$\|EM_i\| * TIC_{TUP}+ \tag{3}$$

$$\|EM_i\| * ((FC + TIC_{TUP}) + FC) \tag{4}$$

$$SF * \|EM_i\| * (TIC_{TUP}) \tag{5}$$

$$IO = (\frac{|C_i|}{PF} * SEEK + SF * |C_i| * READ) * (1 - F) \tag{2}$$

/* $SF * |C_i|$ is a lower bound for the blocks needed to

be read in, as in Figure 5-2 */

Figure 5-3: Pseudocode and cost formulas for DS-Case 4.

In this case, each of the input position lists and the output position list are each encoded as ranges. The pseudocode and cost analysis for this case is shown in Figure 5-4. Since this operator is a streaming operator it incurs no I/O.

```
AND(POSLIST inpos 1 ... inpos k)
 1. iterate through position lists
 2. perform AND operations
 3. produce output lists
```

$$\text{Let } M = max(\|inpos_i\|/RL_{p_i}, i \in 1\ldots k)$$

$$COST =$$

$$\sum_{i=1\ldots k} (TIC_{COL} * \|inpos_i\|/RL_{p_i}) + \tag{1}$$

$$M * (k-1) * FC + \tag{2}$$

$$M * TIC_{COL} * FC \tag{3}$$
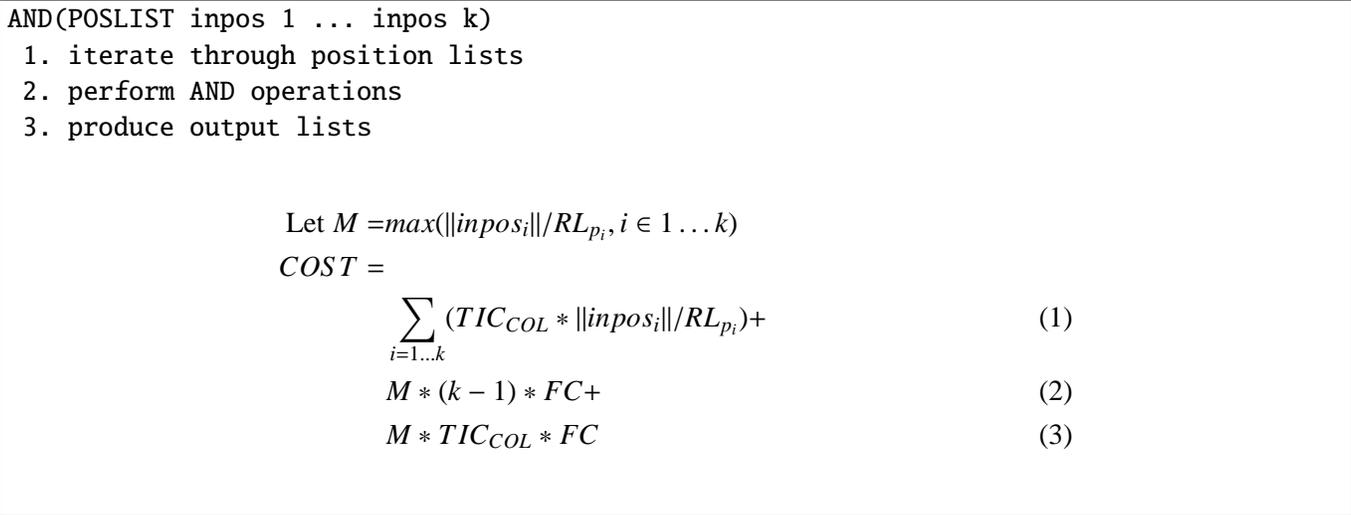
Figure 5-4: Psuedocode and cost formulas for AND, Case 1.

```
Merge(Col s1,..., Col sk)
 1. iterate through all k cols of len. ||VAL_i||
 2. produce output tuples
```

$$COST =$$

// Access values as vector (don't use iterator)

$$\|VAL_i\| * k * FC + \tag{1}$$

// Produce tuples as array (don't use iterator)

$$\|VAL_i\| * k * FC) \tag{2}$$
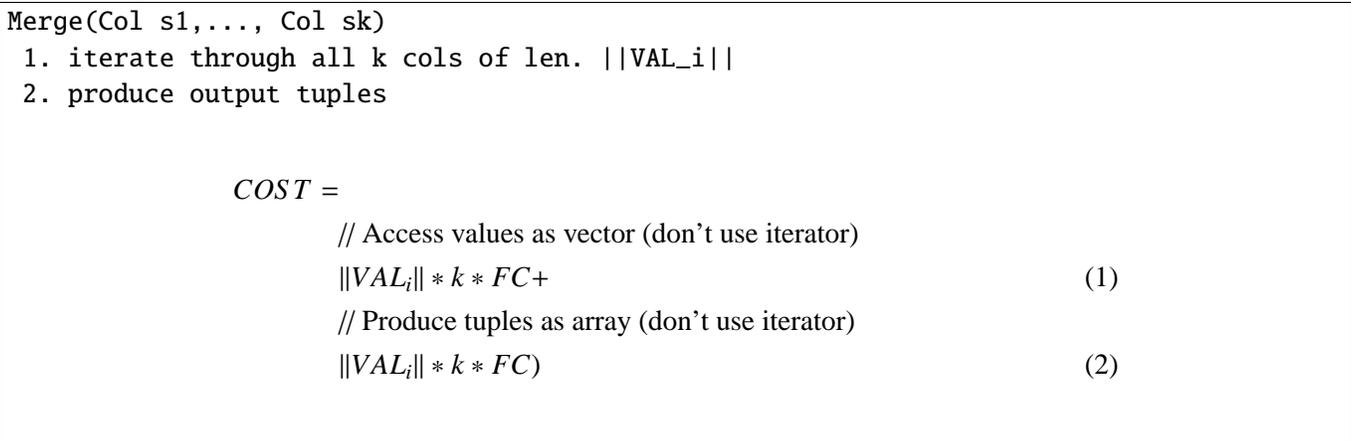
Figure 5-5: Pseudocode and cost formulas for Merge.

Case 2: Bit-list inputs, bit-list output

In this case each of the input position lists and the output position lists are bit vectors. The input position lists are "ANDed" 32 bits at a time. The cost formula for this case is identical to Case 1 except that every instance of $\|inpos_i\|/RL_{p_i}$ in Figure 5-4 is replaced with $\|inpos_i\|/32$ (32 is the processor word size).

Case 3: Mix of Bit-list and range inputs, bit-list output

In this case the input position lists to the AND operator are a mix of range and bit lists. The output is a bit-list. Execution occurs in three steps. First the range lists are intersected to produce one range list. Second, the bit-lists are anded together to produce a single bit list. Finally, the single range and bit lists are combined to produce the output list.

5.3.4 Tuple Construction Operators

The final two operators we consider are tuple construction operators. The first, the MERGE operator, takes k sets of values $VAL_1 \ldots VAL_k$ and produces a set of k-ary tuples. This operator is used to construct tuples at the top of

```
SPC(Col c1,..., Col ck, Pred p1,..., Pred pk)
 1. for each column, C_i
 2.   for each block b in C_i
 3.     read b from disk
 4.     call Merge sub-routine
 5.     check predictates
 6.     output matching tuples
```

$$CPU =$$

$$|C_i| * BIC+ \tag{2}$$

$$Merge(c1, ..., ck)+ \tag{4}$$

$$\|C_i\| * FC * \prod_{j=1...(i-1)} (SF_j)+ \tag{5}$$

$$\|C_k\| * FC * \prod_{j=1...k} (SF_j)+ \tag{6}$$

$$IO = (\frac{|C_i|}{PF} * SEEK + |C_i| * READ) \tag{3}$$

Figure 5-6: Pseudocode and cost formulas for SPC.

an LM plan. The pseudocode and cost of this operation is shown in Figure 5-5. The analysis assumes the k sets of values are resident in main memory, since they are produced by a child operator, and that each set has the same cardinality.

The second tuple construction operator is the SPC (Scan, Predicate, and Construct) operator which can sit at the bottom of EM plans. SPC takes a set of columns $VAL_1 \ldots VAL_k$, reads them off disk, optionally takes a set of predicates to apply on the column values, and constructs tuples if all predicates pass. The pseudocode and cost of this operation is shown in Figure 5-6.

5.3.5   Example Query Plans

The use of the above presented operators is illustrated in Figures 5-7 and 5-8 for the query:

```
(1) SELECT shipdate, linenum FROM lineitem
    WHERE  shipdate < CONST1 AND linenum < CONST2
```

where lineitem is a table taken from TPC-H [7], a benchmark that models data typically found in decision support and data warehousing applications.

One EM query plan, shown in Figure 5-7(a), uses a DS2 operator (Data Scan Case 2) operator to scan the shipdate column, producing a stream of $(pos, shipdate)$ tuples that satisfy the predicate $shipdate < CONST1$. This stream is used as one input to a DS4 operator along with the linenum column and the predicate $linenum < CONST2$ to produce a stream of $(shipdate, linenum)$ result tuples.

Another possible EM query plan, shown in Figure 5-7(b), constructs tuples at the very beginning of the plan - merging all needed columns at the leaf node as it applies the predicates using a SPC operator. The key difference between these early materialization strategies is that while the latter strategy has to scan and process all blocks for all input columns, the former plan applies each predicate in turn and constructs tuples incrementally, adding one attribute per operator. For non-selective predicates this is more work, but for selective predicates only subsets of

EM-pipelined
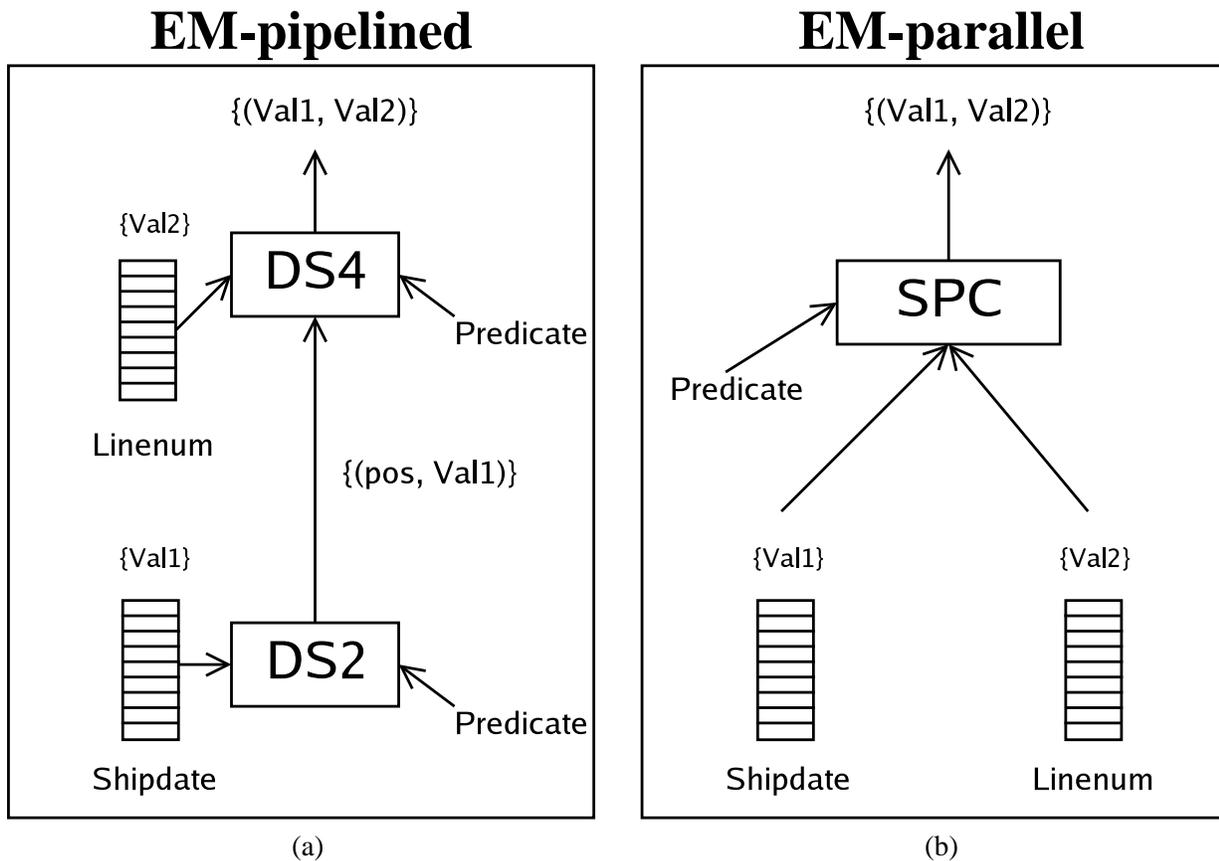
EM-parallel

Figure 5-7: Query plans for EM-pipelined (a) and EM-parallel (b) strategies. DS2 is shorthand for DS_Scan-Case2. (Similarly for DS4).

blocks need to be processed (or in some cases the entire block can be skipped). We call the former strategy EM-pipelined and the latter strategy EM-parallel. The choice of which EM plan to use depends on the selectivity of the predicates; EM-pipelined is likely better if there are highly selective predicates.

As in EM, there are both pipelined and parallel late materialization strategies. LM-parallel is shown in Figure 5-8(a) and LM-pipelined is shown in Figure 5-8(b). LM-parallel begins with two DS1 operators, one for the shipdate and linenum columns. Each DS1 operator scans its column, applying the appropriate predicate to produce a position list of those values that satisfy the predicate. The two position lists are streamed into an AND operator which intersects the two lists. The output position list is then streamed into two DS3 operators to obtain the corresponding values from the shipdate and linenum columns. As these values are obtained they are streamed into a merge operator to produce a stream of (shipdate, linenum) result tuples.

LM-pipelined works similarly to LM-parallel, except that it applies the DS1 operators one at a time, pipelining the positions of the shipdate values that passed the shipdate predicate to a DS3 operator for the linenum column which produces the column values at these input set of positions and sends these values to the linenum DS1 operator which only needs to apply its predicate to this value subset (rather than at all linenum positions). As a side effect, the need for the AND operator is eliminated.

5.3.6 LM Optimization: Multi-Columns

Note that in DS Case 3, used in LM strategies to produce values from positions, the I/O cost is assumed to be zero if the column has already been accessed earlier in the plan, even if the column size is larger than available memory. This is made possible through a specialized data structure for representing intermediate results, designed to facilitate

LM-parallel   **LM-pipelined**

(a)   (b)
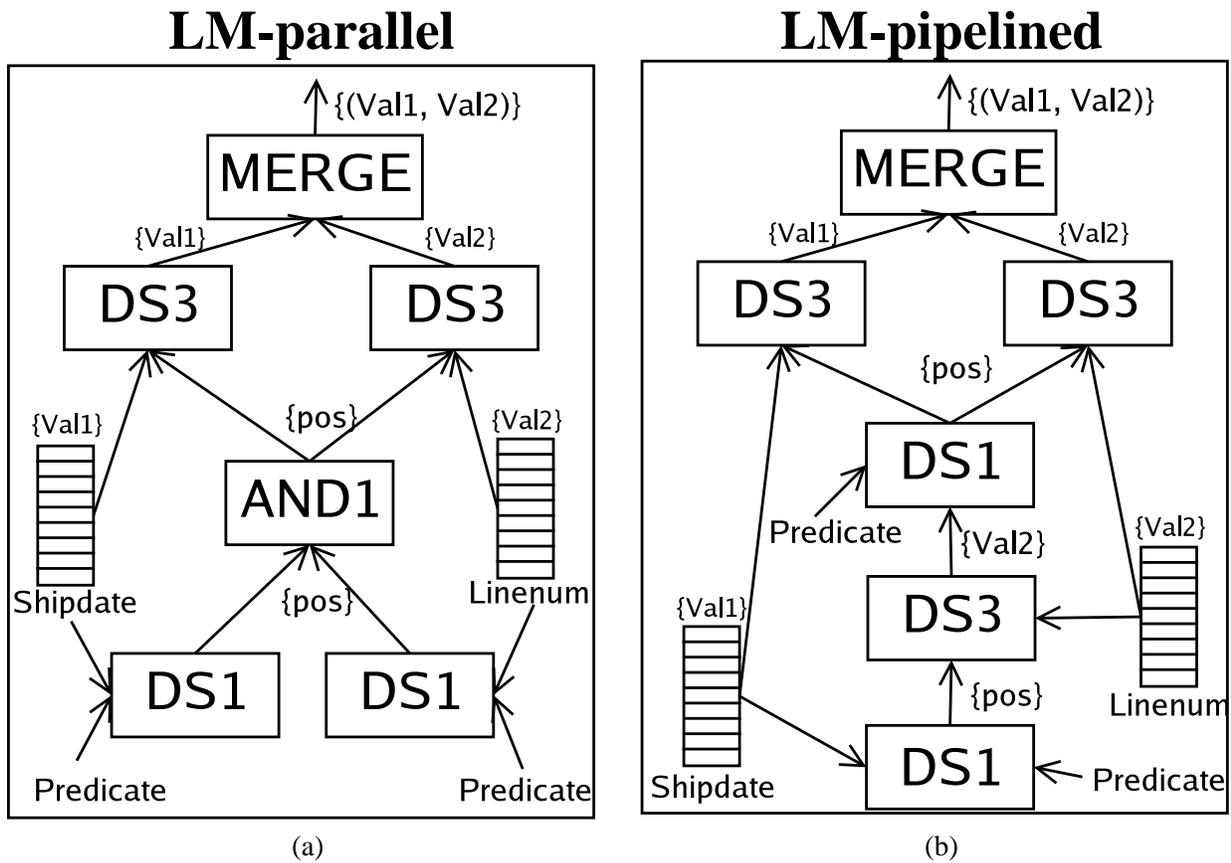
Figure 5-8: Query plans for LM-parallel (a) and LM-pipelined (b) strategies.

query pipelining, that allows blocks of column data to remain in user memory space after the first access so that they can be easily reaccessed again later on. We call this data structure a *multi-column* (see Figure 5-9).

A multi-column contains a memory-resident, horizontal partition of some subset of attributes from a particular relation. It consists of:

A *covering position range* indicating the virtual start position and end position of the horizontal partition (for example, a position range could indicate that rows numbered 1000-2000 are covered in this multi-column).

An array of *mini-columns*. A mini-column is the set of corresponding values for a specified position range of a particular attribute (MonetDB [28] calls this a *vector*, PAX [21] calls this a *mini-page*). Using the previous example, a mini-column for column X would contain 1001 values - the 1000th-2000th values in this column. The *degree* of a multi-column is the size of the mini-column array which is the number of included attributes. Each mini-column is kept compressed the same way as it was on disk.

A *position descriptor* indicating which positions in the position range remain valid. Positions are made invalid as predicates are applied on the multi-column. The position descriptor may take one of three forms:

- *Ranged positions*: All positions between a specified start and end position are valid.

- *Bit-mapped positions*: A bit-vector of size equal to the multi-column covering position range is given, with a '1' at a corresponding position if that position is valid. For example, for a position coverage of 11-20, a bit-vector of 0111010001 would indicate that positions 12, 13, 14, 16, and 20 are valid.

- *Listed positions* A list of valid positions inside the covering position range is given. This is particularly useful when few positions inside a multi-column are valid.

76

When a page from a column is read from disk (e.g., by a DS1 operator), a mini-column is created (which is essentially just a pointer to the page in the buffer pool) with a position descriptor indicating that all positions are valid. The DS1 operator then iterates through the column, applying the predicate to each value and produces a new list of valid positions. The multi-column then replaces its position descriptor with the new position list (the mini-column remains untouched).

An AND operator takes two multi-columns with overlapping covering position ranges and creates a new multi-column where the covering position range and position descriptor are equal to the intersection of the position range and position descriptors of the input multi-columns and the set of mini-columns is equal to the union of the input set of mini-columns. Thus, ANDing multi-columns is in essence the same operation as ANDing normal position lists. The only difference is that in addition to performing the intersection of the position lists, ANDing multi-columns requires copying pointers to mini-columns to the output multi-column, but this is a low cost operation.

If the AND operator produces multi-columns rather than just positions as an input to a DS3 operator, then the operator does not need to reaccess the column, but rather can work directly on one multi-column block at a time – iterating through the appropriate mini-column to produce only those values whose positions are valid according to the position descriptor. Single multi-column blocks are worked on in each operator iteration, so that column-subsets can be pipelined up the query tree. With this optimization, there is no DS3 I/O cost for a reaccessed column.
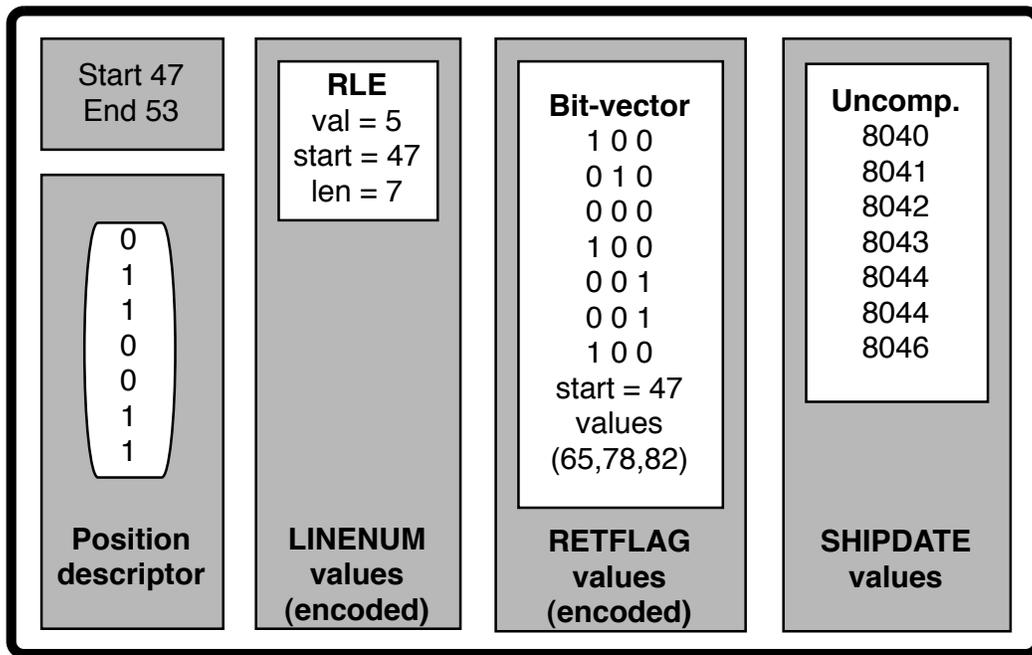
Figure 5-9: An example multi-column block containing values for the SHIPDATE, RETFLAG, and LINENUM columns. The block spans positions 47 to 53; within this range, positions 48, 49, 52, and 53 are active.

5.3.7   Predicted versus Actual Behavior

To gauge the accuracy of the analytical model, we compared the predicted execution time for the selection query (1) in Section 5.3.5 above with the actual execution time obtained using the C-Store prototype and a TPC-H scale 10 version of the lineitem projection primarily and secondarily sorted by *returnflag* and *shipdate*, respectively. The results obtained are presented in Figures 5-10(a) and 5-10(b), which plot response time as a function of the selectivity of the predicate *shipdate* < *CONST*2 for the the late and early materialization strategies, respectively. For these results, we encoded the shipdate column (into one block) using RLE encoding and kept the linenum column
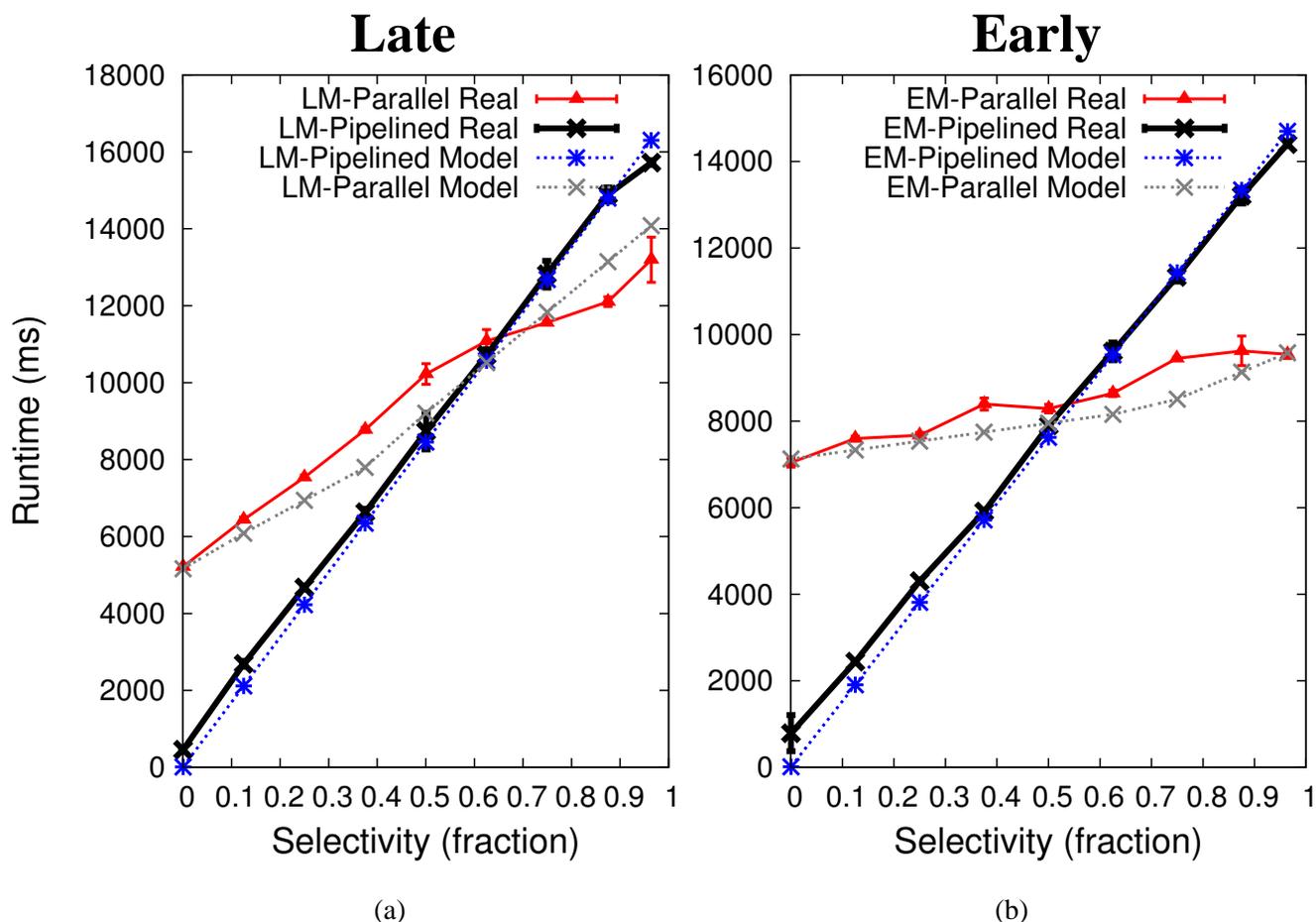
Figure 5-10: Predicted and observed performance for late (a) and early (b) materialization strategies on selection queries.

uncompressed (the 60,000,000 linenum tuples occupy 3696 64KB blocks). Table 5.2 contains the constant values used by the analytical model. These constants were obtained from micro-benchmarks on the C-Store system; they were not reverse-engineered to make the model match the experiments. Both the model and the experiments incurred an additional cost at the end of the query to iterate through the output tuples ($numOutTuples * TIC_{TUP}$). We assume a two thirds overlap of I/O and CPU (which is what we have generally found when running C-Store on our testbed machines).

Here, the important observation is that the models' predictions are quite accurate (at least for this query), which helps validate our understanding of the two strategies. The actual results will be further discussed in Section 5.4. Additionally, we tested our model on several other cases, including the same query presented here but using an RLE-compressed linenum column (occupying only 7 disk blocks) as well as additional queries in which both the shipdate and linenum predicates were varied. We consistently found the model to reasonably predict our experimental results.

5.4 Experiments

To evaluate the trade-offs between the early materialization and late materialization strategies, we ran two queries under a variety of configurations. These queries were run over data generated from the TPC-H dataset. Specifically, we generated an instance of the TPC-H data at scale 10, which yields a total database size of approximately 10 GB with the biggest table (lineitem) containing 60,000,000 tuples. We then created a C-Store projection from a table (all sorted in the same order) containing the SHIPDATE, LINENUM, QUANTITY, and RETURNFLAG

BIC	0.020 microsecs
TIC_{TUP}	0.065 microsecs
TIC_{COL}	0.014 microsecs
FC	0.009 microsecs
PF	1 block
$SEEK$	2500 microsecs
$READ$	1000 microsecs

Table 5.2: Constants used for Analytical Models

columns; the projection was primarily sorted on RETURNFLAG, secondarily sorted on SHIPDATE, and tertiarily sorted on LINENUM. The RETURNFLAG and SHIPDATE columns were compressed using run-length encoding, the LINENUM column was stored redundantly using uncompressed, RLE, and bit-vector encodings, and the QUANTITY column was left uncompressed.

We ran the two queries on these data. First, we ran the selection query from Section 5.3.5:

```
SELECT SHIPDATE, LINENUM  FROM LINEITEM
WHERE SHIPDATE < X AND LINENUM < Y
```

where *X* and *Y* are both constants. Second, we ran an aggregation version of this query:

```
SELECT SHIPDATE, SUM(LINENUM) FROM LINEITEM
WHERE SHIPDATE < X AND LINENUM < Y
GROUP BY SHIPDATE
```

again with *X* and *Y* as constants. While these queries are simpler than those that one would expect to see in a production environment, their simplicity aids in distilling the essential differences in performance between the materialization strategies. We consider joins separately in Section 5.4.3.

To explore the performance of the strategies as a function of the selectivity of the query, we varied *X* across the entire shipdate domain and kept Y constant at 7 (96% selectivity). In other experiments we varied Y and kept X constant and observed similar results (unless otherwise stated).

Additionally, at each point in this sample space, we varied the encoding of the LINENUM column among uncompressed, RLE, and bit-vector encodings (SHIPDATE was always RLE encoded). We experimented with the four different query plans described in Section 5.3.5: *EM-pipelined, EM-parallel, LM-pipelined, and LM-parallel.* Both LM strategies were implemented using the multi-column optimization.

Experiments were run on a Dell Optiplex GX620 DT with a 3.8 GHz Intel Pentium 4 processor 670 with Hyper-Threading, 2MB of cache, and a 800 MHz FSB. The system had 4GB of main memory installed, of which 3.5GB were available to the database. The hard drive used was a 250GB Western Digital WD2500JS-75N.

5.4.1   Simple Selection Query

For this set of experiments, we consider the simple selection query presented both in Section 5.3.5 and in the introduction to this section above. Figures 5-11 (a) and (b) show the total end-to-end query time for the four materialization strategies when the LINENUM column is stored uncompressed and RLE encoded, respectively.

For the uncompressed LINENUM experiment (Figure 5-11 (a)), the pipelined strategies are the clear winners at low selectivities. The pipelined algorithm reduces the I/O and CPU costs of reading in and applying the predicate to the large (250 MB) uncompressed LINENUM column because the first predicate is highly selective and the matching tuples are sufficiently localized (since the SHIPDATE column is secondarily sorted) that entire LINENUM blocks can be completely skipped. At high selectivities, however, pipelined strategies perform poorly since the CPU cost of jumping to each matching position is more expensive than simply iterating through the block one position at a time, and this additional CPU cost eventually dominates query time. At high selectivities, immediately making complete
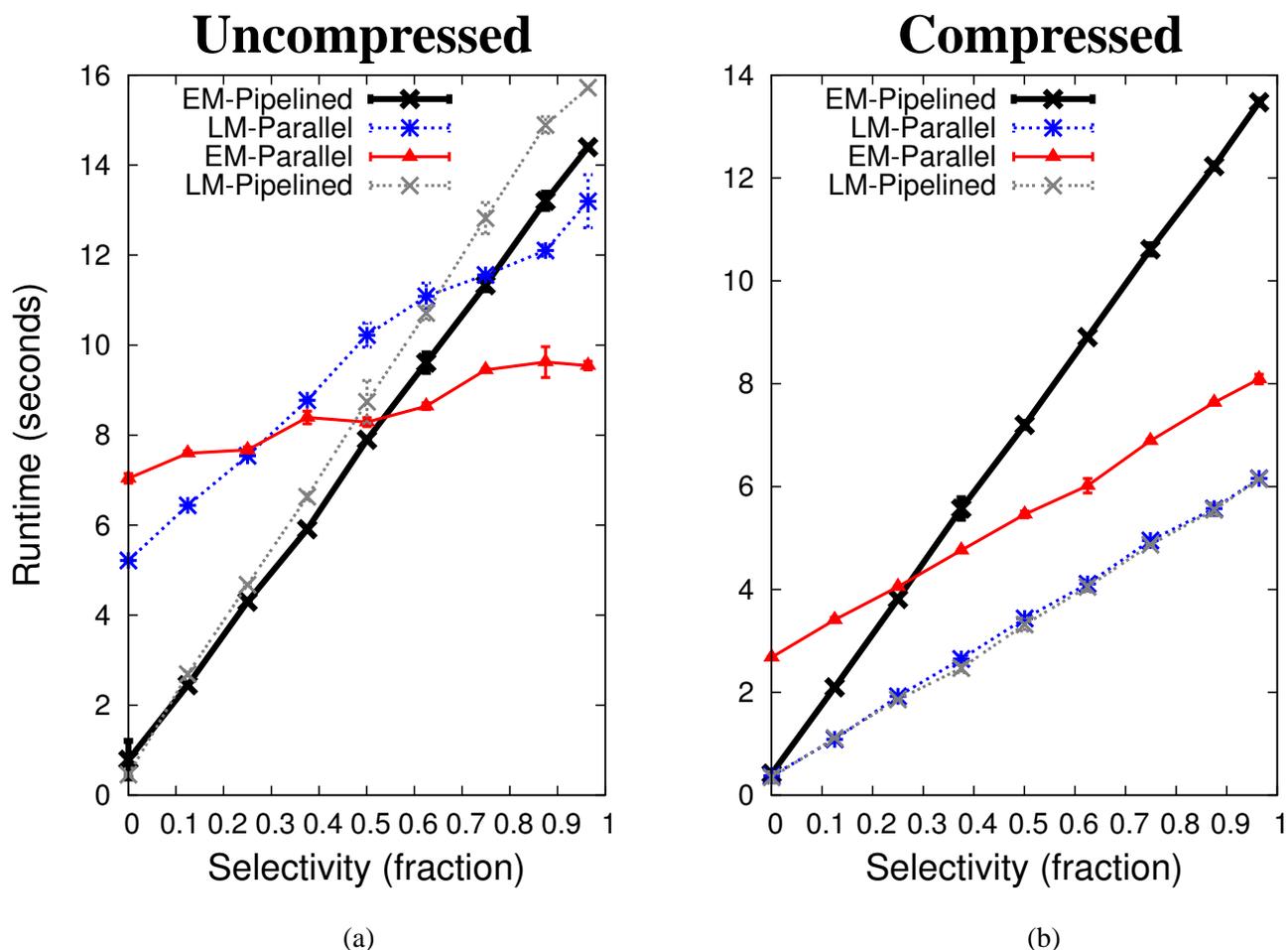
Figure 5-11: Run-times for four materialization strategies on selection queries with uncompressed (a) and RLE compressed (b) LINENUM column.

tuples at the bottom of the query plan (EM-parallel) is the best option; almost all tuples will need to be materialized, and EM-parallel has the lowest per-tuple construction cost of any of the strategies.

In other experiments, we varied the LINENUM predicate across the LINENUM domain and observed that if both the LINENUM and the SHIPDATE predicate have medium selectivities, LM-parallel can beat EM-parallel (this is due to the LM advantage of waiting until the end of the query to construct tuples and thus it can avoid creating tuples that will ultimately not be output).

For the RLE-compressed LINENUM experiment (Figure 5-11 (b)), the I/O cost for all materialization strategies is negligible (the RLE encoded LINENUM column occupies only seven 64k blocks on disk). At low query selectivities, the CPU cost is also low for all strategies. However, as the query selectivity increases, we observe the difference in costs of the strategies. Both EM strategies under-perform the LM strategies since tuples are constructed at the beginning of the query plan and tuple construction requires the RLE-compressed data to be decompressed (Section 5.2.1), precluding the performance advantages of operating directly on compressed data discussed in Chapter 4. In fact, the CPU cost of operating directly on compressed data is so small that almost the entire query time for the LM strategies is the construction of tuples and subsequent iteration over the results; hence both LM strategies perform similarly.

We also ran experiments when the LINENUM column column was bit-vector compressed. The dominant cost factor for these sets of experiments was decompression, so EM and LM performed similarly.

80
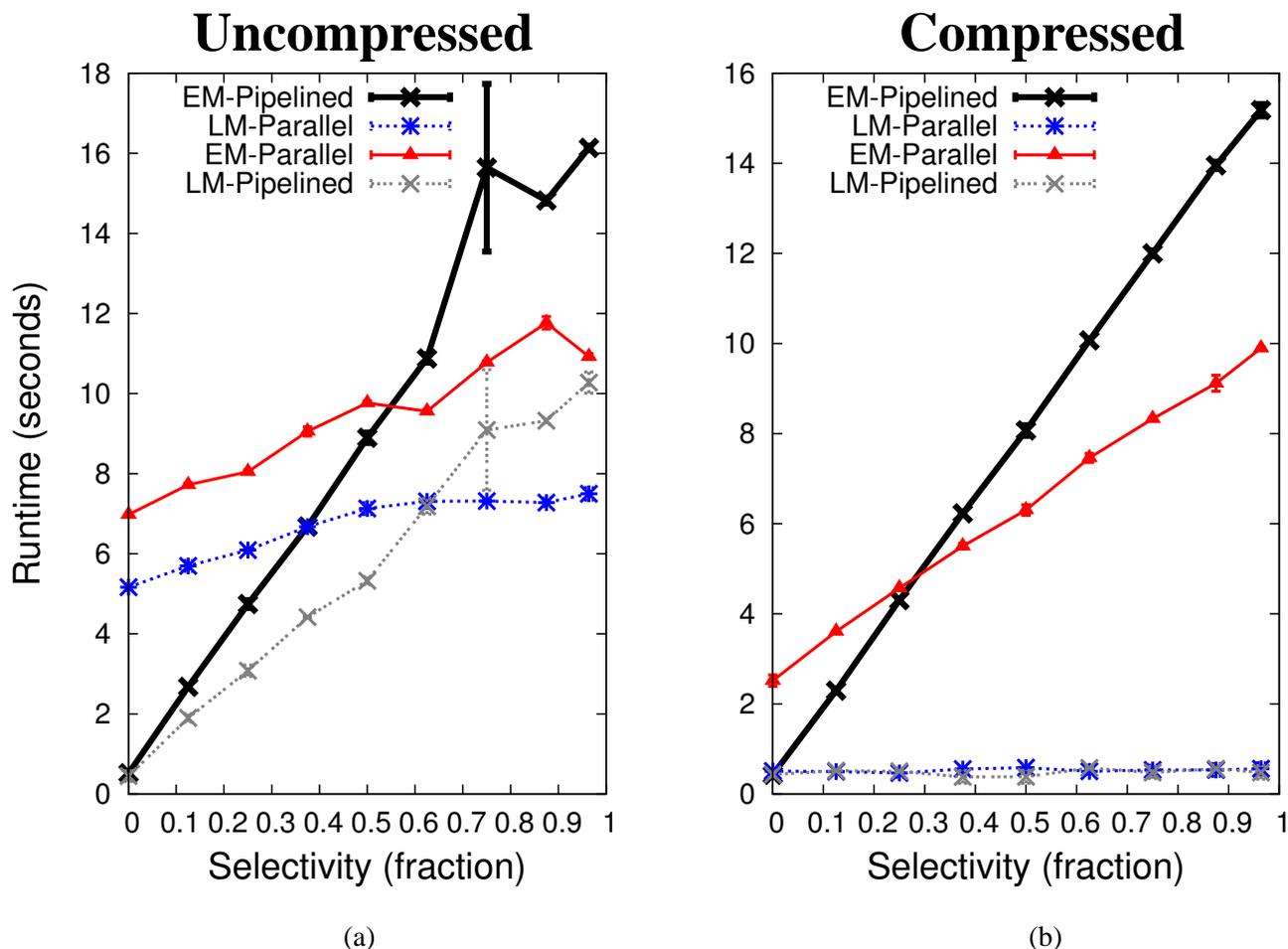
Figure 5-12: Run-times for four materialization strategies on aggregation queries with uncompressed (a) and RLE compressed (b) LINENUM column.

5.4.2 Aggregation Queries

For this set of experiments, we consider adding an aggregation operator on top of the selection query plan (the full query is presented in the introduction to this section above). Figure 5-12 (a) and (b) shows the total end-to-end query time for the four materialization strategies when the LINENUM column is stored uncompressed and RLE encoded, respectively.

In each of these graphs, the EM strategies perform similarly to their counterparts in Figure 5-11: the CPU cost of producing and iterating through the early-materialized tuples easily dominates the cost of producing and iterating through the aggregate result. By contrast, the LM strategies all perform significantly better than before. In both the uncompressed and compressed cases (Figure 5-12(a) and (b), respectively), delaying tuple construction allows the plan to construct only those few tuples produced as output by the aggregator and to skip constructing the tuples corresponding to its inputs. Additionally, in the compressed case, the cost of aggregation is also reduced because the aggregator is able to operate extremely efficiently on compressed data (as described in Chapter 4).

5.4.3 Joins

We now look at the effect of materialization strategy on join performance. If an early materialization strategy is used relative to a join, tuples have already been constructed before reaching the join operator, so the join functions as it would in a standard row-store system and outputs tuples. An alternative algorithm can be used with a late

materialization strategy, however. In this case, only the columns that compose the join predicate are input to the join. The output of the join is a set of pairs of positions in the two input relations for which the predicate succeeded. For example (this same example was presented in Chapter 4), the figure below shows the results of a join of a column of size 5 with a column of size 3.

$$
\begin{array}{c}
\boxed{42} \\
\boxed{36} \\
\boxed{42} \\
\boxed{44} \\
\boxed{38}
\end{array}
\;\bowtie\;
\begin{array}{c}
\boxed{38} \\
\boxed{42} \\
\boxed{46}
\end{array}
\;=\;
\begin{array}{cc}
\boxed{1}\;\boxed{2} \\
\boxed{3}\;\boxed{2} \\
\boxed{5}\;\boxed{1}
\end{array}
$$
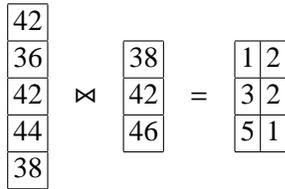
For many join algorithms, the output positions for the left (outer) input relation will be sorted while the output positions of the right (inner) input relation will not. This is because the positions in the left column are usually iterated through in order, while the right relation is probed for join predicate matches. This asymmetric nature of join positional output implies that restricting other columns from the left input relation using the join output positions will be relatively fast, since the standard merge join of positions can be used to extract column values. Restricting other columns from the right input relation using the join output positions can be significantly more expensive, however, as the out-of-order positions preclude the use of a merge-join on position to retrieve column values.

Of course, a hybrid approach could be used in which the right relation sends tuples to the join operator while the left relation sends only the single join predicate column. The join result would then be a set of tuples from the right relation and an ordered set of positions from the left relation; the positions from the left relation could easily be used to retrieve additional columns from that relation and complete the tuple construction process. This approach has the advantage of only materializing values in the left relation corresponding to tuples that pass the join predicate while avoiding the penalty of materializing values from the right relation using unordered positions.

Multi-columns provide another option for the representation of the right (inner) relations. All relevant columns (i.e., columns to be materialized after the join plus the predicate column) are input to the join operator as a multi-column. As inner table values match the join predicate, the position of the value is used to retrieve the values for other columns, and tuples are constructed on the fly. This hybrid technique is useful when the join selectivity is low and few tuples need to be constructed, but is otherwise expensive, since it potentially requires a particular tuple from the inner relation to be constructed multiple times.

To further examine the differences between these three materialization approaches for the inner table in a join operator (send just the unmaterialized join predicate column, send the unmaterialized relevant columns in a multi-column, or send materialized tuples), we ran a standard star schema join query on our TPC-H data between the orders table and the customers table on customer key (customer key is a foreign key in the orders table and the primary key for the customers table), where the less-than predicate on customer key is varied to obtain the desired selectivity:

```
SELECT Orders.shipdate
       Customer.nationcode
FROM   Orders, Customer
WHERE  Orders.custkey=Customer.custkey
   AND Orders.custkey < X
```

For TPC-H scale 10 data, the orders table contains 15,000,000 tuples and the customer table 1,500,000 tuples. Since this is a foreign key-primary key join, the join result will also have at most 15,000,000 tuples (the actual number is determined by the Orders predicate selectivity). The results of this experiment can be found in Figure 5-13. Sending either early materialized tuples or multi-columns as the right-side input of the join operator results in similar performance, as the multi-column advantage of only materializing relevant tuples is not helpful for a foreign key-primary key join where there are exactly as many join results as join inputs. Sending just the join predicate column performs poorly due to the overhead of subsequent materialization using unordered positions. If the entire set of positions were not able to be kept in memory, late materialization would have performed even more poorly.
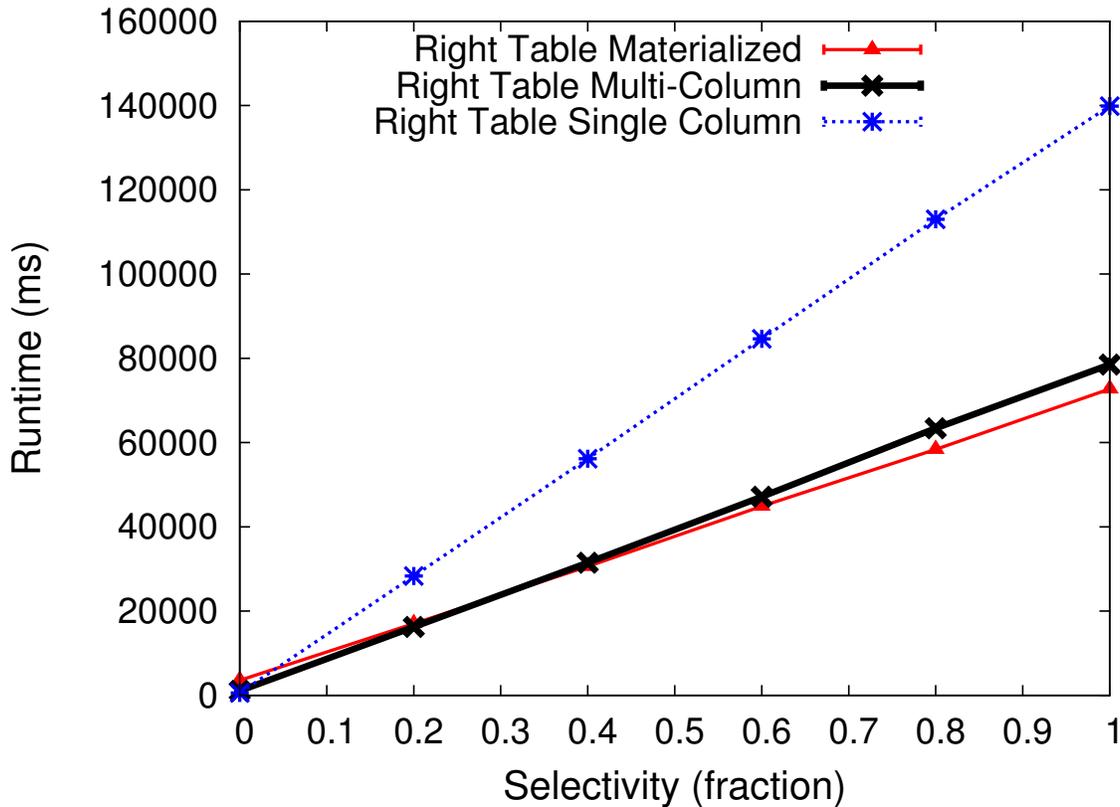
Figure 5-13: Run-times for three different materialization strategies for the inner table of a join query. Late materialization is used for the outer table.

We do not present results for varying the materialization strategy of the left-side input table to the join operator since the trade-offs are identical to those discussed in previous experiments: if the join is highly selective or if the join results will be aggregated, a late materialization strategy should be used. Otherwise, EM-parallel should be used.

5.5 Related Work

To the best of our knowledge, this chapter (excepting the companion paper published in ICDE [19]) contains the only study of multiple tuple creation strategies in a column-oriented system. C-Store used LM-parallel only (until we extended it with additional strategies). Published descriptions of Sybase IQ [51] seem to indicate that it also uses LM-parallel. Papers by Halverson et al. [42] and Harizopoulos et al. [43] that further explore the trade-offs between row- and column-stores use early materialization approaches for the column-store they implemented (the former uses EM-parallel, the latter uses EM-pipelined). MonetDB/X100 [28] uses late materialization implemented using a similar multi-column approach; however, their version of position descriptors (they call them *selection vectors*) is kept separately from column values and data is decompressed in the cache, precluding the potential performance benefits of operating directly on compressed data both on position descriptors and on column values. Thus, previous work has tended to choose a materialization strategy a priori without justification and has not examined trade-offs

between these choices.

The multi-column optimization of combining chunks of columns covering the same position range together into one data structure is similar to the PAX [21] idea of taking a row-store page and splitting it into multiple *mini-pages* where each tuple attribute is stored contiguously. PAX does this to improve cache performance by maximizing inter-record spatial locality within a page. Multi-columns build on PAX in the following ways: first, multi-columns are an in-memory data structure only and are created on the fly from different columns stored separately on disk (where pages for the different columns on disk do not necessarily match-up position-wise). Second, positions are first class citizens in multi-columns and may be accessed and processed separately from attribute values. Finally, mini-columns are kept compressed inside multi-columns in their native compression format throughout the query plan, encapsulated in specialized data structures that facilitate direct operation on compressed data.

5.6   Conclusion

The optimal point at which to perform tuple construction in a column-oriented database is not obvious. This chapter provides a systematic evaluation of a variety of strategies for when tuple construction should occur. We showed that late materialization has many advantages, but potentially incurs additional costs due to re-processing disk blocks, and hence early materialization is sometimes preferable. A good heuristic to use is that if output data is aggregated, or if the query has low selectivity (highly selective predicates), or if input data is compressed using a light-weight compression technique, a late materialization strategy should be used. Otherwise, for high selectivity, non-aggregated, non-compressed data, early materialization should be used. Further, the right input table to a join should be materialized before (or during if a multi-column is input) the join operation. For queries of low selectivity, pipelined query plans should be chosen over parallel plans.

We are also optimistic that our analytical model can be used to predict query performance and help choose a materialization strategy at query planning and optimization time. An interesting avenue of future work would be to allow the system to detect and dynamically switch materialization strategies if the heuristics or analytical model suggest a poor initial strategy.

As we saw in Section 5.4.3, joins present significant materialization complications. We revisit this problem in the next chapter.

Chapter 6

The Invisible Join

In this chapter, we describe the invisible join, a join algorithm designed for data warehouses organized in a star schema. We show that the invisible join can outperform traditional joins by a factor of 3, and that in some cases, joining tables using this technique can outperform queries over tables where joins have already been performed in advance.

6.1 Introduction

Queries over data warehouses, particularly over data warehouses modeled with a star schema, often have the following structure: Restrict the set of tuples in the fact table using selection predicates on one (or many) dimension tables. Then, perform some aggregation on the restricted fact table, often grouping by other dimension table attributes. Thus, joins between the fact table and dimension tables need to be performed for each selection predicate and for each aggregate grouping. A good example of this is Query 3.1 from the Star Schema Benchmark.

```
SELECT c_nation, s_nation, d_year,
       sum(lo_revenue) as revenue
FROM customer, lineorder, supplier, dwdate
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_orderdate = d_datekey
  AND c_region = 'ASIA'
  AND s_region = 'ASIA'
  AND d_year >= 1992 and d_year <= 1997
GROUP BY c_nation, s_nation, d_year
ORDER BY d_year asc, revenue desc;
```

This query finds the total revenue from customers who live in Asia and who purchase a product supplied by an Asian supplier between the years 1992 and 1997 grouped by each unique combination of the nation of the customer, the nation of the supplier, and the year of the transaction.

The traditional plan for executing these types of queries is to pipeline joins in order of predicate selectivity. For example, if c_region = 'ASIA' is the most selective predicate, the join on custkey between the lineorder and customer tables is performed first, filtering the lineorder table so that only orders from customers who live in Asia remain. As this join is performed, the nation of these customers are added to the joined customer-order table. These results are pipelined into a join with the supplier table where the s_region = 'ASIA' predicate is applied and s_nation extracted, followed by a join with the data table and the year predicate applied. The results of these joins are then grouped and aggregated and the results sorted according to the ORDER BY clause.

An alternative to the traditional plan is the late materialized join technique described in the previous chapter. In this case, a predicate is applied on the `c_region` column (`c_region = 'ASIA'`), and the customer key of the customer table is extracted at the positions that matched the predicate. These keys are then joined with the customer key column from the fact table to get a set of positions from the fact table and from the customer table of matching tuples. Values from the country column at this set of positions are then extracted, along with values from the other fact table columns (supplier key, order date, and revenue). Similar joins are then performed with the supplier and date tables.

Each of these plans have a set of disadvantages. In the first (traditional) case, constructing tuples before the join precludes all of the late materialization benefits described in the previous chapter. In the second case, values from dimension table group-by columns need to be extracted in out-of-position order, as was described in Section 5.4.3, which can have significant cost.

In this chapter, we introduce an alternative to these plans. We describe a technique we call the *invisible join* that can be used in column-oriented databases for foreign-key/primary-key joins on star schema style tables. It is a late materialized join, but minimizes the values that need to be extracted out-of-order, thus alleviating both sets of disadvantages described above. It works by rewriting joins into predicates on the foreign key columns in the fact table. These predicates can be evaluated either by using a hash lookup (in which case a hash join is simulated), or by using other more advanced techniques that we discuss later.

By rewriting the joins as selection predicates on fact table columns, they can be executed at the same time as other selection predicates that are being applied to the fact table, and any of the predicate application algorithms described in Chapter 5 can be used. For example, each predicate can be applied in parallel and the results merged together using fast bit-map operations. Alternatively, the results of a predicate application can be pipelined into another predicate application to reduce the number of times the second predicate must be applied. Once all predicates have been applied, the appropriate tuples can be extracted from the relevant dimensions (this can also be done in parallel). By waiting until all predicates have been applied before doing the extraction, the number of out-of-order extractions is minimized.

The invisible join extends previous work on improving performance for star schema joins [54, 66] that are reminiscent of semijoins [26] by taking advantage of the column-oriented layout, and rewriting predicates to avoid hash-lookups, as described below.

6.2   Join Details

The invisible join performs joins in three phases. First, each predicate is applied to the appropriate dimension table to extract a list of dimension table keys that satisfy the predicate. These keys are used to build a hash table that can be used to test whether a particular key value satisfies the predicate (the hash table should easily fit in memory since dimension tables are typically small and the table contains only keys). An example of the execution of this first phase for the above query on some sample data is displayed in Figure 6-1.

In the next phase, each hash table is used to extract the positions of records in the fact table that satisfy the corresponding predicate. This is done by probing into the hash table with each value in the foreign key column of the fact table, creating a list of all the positions in the fact table that satisfy the predicate. An example of the execution of this second phase is displayed in Figure 6-2. Note that a position list may be an explicit list of positions, or a bitmap as shown in the example.

The third phase of the join occurs once all of the predicates on each foreign key in the fact table have been applied. First, the position lists from all of the predicates are intersected to generate a list of satisfying positions P in the fact table. Then, for each column C in the fact table containing a foreign key reference to a dimension table that is needed to answer the query (e.g., where the dimension column is referenced in the select list, group by, or aggregate clauses), foreign key values from C are extracted using P and are looked up in the corresponding dimension table. Note that if the dimension table key is a sorted, contiguous list of identifiers starting from 1 (which is the common case), then the foreign key actually represents the position of the desired tuple in dimension table. This means that

Apply "region = 'Asia'" On Customer Table

custkey	region	nation	...
1	ASIA	CHINA	...
2	EUROPE	FRANCE	...
3	ASIA	INDIA	...

→ Hash Table Containing
Keys 1 and 3

Apply "region = 'Asia'" On Supplier Table

suppkey	region	nation	...
1	ASIA	RUSSIA	...
2	EUROPE	SPAIN	...

→ Hash Table Containing
Key 1

Apply "year in [1992,1997]" On Date Table

dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

→ Hash Table Containing
Keys 01011997, 01021997,
and 01031997

Figure 6-1: The first phase of the joins needed to execute Query 7 from the Star Schema benchmark on some sample data

the needed dimension table columns can be extracted directly using this position list (and this is simply a fast array look-up). This is the reason why this join does not suffer from the pitfalls of the late materialized join approaches from Chapter 5 where this final position list extraction is very expensive. An example of the execution of this third phase is displayed in Figure 6-3. Note that for the date table, the key column is not a sorted, contiguous list of identifiers starting from 1, so a full join must be performed (rather than just a position extraction).

Although the lookups in this third phase are typically randomly distributed across the "inner" dimension table, since dimension tables are small, the column being looked up can often fit inside the L2 cache, so this random access is not expensive. Note that since this is a foreign-key primary-key join, and since all predicates have already been applied, there is guaranteed to be one and only one result in each dimension table for each position in the unioned position list. This means that there are the same number of results for each dimension table join from this third phase, so each join can be done separately and the results combined (stitched together) at a later point in the query plan.

As described thus far, this algorithm is simply another way of thinking about a column-oriented semijoin or a late materialized hash join. Even though the hash part of the join is expressed as a predicate on a fact table column, practically there is little difference between the way the predicate is applied and the way a (late materialization) hash join is executed. The advantage of expressing the join as a predicate comes into play in the surprisingly common case (for star schema joins) where the set of keys in dimension table that remain after a predicate has been applied are contiguous. When this is the case, a technique we call "between predicate rewriting" can be used, where the predicate can be rewritten from a hash-lookup predicate on the fact table to a "between" predicate where the foreign key falls between two ends of the key range. For example, if the contiguous set of keys that are valid after a predicate has been applied are keys 1000-2000, then instead of inserting each of these keys into a hash table and probing the hash table for each foreign key value in the fact table, we can simply check to see if the foreign key is in between 1000 and 2000. If so, then the tuple joins; otherwise it does not. Between predicates are faster to execute for obvious reasons as they can be evaluated directly without looking anything up.

Original Fact Table

orderkey	custkey	suppkey	orderdate	revenue
1	3	1	01011997	43256
2	3	2	01011997	33333
3	2	1	01021997	12121
4	1	1	01021997	23233
5	2	2	01021997	45456
6	1	2	01031997	43251
7	3	2	01031997	34235

$$
\begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \& \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \& \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}
$$

Hash Table Containing Keys 1 and 3

+

custkey	
3	1
3	1
2	0
1	1
2	0
1	1
3	1

Hash Table Containing Key 1

+

suppkey	
1	1
2	0
1	1
1	1
2	0
2	0
2	0

Hash Table Containing Keys 01011997, 01021997, and 01031997

+

orderdate	
01011997	1
01011997	1
01021997	1
01021997	1
01021997	1
01031997	1
01031997	1

Figure 6-2: The second phase of the joins needed to execute Query 7 from the Star Schema benchmark on some sample data

The ability to apply this optimization hinges on the set of these valid dimension table keys being contiguous. In many instances, this property does not hold. For example, a range predicate on a non-sorted field results in non-contiguous result positions. And even for predicates on sorted fields, the process of sorting the dimension table by that attribute likely reordered the primary keys so they are no longer an ordered, contiguous set of identifiers. However, the latter concern can be easily alleviated through the use of dictionary encoding for the purpose of key reassignment (rather than compression). Since the keys are unique, dictionary encoding the column results in the dictionary keys being an ordered, contiguous list starting from 0. As long as the fact table foreign key column is encoded using the same dictionary table, the hash-table to between-predicate rewriting can be performed.

Further, the assertion that the optimization works only on predicates on the sorted column of a dimension table is not entirely true. In fact, dimension tables in data warehouses often contain sets of attributes of increasingly finer granularity. For example, the date table in SSBM has a year column, a yearmonth column, and the complete date column. If the table is sorted by year, secondarily sorted by yearmonth, and tertiarily sorted by the complete date, then equality predicates on any of those three columns will result in a contiguous set of results (or a range predicate on the sorted column). As another example, the supplier table has a region column, a nation column, and a city column (a region has many nations and a nation has many cities). Again, sorting from left-to-right will result in predicates on any of those three columns producing a contiguous range output. Data warehouse queries often access these columns, due to the OLAP practice of rolling-up data in successive queries (tell me profit by region, tell me profit by nation, tell me profit by city). Thus, "between predicate rewriting" can be used more often than one might initially expect, and (as we show in the next section), often yields a significant performance gain.

Note that predicate rewriting does not require changes to the query optimizer to detect when this optimization can be used. The code that evaluates predicates against the dimension table is capable of detecting whether the result set is contiguous. If so, the fact table predicate is rewritten at run-time.
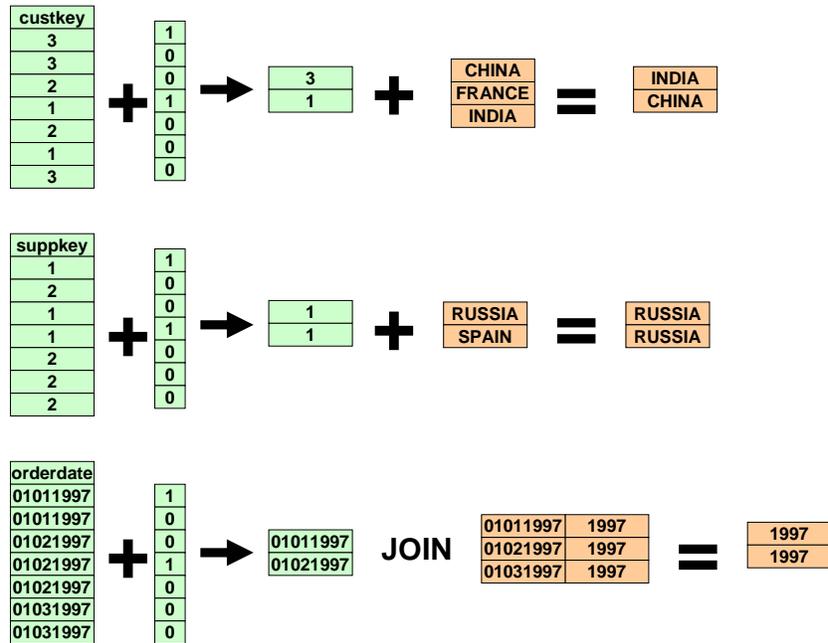
Figure 6-3: The third phase of the joins needed to execute Query 7 from the Star Schema benchmark on some sample data

6.3 Experiments

In order to understand the performance benefits of the invisible join relative to other join algorithms discussed in this dissertation (namely the late materialized join and the early materialized join), we ran some experiments on our C-Store implementation.

All of our experiments were run on a 2.8 GHz single processor, dual core Pentium(R) D workstation with 3 GB of RAM running RedHat Enterprise Linux 5. The machine has a 4-disk array, managed as a single logical volume with files striped across it. Typical I/O throughput is 40 - 50 MB/sec/disk, or 160 - 200 MB/sec in aggregate for striped files. The numbers we report are the average of several runs, and are based on a "warm" buffer pool (in practice, we found that this yielded about a 30% performance increase; the gain is not particularly dramatic because the amount of data read by each query exceeds the size of the buffer pool).

Section 6.3.1 presents the results of these experiments. Section 6.3.2 then discusses the implications of fast join performance (using the invisible join) on schema design.

6.3.1 Comparison of Invisible Join to Other Join Techniques

Given that the invisible join is designed for star schema joins, we used the same "Star Schema Benchmark" (SSBM) we used in Chapter 2 to evaluate performance. We implemented the invisible join (with the "between predicate rewriting" optimization) in our C-Store prototype, and rewrote the hand-coded query plans to use the invisible join instead of the early materialized join used for the experiments in Chapter 2. We allowed compression to be used for these experiments since this is the more realistic scenario (compression was not used in Chapter 2 since we were exploring bare-bones approaches to building a column-store – this is the only reason why the early materialized join results here are different than the "approach 3" results in Chapter 2 – everything else is identical).

There are many cases in the SSBM where the between predicate rewriting optimization can be used. In the
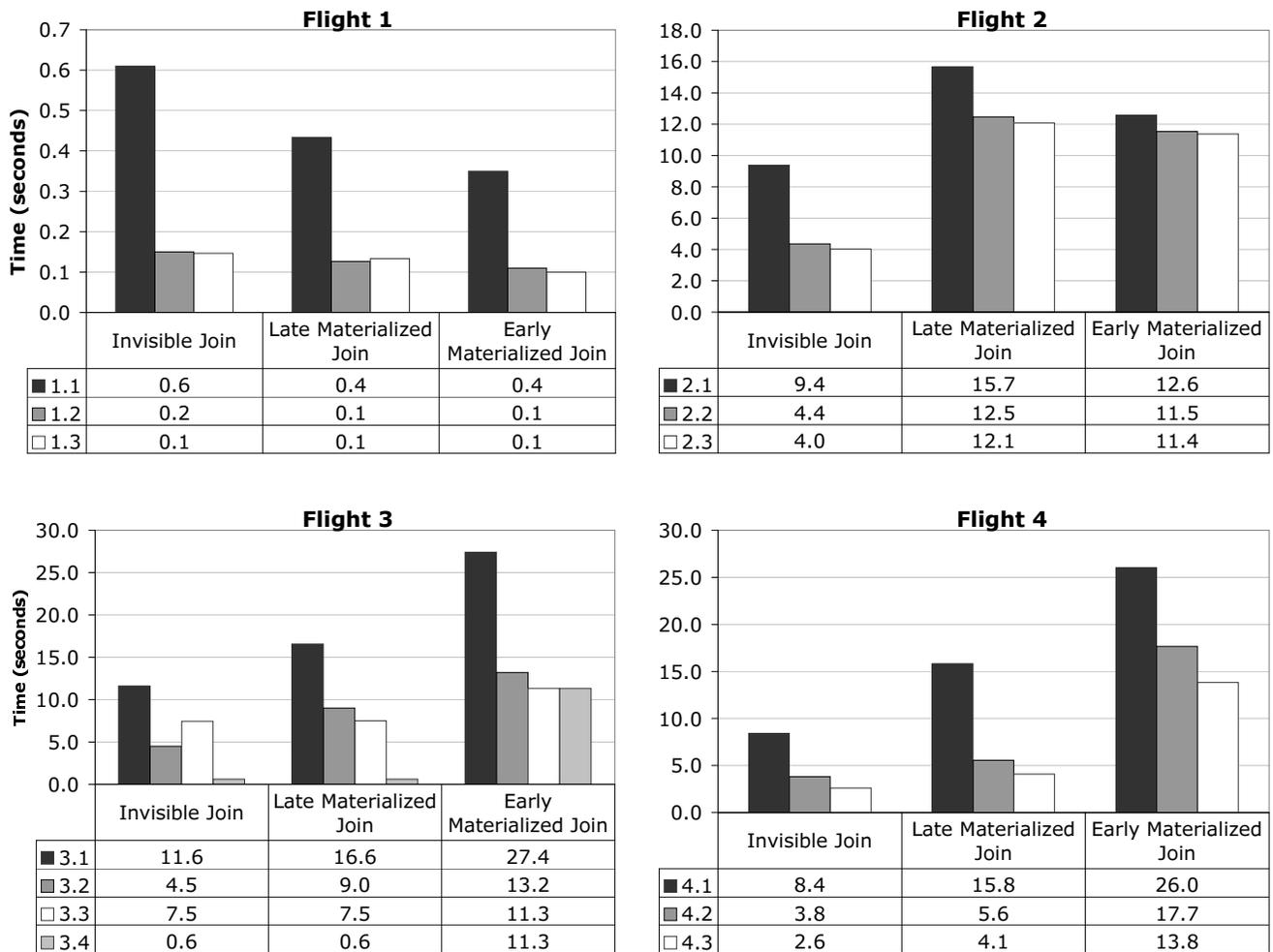
Figure 6-4: Performance numbers for different join variants by query flight.

supplier table, the region, nation, and city columns are attributes of increasingly finer granularity, which, as described above, result in contiguous positional results sets from equality predicate application on any of these columns. The customer table has a similar region, nation, and city column trio. The part table has mfgr, category, and brand as attributes of increasingly finer granularity. Finally, the date table has year, month, and day increasing in granularity. Every query in the SSBM contain one or more joins (all but the first query flight contains more than one join), and for each query, at least one of the joins is with a dimension table that had a predicate on one of these special types of attributes. Hence, it was possible to use the between predicate rewriting optimization at least once per query.

To better understand the impact of the between predicate writing optimization, we also coded the query plans to use late materialization (without the optimization) as a third possibility. Hence, we ran the SSBM under three cases: the full invisible join, the late materialized join, and the early materialized join. The results per query are displayed in Figure 6-4 and the average results are displayed in Figure 6-5.

For query flight 1, all three algorithms perform in under a second due to the fast and selective predicate applications on run-length encoded columns. In all three cases, the join can operate directly on RLE compressed data, so the differences between them are insignificant.

For query flight 2, the late materialization join is outperformed by the early materialization join. This is explained as follows. Since the join with the date table is the most common join in this workload, the fact table is sorted by orderdate (a foreign key to the date table) to improve performance of joins of this type. Since the table is sorted
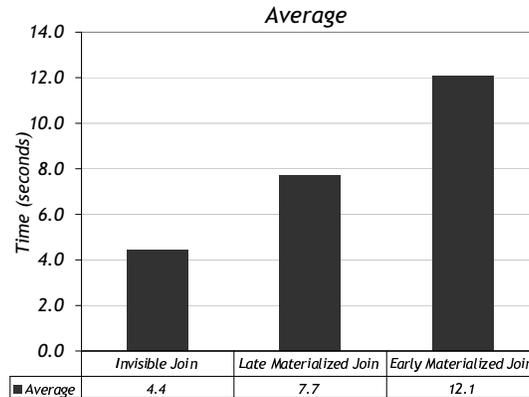
Figure 6-5: Average performance numbers across all queries in the SSBM for different join variants.

on orderdate, the column is run-length encoded. The late materialized join can take advantage of the run-length encoded column and can operate on it directly, as described in Chapters 4 and 5. However, the early materialized join must decompress the column to perform tuple reconstruction. However, flight 2 is the only flight that does not contain a predicate on the date table. Thus, this key advantage of the late materialized join is negated. Thus, all that is left is the disadvantage of the late materialized join in the need to reconstruct tuples out of order after the join. The invisible join, because of between-predicate-rewriting, outperforms the other two algorithms by approximately a factor of 2.

For the other two query flights, the late materialized join outperforms the early materialized join as a result of the advantage of operating on direct data discussed above, and the out-of-order tuple reconstruction cost is not large since the inner dimension tables easily fit in cache, and the queries are sufficiently selective so that only a small percentage of tuples need to be constructed. The invisible join, again because of between-predicate-rewriting, outperforms the other two algorithms.

6.3.2 Implications of Join Performance

In profiling the code, we noticed that in the baseline C-Store case, performance is dominated in the lower parts of the query plan (predicate application) and that the invisible join technique made join performance relatively cheap. In order to explore this observation further we created a denormalized version of the fact table where the fact table and its dimension table are prejoined such that instead of containing a foreign key into the dimension table, the fact table contains all of the values found in the dimension table repeated for each fact table record (e.g., all customer information is contained in each fact table tuple corresponding to a purchase made by that customer). Clearly, this complete denormalization would be more detrimental from a performance perspective in a row-store since this would significantly widen the table. However, in a column-store, one might think this would speed up read-only queries since only those columns relevant for a query need to read in, and joins would be avoided.

Surprisingly, we found this often not to be the case. Figure 6-6 compares the baseline C-Store performance from the previous section (using the invisible join) with the average performance of SSBM on three versions of the single denormalized table where joins do not have to be performed (detailed, by-flight results are shown in Figure 6-7). In the first case, complete strings like customer region and customer nation are included unmodified in the denormalized table. This case performs a factor of 5 worse than the base case. This is because the invisible join converts predicates on dimension table attributes into predicates on fact table foreign key values. When the table is denormalized, predicate application is performed on the actual string attribute in the fact table. In both cases, this predicate application is the dominant step. However, a predicate on the integer foreign key can be performed faster than a predicate on a string attribute since the integer attribute is smaller.

91
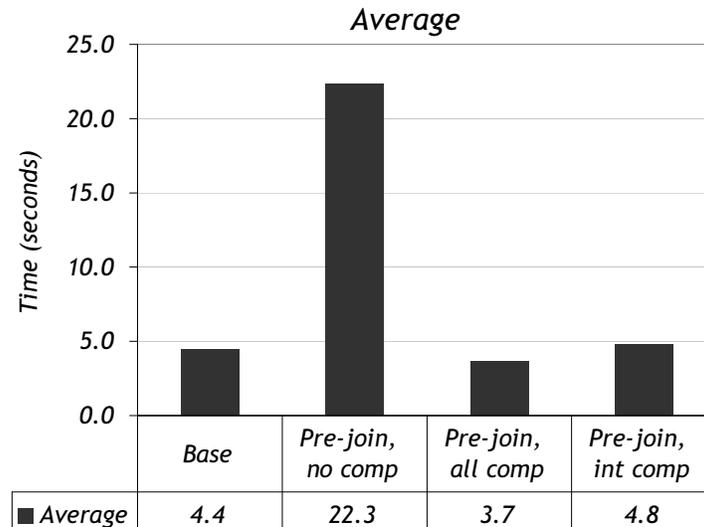
Figure 6-6: Comparison of performance of baseline C-Store on the original SSBM schema with a denormalized version of the schema, averaged across all queries. Denormalized columns are either not compressed, dictionary compressed into integers, or compressed as much as possible.

Of course, the string attributes could have easily been dictionary encoded into integers before denormalization. When we did this (the int-comp case in Figures 6-6 and 6-7), the performance difference between the baseline and the denormalized cases became much smaller. Nonetheless, for quite a few queries, the baseline case still performed faster. The reasons for this are twofold. First, some SSBM queries have two predicates on the same dimension table. The invisible join technique is able to summarize the result of this double predicate application as a single predicate on the foreign key attribute. However, for the denormalized case, the predicate must be be completely applied to both columns in the fact table (remember that for data warehouses, fact tables are generally much larger than dimension tables).

Second, many queries have a predicate on one attribute in a dimension table and group by a different attribute. For the invisible join, this requires iteration through the foreign key column once to apply the predicate, and again (after all predicates from all tables have been applied and intersected) to extract the group-by attribute. But since C-Store uses pipelined execution, blocks from the foreign key column will still be in memory upon the second access. In the denormalized case the predicate column and the group-by column are separate columns in the fact table and both must be iterated through, doubling the necessary I/O.

In fact, many of the SSBM dimension table columns that are accessed in the queries have low cardinality, can be compressed into values that are smaller than the integer foreign keys. When using complete C-Store compression, we found that the denormalization technique was useful more often (shown as the all-comp case in Figures 6-6 and 6-7).

These results have interesting implications. Denormalization has long been used as a technique in database systems to improve query performance, by reducing the number of joins that must be performed at query time. In general, the school of wisdom teaches that denormalization trades query performance for making a table wider, and more redundant (increasing the size of the table on disk and increasing the risk of update anomalies). One might expect that this tradeoff would be more favorable in column-stores (denormalization should be used more often) since one of the disadvantages of denormalization (making the table wider) is not problematic when using a column-oriented layout. However, these results show the exact opposite: denormalization is actually not very useful in column-stores (at least for star schemas). This is because the invisible join performs so well that reducing the number of joins via denormalization provides an insignificant benefit. In fact, denormalization only appears

Flight 1

	Base	Pre-join, no comp	Pre-join, all comp	Pre-join, int comp
■ 1.1	0.6	0.5	0.8	0.3
■ 1.2	0.2	0.1	0.2	0.1
□ 1.3	0.1	0.2	0.2	0.1

Flight 2

	Base	Pre-join, no comp	Pre-join, all comp	Pre-join, int comp
■ 2.1	9.4	33.0	6.9	11.5
■ 2.2	4.4	24.5	2.3	3.0
□ 2.3	4.0	12.2	1.9	2.7

Flight 3

	Base	Pre-join, no comp	Pre-join, all comp	Pre-join, int comp
■ 3.1	11.6	43.6	7.4	11.8
■ 3.2	4.5	44.2	3.7	8.5
□ 3.3	7.5	33.5	7.3	5.6
▨ 3.4	0.6	30.5	6.5	4.2

Flight 4

	Base	Pre-join, no comp	Pre-join, all comp	Pre-join, int comp
■ 4.1	8.4	51.8	7.5	10.3
■ 4.2	3.8	9.2	1.8	2.6
□ 4.3	2.6	6.8	1.1	1.5

Figure 6-7: Detailed performance by SSBM flight for the denormalized strategies in 6-6.

to be useful when the dimension table attributes included in the fact table are sorted (or secondarily sorted) or are otherwise highly compressible.

6.4   Conclusion

In this chapter, we introduced a new join algorithm, the invisible join, designed for data warehouses organized in a star schema, that is designed for the column-oriented data layout of column-stores, and can outperform traditional early materialized joins by a factor of 3. We show that the invisible join is so fast, that in some cases, joining tables using this technique can outperform the case where the tables have already been joined in advance. This observation results in interesting implications on data warehouse schema design: Denormalization, an important but expensive (in space requirements) and complicated (in deciding in advance what tables to denormalize) performance enhancing technique used in data warehouses implemented using row-store DBMS technology, is not necessary in data warehouses implemented using column-store DBMS technology (or can be used with greatly reduced cost and complexity).

Chapter 7

Putting It All Together: Performance On The Star Schema Benchmark

7.1   Introduction

In the previous three chapters of this dissertation, we have examined in detail the performance consequences of three important issues in column-oriented databases: data compression, tuple construction, and the invisible join. Combined, these three techniques can result in dramatic performance improvements relative to naive column-oriented database design and to row-oriented databases. By compressing data, less data needs to be read off disk, saving I/O time. Further, if compressed data can be operated on directly, CPU cycles can also be saved. By constructing tuples using late materialization, only required tuples need be constructed, and fast, column-oriented, vectorized operators can be used. By using the invisible join for star schema joins, the late materialization technique can be applied to joins as well.

So far, each of these performance enhancing techniques were evaluated independently. Hence, in the next two chapters, we put them all together and measure performance of the complete C-Store system. In this chapter, we revisit the Star Schema Benchmark (SSBM) of Chapters 2 and 6 and evaluate performance on the application known to be well-suited for column-stores: data warehousing. In the next chapter, we will evaluate performance on a new application for column-store database systems: the Semantic Web.

Since we have already run the SSBM using a commercial row-store for the numbers presented in Chapter 2, we begin by comparing the complete C-Store system with these row-store numbers in order to add yet another data point in the work on comparing the performance difference of row-stores and column-stores on data warehouse workloads (as we pointed out in Chapter 2, quite a few other data points can be found in the literature [28, 43, 63, 58, 49, 42, 8], but it is important to keep in mind the categorization described in Chapter 2 to distinguish the approaches these publications used in building their respective column-stores).

After making this comparison, we explore in detail how the different performance enhancing techniques discussed in this dissertation contribute to C-Store's performance relative to the row-store. We do this by creating different variants of the C-Store database by removing these techniques one-by-one (in effect, making the C-Store query executor behave more like a row-store and moving from approach 3 to approach 2 from Chapter 2). We carefully measure the performance of these different variants, breaking down the factors responsible for C-Store's good performance. We find that compression can offer order-of-magnitude gains when it is possible, but that the benefits are less substantial in other cases, whereas late materialization offers about a factor of 3 performance gain across the board. Other optimizations offer about a factor 1.5 performance gain on average.

7.2 Review of Performance Enhancing Techniques

In this section, we review the four techniques that we will evaluate in Section 7.3 for their contribution in query execution to improve performance of column-stores. Three of them are discussed in more detail in previous chapters of this dissertation, while the fourth, block iteration, is also an important performance enhancing technique, and is discussed elsewhere [76].

7.2.1 Compression

We showed in Chapter 4 that compressing data using column-oriented compression algorithms and keeping data in this compressed format as it is operated upon can improve query performance by up to an order of magnitude. Compression improves performance (in addition to reducing disk space) since if data is compressed, then less time must be spent in I/O as data is read from disk into memory (or from memory to CPU). In fact, compression can improve query performance beyond simply saving on I/O. If a column-oriented query executor can operate directly on compressed data, decompression can be avoided completely, and, in some cases, multiple values within a column can be operated on at once.

Chapter 4 concluded that the biggest difference between compression in a row-store and compression in a column-store are the cases where a column is sorted (or secondarily sorted) and there are consecutive repeats of the same value in a column. In a column-store, it is extremely easy to summarize these value repeats and operate directly on this summary. In a row-store, the surrounding data from other attributes significantly complicates this process. Thus, in general, compression will have a larger impact on query performance if a high percentage of the columns accessed by that query have some level of order. For the SSBM benchmark we use in this chapter, we do not store multiple copies of the fact table in different sort orders, and so only one of the seventeen columns in the fact table can be sorted (and two others secondarily sorted) so we expect compression to have a somewhat smaller (and more variable per query) effect on performance than it could if more aggressive redundancy was used.

7.2.2 Late Materialization

As described in Chapter 5, late materialization refers to the process of the query executor waiting to perform (partial) tuple reconstruction of the attributes accessed by a query. This is in contrast to "early materialization" where tuple reconstruction occurs at the beginning of a query plan. Chapter 5 discussed the four-fold advantages of late materialization. First, selection and aggregation operators tend to render the construction of some tuples unnecessary (if the executor waits long enough before constructing a tuple, it might be able to avoid constructing it altogether). Second, if data is compressed using a column-oriented compression method, it must be decompressed before the combination of values with values from other columns. This removes the advantages of operating directly on compressed data described above.

Third, cache performance is improved when operating directly on column data, since a given cache line is not polluted with surrounding irrelevant attributes for a given operation (as shown in PAX [21]). Fourth, the block iteration optimization described in the subsection 7.2.4 has a higher impact on performance for fixed-length attributes. If any attribute in a tuple is variable-width, then the entire tuple is variable width. In a late materialized column-store, fixed-width columns can be operated on separately.

7.2.3 Invisible Join

Chapter 6 introduces the invisible join as a technique to improve performance of foreign-key-primary-key joins between fact tables and dimension tables in a star schema. The key observation is that dimension tables are small relative to the fact table and can often fit in cache, making them well suited to serve as the inner table for a late materialized join strategy (since the out-of-order value extraction problem discussed in Chapter 5 is alleviated). Further, join predicates can be rewritten into normal predicates on the foreign key column in the fact table, and in many cases, this predicate application can be sped up through the between-predicate-rewriting optimization. This

	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	Avg.
RS	2.7	2.0	1.8	43.8	44.1	46.0	43.0	42.8	40.0	9.0	44.4	14.1	12.2	26.6
RS (MV)	1.0	2.0	2.0	15.5	13.5	11.8	16.1	6.9	17.9	7.8	29.2	22.4	6.4	11.7
CS	0.6	0.2	0.1	9.4	4.4	4.0	11.6	4.5	7.5	0.6	8.4	3.8	2.6	4.4
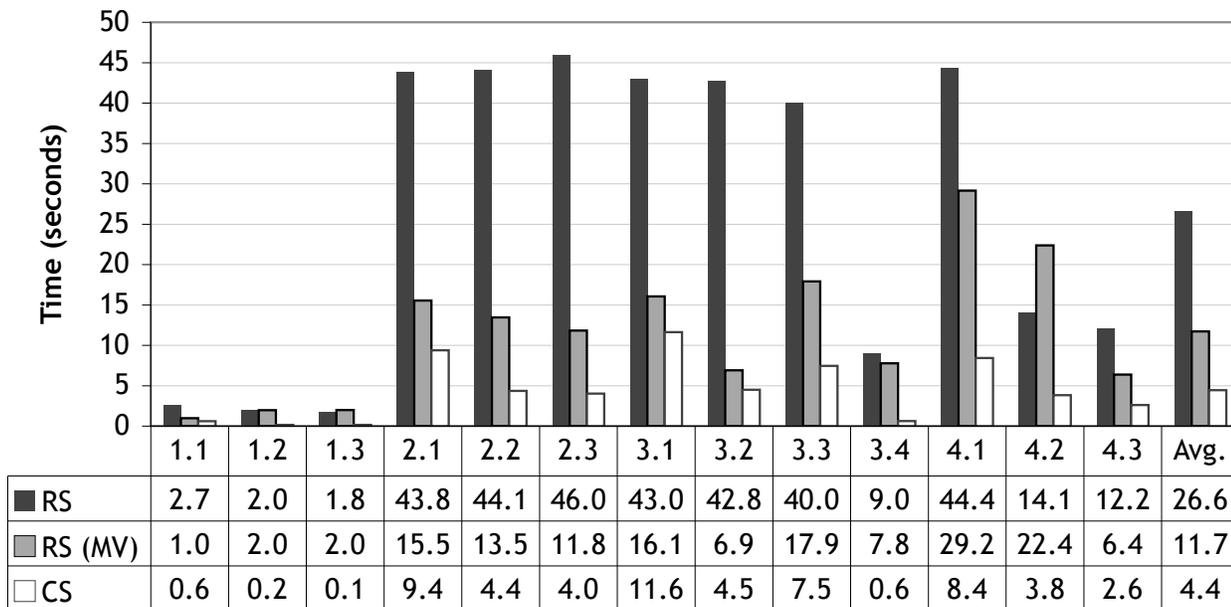
Figure 7-1: Baseline performance of column-store (CS) versus row-store (RS) and row-store w/ materialized views (RS (MV)) on the SSBM.

optimization allows the predicate to evaluate whether a foreign key will pass the join predicate directly, without having to look it up in a hash table.

7.2.4 Block Iteration

In order to process a series of tuples, most DBMS query executors first iterate through each tuple, and then need to extract the needed attributes from these tuples through a tuple representation interface [41]. This leads to tuple-at-a-time processing, where there are 1-2 function calls to extract needed data from a tuple for each operation (which if it is a small expression or predicate evaluation is low cost compared with the function calls) [76]. In column-stores, blocks of values from the same column are sent to an operator in a single function call. Further, if the column is fixed-width, these values can be iterated through directly as an array. This leads to higher CPU efficiency.

7.3 Experiments

We now evaluate these performance enhancing techniques on the SSBM data warehousing benchmark. We first present a comparison of the complete C-Store system with all techniques implemented with the commercial row-store of Chapter 2 ("System X") and then evaluate the performance contribution of each of these techniques.

All of our experiments were run on a 2.8 GHz single processor, dual core Pentium(R) D workstation with 3 GB of RAM running RedHat Enterprise Linux 5. The machine has a 4-disk array, managed as a single logical volume with files striped across it. Typical I/O throughput is 40 - 50 MB/sec/disk, or 160 - 200 MB/sec in aggregate for striped files. The numbers we report are the average of several runs, and are based on a "warm" buffer pool (in practice, we found that this yielded about a 30% performance increase; the gain is not particularly dramatic because the amount of data read by each query exceeds the size of the buffer pool).

7.3.1 System X vs. C-Store

We start with a simple experiment where we ran the complete C-Store database system, with all the query executor performance enhancement techniques implemented, on the SSBM. We compare these results with the numbers

presented in Chapter 2 on System X to get a sense of the performance difference of a complete column-store vs. a commercial row-store on a data warehousing benchmark. We compare against two implementations of System X – the baseline traditional implementation and the best-case System X that uses an optimal collection of materialized views containing minimal projections of tables needed to answer each query.

As shown in Figure 7-1 C-Store outperforms System X by a factor of six in the base case, and a factor of three when System X is using materialized views. Remember from Chapter 2 that this performance difference is conservative. We showed that C-Store, due to its nature as a small university prototype, does not have some of the standard performance optimizations that professional database systems have, such as partitioning and multi-threading. Chapter 2 showed that C-Store was approximately a factor of two slower than System X in a baseline case, mostly due to its lack of its ability to horizontally partition data.

Perhaps the most notable observation in these results is that C-Store is significantly faster than the best-case (I/O) scenario for the row-store, even though C-Store does not save work relative to the row-store since they read the same number of columns; on the contrary – C-Store must do more work since it must perform tuple reconstruction and the row-store does not. From this we conclude that the column-oriented query execution techniques we added to C-Store must have a significant effect on its performance. We explore this further in the next section.

7.3.2 Break-down of C-Store Performance

As described in Section 7.2, four column-oriented query executor techniques all can significantly improve the performance of column-oriented databases. These are compression, late materialization, the invisible join, and block-iteration. Presumably, these techniques are the reason for the performance difference between the column-store and the best-case row-store (materialized views) that has similar I/O patterns as the column-store. In order to verify this presumption, we successively removed these optimizations from C-Store and measured performance after each step.

Average

	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
■ Average	4.4	6.5	7.7	9.3	14.9	16.0	40.7
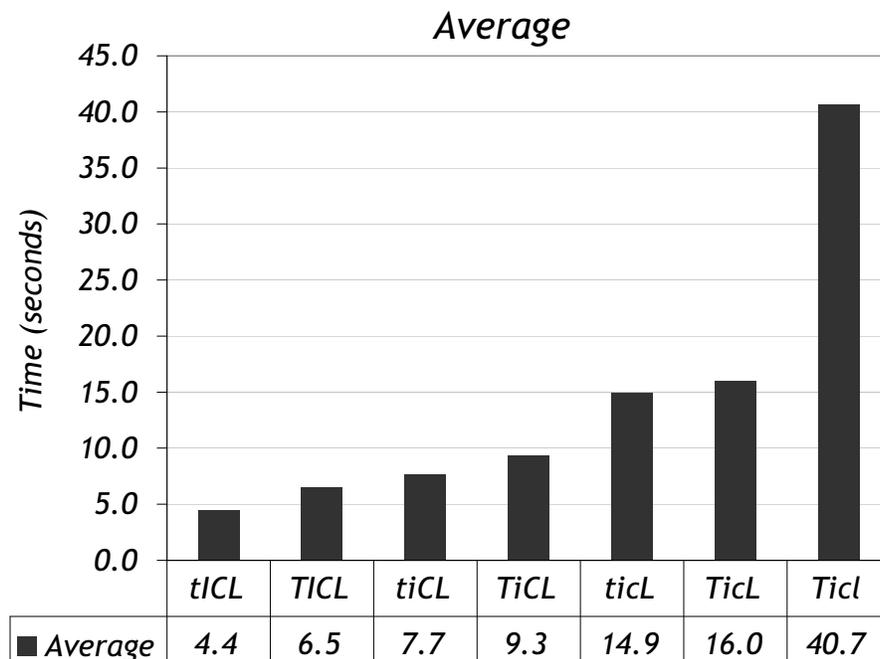
Figure 7-2: Average performance numbers for C-Store with different optimizations removed. The four letter code indicates the C-Store configuration: T=tuple-at-a-time processing was used, t=block processing; I=invisible join enabled, i=disabled; C=compression enabled, c=disabled; L=late materialization enabled, l=disabled.

Flight 1

	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
■ 1.1	0.6	0.3	0.4	0.6	5.4	8.0	32.7
▨ 1.2	0.2	0.1	0.1	0.1	2.2	6.2	27.3
□ 1.3	0.1	0.1	0.1	0.1	2.0	6.1	26.4

Flight 2

	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
■ 2.1	9.4	7.5	15.7	15.1	16.7	15.9	40.8
▨ 2.2	4.4	6.8	12.5	13.6	13.6	14.8	35.9
□ 2.3	4.0	6.5	12.1	13.2	13.3	14.4	34.8

Flight 3

	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
■ 3.1	11.6	17.4	16.6	21.2	31.9	31.4	56.7
▨ 3.2	4.5	11.2	9.0	14.3	15.7	15.5	33.5
□ 3.3	7.5	12.5	7.5	12.5	13.6	13.5	29.8
▨ 3.4	0.6	0.7	0.6	0.7	13.4	13.5	29.8

Flight 4

	tICL	TICL	tiCL	TiCL	ticL	TicL	Ticl
■ 4.1	8.4	11.1	15.8	17.1	29.9	29.5	67.1
▨ 4.2	3.8	5.7	5.6	7.2	20.3	21.4	60.0
□ 4.3	2.6	4.4	4.1	5.5	16.2	17.2	54.0
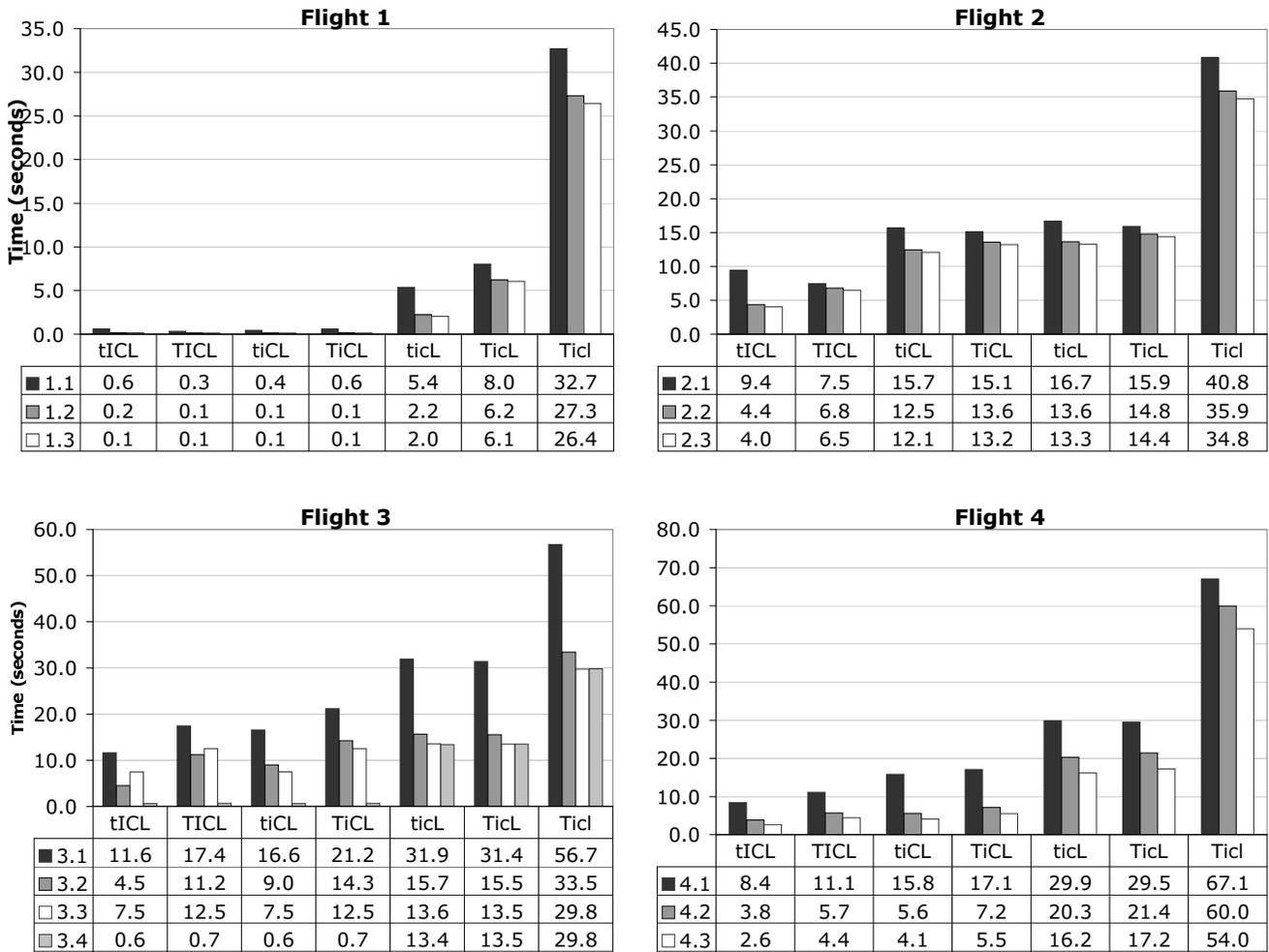
Figure 7-3: Performance numbers for C-Store by SSBM flight with different optimizations removed. The four letter code indicates the C-Store configuration: T=tuple-at-a-time processing was used, t=block processing; I=invisible join enabled, i=disabled; C=compression enabled, c=disabled; L=late materialization enabled, l=disabled.

Removing compression, late materialization, and the invisible join from C-Store was straightforward since we have already performed experiments with and without these techniques in previous chapters of this dissertation. Removing block-iteration was somewhat more difficult than the other three. As mentioned in Chapter 3, C-Store "blocks" of data can be accessed through two interfaces: "getNext" and "asArray". The former method requires one function call per value iterated through, while the latter method returns a pointer to an array than can be iterated through directly. For the operators used in the SSBM query plans that access blocks through the "asArray" interface, we wrote alternative versions that use "getNext". We only noticed a significant difference in the performance of selection operations using this method.

Figure 7-2 shows the results of successively removing these optimizations averaged across all queries, with detailed, per-query results shown in Figure 7-3. Block-processing can improve performance anywhere from a factor of only 5% to 50% depending on whether compression has already been removed (when compression is removed, the CPU benefits of block processing is not as significant since I/O becomes a factor). The invisible join improves performance by 50-75%.

Clearly, the most significant two optimizations are compression and late materialization. Compression improves performance by almost a factor of two on average. However, as mentioned in Section 7.2, we do not redundantly

store the fact table in multiple sort orders to get the full advantage of compression (only one column – the orderdate column – is sorted, and two others secondarily sorted – the quantity and discount columns). The columns in the fact table that are accessed by the SSBM queries are not very compressible if they do not have order to them, since they are either keys (which have high cardinality) or are random values. The first query flight, which accesses each of the three columns that have order to them, demonstrates the performance benefits of compression when queries access highly compressible data. In this case, compression results in an order of magnitude performance improvement. This is because runs of values in the three ordered columns can be run-length encoded (RLE). Not only does run-length encoding yield a good compression ratio and thus reduced I/O overhead, but RLE is also very simple to operate on directly (for example a predicate or an aggregation can be applied to an entire run at once). The primary sort column, orderdate, only contains 2405 unique values, and so the average run-length for this column is almost 25,000. This column takes up less than a block on disk.

The other significant optimization is late materialization. This optimization was removed last since data needs to be decompressed in the tuple construction process, and early materialization results in row-oriented processing which precludes invisible joins or block-iteration. Late materialization results in almost a factor of three performance improvement. This is primarily because of the selective predicates in some of the SSBM queries. The more selective the predicate, the more wasteful it is to construct tuples at the start of a query plan, since such are tuples immediately discarded.

Note that once all of these optimizations are removed, there are no column-specific optimizations left in the query executor, and the column-store acts like "approach 2" from Chapter 2 which uses a row-store query executor. Hence the numbers are identical to the results presented in that chapter.

7.4   Conclusion

In this chapter, we benchmarked the performance of the complete C-Store database system on a data warehousing benchmark. We demonstrated how the compression and materialization techniques proposed in previous chapters of this dissertation have a significant impact on query performance on the SSBM. Overall, C-Store performance is improved by an order of magnitude relative to the second column-store implementation approach described in Chapter 2, and a factor of 6 relative to a commercial row-store (and this factor of 6 is conservative). We then broke down the reasons why a column-store is able to process column-oriented data so effectively, finding that late materialization improves performance by a factor of three, and that compression provides about a factor of two on average, or an order-of-magnitude on queries that access sorted data.

Although the results presented in the first part of this chapter are interesting, the fact that column-stores outperform row-stores on data warehouse benchmarks is not a new revelation. In the next chapter, we benchmark column-store performance in an application outside of their traditional sweet-spot: the Semantic Web.

Chapter 8

Scalable Semantic Web Data Management

8.1 Introduction

The Semantic Web is an effort by the W3C [12] to enable integration and sharing of data across different applications and organizations. Though called the Semantic *Web*, the W3C envisions something closer to a global database than to the existing World-Wide Web. In the W3C vision, users of the Semantic Web should be able to issue structured queries over all of the data on the Internet, and receive correct and well-formed answers to those queries from a variety of different data sources that may have information relevant to the query. Building the Semantic Web requires surmounting many of the semantic heterogeneity problems faced by the database community over the years. In fact – as in many database research efforts – the W3C has proposed schema matching, ontologies, and schema repositories for managing semantic heterogeneity.

One area in which the Semantic Web community differs from the relational database community is in its choice of data model. The Semantic Web data model, called the "Resource Description Framework," [13] or RDF, represents data as statements about resources using a graph connecting resource nodes and their property values with labeled arcs representing properties. Syntactically, this graph can be represented using XML syntax (RDF/XML). This is typically the format for RDF data exchange; however, structurally, the graph can be parsed into a series of triples, each representing a statement of the form $< subject, property, object >$, which is the notation that will be followed in this chapter. These triples can then be stored in a relational database with a three-column schema. For example, to represent the fact that Serge Abiteboul, Rick Hull, and Victor Vianu wrote a book called "Foundations of Databases" we would use seven triples[1]:

```
person1 isNamed ''Serge Abiteboul''
person2 isNamed ''Rick Hull''
person3 isNamed ''Victor Vianu''
book1 hasAuthor person1
book1 hasAuthor person2
book1 hasAuthor person3
book1 isTitled ''Foundations of Databases''
```

The commonly stated advantage of this approach is that it is very general (almost any type of data can be expressed in this format – it's easy to shred both relational and XML databases into RDF triples) and it's easy to build tools that manipulate RDF. These tools won't be useful if different users describe objects differently, so the Semantic Web community has developed a set of standards for expressing schemas (RDFS and OWL); these make it possible, for example, to say that every book should have an author, or that the property "isAuthor" is the same as the property "authored."

---

[1]In practice, RDF uses Universal Resource Identifiers (URIs), which look like URLs and often include sequences of numbers to make them unique. More readable names will be used in examples in this chapter.

```
SELECT p5.obj
FROM rdf AS p1, rdf AS p2, rdf AS p3,
     rdf AS p4, rdf AS p5
WHERE p1.prop = 'title' AND p1.obj ~= 'Transaction'
  AND p1.subj = p2.subj AND p2.prop = 'type'
  AND p2.obj = 'book' AND p3.prop = 'type'
  AND p3.obj = 'auth' AND p4.prop = 'hasAuth'
  AND p4.subj = p2.subj AND p4.obj = p3.subj
  AND p5.prop = 'isnamed' AND p5.subj = p4.obj;
```

Figure 8-1: SQL over a triple-store for a query that finds all of the authors of books whose title contains the word "Transaction".

This data representation, though flexible, has the potential for serious performance issues, since there is only one single RDF table, and almost all interesting queries involve many self-joins over this table. For example, to find all of the authors of books whose title contains the word "Transaction" it is necessary to perform the five-way self-join query shown in Figure 8.1.

This query is potentially very slow to execute, since as the number of triples in the library collection scales, the RDF table may well exceed the size of memory, and each of these filters and joins will require a scan or index lookup. Real world queries involve many more joins, which complicates selectivity estimation and query optimization, and limits the benefit of indices.

It is tempting to dismiss RDF, as the data model seems to offer inherently limited performance for little – or no – improvement in expressiveness or utility. Regardless of one's opinion of RDF, however, it appears to have a great deal of momentum in the web community, with several international conferences (ISWC, ESWC) each drawing more than 250 full paper submissions and several hundred attendees, as well as enthusiastic support from the W3C (and its founder, Tim Berners-Lee.) Further, an increasing amount of data is becoming available on the Web in RDF format, including the UniProt comprehensive catalog of protein sequence, function, and annotation data (created by joining the information contained in Swiss-Prot, TrEMBL, and PIR) [9] and Princeton University's WordNet (a lexical database for the English language) [11]. The online Semantic Web search engine Swoogle [6] reports that it indexes 2,171,408 Semantic Web documents at the time of the publication of this thesis.

Hence, the goal of this chapter is to explore ways to improve RDF query performance, since it appears that it will be an important way for people to represent data on (or about) the web. The focus is on using a relational query processor to execute RDF queries, as it is likely to be the best performing approach (this belief is shared by several other research groups [31, 33, 44, 68]) . The gist of the technique presented in this chapter is based on a simple and familiar observation to proponents of relational technology: just as with relations, RDF does not have to be a proposal for physical storage – it is merely a logical data model. RDF databases are free to store RDF data as they see fit – including in ways that offer much better performance than actually storing collections of triples in memory or on disk.

The chapter studies two different physical organization techniques for RDF data. The first, called the *property table* technique, denormalizes RDF tables by physically storing them in a wider, flattened representation more similar to traditional relational schemas. One way to do this flattening, as suggested in [33] and [69], is to find sets of properties that tend to be defined together; i.e., clusters of subjects tend to have these properties defined. For example, "title," "author," and "isbn" might all be properties that tend to be defined for subjects that represent book entities. Thus a table containing subject as the key and "title," "author," and "isbn" as the other attributes might be created to store entities of type "book." This flattened property table representation will require many fewer joins to access, since self-joins on the subject column can be eliminated. One can use standard query rewriting techniques to translate queries over the RDF triple-store to queries over the flattened representation.

There are several issues with this property table technique, including:

NULLs. Because not all properties will be defined for all subjects in the subject cluster, wide tables will have

(possibly many) NULLs. For very wide tables with many sparse attributes, the space overhead of these NULLs can potentially dominate the space of the data itself.

Multi-valued Attributes. Multi-valued attributes (such as a book with multiple titles in different languages) and many-to-many relationships (such as the book authorship relationship where a book can have multiple authors and an author can write multiple books) are somewhat awkward to express in a flattened representation. Anecdotally, many RDF datasets make heavy use of multi-valued attributes, so this may be of more concern here than in other database applications.

Proliferation of union clauses and joins. In the above example, queries are simple if they can be isolated to querying a single property table like the one described above. But if, for example, the query does not restrict on property value, or if the value of the property will be bound when the query is processed, all flattened tables will have to be queried and the results combined with either complex union clauses, or through joins.

To address these limitations, we apply the column-store technology studied thus far this dissertation as a physical organization technique for RDF data. First, we use the vertical partitioning technique described in Chapter 2. A two-column table is created for each unique property in the RDF dataset where the first column contains subjects that define the property and the second column contains the object values for those subjects. For the library example, tables would be created for the "title," "author," "isbn," etc. properties, each table listing subject URIs with their corresponding value for that property. Multi-valued subjects are thus represented as multiple rows in the table with the same subject and different values. Although many joins are still required to answer queries over multiple properties, each table is sorted by subject, so fast (linear) merge joins can be used. Further, only those properties that are accessed by the query need to be read off disk (or from memory), saving I/O time.

As we showed in Chapter 2, although vertically partitioning a database can be done in a normal DBMS, these databases are not optimized for these narrow schemas (for example, the tuple header dominates the size of the actual data resulting in table scans taking 4-5 times as long as they need to), while using a column-store with a storage manager and/or a query executor designed for column-orientation can perform much better.

In this chapter, we present a comparison of the performance differences of RDF storage schemes on a real world RDF dataset. The Postgres open source DBMS is used to show that both the property table and the vertically partitioned approaches outperform the standard triple-store approach by more than a factor of 2 (average query times go from around 100 seconds to around 40 seconds) and have superior scaling properties. We then show that one can get another order of magnitude in performance improvement by using C-Store since it is designed for column-oriented data layout (queries now run in an average of 3 seconds).

The main contributions of this chapter are: an overview of the state of the art for storing RDF data in databases, a proposal to vertically partition RDF data (either using a traditional DBMS, or, even better, using a column-store such as C-Store) as a simple way to improve RDF query performance relative to the state of the art, and a performance evaluation of these different proposals. Ultimately, the column-oriented DBMS is able to obtain near-interactive performance (on non-trivial queries) over real-world RDF datasets of many millions of records, something that (to the best of our knowledge) no other RDF store has been able to achieve.

The remainder of this chapter is organized as follows. In Section 8.2, the state of the art of storing RDF data in relational databases is presented, with an extended look at the property table approach. In Section 8.3, the vertically partitioned (column-oriented) approach is discussed. In Section 8.4, an additional optimization to improve performance on RDF queries is presented: materializing path expressions in advance. In Section 8.5, the library benchmark used for evaluating the performance of an RDF database is summarized, and then the performance of the different RDF storage approaches is compared in Section 8.6.

8.2 Current State of the Art

In this section, the state of the art of storing RDF data in relational databases is discussed, with an extended look at the property table approach.

8.2.1 RDF In RDBMSs

Although there have been non-relational DBMS proposals for storing RDF data [30], the majority of RDF data storage solutions use relational DBMSs, such as Jena [68], Oracle[33], Sesame [31], and 3store [44]. These solutions generally center around a giant triples table, containing one row for each statement. For example, the RDF triples table for a small library dataset is shown in Table 8.1(a).

Since URIs and literal values tend to be long strings (rather than those shown in the simplified example in 8.1(a)), many RDF stores choose not to store entire strings in the triples table; instead they store shortened versions or keys. Oracle and Sesame map string URIs to integer identifiers so the data is normalized into two tables, one triples table using identifiers for each value, and one mapping table that maps the identifiers to their corresponding strings. This can be thought of as dictionary encoding the string data. 3store does something similar, except the identifiers are created by applying a hash function to each string. Jena prefers to just dictionary encode the namespace prefixes in the URIs and only normalizes the particularly long strings into a separate table.

Each of the above listed RDF storage solutions implements a multi-layered architecture, where RDF-specific functionality (for example, query translation) is performed in a layer above the RDBMS (which sits in the lowest layer). This removes any dependence on the particular RDBMS used (though Sesame will take advantage of specific features of an object relational DBMS such as Postgres to use subtables to model class and property subsumption relations). Queries are issued in an RDF-specific querying language (such as SPARQL [16] or RDQL [14]), converted to SQL in the higher level RDF layers, and then sent to the RDBMS which will optimize and execute the SQL query over the triple-store.

For example, the SPARQL query that attempts to get the title of the book(s) Joe Fox wrote in 2001:

```
SELECT ?title
FROM table
WHERE { ?book author ''Fox, Joe''
        ?book copyright ''2001''
        ?book title ?title }
```

would get converted into the SQL query shown in Table 8.1(b) run over the data in Table 8.1(a).

Note that this simple query results in a three-way self-join over the triples table (in fact, another join will generally be needed if the strings are normalized into a separate table, as described above). If the predicates are selective, this 3-way join is not expensive (assuming the triples table is indexed – typically there will be indexes on all three columns). However, the less selective the predicates, the more problematic the joins become. As a result, both Jena and Oracle propose changes to the schema to reduce the number of joins of this type: *property tables*. These data structures are now examined in more detail.

8.2.2 Property Tables

Researchers developing the Jena Semantic Web toolkit, Jena2 [69, 68], were the first to propose the use of property tables to speed up queries over triple-stores. They proposed two types of property tables. The first type, which we call a *clustered property table*, contains clusters of properties that tend to be defined together. For example, for the raw data in Table 8.1(a), type, title, and copyright date tend to be defined as properties for similar subjects. Thus, a property table containing these three properties as attributes along with subject as the table key can be created, which stores the triples from the original data whose property is one of these three attributes. The resulting property table, along with the left-over triples that are not stored in this property table, is shown in Table 8.1(c). Multiple property tables with different clusters of properties may be created; however, a key requirement for this type of property table is that a particular property may only appear in at most one property table.

The second type of property table, termed a property-class table, exploits the type property of subjects to cluster similar sets of subjects together in the same table. Unlike the first type of property table, a property may exist in multiple property-class tables. Table 8.1(d) shows two example property tables that may be created from the same set of input data as Table 8.1(c). Jena2 found property-class tables to be particularly useful for the storage of

Subj.	Prop.	Obj.
ID1	type	BookType
ID1	title	"XYZ"
ID1	author	"Fox, Joe"
ID1	copyright	"2001"
ID2	type	CDType
ID2	title	"ABC"
ID2	artist	"Orr, Tim"
ID2	copyright	"1985"
ID2	language	"French"
ID3	type	BookType
ID3	title	"MNO"
ID3	language	"English"
ID4	type	DVDType
ID4	title	"DEF"
ID5	type	CDType
ID5	title	"GHI"
ID5	copyright	"1995"
ID6	type	BookType
ID6	copyright	"2004"

(a) Some Example RDF Triples

```
SELECT C.obj.
FROM TRIPLES AS A,
     TRIPLES AS B,
     TRIPLES AS C
WHERE A.subj. = B.subj.
  AND B.subj. = C.subj.
  AND A.prop. = 'copyright'
  AND A.obj. = ''2001''
  AND B.prop. = 'author'
  AND B.obj. = ''Fox, Joe''
  AND C.prop. = 'title'
```

(b) Example SQL Query Over RDF Triples Table From (a)

Property Table

Subj.	Type	Title	copyright
ID1	BookType	"XYZ"	"2001"
ID2	CDType	"ABC"	"1985"
ID3	BookType	"MNP"	NULL
ID4	DVDType	"DEF"	NULL
ID5	CDType	"GHI"	"1995"
ID6	BookType	NULL	"2004"

Left-Over Triples

Subj.	Prop.	Obj.
ID1	author	"Fox, Joe"
ID2	artist	"Orr, Tim"
ID2	language	"French"
ID3	language	"English"

(c) Clustered Property Table Example

Class: BookType

Subj.	Title	Author	copyright
ID1	"XYZ"	"Fox, Joe"	"2001"
ID3	"MNP"	NULL	NULL
ID6	NULL	NULL	"2004"

Class: CDType

Subj.	Title	Artist	copyright
ID2	"ABC"	"Orr, Tim"	"1985"
ID5	"GHI"	NULL	"1995"

Left-Over Triples

Subj.	Prop.	Obj.
ID2	language	"French"
ID3	language	"English"
ID4	type	DVDType
ID4	title	"DEF"

(d) Property-Class Table Example

Table 8.1: Some sample RDF data and possible property tables.

reified statements (statements about statements) where the class is rdf:Statement and the properties are rdf:Subject, rdf:Property, and rdf:Object.

Oracle [33] also adopts a property table-like data structure (they call it a "subject-property matrix") to speed up queries over RDF triples. Their utilization of property tables is slightly different from Jena2 in that they are not used as a primary storage structure, but rather as an auxiliary data structure – a materialized view – that can be used to speed up specific types of queries.

The most important advantage of the introduction of property tables to the triple-store is that they can reduce subject-subject self-joins of the triples table. For example, the simple query shown in Section 8.2.1 ("return the title of the book(s) Joe Fox wrote in 2001") resulted in a three-way self-join. However, if title, author, and copyright were all located inside the same property table, the query can be executed via a simple selection operator.

To the best of our knowledge, property tables have not been widely adopted except in specialized cases (like reified statements). One reason for this may be that they have a number of disadvantages. Most importantly, as Wilkinson points out in [69], while property tables are very good at speeding up queries that can be answered from a single property table, most queries require joins or unions to combine data from several tables. For example, for the data in Table 8.1, if a user wishes to find out if there are any items in the catalog copyrighted before 1990 in a language other than English, the following SQL queries could be issued:

```
SELECT T.subject, T.object
FROM TRIPLES AS T, PROPTABLE AS P
WHERE T.subject == P.subject
  AND P.copyright < 1990
  AND T.property = 'language'
  AND T.object != ''English''
```

for the schema in 8.1(c), and

```
(SELECT T.subject, T.object
 FROM TRIPLES AS T, BOOKS AS B
 WHERE T.subject == B.subject
   AND B.copyright < 1990
   AND T.property = 'language'
   AND T.object != ''English'')
UNION
(SELECT T.subject, T.object
 FROM TRIPLES AS T, CDS AS C
 WHERE T.subject == C.subject
   AND C.copyright < 1990
   AND T.property = 'language'
   AND T.object != ''English'')
```

for the schema in 8.1(d). As can be seen, join and union clauses get introduced into the queries, and query translation and plan generation get complicated very quickly. Queries that do not select on class type are generally problematic for property-class tables, and queries that have unspecified property values (or for whom property value is bound at run-time) are generally problematic for clustered property tables.

Another disadvantage of property tables is that RDF data tends not to be very structured, and not every subject listed in the table will have all the properties defined. The less structured the data, the more NULL values will exist in the table. In fact, these representations can be extremely sparse – containing hundreds of NULLs for each non-NULL value. These NULLs impose a substantial performance overhead, as has been noted in previous work [17, 20, 25].

The two problems with property tables are at odds with one another. If property tables are made narrow, with few property columns that are highly correlated in their value definition, the average value density of the table increases and the table is less sparse. Unfortunately, the likelihood of any particular query being able to be confined to a single

property table is reduced. On the other hand, if many properties are included in a single property table, the number of joins and union clauses per query decreases, but the number of NULLs in the table increases (it becomes more sparse), bloating the table and wasting space. Thus there is a fundamental trade-off between query complexity as a result of proliferation of joins and unions and table sparsity (and its resulting impact on query performance). Similar problems have been noted in attempts to shred and store XML data in relational databases [38, 62].

A third problem with property tables is the abundance of multi-valued attributes found in RDF data. Multi-valued attributes are surprisingly prevalent in the Semantic Web; for example in the library catalog data we work with in Section 8.5, properties one might think of as single-valued such as title, publisher, and even entity type are multi-valued. In general, there always seem to be exceptions, and the RDF data model provides no disincentives for making properties multi-valued. Further, our experience suggests that RDF data is often unclean, with overloaded subject URIs used to represent many different real-world entities.

Multi-valued properties are problematic for property tables for the same reason they are problematic for relational tables. They cannot be included with the other attributes in the same table unless they are represented using list, set, or bag attributes. However, this requires an object-relational DBMS, results in variable width attributes, and complicates the expression of queries over these attributes.

In summary, while property tables can significantly improve performance by reducing the number of self-joins and typing attributes, they introduce complexity by requiring property clustering to be carefully done to create property tables that are not too wide, while still being wide enough to answer most queries directly. Ubiquitous multi-valued attributes cause further complexity.

8.3 A Simpler Alternative

This section proposes an alternative to the property table solution to speed up queries over a triple-store. In Section 8.3.1 the vertically partitioned approach to storing RDF triples is discussed. Then, in Section 8.3.2, we review the findings from Chapter 2 on why a column-store designed for column-oriented layout is likely to perform better than a row-store that implements the vertically partitioned approach. However, unlike the approaches taken in Chapter 2, here we propose that the column-store makes the same modifications to the logical schema as a row-store that implements the vertically partitioning approach. We then provide a description of what we needed to add to C-Store to manage RDF data.

8.3.1 Vertically Partitioned Approach

We propose storage of RDF data using a fully decomposed storage model (DSM) [35, 37]. The triples table is rewritten into *n* two column tables where *n* is the number of unique properties in the data. In each of these tables, the first column contains the subjects that define that property and the second column contains the object values for those subjects. For example, the triples table from Table 8.1(a) would be stored as:

Type

ID1	BookType
ID2	CDType
ID3	BookType
ID4	DVDType
ID5	CDType
ID6	BookType

Author

ID1	"Fox, Joe"

Title

ID1	"XYZ"
ID2	"ABC"
ID3	"MNO"
ID4	"DEF"
ID5	"GHI"

Artist

ID2	"Orr, Tim"

Copyright

ID1	"2001"
ID2	"1985"
ID5	"1995"
ID6	"2004"

Language

ID2	"French"
ID3	"English"

Each table is sorted by subject, so that particular subjects can be located quickly, and that fast merge joins can be used to reconstruct information about multiple properties for subsets of subjects. The value column for each table

can also be optionally indexed (or a second copy of the table can be created clustered on the value column).

The advantages of this approach (relative to the property table approach) are:

Support for multi-valued attributes. A multi-valued attribute is not problematic in the decomposed storage model. If a subject has more than one object value for a particular property, then each distinct value is listed in a successive row in the table for that property. For example, if ID1 had two authors in the example above, the table would look like:

<div align="center">

Author

ID1	"Fox, Joe"
ID1	"Green, John"

</div>

Support for heterogeneous records. Subjects that do not define a particular property are simply omitted from the table for that property. In the example above, author is only defined for one subject (ID1) so the table can be kept small (NULL data need not be explicitly stored). The advantage becomes increasingly important when the data is not well-structured.

Only those properties accessed by a query need to be read. I/O costs can be substantially reduced.

No clustering algorithms are needed. This point is the basis behind our claim that the vertically partitioned approach is simpler than the property table approach. While property tables need to be carefully constructed so that they are not too wide, but yet wide enough to independently answer queries, the algorithm for creating tables in the vertically partitioned approach is straightforward and need not change over time.

Fewer unions and fast joins. Since all data for a particular property is located in the same table (unlike the property-class schema of Figure 8.1(d)), union clauses in queries are less common. And although the vertically partitioned approach will require more joins relative to the property table approach, properties are joined using simple, fast (linear) merge joins.

Of course there are several disadvantages to this approach relative to property tables. When a query accesses several properties, these 2-column tables have to be merged. Although a merge join is not expensive, it is also not free. Also, inserts can be slower into vertically partitioned tables, since multiple tables need to be accessed for statements about the same subject. However, we have yet to come across an RDF application where the insert rate is so high that buffering the inserts and batch rewriting the tables is unacceptable.

In Section 8.6 we will compare the performance of the property table approach and the vertically partitioned approach to each other and to the triples table approach. Before we present these experiments, we describe how a column-oriented DBMS can be extended to implement the vertically partitioned approach.

8.3.2   Extending C-Store

Recall from Chapter 2 that one advantage of moving from the vertical partitioning approach to the other column-store implementation approaches that make modifications to the storage layer and the query executor is that the first column of the two-column tables need not be stored. The ith value in each column simply match up, rather than relying on a key to help with the matching. In essence, the position of the value in a column was a "virtual key". However, in the case of RDF data management, with the large amount of NULL data and multi-value attributes, virtual keys are no longer useful. The key must be physically stored so that it is clear whether the $i + 1$th value in a column is really the attribute value for the $i + 1$th tuple, or whether it is another value of a multi-valued attribute for the ith attribute. Further, NULL data must be explicitly stored under the virtual key scheme or else columns will no longer line up.

Thus, we propose that the column-store uses the same DSM schema. At first blush, it might seem strange to use a column-store to store a set of two-column tables since column-stores excel at storing big wide tables where only a few attributes are queried at once. However, column-stores are actually well-suited for schemas of this type, for the following reasons:

Tuple headers are stored separately. Databases generally store tuple metadata at the beginning of the tuple. For example, Postgres contains a 27 byte tuple header containing information such as insert transaction timestamp, number of attributes in tuple, and NULL flags. In contrast, the rest of the data in the two-column tables will generally not take up more than 8 bytes (especially when strings have been dictionary encoded). A column-store puts header information in separate columns and can selectively ignore it (a lot of this data is no longer relevant in the two column case; for example, the number of attributes is always two, and there are never any NULL values since subjects that do not define a particular property are omitted). Thus, the effective tuple width in a column-store is on the order of 8 bytes, compared with 35 bytes for a row-store like Postgres, which means that table scans perform 4-5 times quicker in the column-store.

Optimizations for fixed-length tuples. In a row-store, if any attribute is variable length, then the entire tuple is variable length. Since this is the common case, row-stores are designed for this case, where tuples are located through pointers in the page header (instead of address offset calculation) and are iterated through using an extra function call to a tuple interface (instead of iterated through directly as an array). This has a significant performance overhead [29, 28]. In a column-store, fixed length attributes are stored as arrays. For the two-column tables in our RDF storage scheme, both attributes are fixed-length (assuming strings are dictionary encoded).

Column-oriented data compression. As described in Chapter 4, since each attribute is stored separately, each attribute can be compressed separately using an algorithm best suited for that column. This can lead to significant performance improvement through I/O savings. For example, the subject ID column, a monotonically increasing array of integers, is very compressible.

Carefully optimized column merge code. As described in Chapter 5, since merging columns is a very frequent operation in column-stores, the merging code is carefully optimized to achieve high performance. For example, extensive prefetching is used when merging multiple columns, so that disk seeks between columns (as they are read in parallel) do not dominate query time. Merging tables sorted on the same attribute can use the same code.

Direct access to sorted files rather than indirection through a B tree. While not strictly a property of column-oriented stores, the increased dependence on merge joins necessitates that heap files are maintained in guaranteed sorted order, whereas the order of heap files in many row-stores, even on a clustered attribute, is only guaranteed through an index. Thus, iterating through a sorted file must be done indirectly through the index, and extra seeks between index leaves may degrade performance.

In summary, a column-store vertically partitions attributes of a table. The vertically partitioned scheme described in Section 8.3.1 can be thought of as partitioning attributes from a wide universal table containing all possible attributes from the data domain. Consequently, it makes sense to use a DBMS that is optimized for this type of partitioning.

Implementation details

C-Store was extended to experiment with the ideas presented in this chapter. C-Store, as a bare-bones research prototype, did not have support for temporary tables, the index-nested loops join operator, or the union operator, each of which had to be added. Strings were dictionary encoded similarly to Oracle and Sesame (as described in Section 8.2.1) where only fixed-width integer keys are stored in the data tables, and the keys are decoded at the end of each query using and index-nested loops join with a large strings dictionary table.

8.4   Materialized Path Expressions

In all three RDF storage schemes described thus far (triples schema, property tables, and vertically partitioned tables), querying path expressions (a common operation on RDF data) is expensive. In RDF data, object values can either be literals (e.g., "Fox, Joe") or URIs (e.g., `http://preamble/FoxJoe`). In the latter case, the value can be further described using additional triples (e.g., <BookID1, Author, `http://preamble/FoxJoe`>, <`http:`

//preamble/FoxJoe, wasBorn, "1860">). If one wanted to find all books whose authors were born in 1860, this would require a path expression through the data. In a triples store, this query might look like:

```
SELECT B.subj
FROM triples AS A, triples AS B
WHERE A.prop = wasBorn
AND A.obj = ''1860''
AND A.subj = B.obj
AND B.prop = ''Author''
```

We need to perform a subject-object join to connect information about authors with information on the books they wrote.

In general, in a triples schema, a path expression requires $(n-1)$ subject-object self-joins where n is the length of the path. For a property table schema, $(n-1)$ self-joins are also required if all properties in the path expression are included in the table; otherwise the property table needs to be joined with other tables. For the vertically partitioned schema, the tables for the properties involved in the path expression need to be joined together; however these are joins of the second (unsorted) column of one table with the first column of the other table (and are hence not merge joins).
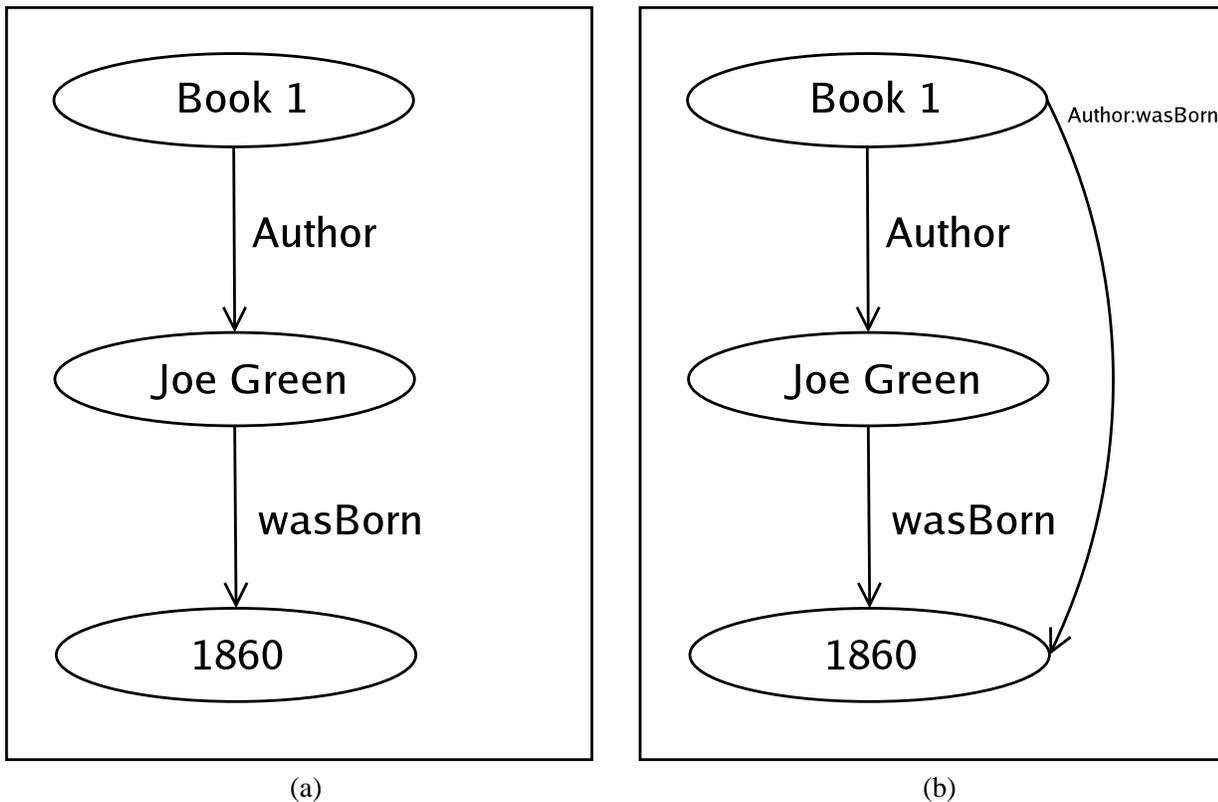
Figure 8-2: Graphical presentation of subject-object join queries.

Graphically, the data is modeled as shown in Figure 8-2(a). Here we use the standard RDF semantic model where subjects and objects are connected by labeled directed edges (properties). The path expression join can be observed through the *author* and *wasBorn* properties. If we could store the results of following the path expression through a more direct path (shown in Figure 8-2(b)), the join could be eliminated:

```
SELECT A.subj
```

```
FROM proptable AS A,
WHERE A.author:wasBorn = ''1860''
```

Using a vertically partitioned schema, this author:wasBorn path expression can be precalculated and the result stored in its own two column table (as if it were a regular property). By precalculating the path expression, we do not have to perform the join at query time. Note that if any of the properties along the path in the path expression were multi-valued, the result would also be multi-valued. Thus, this materialized path expression technique is easier to implement in a vertically partitioned schema than in a property table.

Inference queries (e.g., if X is a part of Y and Y is a part of Z then X is a part of Z), a very common operation on Semantic Web data, are also usually performed using subject-object joins, and can be accelerated through this method.

There is, however, a cost in having a larger number of extra materialized tables, since they need to be recalculated whenever new triples are added to the RDF store. Thus, for read-only or read-mostly RDF applications, many of these materialized path expression tables can be created, but for insert heavy workloads, only very common path expressions should be materialized.

We realize that a materialization step is not an automatic improvement that comes with the presented architectures. However, both property tables and vertically partitioned data lend themselves to allowing such calculations to be precomputed if they appear on a common path expression.

8.5   Benchmark

In this section, we describe the RDF benchmark we have developed for evaluating the performance of our three RDF databases. Our benchmark is based on publicly available library data and a collection of queries generated from a web-based user interface for browsing RDF content.

8.5.1   Barton Data

The dataset we work with is taken from the publicly available Barton Libraries dataset [1]. This data is provided by the Simile Project [5], which develops tools for library data management and interoperability. The data contains records acquired from an RDF-formatted dump of the MIT Libraries Barton catalog, converted from raw data stored in an old library format standard called MARC (Machine Readable Catalog). Because of the multiple sources the data was derived from and the diverse nature of the data that is cataloged, the structure of the data is quite irregular.

We converted the Barton data from RDF/XML syntax to triples using the Redland parser [4] and then eliminated duplicate triples. We then did some very minor cleaning of data, eliminating triples with particularly long literal values or with subject URIs that were obviously overloaded to correspond to several real-world entities (more than 99% of the data remained). This left a total of $50,255,599$ triples in our dataset, with a total of 221 unique properties, of which the vast majority appear infrequently. Of these properties, 82 (37%) are multi-valued, meaning that they appear more than once for a given subject; however, these properties appear more often (77% of the triples have a multi-valued property). The dataset provides a good demonstration of the relatively unstructured nature of Semantic Web data.

8.5.2   Longwell Overview

Longwell [2] is a tool developed by the Simile Project, which provides a graphical user interface for generic RDF data exploration in a web browser. It begins by presenting the user with a list of the values the *type* property can take (such as *Text* or *Notated Music* in the library dataset) and the number of times each type occurs in the data. The user can then click on the types of data to further explore. Longwell shows the list of currently filtered resources (RDF subjects) in the main portion of the screen, and a list of filters in panels along the side. Each panel represents a property that is defined on the current filter, and contains popular object values for that property along with their

corresponding frequencies. If the user selects an object value inside one of these property panels, this filters the working set of resources to those that have that property-object pair defined, updating the other panels with the new frequency counts for this narrower set of resources.

We will now describe a sample browsing session through the Longwell interface, along with screenshots from this sample session. The opening screen with the list of different types is shown in Figure 8-3. Note the list of different types is shown at the top of the screen and their frequencies in the upper-right-hand side of the screen.

The path starts when the user selects *Text* from the *type* panel (at the upper-right-hand side of the screen), which filters the data into a list of text entities. This is shown in Figure 8-4.

On the right side of the screen, we find that popular properties on these entities. As the user scrolls down (see Figure 8-5), we can see that they include "subject," "creator," "genre," and "publisher." Within each property there is a list of the counts of the popular objects within this property. For example, as the user scrolls down even more (see Figure 8-6), we find out that the German object value appears 122 times and the French object value appears 131 times under the language property. By clicking on "fre" (French language), information about the 131 French texts in the database is presented, along with the revised set of popular properties and property values defined on these French texts. This is shown in Figure 8-7.

Currently, Longwell only runs on a small fraction of the Barton data – 9375 records, as its RDF triple store cannot scale to support the full 50 million triple dataset while still allowing interactive response time to queries (we will show this scalability limitation in our experiments).

Our experiments use Longwell-style queries to provide a realistic benchmark for testing the designs proposed. Our goal is to explore architectures and schemas which can provide interactive performance on the full dataset.

8.5.3 Longwell Queries

Our experiments feature seven queries that need to be executed on a typical Longwell path through the data. These queries are based on a typical browsing session, where the user selects a few specific entities to focus on and where the aggregate results summarizing the contents of the RDF store are updated.

The full queries are described at a high level here and are provided in full in Appendix C as SQL queries against a triple-store. We will discuss later how we rewrote the queries for each schema.

Query 1 (Q1). Calculate the opening panel displaying the counts of the different types of data in the RDF store. This requires a search for the objects and counts of those objects with property *Type*.
There are 30 such objects. For example: *Type*: *Text* has a count of $1,542,280$, and *Type*: *NotatedMusic* has a count of $36,441$.

Query 2 (Q2). The user selects *Type*: *Text* from the previous panel. Longwell must then display a list of other defined properties for resources of *Type*: *Text*. It must also calculate the frequency of these properties. For example, the *Language* property is defined $1,028,826$ times for resources that are of *Type*: *Text*.

Query 3 (Q3). For each property defined on items of *Type*: *Text*, populate the property panel with the counts of popular object values for that property (where popular means that an object value appears more than once). For example, the property *Edition* has 8 items with value "[1st_ed._reprinted]."

Query 4 (Q4). This query recalculates all of the property-object counts from Q3 if the user clicks on the "French" value in the "Language" property panel. Essentially this is narrowing the working set of subjects to those whose *Type* is *Text* **and** *Language* is *French*. This query is thus similar to Q3, but has a much higher-selectivity.

Query 5 (Q5). Here we perform a type of *inference*. If there are triples of the form (*X Records Y*) and (*Y Type Z*) then we can *infer* that *X* is of type *Z*. Here *X Records Y* means that *X* records information about *Y* (for example, *X* might be a web page with information on *Y*). For this query, we want to find the inferred type of all subjects that have this *Records* property defined that also originated in the US Library of Congress (i.e. contain triples of the form (*X origin "DLC"*)). The subject and inferred type is returned for all non-*Text* entities.

Query 6 (Q6). For this query, we combine the inference first step of Q5 with the property frequency calculation of
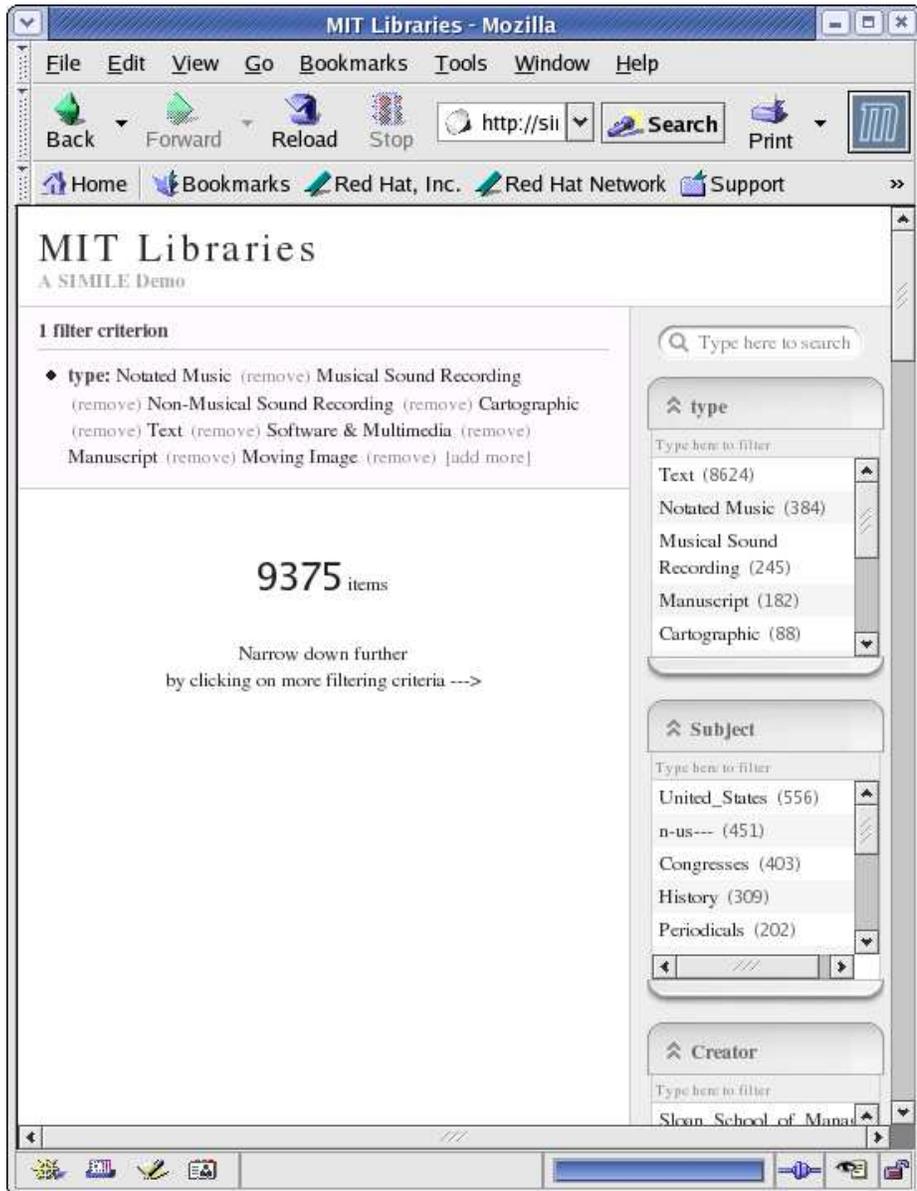
Figure 8-3: Longwell Opening Screen
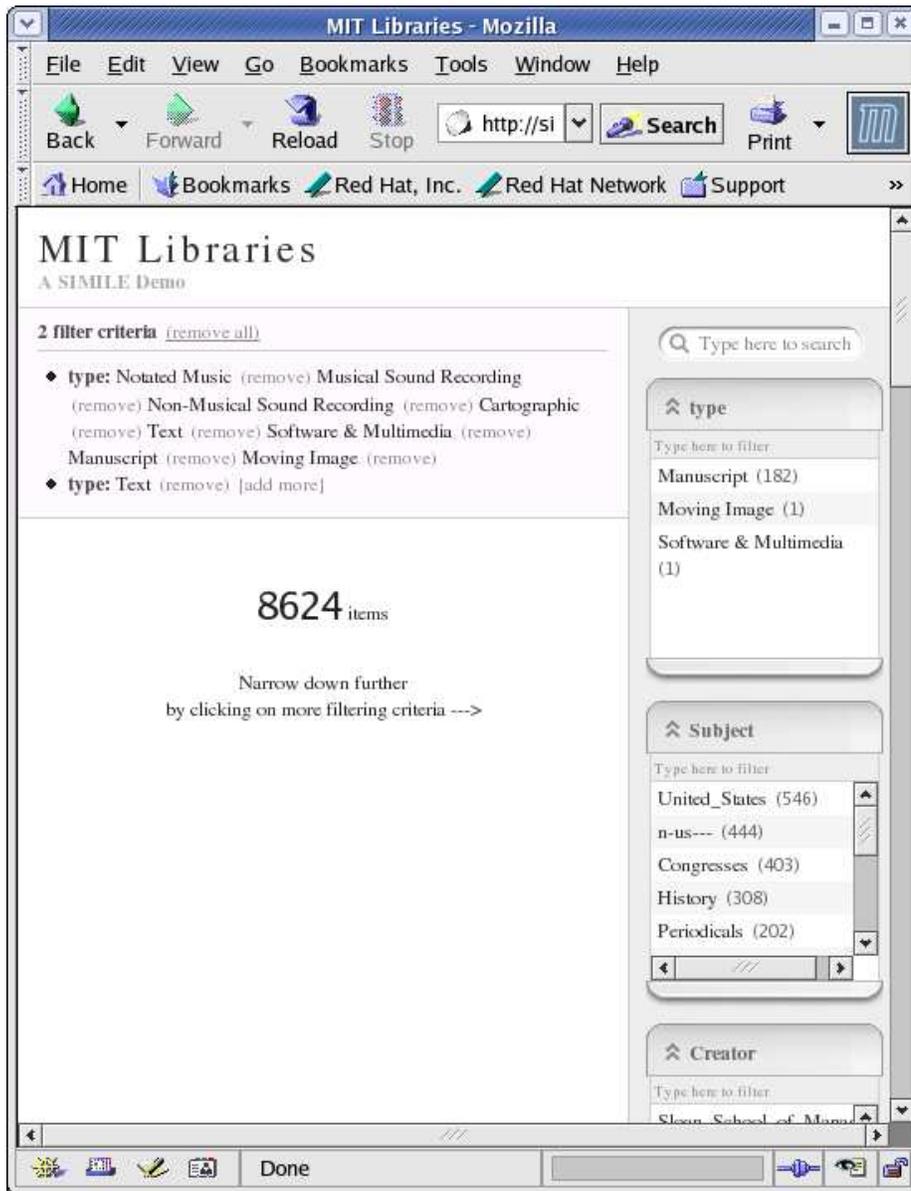
Figure 8-4: Longwell Screen Shot After Clicking on "Text" in the Type Property Panel
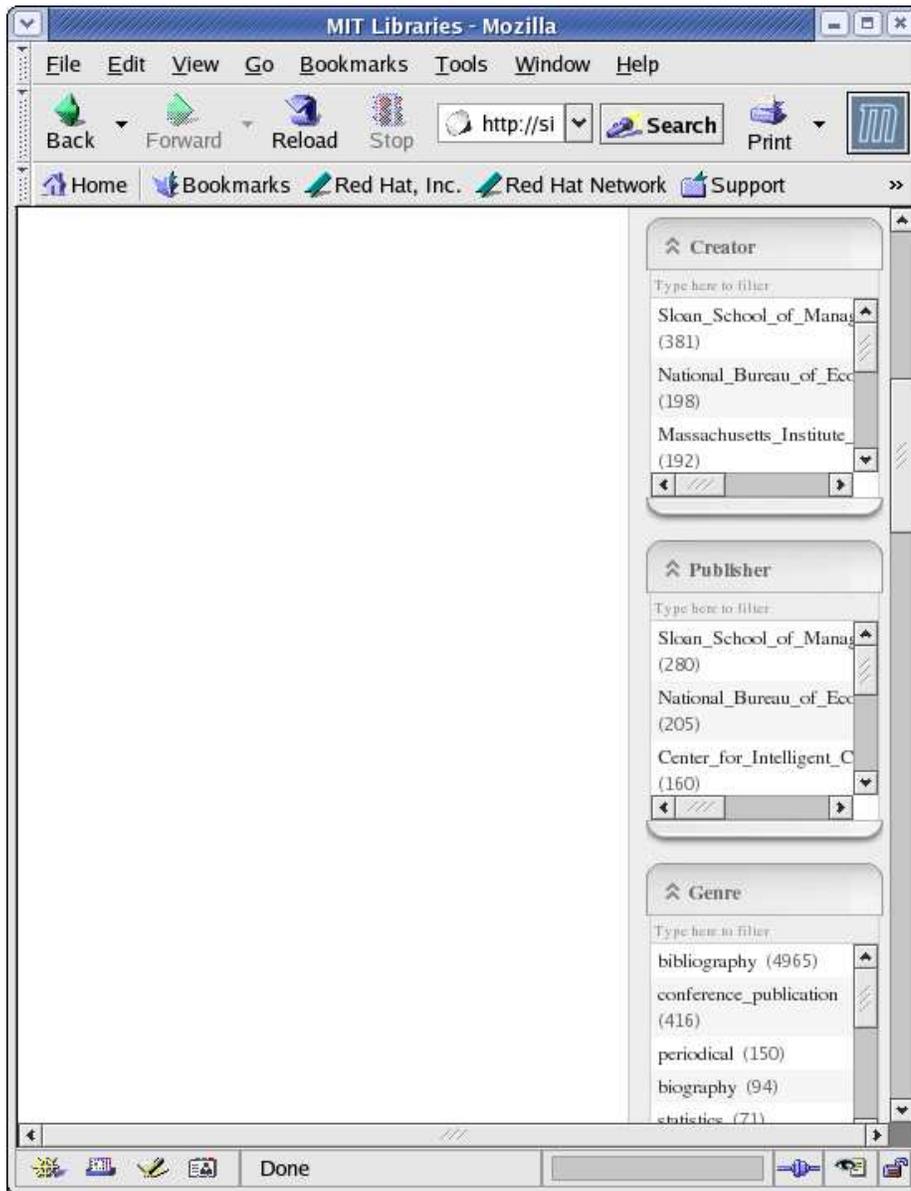
Figure 8-5: Longwell Screen Shot After Clicking on "Text" in the Type Property Panel and Scrolling Down

Figure 8-6: Longwell Screen Shot After Clicking on "Text" in the Type Property Panel and Scrolling Down to the Language Property Panel

Figure 8-7: Longwell Screen Shot After Clicking on "fre" in the Language Property Panel

Q2 to extract information in aggregate about items that are either directly known to be of *Type*: *Text* (as in Q2) or inferred to be of *Type*: *Text* through the Q5 *Records* inference.

Query 7 (Q7). Finally, we include a simple triple selection query with no aggregation or inference. The user tries to learn what a particular property (in this case *Point*) actually means by selecting other properties that are defined along with a particular value of this property. The user wishes to retrieve subject, *Encoding*, and *Type* of all resources with a *Point* value of "end." The result set indicates that all such resources are of the type *Date*. This explains why these resources can have "start" and "end" values: each of these resources represents a start or end date, depending on the value of *Point*.

We make the assumption that the Longwell administrator has selected a set of 28 interesting properties over which queries will be run (they are listed in Appendix D). There are 26,761,389 triples for these properties. For queries Q2, Q3, Q4, and Q6, only these 28 properties are considered for aggregation.

8.6 Evaluation

Now that we have described our benchmark dataset and the queries that we run over it, we compare their performance in three different schemas – a triples schema, a property tables schema, and a vertically partitioned schema. We study the performance of each of these three schemas in a row-store (Postgres) and, for the vertically partitioned schema, also in a column-store (the extension of C-Store).

Our goal is to study the performance tradeoffs between these representations to understand when a vertically partitioned approach performs better (or worse) than the property tables solution. Ultimately, the goal is to improve performance as much as possible over the triple-store schema, since this is the schema most RDF store systems use.

8.6.1 System

Our benchmarking system is a hyperthreaded 3.0 GHz Pentium IV, running RedHat Linux, with 2 Gbytes of memory, 1MB L2 cache, and a 3-disk, 750 Gbyte striped RAID array. The disk can read cold data at 150-180 MB/sec.

PostgreSQL Database

We chose Postgres as the row-store to experiment with because Beckmann et al. [25] experimentally showed that it was by far more efficient dealing with sparse data than commercial database products. Postgres does not waste space storing NULL data: every tuple is preceded by a bit-string of cardinality equal to the number of attributes, with '1's at positions of the non-NULL values in the tuple. NULL data is thus not stored; this is unlike commercial products that waste space on NULL data. Beckmann et al. show that Postgres queries over sparse data operate about eight times faster than commercial systems.

We ran Postgres with *work_mem* = 51200, meaning that 50 Mbytes of memory are dedicated to each sorting and hashing operation. This may seem low, but the *work_mem* value is considered per operation, many of which are highly parallelizable. For example, when multiple aggregations are simultaneously being processed during the UNIONed GROUP BY queries for the property table implementation, a higher value of *work_mem* would cause the query executor to use all available physical memory and thrash. We set *effective_cache_size* to 183500 4KB pages. This value is a planner hint to predict how much memory is available in both the Postgres and operating system cache for overall caching. Setting it to a higher value does not change the plans for any of the queries run. We turned *fsync* off to avoid syncing the write-ahead log to disk to make comparisons to C-Store fair, since it does not use logging [63]. All queries were run at a READ COMMITTED isolation level, which is the lowest level of isolation available in Postgres, again because C-Store was not using transactions.

8.6.2   Store Implementation Details

We now describe the details of our store implementations. Note that all implementations feature a dictionary encoding table that maps strings to integer identifiers (as was described in Section 8.2.1); these integers are used instead of strings to represent properties, subjects, and objects. The encoding table has a clustered B+tree index on the identifiers, and an unclustered B+tree index on the strings. We found that all experiments, including those on the triple-store, went an order of magnitude faster with dictionary encoding.

Triple Store

Of the popular full triple-store implementations, Sesame [31] seemed the most promising in terms of performance because it provides a native store that utilizes B+tree indices on any combination of subjects, properties, and objects, but does not have the overhead of a full database (of course, scalability is still an issue as it must perform many self-joins like all triple-stores). We were unable to test all queries on Sesame, as the current version of its query language, SeRQL, does not support aggregates (which are slated to be included in version 2 of the Sesame project). Because of this limitation, we were only able to test Q5 and Q7 on Sesame, as they did not feature aggregation. The Sesame system implements dictionary encoding to remove strings from the triples table, and including the dictionary encoding table, the triples table, and the indices on the tables, the system took 6.4 GBytes on disk.

On Q5, Sesame took 1400.94 seconds. For Q7, Sesame completed in 79.98 seconds. These results are the same order of magnitude, but 2-3X slower than the same queries we ran on a triple-store implemented directly in Postgres. We attribute this to the fact that we compressed namespace strings in Postgres more aggressively than Sesame does, and we can interact with the triple-store directly in SQL rather than indirectly through Sesame's interfaces and SeRQL. We observed similar results when using Jena instead of Sesame.

Thus, in this chapter, triple-store numbers are reported using the direct Postgres representation, since this seems to be a more fair comparison to the alternative techniques explored (which also directly interact with the database) and allows numbers to be reported for aggregation queries.

Our Postgres implementation of the triple-store contains three columns, one each for subject, property, and object. The table contains three B+ tree indices: one clustered on (subject, property, object), two unclustered on (property, object, subject) and (object, subject, property). We experimentally determined these to be the best performing indices for our query workload. We also maintain the list of the 28 interesting properties described in Section 8.5.3 in a small separate table. The total storage needs for this implementation is 8.3 GBytes (including indices and the dictionary encoding table).

Property Table Store

We implemented clustered property tables as described in Section 8.2.1. To measure their best-case performance, we created a property table for each query containing only the columns accessed by that query. Thus, the table for Q2, Q3, Q4 and Q6 contains the 28 interesting properties described in Section 8.5.3. The table for Q1 stores only subject and *Type* property columns, allowing for repetitions in the subject for multi-valued attributes. The table for Q5 contains columns for subject, *Origin*, *Records*, and *Type*. The Q7 table contains subject, *Encoding*, *Point*, and *Type* columns. We will look at the performance consequences of property tables that are wider than needed to answer particular queries in Section 8.6.7.

For all but Q1, multi-valued attributes are stored in columns that are integer arrays (*int[]* in Postgres), while all other columns are integer types. For single-valued attributes that are used as selection predicates, we create unclustered B+ tree indices. We attempted to use GiST [45] indexing for integer arrays in Postgres[2], but using this access path took more time than a sequential scan through the database, so multi-valued attributes used as selection predicates were not indexed. All tables had a clustered index on subject. While the smaller tables took less space, the property table with 28 properties took 14 GBytes (including indices and the dictionary encoding table).

---

[2]http://www.sai.msu.su/ megera/postgres/gist/intarray/README.intarray

Vertically Partitioned Store in Postgres

The vertically partitioned store contains one table per property. Each table contains a subject and object column. There is a clustered B+ tree index on subject, and an unclustered B+ tree index on object. Multi-valued attributes are represented as described in Section 8.3.1 through multiple rows in the table with the same subject and different object value. This store took up 5.2 GBytes (including indices and the dictionary encoding table).

Column-Oriented Store

Properties are stored on disk in separate files, in blocks of 64 KB. Each property contains two columns like the vertically partitioned store above. Each property has a clustered B+ tree on subject; and single-valued, low cardinality properties have a bit-map index on object. We used the C-Store default of 4MB column prefetching (this reduces seeks in merge joins). This store took up 2.7 GBytes (including indices and the dictionary encoding table).

8.6.3 Query Implementation Details

In this section, we discuss the implementation of all seven benchmark queries in the four designs described above.

Q1. On a triple-store, Q1 does not require a join, and aggregation can occur directly on the object column after the property=*Type* selection is performed. The vertically partitioned table and the column-store aggregate the object values for the *Type* table. Because the property table solution has the same schema as the vertically partitioned table for this query, the query plan is the same.

Q2. On a triple-store, this query requires a selection on property=*Type* and object=*Text*, followed by a self-join on subject to find what other properties are defined for these subjects. The final step is an aggregation over the properties of the newly joined triples table. In the property table solution, the selection predicate *Type=Text* is applied, and then the counts of the non-NULL values for each of the 28 columns is written to a temporary table. The counts are then selected out of the temporary table and unioned together to produce the correct results schema. The vertically partitioned store and column-store select the subjects for which the *Type* table has object value *Text*, and store these in a temporary table, *t*. They then union the results of joining each property's table with *t* and count all elements of the resulting joins.

Q3. On a triple-store, Q3 requires the same selection and self-join on subject as Q2. However, the aggregation groups by both property and object value.

The property table store applies the selection predicate *Type=Text* as in Q2, but is unable to perform the aggregation on all columns in a single scan of the property table. This is because grouping must be per property and then object for each column, and thus each column must group by the object values in that particular column (a single GROUP BY clause is not sufficient). The SQL standard describes GROUP BY GROUPING SETS to allow multiple GROUP BY aggregation groups to be performed in a single sequential scan of a table. Postgres does not implement this feature, and so our query plan requires a sequential scan of the table for each property aggregation (28 sequential scans), which should prove to be expensive. There is no way for us to accurately predict how the use of grouping sets would improve performance, but it should greatly reduce the number of sequential scans.

The vertical store and the column store work like they did in Q2, but perform a GROUP BY on the object column of each property after merge joining with the subject temporary table. They then union together the aggregated results from each property.

Q4. On a triple-store, Q4 has a selection for property=*Language* and object=*French* at the bottom of the query plan. This selection is joined with the *Type Text* selection (again a self-join on subject), before a second self-join on subject is performed to find the other properties and objects defined for this refined subject list.

The property table store performs exactly as it did in Q3, but adds an extra selection predicate on *Language=French*.

The vertically partitioned and column stores work as they did in Q3, except that the temporary table of subjects is further narrowed down by a join with subjects whose *Language* table has object=*French*.

Q5. On a triple-store, this requires a selection on property=*Origin* and object=*DLC*, followed by a self-join on subject to extract the other properties of these subjects. For those subjects with the *Records* property defined, we do a subject-object join to get the types of the subjects that were objects of the *Records* property.

For the property table approach, a selection predicate is applied on *Origin=DLC*, and the *Records* column of the resulting tuples is projected and (self) joined with the *subject* column of the original property table. The *type* values of the join results are extracted.

On the vertically partitioned and column stores, we perform the object=*DLC* selection on the *Origin* property, join these subjects with the *Records* table, and perform a subject-object join on the *Records* objects with the *Type* subjects to attain the inferred types.

Note that as described in Section 8.4, subject-object joins are slower than subject-subject joins because the object column is not sorted in any of the approaches. We discuss how the materialized path expression optimization described in Section 8.4 affects the results of this query and Q6 in Section 8.6.6.

Q6. On a triple-store, the query first finds subjects that are directly of *Type*: *Text* through a simple selection predicate, and then finds subjects that are inferred to be of *Type Text* by performing a subject-object join through the records property as in Q5. Next, it finds the other properties defined on this working set of subjects through a self-join on subject. Finally, it performs a count aggregation on these defined properties.

The property table, vertical partitioning, and column-store approaches first create temporary tables by the methods of Q2 and Q5, and perform aggregation in a similar fashion to Q2.

Q7. To implement Q7 on a triple-store, the selection on the *Point* property is performed, and then two self-joins are performed to extract the *Encoding* and *Type* values for the subjects that passed the predicate.

In the property table schema, the property table is narrowed down by a filter on *Point*, which is accessed by an index. At this point, the other three columns (subject, *Encoding*, *Type*) are projected out of the table. Because *Type* is multi-valued, we treat each of its two possible instances per subject separately, unioning the result of performing the projection out of the property table once for each of the two possible array values of *Type*.

In the vertically partitioned and column-store approaches, we join the filtered *Point* table's subject with those of the *Encoding* and *Type* tables, returning the result.

Since this query returns slightly less than 75,000 triples, we avoid the final join with the string dictionary table for this query since this would dominate query time and is the same for all four approaches. We are exploring intelligent caching techniques to reduce the cost of this final dictionary decoding step for high cardinality queries.

8.6.4 Results

The performance numbers for all seven queries on the four architectures are shown in Figure 8-8. All times presented in this section are the average of three runs of the queries. Between queries we copy a 2 GByte file to clear the operating system cache, and restart the database to clear any internal caches.

The property table and vertical partitioning approaches both perform a factor of 2-3 faster than the triple-store approach (the geometric mean[3] of their query times was 38 and 36 seconds respectively compared with 97 seconds for the triple-store approach[4]. C-Store added another factor of 10 performance improvement with a geometric mean of 3 seconds (and so is a factor of 32 faster than the triple-store).

To better understand the reasons for the differences in performance between approaches, we look at the performance differences for each query. For Q1, the property table and vertical partitioning numbers are identical because we use the idealized property table for each query, and since this query only accesses one property, the idealized property table is identical to the vertically partitioned table. The triple-store only performs a factor of two slower since it does not have to perform any joins for this query. Perhaps surprisingly, C-Store performs an order of magnitude better. To understand why, we broke the query down into pieces. First, we noted that the type property table in

---

[3]We use geometric mean – the *n*th root of the product of *n* numbers – instead of the arithmetic mean since it provides a more accurate reflection of the total speedup factor over the workload.

[4]If we hand-optimized the triple-store query plans rather than use the Postgres default, we were able reduce its geometric mean to 79 seconds; this demonstrates the fact that by introducing a number of self-joins, queries over a triple-store schema are very hard to optimize.

|	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geo. Mean
Triple Store	24.63	157	224.3	27.67	408.7	212.7	38.37	97
Prop. Table	12.66	18.37	579.8	28.54	47.85	101	6.1	38
Vert. Part.	12.66	41.7	71.3	35.49	52.34	84.6	13.25	36
C-Store	0.66	1.64	9.28	2.24	15.88	10.81	1.44	3
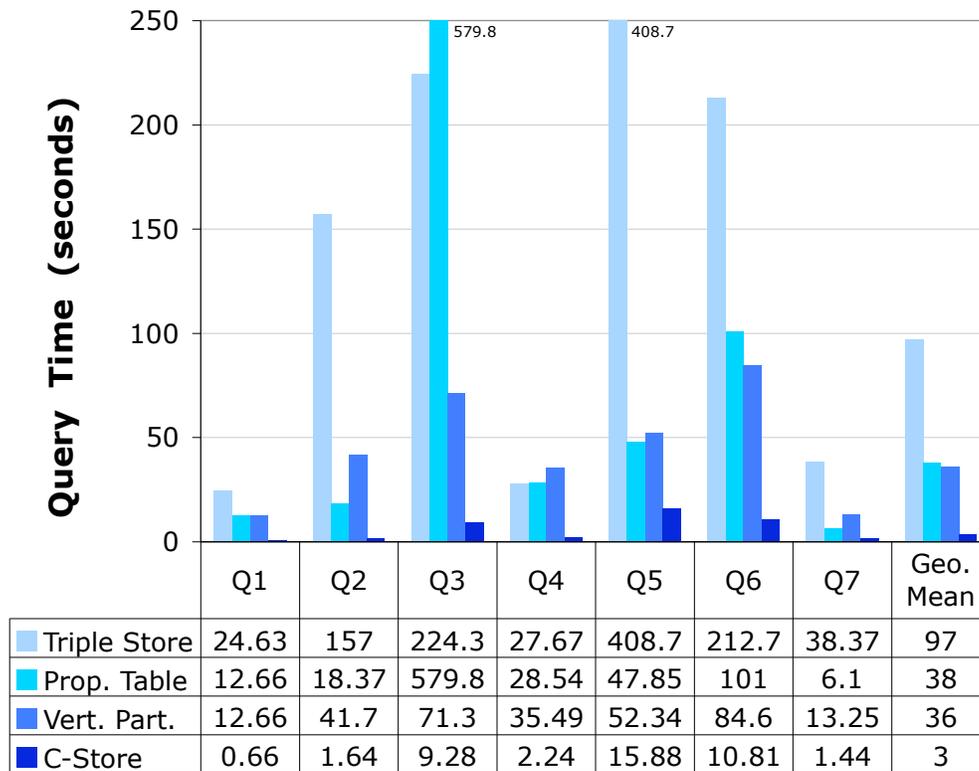
Figure 8-8: Performance comparison of the triple-store schema with the property table and vertically partitioned schemas (all three implemented in Postgres) and with the vertically partitioned schema implemented in C-Store. Property tables contain only the columns necessary to execute a particular query.

Postgres takes 472MB compared to just 100MB in C-Store. This is almost entirely due to the fact that the Postgres tuple header is 27 bytes compared with just 8 bytes of actual data per tuple and so the Postgres table scan needs to read 35 bytes per tuple (actually, more than this if one includes the pointer to the tuple in the page header) compared with just 8 for C-Store.

Another reason why C-Store performs better is that it uses an index nested loops join to join keys with the strings dictionary table while Postgres chooses to do a merge join. This final join takes 5 seconds longer in Postgres than it does in C-Store (this 5 second overhead is observable in the other queries as well). These two reasons account for the majority of the performance difference between the systems; however the other advantages of using a column-store described in Section 8.3.2 are also a factor.

Q2 shows why avoiding the expensive subject-subject joins of the triple-store is crucial, since the triple-store performs much more slowly than the other systems. The vertical partitioning approach is outperformed by the property table approach since it performs 28 merge joins that the property table approach does not need to do (again, the property table approach is helped by the fact that we use the optimal property table for each query).

As expected, the multiple sequential scans of the property table hurt it in Q3. Q4 is so highly selective that the query results for all but C-Store are quite similar. The results of the optimal property table in Q5-Q7 are on par with those of the vertically partitioned option, and show that subject-object joins hurt each of the stores significantly.

On the whole, vertically partitioning a database provides a significant performance improvement over the triple-store schema, and performs similarly to property tables. Given that vertical partitioning in a row-oriented database is competitive with the optimal scenario for a property table solution, we conclude that they are the preferable solution since they are simpler to implement. Further, if one uses a database designed for vertically partitioned data such as C-Store, additional performance improvement can be realized. C-Store achieved nearly-interactive time on our benchmark running on a single machine that is two years old.

We also note that multi-valued attributes play a role in reducing the performance of the property table approach. Because we implement multi-valued attributes in property tables as arrays, simple indexing can not be performed on these arrays, and the GiST [45] indexing of integer arrays performs worse than a sequential scan of the property table.

Finally, we remind the reader that the property tables for each query are idealized in that they only include the subset of columns that are required for the query. As we will show in Section 8.6.7, poor choice in columns for a property table will lead to less-than-optimal results, whereas the vertical partitioning solution represents the best- and worst-case scenarios for all queries.

Postgres as a Choice of RDBMS

There are several notes to consider that apply to our choice of Postgres as the RDBMS. First, for Q3 and Q4, performance for the property table approach would be improved if Postgres implemented GROUP BY GROUPING SETs.

Second, for the vertically partitioned schema, Postgres processes subject-subject joins non-optimally. For queries that feature the creation of a temporary table containing subjects that are to be joined with the subjects of the other properties' tables, we know that the temporary list of subjects will be in sorted order, as it comes from a table that is clustered on subject. Postgres does not carry this information into the temporary table, and will only perform a merge join for intermediate tuples that are guaranteed to be sorted. To simulate the fact that other databases would maintain the metadata about the sorted temporary subject list, we create a clustered index on the temporary table before the UNION-JOIN operation. We only included the time to create the temporary table and the UNION-JOIN operations in the total query time, as the clustering is a Postgres implementation artifact.

Further, Postgres does not assume that a table clustered on an attribute is in perfectly sorted order (due to possible modifications after the cluster operation), and thus can not perform the merge join directly; rather it does so in conjunction with an index scan, as the index is in sorted order. This process incurs extra seeks as the leaves of the B+ tree are traversed, leading to a significant cost effect compared to the inexpensive merge join operations of C-Store.

With a different choice of RDBMS, performance results might differ, but we remain convinced that Postgres was a good choice of RDBMS, given that it handles NULL values so well, and thus enabled us to fairly benchmark the property table solutions.

8.6.5   Scalability

Although the magnitude of query performance is important, an arguably more important factor to consider is how performance scales with size of data. In order to determine this, we varied the number of triples we used from the library dataset from one million to fifty million (we randomly chose what triples to use from a uniform distribution) and reran the benchmark queries. Figure 8-9 shows the results of this experiment for query 6. Both vertical partitioning schemes (Postgres and C-Store) scale linearly, while the triple-store scales super-linearly. This is because all joins for this query are linear for the vertically partitioned schemes (either merge joins for the subject-subject joins, or index scan merge joins for the subject-object inference step); however the triple-store sorts the intermediate results after performing the three selections and before performing the merge join. We observed similar results for all queries except queries 1, 4, and 7 (where the triple-store also scales linearly, but with a much higher slope relative to the vertically partitioned schemes).

8.6.6   Materialized Path Expressions

As described in Section 8.4, materialized path expressions can remove the need to perform expensive subject-object joins by adding additional columns to the property table or adding an extra table to the vertically partitioned and column-oriented solutions. This makes it possible to replace subject-object joins with cheaper subject-subject joins.
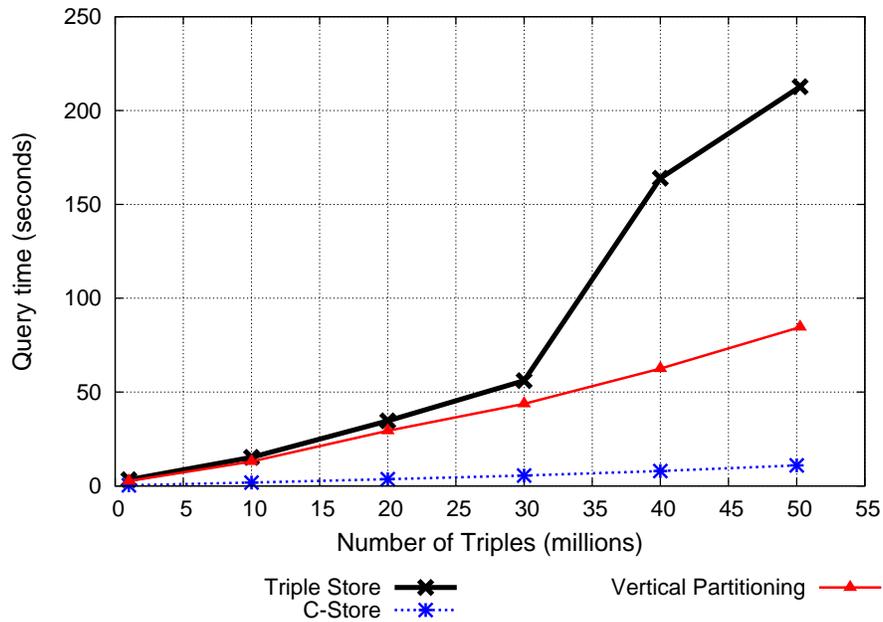
Figure 8-9: Query 6 performance as number of triples scale.

	Q5	Q6
Property Table	39.49 (17.5% faster)	62.6 (38% faster)
Vertical Partitioning	4.42 (92% faster)	65.84 (22% faster)
C-Store	2.57 (84% faster)	2.70 (75% faster)

Table 8.2: Query times (in seconds) for Q5 and Q6 after the Records:Type path is materialized. % faster $= \frac{100|original-new|}{original}$.

Since Queries 5 and 6 contain subject-object joins, we reran just those experiments using materialized path expressions. Recall that in these queries we join object values from the *Records* property with subject values to get those subjects that can be inferred to be a particular type through the *Records* property.

For the property table approach, we widened the property table by adding a new column representing the materialized path expression: *Records:Type*. This column indicates the type of entity that is related to a subject through the *Records* property (if a subject does not have a *Records* property defined, its value in this column will be NULL). Similarly, for the vertically partitioned and column-oriented solutions, we added a table containing a subject column and a *Records:Type* object column, thus allowing one to find the *Type* of objects that a resource *Records* with a cheap subject-subject merge join. The results are displayed in Table 8.2.

It is clear that materializing the path expression and removing the subject-object join results in significant improvement for all schemas. However, the vertically partitioned schemas see a greater benefit since the materialized path expression is multi-valued (which is the common case, since if at least one property along the path is multi-valued, then the materialized result will be multi-valued).

In summary, Q5 and Q6, which used to take 400 and 200 seconds respectively on the triple-store, now take less than three seconds on the column-store. This represents a two orders of magnitude performance improvement!

8.6.7 The Effect of Further Widening

Given that semantic web content is likely to have an unstructured schema, clustering algorithms will not always yield property tables that are the perfect width for all queries. We now experimentally demonstrate the effect of

Query	Wide Property Table	Property Table % slowdown
Q1	60.91	381%
Q2	33.93	85%
Q3	584.84	1%
Q4	44.96	58%
Q5	76.34	60%
Q6	154.33	53%
Q7	24.25	298%

Table 8.3: Query times in seconds comparing a wider than necessary property table to the property table containing only the columns required for the query. % Slowdown = $\frac{100|original-new|}{original}$. Vertically partitioned stores are not affected.

property tables that are wider than they need to be for the same queries run in the experiments above. Row-stores traditionally perform poorly relative to vertically partitioned schemas and column-stores when queries need to access only a few columns of a wide table, so we expect the performance of the property table implementation to degrade with increasing table width. To measure this, we synthetically added 60 non-sparse random integer-valued columns to the end of each tuple in the widest property table in Postgres. This resulted in an approximately 7 GByte increase in database size. We then re-ran Q1-Q7 on this wide property table. The results are shown in Table 8.3.

Since each of the queries (except query 1) executes in two parts, first creating a temporary table containing the subset of the relevant data for that query, and then executing the rest of the query on this subset, we see some variance in % slowdown numbers, where smaller slowdown numbers indicate that a majority of the query time was spent on the second stage of query processing. However, each query sees some degree of slowdown. These results support the notion that while property tables can sometimes outperform vertical partitioning on a row-oriented store, a poor choice of property table can result in significantly poorer query performance. The vertically partitioned solutions are impervious to such effects.

8.7 Conclusion

The emergence of the Semantic Web necessitates high-performance data management tools to manage the tremendous collections of RDF data being produced. Current state of the art RDF databases – triple-stores – scale extremely poorly since most queries require multiple self-joins on the triples table. The previously proposed "property table" optimization has not been adopted in most RDF databases, perhaps due to its complexity and inability to handle multi-valued attributes. This chapter showed that a poorly-selected property table can result in a factor of 3.8 slowdown over an optimal property table, thus making the solution difficult to use in practice. As an alternative to property tables, the chapter proposed vertically partitioning tables and demonstrated that they achieve similar performance as property tables in a row-oriented database, while being simpler to implement. Further, on C-Store, it was possible to achieve a factor of 32 performance improvement over the current state of the art triple store design. Queries that used to take hundreds of seconds can now be run in less than ten seconds, a significant step toward interactive-time semantic web content storage and querying.

Chapter 9

Conclusions And Future Work

Column-oriented database systems, due to their I/O efficiency on read-mostly, attribute-oriented queries are being increasingly adopted in the commercial marketplace. This trend has resulted in a variety of venture-backed column-oriented database start-ups bursting onto the scene, including Vertica, ParAccel, CalPont, and in the increasing popularity of column-oriented databases that have been around for a while (including Sybase IQ, Sand/DNA Analytics, and SenSage). It is clear that column-stores are well suited for analytical workloads like those found in data warehouses that serve customer relationship management and business intelligence applications.

Both in the commercial marketplace and in the research literature, these different variants of column-stores make different architectural decisions and make different claims on their performance relative to standard row-store technology. In this dissertation, we classified these different column-store variants into three primary implementation approaches, and showed that fundamental differences between these approaches result in significantly different performance. In essence, we found that if the DBMS is designed from scratch for column-oriented data layout, building a storage layer and a query executor around this this data layout, a significant performance improvement can be obtained over less aggressive (but easier to implement) approaches.

As a result of this finding, we set out to build such a column-store, with a specially designed storage layer and query executor. We presented the architecture and implementations of this new column-store: C-Store. We then presented three column-oriented optimizations to its query executor: operating on compressed data, late materialization, and the invisible join, that further increase system performance. After adding these optimizations, we benchmarked the performance of the complete C-Store system on two different applications: data warehousing and the Semantic Web.

On the data warehousing benchmark, we found that the complete C-Store system performed an order of magnitude faster than an alternative approach to building column-stores where the storage layer is designed around the column-oriented data layout, but the query executor is not (relevant columns are merged before query execution and standard row-oriented query execution is used), with queries running in 4.4 seconds (on average) instead of 40.7 seconds. In turn, both approaches are faster than the simple approach of vertically partitioning a row-store to emulate a column-store (which took 79.9s on average). Thus, the vertical partitioning strategy was a factor of 18 slower than C-Store.

Further, we compared C-Store's performance to a row-store on the same benchmark. We found that C-Store, despite being at a disadvantage since it does not implement some basic performance enhancing techniques such as horizontal partitioning and multi-threading, still outperforms a commercial row-store by a factor of 6 on average. These results are nicely summarized in Figure 9-1 which presents results for Query 2.3 of the SSBM, where the row-store does not benefit from its horizontal partitioning feature. The order of magnitude performance difference is apparent.

The other application that was benchmarked in this dissertation was one not traditionally thought of as an application well-suited for column-stores. The application was a Semantic Web application with data in RDF "triples" format. Performance and scalability issues are becoming increasingly pressing as Semantic Web technology is applied to real-world applications. We compared the performance of C-Store with other current data management

Sample SSBM Query (2.3)

|	Row-Store	Vertical-Partitioning	Column-Store Approach 2	Column-Store Approach 3	Column-Store Approach 3 (revisited)
Sample SSBM Query (2.3)	46.0	39.0	34.8	13.3	4.0
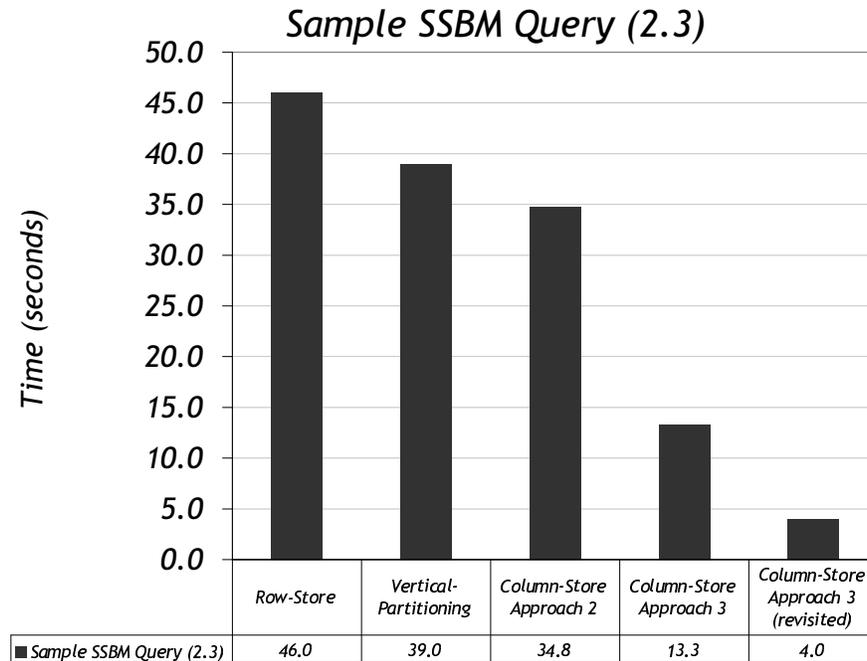
Figure 9-1: Another comparison of different column-oriented database implementation techniques (updated from Figure 1-1. Here "Column-Store Approach 2" refers to the column-store implementation technique of Chapter 2 where the storage layer but not the query executor is modified for column-oriented data layout, and "Column-Store Approach 3" refers to the column-store implementation technique of Chapter 2 where both the storage layer and the query executor are modified for column-oriented data layout. "Column-Store Approach 3 (revisited)" refers to the same implementation approach, but this time with all of the column-oriented query executor optimizations presented in this dissertation implemented.

solutions for RDF data using queries generated by a Web-based RDF browser over a large-scale (more than 50 million triples) catalog of library data. Our results showed that C-Store achieves an order magnitude performance improvement relative to other state-of-the-art techniques (query times drop from minutes to several seconds), while being much simpler to design and having superior scaling properties.

In Chapter 7, we broke down the reasons for C-Store's high performance relative to other column-store implementation approaches and to row-stores. We found that two column-oriented query executor optimizations contributed most: compression and late materialization. By compressing data to reduce I/O, and by delaying tuple construction until an auspicious point in the query plan so that only necessary tuples need be constructed, fast, vectorized operations can be performed on columns, and data can be compressed and operated on directly through-out a query plan. We discuss these two optimizations further in the next subsections.

Compression

Column-oriented database system architectures invite a re-evaluation of how and when data in databases is compressed. Storing data in a column-oriented fashion greatly increases the similarity of adjacent records on disk and thus opportunities for compression. The ability to compress many adjacent tuples at once lowers the per-tuple cost of compression, both in terms of CPU and space overheads.

In Chapter 4, we discussed how we built a compression sub-system for C-Store. We showed how compression schemes not traditionally used in row-oriented DBMSs can be implemented in column-oriented systems. We then

evaluated a set of compression schemes and showed that the best scheme depends not only on the properties of the data, but also on the nature of the query workload.

Finally, we showed how the query executor of a column-oriented database can be modified so that compressed data can be directly operated upon, which reduces CPU costs of decompression and in some cases allows for multiple values to be processed simultaneously.

Tuple Materialization

In order for column-stores to be readily adopted as a replacement for row-stores, they must present the same interface to client applications as do row stores, which implies that they must output row-store-style tuples. In other words, the input columns stored on disk must be converted to rows at some point in the query plan. However, the optimal point at which to do the conversion is not obvious. This problem can be considered as the opposite of the projection problem in row-store systems: while row-stores need to determine where in query plans to place projection operators to make tuples narrower, column-stores need to determine when to combine single-column projections into wider tuples.

In Chapter 5, we described a variety of strategies for tuple construction and intermediate result representations and provided a systematic evaluation of these strategies. In many cases, waiting as long as possible to construct tuples is advantageous, since selection predicates and aggregations reduce the number of tuples that ultimately need to be constructed. Further, by keeping data in columns, data can be iterated through directly as an array rather than indirectly through a tuple iterator, which can greatly speed up predicate and expression evaluation. Finally, the advantages of processing multiple (column-oriented) compressed values simultaneously can only occur before the materialization point.

However, since late materialization potentially incurs additional costs due to re-processing disk blocks, early materialization is sometimes preferable. A good heuristic to use is that if output data is aggregated, or if the query has low selectivity (highly selective predicates), or if input data is compressed using a light-weight compression technique, a late materialization strategy should be used. Otherwise, for high selectivity, non-aggregated, non-compressed data, early materialization should be used. Further, the right input table to a join should be materialized before (or during if a multi-column is input) the join operation unless, as was seen in the star schema benchmark experiments, the join column of the inner table is a primary key and fits in memory.

Chapter 5 also presented an analytical model that can be used to predict query performance and help choose a materialization strategy at query planning and optimization time.

The Bottom Line

We conclude that in order to reap the advantages of a column-oriented database, relying on improved I/O from reading fewer columns only yields a fraction of the performance potential. The rest of the performance improvements are gained by building an executor designed and optimized for the column-oriented layout. The ability to operate on compressed data and perform tuple construction late in a query plan are key components of such a query executor.

9.1   Future Work

Although the C-Store query executor achieves impressive performance on its own, there is still very little interaction between the executor and the rest of the database system. Most importantly, many of the techniques described in this dissertation have not been incorporated into the C-Store optimizer. The optimizer is not aware of the decompression costs of the various compression algorithms, nor able to use the analytical model presented in this dissertation to decide on a materialization strategy. Further, since column-stores are very I/O efficient (as a result of only having to read relevant columns off disk, data compression, and extensive prefetching), we have found that most queries are CPU (and not disk) limited. Thus, good CPU cost models for each step in a query execution plan need to be

incorporated into a cost-based optimizer. Hence, the design of an optimizer designed for a column-oriented database (and perhaps also for data warehouse workloads) remains an interesting area for future work.

This dissertation also has implications on database design. The chapter on compression showed that many narrow materialized views are preferable to fewer wider materialized views since a higher percentage of columns can be sorted (or secondarily sorted) in narrow tables. The chapter on the invisible join showed that in many cases, it does not help to include dimension table columns in fact table materialized views. The design of an automatic database designer that decides what materialized views to create and how data should be sorted/compressed in column-oriented databases/data warehouses remains another interesting area for future work.

Column-stores can be thought of as an extreme opposite of row-stores (at least with regard data layout). This observation leads to the question of whether there are situations where a *Goldilocks* (hybrid) store might be able to do better than a column- or row-store could do individually. There has been published work on using redundancy to get the best of both worlds (have one copy of a database as a column-store and another as a row-store) [58], but there does not seem to be much work on combining column- and row-stores inside the same database table. For example, one might want to store sorted attributes and variable-width attributes in a columns, and fixed-width attributes in (dense-packed) rows. Or one might want to store the less frequently accessed attributes in rows and the more frequently accessed attributes in columns. Goldilocks-stores could be an interesting area of research.

Load times into column-stores is another avenue for future work. Certainly, it is expected that loading a column-oriented database will be slower than loading a row-oriented database (especially if there are many materialized views), but it would be interesting to evaluate the extent of this disadvantage look at algorithms to alleviate it.

Finally, the chapter on processing Semantic Web data also only focused on query execution. The design of a complete database for Semantic Web applications, including parsing from SPARQL to SQL, query rewriting to a vertically partitioned schema, and path-expression optimization, would be an exciting system to build.

Appendix A

C-Store Operators

There are a total of 19 operators implemented in C-Store. Here, we present each of these operators in alphabetical order. Note that whenever the term "stream" is used in the attribute descriptions, this refers to the sequence of blocks (or rather, iterators pointing to blocks) that are input to the operator. Some operations, like "compress" or "decompress" are not listed here since they are done at the block-level and performed only if necessary and on-the-fly at runtime. This is explained further in Chapter 4.

- **Aggregate** takes a group-by column and an aggregation column and produces an aggregate value for each unique value in the group-by column. If no group-by column is provided, a single aggregation on the entire column is performed. C-Store supports sum, average, count, maximum, and minimum aggregation functions. C-Store uses one of two algorithms to perform the aggregation (both are single pass algorithms). The most common algorithm is a "hash aggregation", where a hash table maps a group-by value to the running aggregation total. Alternatively, if the group-by column is sorted, a "pipelined" aggregation can be used, where a hash table is not needed since once a particular group-by value is seen, it is guaranteed not to appear again.

- **AttributeCombine** takes n columns and creates a single column whose ith value is the concatenation of the ith values from each of the n columns. This operator often appears before an aggregation operator with a multi-dimensional group-by clause (each dimension in the group by clause is combined together so that the aggregation operator only has to group by a single attribute). Note that if all values from a table are input to this operator, the result is a set of original "tuples", so this operator can also be used to reconstruct tuples early in a query plan.

- **AttributeUncombine** performs the converse operation to the AttributeCombine operator. It takes single column and divides it up into multiple columns. This operator often appears after an aggregation operator with a multi-dimensional group-by clause (each dimension in the group by clause was combined together before the aggregation operator and might need to be separated for further operation after being aggregated).

- **BlockPrinter** takes n input columns and merges them together into rows to be output to a file or standard output. Its function is very similar to AttributeCombine except that human readable space is used to separate attributes. Every C-Store query plan contains a BlockPrinter operator at the root (and nowhere else in the query plan).

- **Coagulate** reads in a sequence of small blocks and combines them together into larger (64K) blocks. Although all blocks on disk are stored in 64K blocks, they often get smaller as they are processed over the course of a query plan as data is filtered or aggregated. This operator exists solely as a performance enhancement tool since operators must make a call to getNextBlock once for each input block, and this method call takes a proportionally larger amount of CPU time as the number of values included in a block decreases (the extreme case, where each block contains one value, is known as the *tuple iterator model* in database literature and performs poorly). Further, the column-oriented performance enhancement of operating directly on arrays (see

```
currpos = 1
while both inputs still have data
   for each input stream, keep reading in blocks until we reach the one containing currpos
   endpos = minimum end position of two input blocks
   create an iterator from currpos to endpos on both input blocks
   call PosAnd on one of the input interators sending it a pointer to the second
   currpos = endpos+1
```

Figure A-1: Pseudocode for PosAnd Operator

Section 3.6) is only observed when arrays are of reasonable size. Thus, combining multiple small blocks into larger blocks can be a useful performance enhancement tool.

- **DataSource** operators are the leaf operators in query plans. They read columns off storage, optionally applying a predicate as this is done. They can produce either column values, or column positions of values that passed a predicate. DataSources can optionally take a position filter where they read only input from these specified positions rather than all positions in a column.

- **Duplicate** allows the result of an operator to be used an input to multiple parent operators. This is not done by copying the result, rather an iterator on the result block is created and sent to each parent. Consequently, each parent must read the input at the same rate as only one block is held in memory by the operator at a time, and iterators are invalidated when a new block is read in.

- **Eval** takes two input columns and produces a single output column whose ith value is calculated by performing an arithmetic operation on the ith value of the input columns (addition, subtraction, multiplication or division).

- **FlattenWithPos** converts a sparse (many NULLs) block into a normal block with no NULL values. The positions of the non-NULL values are returned in a position block.

- **FlattenWithoutPos** converts a sparse block into a normal block with no NULL values. The positions of the non-sparse values are remapped to consecutive positions starting from 1.

- **FlattenGetPos** performs the same operation as FlattenWithPos, except that it only returns the position block indicating the location of the non-NULL values (the actual non-NULL values are ignored).

- **Ident** forwards all blocks that are input to it to its parent operator.

- **Merge and Union** operators merge results from multiple sub-query plans (e.g. from the query plans for the RS and WS). Union simply serializes the results, Merge will regroup and reaggregate if the sub-queries contain aggregation operations.

- **NLJoin and HashJoin**. C-Store contains both a generic nested loops join and an in-memory hash join. Unlike row-stores, a column-store has three different alternatives for how data should be input and output from a join. This will be described in more detail in later chapters.

- **PosAnd** takes two (sorted) position streams as input and produces a single position stream as output that contains the intersection of the positions in the input streams. Most of the PosAnd code is performed inside iterators on position blocks (these are just like iterators on value blocks described above). The basic pseudocode for this operator is shown in Figure A-1 (the actual code has a few minor optimizations).

  The reason why the actual intersection of positions is done inside the position block iterator is that different algorithms can be used depending on the specific way the position block represents positions. Range position blocks (i.e. a set of consecutive positions like 1-2,000), for example, know that the intersection of a range block with any other block is equal to that other block. As another example, bit-vector position blocks can use

bit-and operations. Coding this knowledge inside the position blocks makes C-Store more extensible, as the PosAnd operator does not need to contain knowledge about all possible position representations.

- **PosOr** takes two (sorted) position streams as input and produces a single position stream as output that contains the union of the positions in the input streams. Like PosAnd, most of the PosOr code is performed inside position block iterators.

- **RowSelect** performs a row-store selection operation. This operator is only used when the selection can not be pushed down to a DataSource leaf operator and columns have been merged into rows by an AttributeCombine operator..

- **SColumnExtracter** reads in an ASCII file from disk, partitions it into columns, and converts the data to blocked binary format so that it can be operated on using C-Store operators or loaded into C-Store.

- **Serialize** takes a set of input columns and outputs a single column representing a vertical concatenation of the inputs (rather than a horizontal concatenation as in AttributeCombine). The entire first column is returned followed by the second, etc.

Appendix B

Star Schema Benchmark Queries

Query 1:

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
  and d_year = 1993
  and lo_discount between 1 and 3
  and lo_quantity < 25;
```

Query 2:

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
  and d_yearmonthnum = 199401
  and lo_discount between 4 and 6
  and lo_quantity between 26 and 35;
```

Query 3:

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
  and d_weeknuminyear = 6 and d_year = 1994
  and lo_discount between 5 and 7
  and lo_quantity between 36 and 40;
```

Query 4:

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
  and lo_partkey = p_partkey
  and lo_suppkey = s_suppkey
  and p_category = 'MFGR#12' -- OK to add p_mfgr = 'MFGR#1'
  and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Query 5:

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
  and lo_partkey = p_partkey
  and lo_suppkey = s_suppkey
  -- OK to add p_mfgr='MFGR#2'
  -- OK to add p_category='MFGR#22'
  and p_brand1 between 'MFGR#2221' and 'MFGR#2228'
  and s_region = 'ASIA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Query 6:

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
  and lo_partkey = p_partkey
  and lo_suppkey = s_suppkey
  -- OK to add p_mfgr='MFGR#2'
  -- OK to add p_category='MFGR#22'
  and p_brand1 = 'MFGR#2221'
  and s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Query 7:

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, dwdate
where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_orderdate = d_datekey
  and c_region = 'ASIA'
  and s_region = 'ASIA'
  and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Query 8:

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, dwdate
where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_orderdate = d_datekey
  and c_nation = 'UNITED STATES'
  and s_nation = 'UNITED STATES'
  and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Query 9:

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, dwdate
where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_orderdate = d_datekey
  and c_nation = 'UNITED KINGDOM'
  and (c_city='UNITED KI1' or c_city='UNITED KI5')
  and (s_city='UNITED KI1' or s_city='UNITED KI5')
  and s_nation = 'UNITED KINGDOM'
  and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Query 10:

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, dwdate
where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_orderdate = d_datekey
  and c_nation = 'UNITED KINGDOM'
  and (c_city='UNITED KI1' or c_city='UNITED KI5')
  and (s_city='UNITED KI1' or s_city='UNITED KI5')
  and s_nation = 'UNITED KINGDOM'
  and d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Query 11:

```
select d_year, c_nation, sum(lo_revenue-lo_supplycost) as profit1
from dwdate, customer, supplier, part, lineorder
where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_partkey = p_partkey
  and lo_orderdate = d_datekey
  and c_region = 'AMERICA'
  and s_region = 'AMERICA'
  and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation;
```

Query 12:

```
select d_year, s_nation, p_category, sum(lo_revenue-lo_supplycost) as profit1
from dwdate, customer, supplier, part, lineorder
where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_partkey = p_partkey
  and lo_orderdate = d_datekey
```

```
  and c_region = 'AMERICA'
  and s_region = 'AMERICA'
  and (d_year = 1997 or d_year = 1998)
  and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category;
```

Query 13:

```
select d_year, s_city, p_brand1, sum(lo_revenue-lo_supplycost) as profit1
from dwdate, customer, supplier, part, lineorder
where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_partkey = p_partkey
  and lo_orderdate = d_datekey
  and c_region = 'AMERICA'
  and s_nation = 'UNITED STATES'
  and (d_year = 1997 or d_year = 1998)
  and p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1;
```

Appendix C

Longwell Queries

The queries below are the seven benchmark queries used in Chapter 8 as implemented on a triple-store. Note that while these queries are accurately described, we dictionary encode all strings into their own table, and thus the triples table contains integer IDs which are foreign references into the string table. The actual queries feature selection predicates on integer values, and have a post-processing step of joining the strings back onto the result set.

    The properties table listed in these queries contains the list of 28 properties that are processed for queries 2, 3, 4, and 6. This table is presented in full in the Appendix D.

Query1:

```
SELECT A.obj, count(*)
FROM triples AS A
WHERE A.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
GROUP BY A.obj
```

Query2:

```
SELECT B.prop, count(*)
FROM triples AS A, triples AS B,
  properties AS P
WHERE A.subj = B.subj
  AND A.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
  AND A.obj = "<http://simile.mit.edu/2006/01/ontologies/mods3#Text>"
  AND P.prop = B.prop
GROUP BY B.prop
```

Query3:

```
SELECT B.prop, B.obj, count(*)
FROM triples AS A, triples AS B,
  properties AS P
WHERE A.subj = B.subj
  AND A.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
  AND A.obj = "<http://simile.mit.edu/2006/01/ontologies/mods3#Text>"
  AND P.prop = B.prop
GROUP BY B.prop, B.obj
HAVING count(*) > 1
```

Query4:

```
SELECT B.prop, B.obj, count(*)
FROM triples AS A, triples AS B,
   triples AS C, properties AS P
WHERE A.subj = B.subj
  AND A.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
  AND A.obj = "<http://simile.mit.edu/2006/01/ontologies/mods3#Text>"
  AND P.prop = B.prop
  AND C.subj = B.subj
  AND C.prop = "<http://simile.mit.edu/2006/01/ontologies/mods3#language>"
  AND C.obj =
    "<http://simile.mit.edu/2006/01/language/iso639-2b/fre>"
GROUP BY B.prop, B.obj
HAVING count(*) > 1
```

Query5:

```
SELECT B.subj, C.obj
FROM triples AS A, triples AS B,
     triples AS C
WHERE A.subj = B.subj
  AND A.prop = "<http://simile.mit.edu/2006/01/ontologies/mods3#origin>"
  AND A.obj = "<info:marcorg/DLC>"
  AND B.prop = "<http://simile.mit.edu/2006/01/ontologies/mods3#records>"
  AND B.obj = C.subj
  AND C.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
  AND C.obj != "<http://simile.mit.edu/2006/01/ontologies/mods3#Text>"
```

Query6:

```
SELECT A.prop, count(*)
FROM triples AS A, properties AS P (
 (SELECT B.subj
  FROM triples AS B
  WHERE B.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
  AND B.obj = "<http://simile.mit.edu/2006/01/ontologies/mods3#Text>")
UNION
 (SELECT C.subj
  FROM triples AS C, triples AS D
  WHERE C.prop = "<http://simile.mit.edu/2006/01/ontologies/mods3#records>"
  AND C.obj = D.subject
  AND D.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"
  AND D.obj = "<http://simile.mit.edu/2006/01/ontologies/mods3#Text>")
) AS uniontable
WHERE A.subj = uniontable.subj
  AND P.prop = A.prop
GROUP BY A.prop;
```

Query7:

```
SELECT A.subj, B.obj,
  C.obj
```

```
FROM triples AS A, triples AS B,
     triples AS C
WHERE A.prop = "<http://simile.mit.edu/2006/01/ontologies/mods3#Point>"
  AND A.obj = '"end"'
  AND A.subj = B.subject
  AND B.prop = "<http://simile.mit.edu/2006/01/ontologies/mods3#Encoding>"
  AND A.subj = C.subject
  AND C.prop = "<http://www.w3.org/1999/02/22-rdf-syntax-ns#Type>";
```

Appendix D

Properties Table

The 28 properties contained in the properties table used in Chapter 8 are:

```
<http://simile.mit.edu/2006/01/ontologies/mods3#access>
<http://simile.mit.edu/2006/01/ontologies/mods3#address>
<http://simile.mit.edu/2006/01/ontologies/mods3#affiliation>
<http://simile.mit.edu/2006/01/ontologies/mods3#authority>
<http://simile.mit.edu/2006/01/ontologies/mods3#catalogingLanguage>
<http://simile.mit.edu/2006/01/ontologies/mods3#code>
<http://simile.mit.edu/2006/01/ontologies/mods3#contents>
<http://simile.mit.edu/2006/01/ontologies/mods3#copyrightDate>
<http://simile.mit.edu/2006/01/ontologies/mods3#dateCreated>
<http://simile.mit.edu/2006/01/ontologies/mods3#dates>
<http://simile.mit.edu/2006/01/ontologies/mods3#edition>
<http://simile.mit.edu/2006/01/ontologies/mods3#encoding>
<http://simile.mit.edu/2006/01/ontologies/mods3#extent>
<http://simile.mit.edu/2006/01/ontologies/mods3#fullName>
<http://simile.mit.edu/2006/01/ontologies/mods3#issuance>
<http://simile.mit.edu/2006/01/ontologies/mods3#language>
<http://simile.mit.edu/2006/01/ontologies/mods3#nonSort>
<http://simile.mit.edu/2006/01/ontologies/mods3#origin>
<http://simile.mit.edu/2006/01/ontologies/mods3#partName>
<http://simile.mit.edu/2006/01/ontologies/mods3#partNumber>
<http://simile.mit.edu/2006/01/ontologies/mods3#point>
<http://simile.mit.edu/2006/01/ontologies/mods3#qualifier>
<http://simile.mit.edu/2006/01/ontologies/mods3#records>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://simile.mit.edu/2006/01/ontologies/mods3#sub>
<http://simile.mit.edu/2006/01/ontologies/mods3#changed>
<http://simile.mit.edu/2006/01/ontologies/mods3#created>
<http://simile.mit.edu/2006/01/ontologies/mods3#physicalDescription>
```

Bibliography

[1] Library catalog data. `http://simile.mit.edu/rdf-test-data/barton/`.

[2] Longwell website. `http://simile.mit.edu/longwell/`.

[3] LZOP compression code. `http://www.lzop.org`.

[4] Redland RDF Application Framework. `http://librdf.org/`.

[5] Simile website. `http://simile.mit.edu/`.

[6] Swoogle. `http://swoogle.umbc.edu/`.

[7] TPC-H. `http://www.tpc.org/tpch/`.

[8] TPC-H Result Highlights Scale 1000GB. `http://www.tpc.org/tpch/results/tpch_result_detail.asp?id=107102903`.

[9] UniProt RDF Dataset. `http://dev.isb-sib.ch/projects/uniprot-rdf/`.

[10] Vertica. `http://www.vertica.com/`.

[11] WordNet RDF Dataset. `http://www.cogsci.princeton.edu/~wn/`.

[12] World Wide Web Consortium (W3C). `http://www.w3.org/`.

[13] RDF Primer. W3C Recommendation. `http://www.w3.org/TR/rdf-primer`, 2004.

[14] RDQL - A Query Language for RDF. W3C Member Submission 9 January 2004. `http://www.w3.org/Submission/RDQL/`, 2004.

[15] C-Store code release under BSD license. `http://db.csail.mit.edu/projects/cstore/`, 2005.

[16] SPARQL Query Language for RDF. W3C Working Draft 4 October 2006. `http://www.w3.org/TR/rdf-sparql-query/`, 2006.

[17] Daniel J. Abadi. Column stores for wide and sparse data. In *CIDR*, Asilomar, CA, USA, 2007.

[18] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, Chicago, IL, USA, 2006.

[19] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, pages 466–475, Istanbul, Turkey, 2007.

[20] Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and Querying of E-Commerce Data. In *VLDB*, 2001.

[21] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[22] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. In *VLDB*, pages 329–338, 2000.

[23] G. Antoshenkov. Byte-aligned data compression. U.S. Patent Number 5,363,098.

[24] Gennady Antoshenkov, David B. Lomet, and James Murray. Order preserving compression. In *ICDE '96*, pages 655–663. IEEE Computer Society, 1996.

[25] Jennifer Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *ICDE*, 2006.

[26] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

[27] Peter Boncz, Stefan Manegold, and Martin Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[28] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[29] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal: Very Large Data Bases*, 8(2):101–119, 1999.

[30] Valerie Bonstrom, Annika Hinze, and Heinz Schweppe. Storing RDF as a graph. In *Proc. of LA-WEB*, 2003.

[31] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *ISWC*, pages 54–68, 2002.

[32] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In *SIGMOD '01*, pages 271–282, 2001.

[33] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *Proc. of VLDB*, pages 1216–1227, 2005.

[34] George Copeland and Setrag Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[35] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proc. of SIGMOD*, pages 268–279, 1985.

[36] Gordon V. Cormack. Data compression on a database system. *Commun. ACM*, 28(12):1336–1342, 1985.

[37] John Corwin, Avi Silberschatz, P. L. Miller, and L. Marenco. Dynamic tables: An architecture for managing evolving, heterogeneous biomedical data in relational database management systems. *Journal of the American Medical Informatics Association*, 14(1):86–93, 2007.

[38] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[39] G.Graefe and L.Shapiro. Data compression and database performance. In ACM/IEEE-CS Symp. On Applied Computing pages 22 -27, April 1991.

[40] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *ICDE '98*, pages 370–379, 1998.

[41] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. 6:120–135, 1994.

[42] Alan Halverson, Jennifer L. Beckmann, Jeffrey F. Naughton, and David J. Dewitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University of Wisconsin-Madison, 2006.

[43] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel R. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, Seoul, Korea, 2006.

[44] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *In Proc. of PSSS'03*, pages 1–15, 2003.

[45] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proc. of VLDB 1995, Zurich, Switzerland*, pages 562–573.

[46] D. Huffman. A method for the construction of minimum-redundancy codes. Proc. IRE, 40(9):1098-1101, September 1952.

[47] Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In *VLDB '94*, pages 695–704, 1994.

[48] Theodore Johnson. Performance measurements of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

[49] Setrag Khoshafian, George Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In *ICDE*, pages 636–643, 1987.

[50] Clifford A. Lynch and E. B. Brownrigg. Application of data compression to a large bibliographic data base. In *VLDB '81, Cannes, France*, pages 435–447, 1981.

[51] Roger MacNicol and Blaine French. Sybase IQ multiplex - designed for analytics. In *VLDB, pp. 1227-1230*, 2004.

[52] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1):1–9, 1994.

[53] Carl Olofson. Worldwide RDBMS 2005 vendor shares. Technical Report 201692, IDC, May 2006.

[54] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3):8–11, 1995.

[55] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[56] Patrick E. O'Neil, Xuedong Chen, and Elizabeth J. O'Neil. Adjoined Dimension Column Index (ADC Index) to Improve Star Schema Query Performance. In *Proc. of ICDE*, 2008.

[57] Patrick E. O'Neil, Elizabeth J. O'Neil, and Xuedong Chen. The Star Schema Benchmark (SSB). `http://www.cs.umb.edu/˜poneil/StarSchemaB.PDF`.

[58] Ravishankar Ramamurthy, David Dewitt, and Qi Su. A case for fractured mirrors. In *VLDB*, pages 89 – 101, 2002.

[59] Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, 1995.

[60] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, 1993.

[61] Dennis G. Severance. A practitioner's guide to data base compression - tutorial. *Inf. Syst.*, 8(1):51–62, 1983.

[62] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[63] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564, Trondheim, Norway, 2005.

[64] Michael Stonebraker, Chuck Bear, Ugur Cetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One size fits all? - part 2: Benchmarking results. In *Proceedings of the Third International Conference on Innovative Data Systems Research (CIDR)*, January 2007.

[65] Dan Vesset. Worldwide data warehousing tools 2005 vendor shares. Technical Report 203229, IDC, August 2006.

[66] Andreas Weininger. Efficient execution of joins in a star schema. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 542–545, 2002.

[67] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, 2000.

[68] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in jena2. In *SWDB*, pages 131–150, 2003.

[69] Kevin Wilkinson. Jena property table implementation. In *SSWS*, 2006.

[70] K. Wu, E. Otoo, and A. Shoshani. Compressed bitmap indices for efficient query processing. Technical Report LBNL-47807, 2001.

[71] K. Wu, E. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In SSDBM'02, pages 99–108, 2002. LBNL-49627., 2002.

[72] K. Wu, E. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, 2001.

[73] A. Zandi, Balakrishna R. Iyer, and Glen G. Langdon Jr. Sort order preserving data compression for extended alphabets. In *Data Compression Conference*, pages 330–339, 1993.

[74] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[75] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[76] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, June 2005.

[77] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.