Compression and Query Execution within Column Oriented Databases

by

Miguel C. Ferreira

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

Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by . . . .
Samuel Madden
Assistant Professor, Theses Supervisor
Thesis Supervisor

Accepted by . . . .
Professor Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Compression and Query Execution within Column Oriented Databases

by

Miguel C. Ferreira

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering

Abstract

Compression is a known technique used by many database management systems("DBMS") to increase performance[4, 5, 14]. However, not much research has been done in how compression can be used within column oriented architectures. Storing data in column increases the similarity between adjacent records, thus increase the compressibility of the data. In addition, compression schemes not traditionally used in row-oriented DBMSs can be applied to column-oriented systems.

This thesis presents a column-oriented query executor designed to operate directly on compressed data. We show that operating directly on compressed data can improve query performance. Additionally, the choice of compression scheme depends on the expected query workload, suggesting that for ad-hoc queries we may wish to store a column redundantly under different coding schemes. Furthermore, the executor is designed to be extensible so that the addition of new compression schemes does not impact operator implementation. The executor is part of a larger database system, known as CStore [10].

Thesis Supervisor: Samuel Madden
Title: Assistant Professor, Theses Supervisor

Acknowledgments

The implementation of the executor was a large team project. Daniel Abadi and I worked together in defining the details of the CStore executor and its subsequent implementation. To Daniel a big thanks for the many long nights spend coding, writing and thinking[1] about CStore. Part of the same team was Edmond Lau, who designed the aggregation operators and was involved in many of our discussions. Involved in an earlier implementation of the executor was Amerson Lin.

I would also like to thank Professor Michael Stonebraker for the conception of CStore, and my advisor Professor Samuel Madden for his prompt reviews of our ideas and our discussions. Thanks Sam for all your help over the last couple of terms and all the last minute reviewing. It was a pleasure to work for you this last year.

CStore is a large, inter-university project, and I would like to thank the rest of the CStore team for their ideas and input. In particular, I would like to thank Alexander Rasin and Xuedong Chen for their help with implementation, as well as Betty O'Neil for her code reviews.

I wish the CStore team and the CStore project the very best.

[1]and dreaming too...

Contents

Chapter 1

Introduction

Compression is a technique used by many DBMSs to increase performance[4, 5, 14]. Compression improves performance by reducing the size of data on disk, decreasing seek times, increasing the data transfer rate and increasing buffer pool hit rate. The I/O benefits of compression can substantially outweigh the processor costs associated with decompression, particularly in the case of light weight compression schemes, such as run length encoding [5, 19, 2], which reduces data size by eliminating repeated adjacent values. In most database systems[19, 5, 15, 6] compression is simply an optimization on disk storage. Compressed pages are read from disk, decompressed in memory and is then presented to the query executor. Some systems [4, 5, 19] include lazy decompression where data is only decompressed if it is needed by the query executor. This improves performance as processor time is not unnecessarily wasted decompressing unused data. This thesis presents the CStore query executor, a query executor designed to always operate directly on compressed data. By operating directly on compressed data we are able to completely eliminate the decompression cost, lower the memory requirements on the system and exploit the structure of compressed data to improve the overall performance.

CStore [10] is a column-oriented database currently under development. Column oriented databases (”column stores”) differ from row oriented databases (”row stores”) in the layout of data on disk. In a column store each value of an attribute (column) is stored contiguously on disk; in a row store the values of each attribute in a tuple are stored contiguously. Column stores work more naturally with compression because compression schemes capture the correlation between values; therefore highly correlated data can be compressed more efficiently than uncorrelated data. The correlation between values of the same attribute is typically greater than the correlation between values of different attributes. Since a column is a sequence of values from a single attribute, it is usually more compressible than a row.

Operating directly on compressed data introduces a number of complexities to the query executor. In a naive approach, operating on compressed data would require every operator to have code to deal with each compression scheme's data structures. This increases implementation complexity and reduces extensibility, as the addition of a new compression scheme would require altering each operator's code. A second complexity introduced by compression is disk access. A particular compression scheme may alter the physical order in which records in a column are laid out on disk. This implies that reading a column in its original order will require random I/O. On the flip side, some compression schemes provide properties that can be exploited to improve performance. These properties alter the functioning of access methods and indexes and allow for new opportunities for optimizations within the executor.

Clearly, it is important to hide the details of compression schemes from the query executor. We achieve this by providing one common interface to manipulate compressed data. However, we do not want to hide all structure compression schemes have created on the data, as these may allow for new optimizations. A mechanism is necessary to expose to operators relevant properties of the compressed data. We propose a classification of compressed data, under which all compression schemes are classified by a set of properties. The classification of compressed data is exposed to operators, and based on this classification, operators can optimize their operations on compressed data.

In this thesis, we show how operating directly on compressed data can improve query performance. Additionally, we show that the choice of compression scheme depends on the expected query workload, suggesting that for ad-hoc queries we may wish to store a column redundantly under different coding schemes. We also present the architecture of an executor that can operator directly on compressed data, without greatly increasing operator complexity.

The next Chapter provides a high level overview of the architecture of the CStore query executor. Subsequently, we describe the system from the bottom up, describing the compression schemes employed in Chapter 3 and describing how data is stored on disk in Chapter 4. Chapter 5 describes the executor software architecture. Results are presented in Chapter 6 followed by a discussion of related work in Chapter 7, and future work in Chapter 8. Finally, we conclude in Chapter 9.

Chapter 2

Design Overview

In this chapter we present a high level introduction to the CStore query executor. CStore differs from a traditional DBMS in two key respects:

1. CStore is a a column store database system. Traditional DBMSs are row stores.

2. CStore uses compression throughout the executor. Traditional DBMSs do not usually expose compressed data to the executor.

These characteristics alter the architecture of the executor and the query plans for a particular query. In this section, we introduce the design of the CStore executor including the interfaces to operators and access methods. We introduce the design in two steps: firstly we describe how a column store architecture alters the traditional query executor and subsequently we extend the column store query executor to support compression.

2.1 Column Stores

A traditional database provides a set of access methods, as shown in Figure 2-1. Operators access data through a variety of access methods, such as sequential and index scans, supported by B-trees, R-trees or a variety of other indexing data structures. On execution, operators are instantiated in a query plan. Query plans in traditional database systems are trees where the root operator produces the query results.

As an example, consider the lineitem table:

lineitem `<orderID, suppkey, shipdate, etc...>`

where we have a primary index on shipdate. Now suppose we want to know the number of orders per supplier after the 1/1/05.

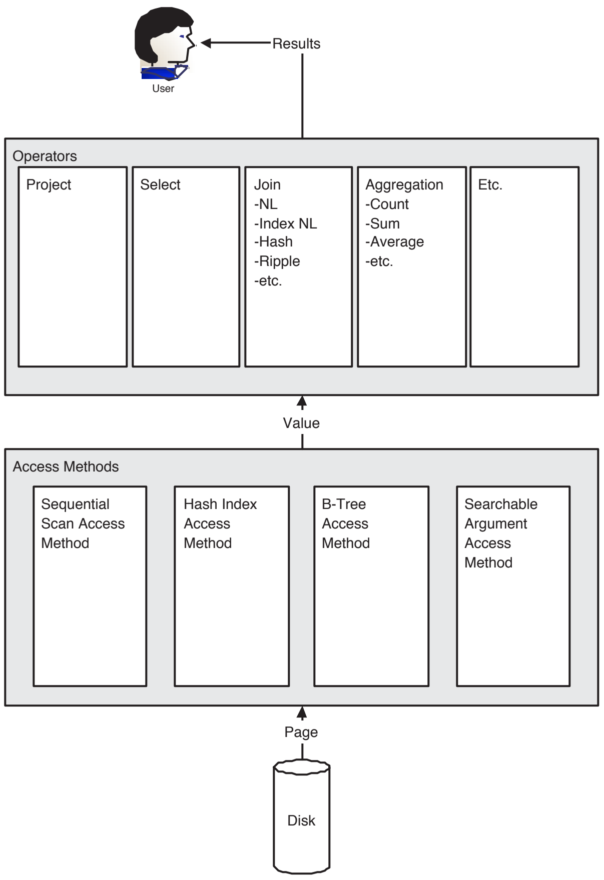


Figure 2-1: Architecture of a Row Store Query Executor

In SQL such a query could be expressed as:

```
SELECT count(orderID), suppKey
FROM lineitem
WHERE shipdate > 1/1/05
GROUP BY suppkey
```

After parsing and optimization, this query would likely correspond to the exection of the plan in Figure 2-2. In this query plan there is a single access method used, a searchable arguement ("SARG") access method that presumely uses the index on `shipdate` to avoid reading pages that would not match the predicate. Subsequently, we project out all columns not required for output, leaving just the `orderID` and `suppkey` as inputs for aggregation.

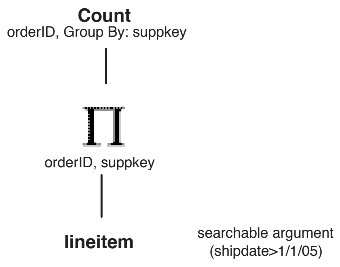


Figure 2-2: Query Plan for Traditional Row Store Executor

The row store query plan, shown in Figure 2-2 could not be used under the column store architecture. Under a column store, we store a columns independently, thus requiring us to access each column independently. We no longer have a single access method, but rather an access method per column. In a naive implementation, we could implement the query above with a plan as shown in Figure 2-3.

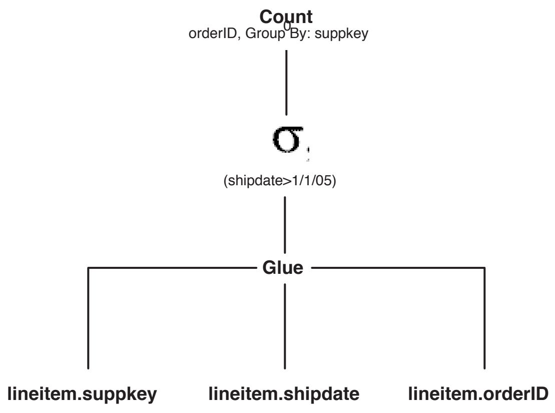


Figure 2-3: Naieve Query Plan for Column Store Executor

The glue operator essentially would perform the opposite of the project operator, gluing multiple columns together so tuples have more than one attribute. In this query, the glue operator converts the the column representation back to a row representation. However, it will execute slower than the row store plan as we perform full sequential scans on each column. We are not able to push the `shipdate > 1/1/05` selection predicate below the glue as we do not know what the corresponding values for the `orderID` and `suppkey` will be for a given `shipdate`.

To allow us to push predicates into the access methods, we introduce the notion of positions. Unlike in a row store, where the access method simply returns an entire row, in CStore the column access methods return the value of the column and the position at which that value occured. We also store position indexes on each column to allow us to quickly lookup the value at a particular position. In this way, we are now able to implement the a plan shown in Figure 2-4. We note that representing positions is a known technique in the literature, and can be represented as a bitmap or a list of positions [13]. Position bitmaps or lists are both compressible. A position list is an intermediate result in the executor that enumerates the positions in the table that are part of the result. To retrieve the values for the positions, we can apply these position lists as filters on the columns we wish to read. In the CStore executor this is known as position filtering.

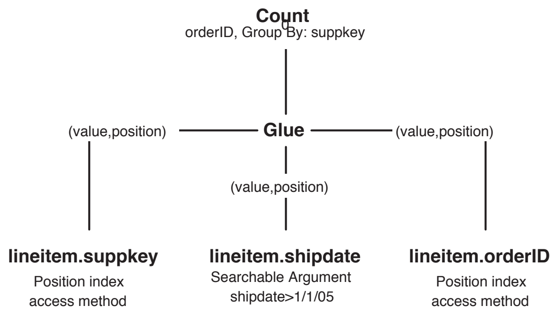


Figure 2-4: Better Query Plan for Column Store Executor

In Figure 2-4, we use indexes on position to filter the `orderID` and `suppkey` columns for the values, where `shipdate > 1/1/05`.

2.1.1 Benefits of Column Store

Our column oriented architecture has allowed us to read only attributes that are used in a query, rather than having to project out irrelevant attributes. But at what cost? In a row store we can read data in rows sequentially off disk. However, in a column store we can read columns sequentially but reading different attributes requires random I/O.

To get a feel for the relative performance of the system, let us analyze the relative performance of the example query above. Let us assume the fields in the employee table are all fixed width with sizes:

lineitem `<orderID[4], suppkey[4], shipdate[4], etc...[x]>`

Assume the table contains a total of 200,000 entries, disk pages are 64Kb and the predicate `shipdate > 1/1/05` has a selectivity of 7.5%. We recall that there is a primary index on shipdate, meaning the table is physically sorted in `shipdate` order and therefore each column is stored in `shipdate` order. This means that the suppkey value for the sixth smallest `shipdate` will be the sixth value in the suppkey column. Let us also assume random I/O is five times[1] slower than sequential reads. We denote the time for to read a page from disk sequentially as C. Given this setup, we have to read 15,000 entries.

In the row store case, each tuple read from disk has a size of $12 + x$ bytes. There are $\frac{65536}{12+x}$ tuples per page. Therefore to reply to this query we need to read $\frac{15000(12+x)}{65536} \approx \frac{1}{4}x$ pages. The total time is: $5C + 3C + \lceil \frac{1}{4}xC \rceil = 8C + \lceil \frac{1}{4}x \rceil C$

In the column store case we are reading three fields, each with a size of 4 bytes. However, we are required to perform a seek when reading each page. For each column, we can hold $\frac{65536}{4} = 16384$ values per page. Since we need to read 15000 values, and columns are stored in shipdate order, we need only to read the last page in the three columns[2] The total cost is:

$1(5)C + 1(5)C + 1(5)C = 15C$

The relative performance depends on the total size of each tuple. We note that if $x > 28$ the column store will out perform the row store. In this analysis we have made two key assumptions that favor the row store's performance performance. Firstly, we assumed that reading any page from the column store would require a disk seek. Secondly, we assumed that all I/O in the column store is sequential, thereby we only incur a disk seek at the start of the query. However, in practice these assumptions do not holds as:

- Other queries run concurrently. Therefore, there is a chance that a given row store query is interupted. This means that we cannot assume for the row store that all I/O is sequential.

- Modern disks have per track buffers. This is equivalent of reading more than one page at a time and holding it in memory. Although, this is advantageous to both the row store and column store architecture, the yield is higher in the case of the column store as all data read is used in the query (rather than some being projected out). If different columns are on different tracks, the benefit is even larger.

- Columns are more compressible, thereby giving us higher I/O bandwidth. With each page read, we retrieve more values.

[1]This is consist with our microbenchmarks, see Chapter 6

[2]May be two pages, in the case were the last page of values contains less than 15000 values

2.2 Compression

Through the example above, we have illustrated how column store systems have different query plans than row store systems and how column stores can improve performance over row stores. Compression provides a further performance advantage at the expense of added complexity in the executor. In a simple design, the introduction of compression might have no influence on the executor. If we simply decompress data in access methods and present data uncompressed to operators, then there is no alteration to the query plans shown in Figure 2-4. Decompressing data has several disadvantages:

1. Decompressing data increases the CPU costs of the query.

2. Decompressed data requires more memory than compressed data, reducing the amount of memory available for the buffer manager.

3. Decompression may reduce our ability to optimize operations. Compression schemes may introduce certain kinds of structure to data, which can be exploited by some operators. This implies that, for some operators, it is faster to operate on compressed data then uncompressed data.

The CStore query executor avoids decompressing data by delivering compressed data to operators. This feature significantly alters the design of the query executor when compared to traditional column stores. In a row oriented DBMS, an iterator interface delivers data to operators. Operators call `getNextTuple()`, pulling a tuple up from the preceding operator in the query tree. In a traditional column store, we deal with each attribute independently, thus the column oriented iterator interface transfers single values between operators through `getNextValue()` call. The basic unit of data delivered by the traditional column oriented iterator is a single value.

The per record iterator interface prohibits the delivery of compressed data to operators. Compression algorithms operate by exploiting similarity present in a sequences of values. These algorithms build a data structure that represents the same sequence in a smaller space. Extracting single values from these data structures is equivalent to decompression.

The solution is to alter the basic unit of data delivered by the iterator. In the CStore query executor, calls to the iterator deliver `Block`s of data. A `Block` is an interface to an arbitrary set of values, which are held compressed in memory. The iterator call is now `getNextValueBlock()`.

We do not place any restrictions on what values a `Block` can hold. Consider a column as a sequence of three values: A, B and C. Let us define the phrase *original order*, as the order in which values would be stored on disk in a uncompressed column store. Now consider the following two *original orders* of those values:

1. AAAAABBBBBCCCC

2. ABCABCBACABCBA

Now suppose, irrespective of the *original order*, the `Block` iterator delivers first a `Block` with all the occurences of A, followed `Block` with all the occurences of B, and a third `Block` with all occurence of C. If the column has the *original order*, then the `Block` iterator have delivered values to the executor in the *original order*. However, if the column had the second *original order*, then the compression scheme signficantly altered the order in which values are delivered. It may seem that the `Block` order does not matter, since SQL operators are order independent. However, the entire table is expected by operators to be stored in the same order. As we store columns independently, under different compression schemes this may not be the case. There are two solutions to the above problem:

- Always deliver `Block`s in the same order as the original sequence of values in the column.
- Do not impose any ordering requirement on `Block`s. Operators must provide support for dealing with out of order values.

Choosing the first option essentially a whole class of possible compression schemes[3]. For example, under delta coding on position, the original column is split into multiple sub-columns with lists of the positions where each value occured. These position lists are then coded and individually stored on disk. The compressed data structure is the compressed position list. To avoid decompression, each value's position list must be placed intact inside separte `Block`s. Requiring that the stream of `Block`s deliver the values of the column in the original order would require us to decompress the position lists for each value to rebuild the order of the column. Outlawing such compression schemes is suboptimal, as there is a large class of data where such schemes provide us with high compression ratios. Instead, operators in the CStore query executor support dealing with out of order `Block`s. We discuss this further in Chapter 5.

`Block`s provide a number of ways to access compressed data, and set of properties that describe the structure of the compressed data contained by the `Block`. Operators uses these properties to decide the optimal way to perform their function. For example, if a particular `Block` contains only one value (occurring at multiple positions) then the selection operator need only look at a single value contained by the `Block` to know whether or not the entire `Block` matches a selection predicate. `Block` properties are discussed more in depth in Section 3.5.

Compression algorithms also change disk storage. Different compression have different access methods, with diferent access costs. We introduce a compression specific layer that provides access methods for each compression scheme, known as the `Datasource` layer. A `Datasource` exists for every compression scheme present in the database, as can be seen in Figure 2-5. Within CStore, we access the disk using both sequential and index scans. However, the data that is indexed is compression specific, therefore, these scans

[3]Any scheme that codes on position

are performed by the `Datasource` layer. `Datasource`s then provide `Block`s to the CStore operators that execute the query. Finally, the results are written by the `BlockPrinter`.

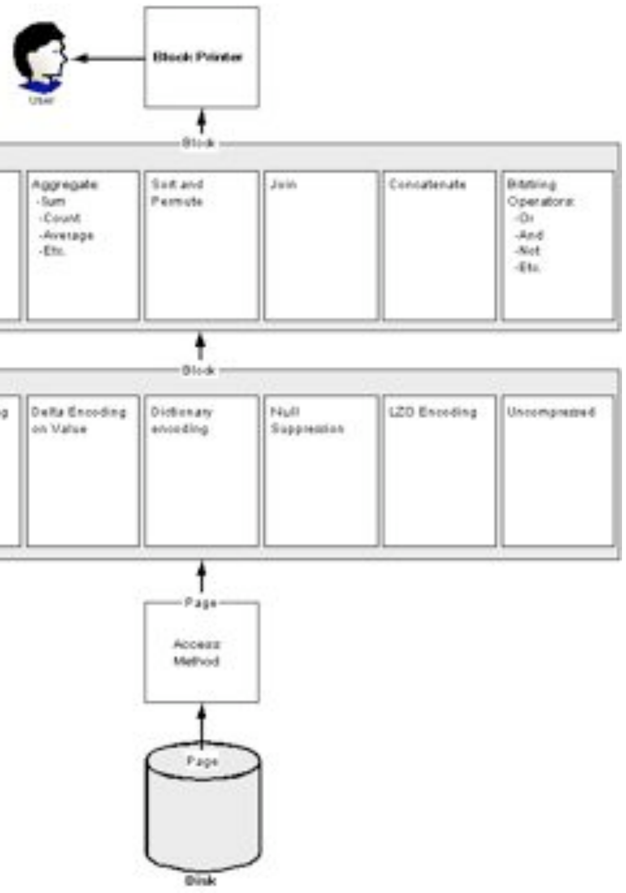


Figure 2-5: CStore Query Executor Architecture

In many cases the structures introduced by compression can actually be used to provide faster lookups than a traditional index. `Datasource`s provide compression scheme specific implementations of predicates (analagous to SARGable predicates) and capture the functionality of indexes through value and position filtering methods. In both cases a filter is represented as stream of `Block`s, `ValueBlock`s for value filtering and `PositionBlock`s for postion filtering. The source of these these filters is another operator in the query plan. This allows us to filter values on disk based on intemediate results in a query. `Datasource`s look at the values or positions in the filters and apply all possible optimizations to read those values/positions efficiently. Filtering is further described in Chapter 5.

To return to the previous example query introduced in Section 2.1, we can now consider how compression alters the execution of the query:

```
SELECT count(orderID), suppKey
```

```
FROM lineitem
WHERE shipdate > 1/1/05
GROUP BY suppkey
```

For the query above, we push a predicate into the `shipdate Datasource` as shown earlier in Figure 2-4. However, the the `shipdate Datasource` now produces `PositionBlock`s used to filter the `orderID` and `suppkey` columns, as shown in Figure 2-6.

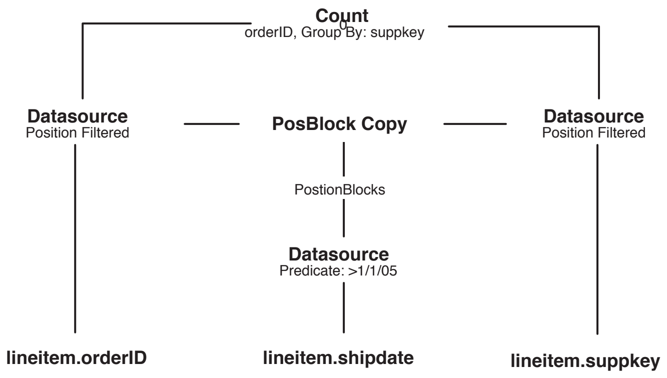


Figure 2-6: Sample Query Plan for CStore Query Executor

This example illustrates several points about query plans in CStore. First, we note that the query plan is not a tree, but a directed acyclic graph. We also note that `Block`s may be used by more than one operator. This is the case in Figure 2-6, where both the `suppkey` and `orderid Datasource`s use `PositionBlock`s from the `shipdate Datasource` to filter their output. Finally, we note that the two different types of `Block` streams, `ValueBlock` and `PositionBlock` streams, are used depending on the input requirements of operators.

2.3 Compression in a Column Store

In summary, the CStore executor is a column store executor that operates on compressed data. Being a column oriented database forces us to use different access methods for each attribute in the query. To avoid always having to perform a full sequential scan over the attributes used by a query, we must track the position of each value throughout executor so to use position indexes over the data. Compression alters the layout of data on disk, thus compression specific access methods, named `Datasource`s, are introduced to provide a standard interface for indexing. Compression also alters the way in which data is delivered to operators. The basic unit of data within the CStore executor, the `Block`, is a set of compressed values. There are no requirements on what values a `Block` can contain. `Block`s also provide a set of properties that expose to operators the structure of the compressed data in order to allow operators to optimize their operations. The query executor is designed to be extensible, such that any compression schemes can be supported simply by writing `Datasource` and `Block` code specific for that compression scheme.

We note that the discussion up to this point has not discussed any of the specific coding schemes used by CStore. In the next Chapter we describe the schemes currently employed by CStore.

Chapter 3

Compression Schemes

A wide variety of coding schemes exist for many purposes. In this section, we introduce the set of schemes that have been implemented in CStore. In general, they can be divided into three categories:

1. Storage optimizations: Compression on a single value.

2. Lightweight schemes: Compression on a sequence of values.

3. Heavyweight schemes: Compression on an array of bytes.

Storage Optimizations are schemes that do not code data and are smart about how they store values on disk. Lightweight schemes code the data by noting some relationship between successive values. Schemes such as run length encoding (RLE) and delta encoding are examples of light weight schemes. Heavy weight schemes code based on patterns found in the data, ignoring the boundaries between values, and treating the data input as an array of bytes. In the subsequent sections, the particular schemes under each of these categories are detailed.

3.1 Storage Optimizations

Null suppression is the only scheme currently implemented in CStore that falls under this category. Null suppression works by suppressing leading zeros in the data. For example, when storing an integer with the value 123, we need not store 4 bytes on disk, 1 byte will suffice. However, there is a caveat: each value on disk is no longer of fixed length. A coded column is a sequence of variable length fields. There are many proposed solutions to this problem[15, 19]. We solve this by coding a table that indicates the length of each field, much in the same way as stated in [15]. We write a byte sized table to code the length of the subsequent four values, for every four values. The byte is divided into sets of two bits, with the first two

numValues[32]	$sizeTable_1$[8]	val_1	val_2	val_3	val_4	$sizeTable_2$[8]	val_5	...

Figure 3-1: Null Suppression (size in bits)

bits encoding the length of the first value, the second two bits encoding the length of the second value, and so on as shown in Figure 3-1. The header in a null suppression coded sequence contains just an integer to represent the total number of values in the sequence.

Null suppression is most used when we have a random sequence of small integers. Null suppression introduces a byte of overhead for every 4 values written, therefore the total number of bytes used coding a sequence with N values, where the i^{th} value can be coded in n_i bytes is:

$$\text{Total Size} = \sum_{i=1}^{N} n_i + \lceil \frac{N}{4} \rceil \tag{3.1}$$

From this analysis we conclude that we should use null suppression if the average value in a column requires less than $3.5 * 8 = 28$ bits to code.

3.2 Light Weight schemes

Light weight compression algorithms work on the basis of some relationship between values, such as when a particular values occurs often, or if we encounter long runs of repeated values. To exploit this we use three main techniques: run length encoding, delta encoding and dictionary encoding.

3.2.1 Run Length Encoding

Run length encoding (RLE) is useful for data with large runs of repeated values, which typically occurs on sorted columns with a few number of values. Consider the following sequence of values:

- {1,1,1,2,2,3,3,3,3,3,3,3,3,3,3,…}

By counting the number of repetitions, we can code such a sequence as {value, numRepetitions} pairs. The sequence above could be represented as:

- {{1,3},{2,2},{3,10}}

The data structure {`value`, `numRepetitions`} is known as a `RLEDouble`. The header for this encoding, shown in Figure 3-2, indicates the number of `RLEDouble`s in the sequence, the starting position of the first `RLEDouble` within the sequence, and size in bits of the value and `numRepititions` fields.

In the context of a column-store database, it may be useful to code some redundant data. Consider we finding the value at the 12^{th} position in this sequence. To find this position, we must start at the first position and sequentially add `numRepetitions` of the `RLEDouble`s until we find the first `RLEDouble` where this running sum is greater that or equal to 12, and then read the value from the `RLEDouble`. Finding a position on this data structure requires an $O(n)$ operation, where n is the number of `RLEDoubles`. By storing the starting position in each `RLEDouble` we can improve performance. The sequence above would now be coded as:

{{1,1,3},{2,4,2},{3,6,10}}

The data structure {`value`, `startPosition`, `numRepetitions`} is known as an `RLETriple`. `RLETriple`s allow us to binary search to locate a particular position. This improves the search bounds to $O(\log_2(n))$.

When encoding RLE we must decide on the size in bits of each field within the `RLETriple` This information is maintained in a header that precedes all RLE encoded data, shown in Figure 3-2. Under CStore both RLE schemes are employed. We denote the scheme using `RLETriple`s as RLE and the scheme storing `RLEDouble`s as RLEII. RLE is the compression algorithm of choice for sequences with large runs of repeated values.

RLE	sizeValue[8]	sizeStartPos[8]	sizeReps[8]	sizePage[32]	$RLETriple_1$	...
RLEII	sizePage[32]	sizeValue[8]	sizeReps[8]	startPos[32]	$RLEDouble_1$	...

Figure 3-2: RLE and its variants

3.2.2 Delta Coding

Delta encoding involves storing the difference two adjacent values. Delta coding is useful when we have a sequence of values where the difference between successive values can be coded in less bits than the values themselves. Consider the sequence of values:

- {1200, 1400, 1700, 8000}

The differences between the values are:

- {200, 300, 6300}

To rebuild the sequence, however, we need to know the initial value. Therefore, we code the sequence as the initial value followed by the set of differences between adjacent values:

- {1200,{200,300,6300}}.

In delta encoding the difference list is known as the delta list. On disk, the delta encoded data structure is laid out as shown in Figure 3-3.

value[32]	startPos[32]	numDeltas[32]	*DeltaList ...*

Figure 3-3: Delta Coding

There is a limit for the largest delta as we have chosen fixed sized field. In a sequence, there could be a small number of large deltas, so we want to be able to accomodate these jumps. If a particular delta is larger than ($delta \geq 2^{sizeOfDelta} - 1$), the value $2^{sizeOfDeltaInBits} - 1$ is written and the delta is encoded in the next 32 bits. In this way we can accomodate occasional large deltas.

Two flavors of delta encoding are used with in CStore, delta encoding on value and delta encoding on position. The coding scheme is unaltered, the difference is just which sequence of values we are coding.

Delta Encoding on Value

Delta encoding on value is the most natural of the two variants. The sequence coded is the sequence of values in the column. If a column contains the sequence:

{1200, 1400, 1700, 8000}

starting at position 1000, we would code the sequence to;

{1200,1000, 9, 3,{200,300,6300}}.

We note that in this example, deltas 200 and 300 fit in a delta field 9 bits wide. The 6300 however would not, and would require 9+32 bits to encode. Delta coding on value is useful if we have a column with many values, where the difference between values can be coded in less bits than the values themselves. This is typically the case in sorted columns.

Delta Encoding on Position

Under Delta encodind on position, we code the list of postions where each value in the column occurs. Delta encoding on position is effective when the list of positions for each value is large, or equivalently when values are repeated multiple times in a column. Consider the sequence:

{1,2,3,4,5,1,2,3,4,5,5,4,3,2,1,1,2,3,4,5}

Now lets consider the positions where each value occurs:

- The value 1 occurs at positions {1, 6, 15, 16}
- The value 2 occurs at positions {2, 7, 14, 17}
- The value 3 occurs at positions {3, 8, 13, 18}
- The value 4 occurs at positions {4, 9, 12, 19}

- The value 5 occurs at positions {5, 10, 11, 20}

Delta encoding on position, codes each of these position lists using the delta encoding scheme described earlier. Therefore, for the previous example we create 5 different delta encoding data structure :

- {startValue=1, `startPosition`=1, sizeOfDeltaInBits=4, numElementsInDeltaList=3,{5,9,1}}
- {startValue=2, `startPosition`=2, sizeOfDeltaInBits=4, numElementsInDeltaList=3,{5,8,3}}
- {startValue=3, `startPosition`=3, sizeOfDeltaInBits=4, numElementsInDeltaList=3,{5,7,5}}
- {startValue=4, `startPosition`=4, sizeOfDeltaInBits=4, numElementsInDeltaList=3,{5,6,7}}
- {startValue=5, `startPosition`=5, sizeOfDeltaInBits=4, numElementsInDeltaList=3,{5,5,9}}

Each of these data structures is stored separately on disk, under different names. In addition, an uncompressed list is stored with the values that exist in the original sequence. This list takes the form {numberOfValues, {list}}. The format of each value sub-column and the value table is shown in Figure 3-4.

DeltaPos	value[32]	startPos[32]	numDeltas[32]	*DeltaList* ...
DeltaPosValueTable	numValues[32]	$value_1$[32]	$value_2$[32]	...

Figure 3-4: DeltaPos Data Structures.

Delta encoding on position is effective when we have a sequence with a small number of values occuring in random positions in the sequence.

3.2.3 Simple Dictionary Coding

A wide set of dictionary compression schemes in literature[4, 15, 2]. CStore employs two dictionary schemes, the first a simple dictionary scheme detailed in this section and the second a Lempel-Ziv based scheme described in the next section. Simple dictionary encoding is one of the most widely used compression schemes in traditional DBMSs[4, 15, 2]. The crux of dictionary encoding is to replace large bit patterns with smaller tokens. In the simple scheme employed by CStore we replace large values in a sequence with fixed sized tokens. The algorithm for dictionary coding in CStore is shown in Figure 3-5. The algorithm takes one pass a coding the column. For each value, it checks looks up the value in a hash map to retrieve its corresponding token. If the value is not in the hash map, it increments a token counter, and adds the value and the token to the hash map. For each value read, it writed the value's corresponding token.

To decode, however, we need to know the mapping between tokens and values. After compression, we write the uncompressed keys in the `valueTokenMap` sequentially storing them as a column. This is known as the table column, and is show in Figure 3-6. The position of values within the table column provide the mapping to the decompressed value. The first value in the table column corresponds to the 0 token, the

```
1. fieldSize=Pick($fieldSize \geq log_2(E[numberOfValuesInSequence])$)
2. count=0
3. hashmap valueTokenMap
4. buffer=new char[PAGESIZE]
5. count=0
6. while (sequence not empty)
    (a) currentValue=getNextValueFromSequence()
    (b) if (valueTokenMap.containsKey(currentValue))
         i. write(buffer, valueTokenMap[value])
    (c) else
         i. valueTokenMap.put(value, count)
        ii. write(buffer, count)
       iii. count++
7. return buffer
```

Figure 3-5: Simple Dictionary Encoding Scheme

Dictionary	numValues[32]	sizeToken[8]	$token_1$[sizeToken]	$token_2$[sizeToken]	...
DictionaryTable	numValues[32]	$value_{token=0}$[32]	$value_{token=1}$[32]	$value_{token=1}$[32]	...

Figure 3-6: Dictionary Coding Data Structures.

second value corresponds to the 1 token and so on. To decompress, we read the values column and build a token to value hash map. Given this hash map, decoding is simple. To decode, we lookup each token in the hash map to obtain its original value.

This algorithm has a useful property: if the original sequence was value sorted, the compressed sequence is too. This allows us to perform range queries on the compressed sequence.

3.3 Heavy Weight Schemes

3.3.1 LZO

LZO (Lempel Ziv Oberhummer[11]) is a modification of the orginal Lempel Ziv ("LZ77") dictionary coding algorithm[12]. LZ77 works by replacing byte patterns with tokens. Each time the algorithm recognizes a new pattern, it outputs the pattern and then it adds it to a dictionary. The next time it encounters that pattern, it outputs this token from the table. The first 256 tokens are assigned to possible values of a single byte. Subsequent tokens are assigned to larger patterns.

Details on the particular algorithm modifications added by LZO are undocumented, although LZO is a GPL compression library, the code is highly optimized and hard to decipher. LZO is heavily optimized for decompression speed, with the author claiming an average data transfer rate of a third of the memcpy function. It provides the following features:

- Decompression is simple and very fast.
- Requires no memory for decompression.
- Compression is fast.
- The algorithm is thread safe.
- The algorithm is lossless.

LZO's ability to provide fast decompression makes it a candidate for CStore. The CStore executor does not, however, operate directly on LZO data, instead LZO compressed data is decompressed before it is presented to executor.

3.4 Picking a compression scheme

We have described six compression schemes currently implemented in CStore. The compression ratios that each scheme is able to acheive depends greatly on the properties of the sequence of values we are encoding. Figure 3-7 provides a rough guide of the effectiveness of different compression schemes on different types of data.

Properties on Values	Few Values	Many Values
Runs of repeated values	RLE	RLE (runs must exist)
Sorted	RLE (runs must exist)	DeltaOnValue, LZO, Int
Unsorted	Dictionary, DeltaOnPosition	LZO/Int
Large Values	Dictionary	LZO, Int
Small Values	Dictionary, Null Suppression	Null Suppression

Figure 3-7: What Compression Scheme to Pick

We have described a rough selection criteria for compression schemes. However, it is important to note that the choice of compression scheme also depends on the queries that use this data. The performance of a given operator is largely dependent on the structure of the coded data it is operating on. Therefore, to maximize performance we may wish to use a less compact scheme to improve a query's performance. For a varied query load, we may have a particular column stored redundantly under many compression schemes so to increase overall query performance.

3.5 Properties of Column Compression

Compression schemes may alter the ordering of a sequence of values. We introduce a set of properties that allow us to describe the structure of compressed data relative to the original sequence. To begin this discussion, there are two general classes of questions we can ask about a sequence of values:

1. What is the fourth value in the sequence?

2. Where does the value 3 occur?

The first question has a single answer, for every position there is only one value. The second question may not have a unique answer since the value 3 can occur at multiple positions. Therefore, to describe a single instance of a value we have to specify both the value and position. We denote the set {`value, position`} as a `Pair`.

When we consider the compression schemes described in this chapter, not all schemes preserve the original order of the sequence on disk. For example. if we consider delta encoding on position, we code a delta list for each value, and store each of these list contiguously on disk. If a particular value did not occur in runs, but was interspersed in other values then original order of the column is not preserved on disk.

It may seem that storage order does not matter, since SQL operators are order independent. However, the entire table is expected by operators to be stored in the same order. With different compression schemes for different columns, and the delivery of data as it its physical order, operators no longer receive two columns in the same order. We build support in the query executor for out order values and deliver every values to the executor as a stream of pairs. The entire stream of `Pair`s represents the original data. `Pair`s are delivered in the query executor in `Block`s, where a `Block` contains an arbitrary number of `Pair`s. Abstractly, we can think of `Block`s as non-overlapping subsets of the total set of `Pair`s in a column. Let's formalize our discussion of columns by introducing some notation. Let:

- P[i] be the i^{th} `Pair` in a stream of `Pair`s.
- $C = \bigcup_{i=1}^{N} P[i]$ be the set of all `Pair`s in a column.
- $Value(P[i])$ =value part of i^{th} pair
- $Position(P[i])$ =position part of i^{th} pair
- S_i are subsets of C, $C = \bigcup_{i=1}^{m} S_i = C$ where
  - $S_a = \bigcup_{i=j}^{k} P[i]$ where $j \leq k$ & $1 \leq j, k \leq N$
  - $S_a \bigcap S_b = 0$ *iff* $a \neq b$ & $1 \leq a, b \leq m$

We can now consider some properties over the `Pair` stream, and `Block`s of the `Pair`s in that stream. In particular:

1. Are the values sorted? (i.e. Is $Value(P[i]) \leq Value(P[j])\ \forall i < j$?)

2. Are the positions sorted? (i.e. Is $Position(P[i]) \leq Position(P[j])\ \forall i < j$?)

3. Does this Block only hold one value? (i.e. Is $Value(P[i]) = Value(P[j]) = k \ \forall i, j : P[i] \& P[j] \in S_i$?)

4. Does this Block hold contiguous positions? (i.e. Is $Position(P[i]) + 1 = Position(P[i+1]) \ \forall i : P[i] \& P[i+1] \in S_i$?)

5. Is this Block value sorted? (i.e. Is $Value(P[i]) \leq Value(P[j]) \ \forall i < j : i \& j \in S_i$?)

6. Is this Block position sorted? (i.e. Is $Position(P[i]) < Position(P[j]) \ \forall i < j : i \& j \in S_i$?)

With 6 properties, there is a set of $2^6 = 64$ possible combinations of properties that any Block could have. However, we can limit this number by noting the following relations between these properties:

- $1 \rightarrow 5$: If the stream is value sorted, then each subset must be value sorted

- $2 \rightarrow 4$: If the stream is position sorted then all positions in stream are adjacent to each other, therefore each subset is position contiguous.

- $2 \rightarrow 6$: If the stream is position sorted, then each subset must also be position sorted.

- $3 \rightarrow 5$: If the subset holds only one value, then the subset is trivially value sorted

- $4 \rightarrow 6$: If subset is position contiguous, it must be position sorted

This gives us a total of 18 possible combinations. This result is important due to its generality. If we consider any coding scheme we can always classify it according to one of these 18 general classes formed by these properties.

This result has a large impact on the complexity of the query executor. It implies that if we abstract away compression schemes but provide operators with a stream of Blocks, then their are at most 18 cases they have to deal with. In practice, operators deal with fewer than 18 cases for two main reasons. Firstly, only a few of these properties can be used fruitfully for optimization. For example, code that deals with a multi-valued Block is able to deal with single value subsets, as a single value subset is a special case of the more general multi-value subset. Secondly, for some operators, many of these properties are irrelevant. A select operator is only concerned with the values in a Block suggesting that postion oriented properties (i.e. 2,4 and 6) are irrelevant.

We must operate on Blocks instead of the entire column for several reasons: for one, we can only deal with subsets of the column at any point in time, as in general the entire column will not fit in memory. Additionally, most compression schemes create data structures that represent a subset of the data. For example, an RLETriple represents a run of a particular value in the original column. The compressed data structures themselves represent subsets of the entire column, and are therefore well suited to the Block model. Lastly, data on disk is stored in pages, and for reasons described in chapter 4, we code subsets of a column until we have a page worth of data. When we decompress a page we have a subset of the column to operate on. In the next chapter, we look at how compressed sequences are actually stored on disk within the CStore system.

Chapter 4

Storage model for CStore

For the current CStore system, we use Sleepycat's BerkeleyDB for storage management. BerkeleyDB ("BDB") "is an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications."[17] CStore currently uses BDB as a storage manager. We store 64K pages as BLOBs (Binary large objects) within BDB. Pages contained coded data, created by running a coding scheme until we have a page worth of data to write. Each page contains sufficient header information to decompress all its contents.

Within BerkeleyDB we store different types of B-Tree indexes on compressed pages depending on the properties of the compressed data.

1. If pages are sorted by value, we store two indexes: a primary index on value and a secondary index on position.
2. If pages are not sorted by value, we store a single index: a primary index on position.

It is important to note that indices are on pages, but pages have multiple values and positions. We pick the last value and position in each page as the key for the index. To lookup a particular value, we perform a lookup on the primary index. We find the first leaf in the index where the key is greater than or equal to the value we are searching for. The page pointed to by this index entry is guaranteed to be the only page where that value could be located [1].

Some of the schemes described in Chapter 3 require storing more than one column on disk. In particular:

Delta Coding on Position codes a delta list for each value. We store these as separate objects in BDB, each delta list with a primary index on position. A second type of column is also stored that provides a

[1] We can proof this by contradiction: if the value we are searching is in fact contained in another page, then the index entry for that page must be greater than or equal to that value. However, we traverse the index in order, so there can only be one page where the value is contained.

list of all the values for which delta lists exist. This value column is sorted with both value and position indexes. The value column operates as a catalog for the delta lists available, in that it allows us to discover what delta lists are available. To maintain this system within BDB, a strict naming convention is followed. The value column is named <column name>VALS and each value sub-column is named <column name><value>. This layout of data allows us to read down each delta list independently and have an independent position index for each value.

Dictionary Encoding requires storing a value table to allow us to map tokens back to the original value. The table is an uncompressed column named <column name>Table. The table column and coded columns are both sorted if the original data was sorted. The table and coded column therefore have both value and position indexes if the original data was sorted. If the original data was unsorted, both the table and compressed columns are stored with only position indexes.

Data stored and indexed by BerkeleyDB is compression specific. Within the CStore architecture, `Datasource`s are responsible for opening the correct BerkeleyDB objects to provide the executor with `Block`s to operate. In the next chapter, we begin a more detailed discussion of how `Datasource`s, `Block`s and other components operate within the executor.

Chapter 5

Software Architecture

Thus far, we have described at a high level the overall architecture of the CStore query executor. In this chapter, we study the details as to the implementation of the CStore executor. To start the discussion, let us consider the sample query introduced in Chapter 2. In this query, we want to find the number of orders on each day since the beginning of the year. We can express such a query in SQL:

```
SELECT count(orderID), shipdate
FROM lineitem
WHERE shipdate > 1/1/05
GROUP BY shipdate
```

Figure 2-2 illustrates a query plan that might be instantiated within a traditional DBMS and Figure 2-6 illustrates the query plan instantiated within the CStore executor.

We recall that in a traditional DBMS the query plan involves a single access method, a projection and finally a `COUNT` aggregation operator. The CStore query plan differs as:

1. There is an access method for every column.

2. Unlike the traditional plan, the CStore query plan is not trees.

3. `Datasource`s are used to filter positions. The `Datasource`s aggressively tries to limit the number of disk accesses required, by taking advantage of its knowledge of the structure of the compressed column on disk and the properties of the filter `Block`s.

In the next sections, we describe in more detail how we can handle the cases introduced by each of these points. We begin our discussion by describing `Block`s, which are the central component exchanged between all components in the design.

5.1 Blocks

Recall that a `Block` carries compressed data and provides a common interface to access compressed data. A `Block` contains a subset of the compressed column and exposes to operators the set of six properties indicated in section 3.5. These properties indicate to operators the structure of the `Block` as well as the `Block` stream in which it is contained. The interface to a `Block` is shown in Figure 5-1.

`Block* clone(Block&)`	Copy the block.
`bool hasNext()`	Does the iterator contain any remaining `Pairs`.
`bool hasNext(int value_)`	Does the iterator contain any `Pairs` with this value.
`Pair* getNext()`	Get the next `Pair` in the iterator.
`Pair* peekNext()`	Get the next `Pair` in iterator without advancing it.
`Pair* getPairAtLoc(uint loc_)`	Get the `Pair` at this location.
`int getCurrLoc()`	Return the current location of the iterator.
`int getSize()`	Return the number of `Pairs` in this `Block`.
`int getSizeInBits()`	Return the size of the `Block` in bits.
`Pair* getStartPair()`	Get the first `Pair` in the `Block`.
`void resetBlock()`	Reset the iterator to point to the first `Pair`.
`bool isValueSorted()`	Returns true if the `Block` stream is value sorted.
`bool isPosSorted()`	Returns true if the `Block` stream is position sorted.
`bool isOneValue()`	Returns true if the `Block` has a single value.
`bool isPosContiguous()`	Returns true if the `Block` is position contiguous.
`bool isBlockValueSorted()`	Returns true if the `Block` is value sorted.
`bool isBlockPosSorted()`	Returns true if the `Block` stream is position sorted.

Figure 5-1: `ValueBlock` interface

`Block`s provide the ability to decompress on the fly, providing an iterator style interface to the `Pairs` it contains. Our goal, however, is for most operators to use knowledge of a `Block`'s structure to avoid decompression. We can illustrate this most clearly through an example: consider a `Block` stream that is value and position sorted, where each `Block` in the stream contains one value and is position contiguous. Suppose we have a second `Block` stream which is also value and position sorted, and where `Block`s are not single-valued. Now consider a selection filtering based on whether the value is greater than some constant k.

With a single-valued `Block`, all the select operator needs to do is look at the first value and if the value satisfies the predicate it can return that entire `Block`. If the value does not satisfy the predicate, it simply goes on to process the next `Block`. With multi-valued `Block`s, the select would have to test the predicate on each value, effectively requiring the `Block` to decompress each value. It may be that the single value property is irrelevant to some operators. For example, a count aggregation operator aggregating on a `Block` stream never needs to access to values, hence whether the `Block`s are one-valued or not is irrelevant.

`Block`s provide access to uncompressed data through an iterator interface, as shown in Figure 5-1 for `ValueBlocks`. `ValueBlocks` provide a `getNextPair()` call, while `PositionBlocks` provide a `getNextPosition()` call. The iterator provides access to a stream of `Pairs` or `Position`s depending on the type of `Block`. In addition, random access is provided by a call to `getPairAtLoc(int i)` which allows us to jump to i^{th} `Pair` or `Position` in the stream.

A particular `Block` may not be unique in a `Block` stream between operators. For example, there could be multiple instances of a set of values and positions after a join operator. We capture this information in a `numOccurences` field within each `Block`, to avoid copying `Blocks`.

5.1.1 Value Blocks

`ValueBlocks` represent the column value data, as opposed to `PositionBlocks` that just contain the set of positions in the column. In the terminology from chapter 3.5, `Blocks` hold a subset of a coded sequence. A `Block` holds the smallest possible compressed data structure. These data structures are by their nature coding scheme specific, thus an `RLEBlock` is different from a `DeltaOnPositionBlock` ("`DeltaPosBlock`" for brevity). In the next subsections, we look at the current types of `ValueBlocks` and their properties.

5.1.2 BasicBlocks

`BasicBlocks`, as their name implies, are the simplest type of `Blocks` in the CStore query executor. They contain a single pair, have a size of 1 and take 8 bytes in memory[1]. `BasicBlocks` are used for uncompressed data or data that is decompressed before it is exposed to the executor. `BasicBlocks` have properties as shown in Figure 5-2.

Property	**Value**
`Block` Stream value sorted	True if original column is value sorted
`Block` Stream position sorted:	True (as original column is always in position sorted sequence)
`Block` one valued:	True
`Block` position contiguous:	True (trivially)
`Block` value sorted:	True (trivially)
`Block` position sorted:	True (trivially)
`Block` size:	1
`Block` size in bytes:	8

Figure 5-2: `BasicBlock` Properties

5.1.3 RLEBlocks

`RLEBlocks` contain a single `RLETriple`. RLEBlocks are decompressed through calls to `getNext()`. The first call to `getNext()` returns the `Pair` {`value,startPos`}, the second call returns {`value, startPos+1`}, the i^{th} call returns {`value, startPos+i-1`} all the way up to {`value,startPos+reps`}. Random access in `RLEBlocks` is fast, as it only requires us to check if the position requested is contained in the `RLEBlock`,

[1]the size of a `Pair`: 4 bytes for the value, 4 bytes for position

which requires at most two comparisons. We note that it is computationally simple to decode `RLEBlock`s, however the overhead of the function calls adds up. The properties of `RLEBlock`s, are shown in Figure 5-3.

Property	**Value**
`Block` Stream value sorted	True if original column is value sorted
`Block` Stream position sorted:	True
`Block` one valued:	True
`Block` position contiguous:	True
`Block` value sorted:	True
`Block` position sorted:	True
`Block` size:	Arbitrary, `numRepetitions` in `RLETriple`
`Block` size in bytes:	12

Figure 5-3: `RLEBlock` Properties

5.1.4 DeltaPosBlocks

`DeltaPosBlock`s differ greatly from `BasicBlock`s and `RLEBlock`s. We stated earlier that a `Block` contains the smallest possible compressed data structure. In the case of delta encoding on position, the smallest compressed data structure is a disk page of delta on position data[2]. As a reminder, a page of `DeltaPos` data contains a header with the number of deltas found on the page, the initial position and the value this page encodes followed by a delta list that fills the page. `DeltaPosBlock`s' properties are shown in Figure 5-4.

Property	**Value**
`Block` Stream value sorted	True, regardless of whether initial data was sorted.
`Block` Stream position sorted:	False
`Block` one valued:	True
`Block` position contiguous:	False
`Block` value sorted:	True
`Block` position sorted:	True
`Block` size:	Arbitrary, `numDeltasInDeltaList` + 1 (for the `startPos`).
`Block` size in bytes:	One Page

Figure 5-4: `DeltaPosBlock` Properties

`DeltaPosBlock`s are decoded on the fly by walking down the delta list. The i^{th} pair returned by `getNext()` is {`value, position[i]`} where `position[i]` = `position[i-1]` + `delta[i-1]` and `position[1]` = `startPos`. Random access in `DeltaPosBlock`s is slow as we have to walk down the delta list adding and subtracting depending on the direction of traversal.

`DeltaPosBlock`s have the appealing feature that they order the original column. This allows a number of optimizations to be performed on delta on position coded columns as operations can assume the output

[2]As we mentioned previously, coding is always done at a disk page granularity.

sequence is sorted. However, the cost for this feature is that DeltaPosBlocks are no longer position contiguous. An operator in a query is not guaranteed where in the stream of DeltaPosBlocks a particular position of interest will appear. A second implication is that operators also do not know which positions were filtered out by previous operators. The trade offs introduced by DeltaPosBlocks must be considered by the optimizer.

5.1.5 DeltaValBlocks

DeltaOnValueBlocks ("DeltaValBlocks") are similar to DeltaOnPosition Blocks in that a full disk page is maintained inside the Block. DeltaValBlocks' properties are shown in Figure 5-5.

Property	**Value**
Block Stream value sorted	True, if original column is value sorted.
Block Stream position sorted:	True
Block one valued:	False
Block position contiguous:	True
Block value sorted:	True, if original column is value sorted.
Block position sorted:	True
Block size:	Arbitrary, numDeltasInDeltaList + 1 (for the startValue).
Block size in bytes:	One Page

Figure 5-5: DeltaValBlock Properties

DeltaValBlocks are decoded in a similar way to DeltaPosBlocks. The i^{th} call to getNext() returns {value[i], startPos+i-1} where value[i]=value[i-1]+delta[i-1] and value[1]=startValue defined in the delta on value page header. Random access in DeltaValBlocks is slow, as we have to walk down the delta list much like the case in the DeltaPosBlock.

5.1.6 Position Blocks

A position Block is a vector of positions. Intuitively, it is half of a ValueBlock, in that if a ValueBlock is stripped of its values it is transformed into a PositionBlock. If this is the case, then why have position Blocks? We are essentially destroying information, and recovering values will require disk I/O. Why not just pass ValueBlocks around? There are two major reasons for the existence of PositionBlocks:

Compressibility: By removing values, we end up with a sequence of positions. These sequences typically have long runs. Consider the case of a RLE encoded sorted column. If we were to apply a selection on this column, with a low selectivity predicate of the form value>constant, we expect many values to satisfy this predicate. However, as the column is sorted, we expect only one range of positions to satisfy that query. If we RLE encode the positions we have only one RLETriple representing the range of positions that satisfied the predicate, rather than one `RLETriple` for each value that satisfied the predicate. This analysis extends to other compression schemes.

Operators: For some operators, it is more natural to think in terms of position vectors, or to produce position vectors as their output. Logical operators, like `BAnd` and `BOr` take a pair of position vectors and perform some logical operation to produce another position vector.

In terms of implementation, there is little difference between a `PositionBlock` and a `ValueBlock`. `PositionBlock`s, however, expose fewer properties. A `PositionBlock` has three properties:

1. Is the position stream in sorted order?
2. Are positions in the `PositionBlock` contiguous?
3. Are positions in the `PositionBlock` in sorted order?

These properties are related in the same way as in the case for value blocks. The following relations exist:

- $1 \rightarrow 2$: If the stream is sorted, then each `PositionBlock` must contain contiguous positions.
- $1 \rightarrow 3$: If the stream is sorted, then each `PositionBlock` must be sorted.
- $2 \rightarrow 3$: If the `PositionBlock` is position contiguous, then the `PositionBlock` must be sorted.

These gives us a total of four possible classes of `PositionBlock`s.

1. The position stream is in sorted order.
2. The position stream is not in sorted order, but the `PositionBlock` are contiguous.
3. The positions in `PositionBlock`s are in sorted order.
4. None of the properties is true.

CStore currently implements `PositionBlock`s for each of these types:

`PosBasicBlocks`: Like their `ValueBlock` counterparts, these blocks contain a single position. They are used in the case that all properties are false.

`PosBitBlocks`: Essentially, a multi-position `PosBasicBlock`. `PosBitBlocks` start at a given position, `startPos`, and code a bitstring, where the i^{th} bit represents the position `startPos+i`. If the bit is high at this position, then that position is returned in the calls to `getNextPosition()`. Whether `PosBitBlocks` or `PosBasicBlocks` are used depends on an optimizer decision.

`PosDeltaBlock`: In `PosDeltaBlock`s, only individual `PositionBlock`s are position sorted. `PosDeltaBlock`s delta encode the positions, thus amounting to a `DeltaPosBlock`.

`PosRLEBlock`: Essentially like an `RLEBlock`, `PosRLEBlock` are used when there are long runs of positions. `PosRLEBlocks` return true to all three properties.

In summary, `PositionBlocks` are very similar to their `ValueBlock` counterparts. They are employed mainly to strip `ValueBlock`s of values information to provide for greater compressibility and allow us to leverage memory further.

5.2 Operators

Operators have a pull based iterator API. An operator pulls in data from its child and provides data to its parents. Operators are constructed with a pointer to their child. They can request two types of `Block`s from the child:`ValueBlock`s and `PositionBlock`s. Symmetrically, operators provide the same interface to their parents. Note, however, that some operators may only provide one type of output. For example, a logical operators, such as `BAnd`, can only produce `PositionBlocks`. `Select` operators can provide both output types. `Select` can return `ValueBlock`s that satisfied the predicate, or alternatively, it can return `PositionBlock`s indicating all the positions that satisfied the predicate.

We mentioned earlier that properties on `Block`s allow operators to optimize their operations. One way to view `Block` properties is that they provide operators certain guarantees. Given these guarantees, operators know what they can do without breaking correctness. Consider the one-valued property of `Block`s. If an operator knows the `Block` contains only one value, it need only look at the first pair to perform any sort of value operation. A `Select` operator can just look at the first pair and decide whether or not to return the entire `Block`. An aggregation operator that is grouping on a one-valued, position-contiguous `Block`s need only read the first `Pair` to find the value and the size of the `Block` to determine what positions it needs to aggregate into that value's bucket.

In the next few subsections, we look at how some of the more complex operators use properties to perform their functions.

5.2.1 Aggregation

The aggregation in current CStore system was developed by Edmond Lau. There are two main parts to aggregation. Firstly, there is a aggregation function, that is performed over a sequence of values. SQL supports functions like `SUM`, `AVERAGE`, `COUNT` and `MAX`. The second aspect to aggregation is defining the set of data on which to apply the aggregation function. This is done through the SQL `GROUP BY` construct.

The aggregation operators have two inputs. The first input is a `ValueBlock` stream on which the aggregation function will be applied. The second is a `ValueBlock` stream that defines the group to which each position belongs. The description thus far is identical to that of a traditional row store. In CStore, we consider the aggregation to be a row operation, and enforce that the two input columns must be lined up. This means that if we denote A, B as the two input `ValueBlock` streams, where $P_A[i], P_B[i]$ represent the i^{th} `Pair` in the `ValueBlock` streams, the $Position(P_A[i]) = Position(P_B[i])$. Relaxing this requirement is a question for future work.

Aggregation operators are able to optimize if both its input is a one-valued `Block`s. If this is the case, the `COUNT` aggregation function simply adds the size of the `Block`s, and `SUM` aggregation function adds the value times the size of the `Block`. Similarly, on one-valued `Block`s, `MIN` and `MAX` can just compare the running minimum or maximum with the value of that `Block` rather than having to iterate through every occurrence of that value.

5.2.2 Nested Loop Join

The nested loop join operator, `NLJoin`, was developed by Daniel Abadi. The `NLJoin` operator takes in two `ValueBlock` streams and outputs two `PositionBlock` streams, one for each table.

The `NLJoin` operator reads contains two `Block`s at any point in time, an outer `Block` for the outer loop and an inner `Block` for the inner loop. For simplicity, let us assume the blocks are one-valued[3] and that the outer block contains only one position. The `NLJoin` operator now compares the two values to see if they match the join predicate. If they do, the join outputs two `PositionBlock`s: a `PositionBlock` with the positions of the outer `Block` and `PositionBlock` with the positions of the inner `Block`. The `NLJoin` now reads a new inner `Block` and repeats the procedure until the inner `Block` stream is exhausted. At this point it reads the next outer `Block` and resets the inner `Block` stream to the first `Block`, and begins iterating through the inner `Block`s again. The `NLJoin` is complete when the outer `Block` stream no longer has any more `Block`s.

The performance of the `NLJoin` is dependent on the number of positions contained in each inner `Block`. For this reason, `NLJoin` performs well one-valued inner `Block`s, as the predicate need only be matched once for a large number of positions. For multi-valued `Block`s, additional optimizations are possible if the `Block` is value sorted, as the `NLJoin` can optimize predicate matching on the `Block`.

In the previous discussion, we made the assumption that the outer `Block` contains only one position. This could be achieved by decompressing the outer `Block`. However, decompression of the outer `Block` may be unnecessary. For example, if the outer `Block` is one-valued and position contiguous, then we can optimize the `NLJoin` by only requiring one pass over the inner `Block` stream for all positions of the outer

[3]We note that we can make any multi-valued `Block` into a set of one valued `Block`s.

`Block`. A one-valued and position contiguous outer `Block` allows the `NLJoin` to simply output the inner `Block` positions N times for each match of the join predicate, where N is the number of positions contained in the outer `Block`.

5.3 Writing Blocks

Blocks can be converted between types or encoded using writers. There are two types of Writers, value and position writers, value writers write `ValueBlock`s and position writers write `PositionBlock`s. There is a writer for each type of value `Block`. Value `Block` types are listed below:

BasicBlockWriter Writes Basic Blocks. This amounts to decompressing values.

RLEWriter Writes RLEBlocks. Optimizes if possible by taking advantage of position contiguous `Block`s..

DeltaPosWriter Writes DeltaOnPosition Blocks. The entire `Block` stream is progressively decompressed, while `DeltaPosBlock`s are progressively returned as the writer fills a delta list for any given value.

DeltaValWriter Writes DeltaValBlocks Blocks. If the input `Block` stream is position sorted, then all writing is done in memory, if not then writer may spill to disk.

We note that writing of `Block`s is an expensive operation that may amount to recompressing the data as we may be converting between two completely different types of encodings. A writer may have to consume an entire `Block` stream before it can write its first output `Block`. The intermediate results for a particular writer may not fit in memory.

5.4 Datasources

`Datasource`s are responsible for reducing the amount of data read from disk, by using predicates and filters and their knowledge of how data is laid out on disk. In this section we describe how the behavior of `Datasource`'s features and subsequently discuss how these are implemented.

A `Datasource` allows three types of information to be passed to it:

Predicates: A `Datasource` can match `Block`s on predicates, returning only `Block`s whose `Pair`s match the predicate.

Value filters: A `Datasource` will consume a stream of value `Block`s and will only return `Block`s whose values are equal to some value in the stream.

Position filters: A `Datasource` will consume a stream of position `Block`s and will only return `Block`s whose positions are equal to some position in the stream.

Two properties affect how efficiently `Datasource`s can read pages with relevant data.

1. If the `Block` stream is value sorted: predicate and value filters can be applied efficiently.
2. If the `Block` stream is position sorted: position filters can be applied efficiently.

The implementation strategy for each of these features is described in the next few subsections.

5.4.1 Predicates

Predicates filtering can be implemented efficiently if the compressed column can be accessed in value sorted order. Suppose, for example, we have a primary index on value. To implement the predicate $>$ k, we lookup k in a value index and walk down the index until the end of the column. If there is no way to access the column in sorted order, we are forced to perform a sequential scan.

Indices allow us to pick the right pages, but how we handle values within those pages depends on the particular compression scheme for the column. We note that for:

`RLE`: There are multiple `Block`s within a page, therefore after the lookup on the first page we must find the first `Block` that matches the predicate. We are guaranteed that all subsequent `Block`s will satisfy the predicate (as long as the column is value sorted) so these can be returned without evaluating the predicate. If the initial column is unsorted, we must read each `RLETriple` from disk and test whether or not it matches the predicate.

`DeltaOnPosition`: We do not have a traditional value index. However, we have a sorted list of all values in that column. Within this list, we perform a similar lookup and subsequently return all sub-columns that match the predicate.

`DeltaOnValue`: If the original column was sorted, we may have to re-encode a delta list to remove all values that did not match the predicate. If the original column was not sorted, we are forced to decompress to ensure the predicate is satisfied for each value.

An extension of the predicate optimization is the support for value filters described in the next section.

5.4.2 Value Filters

Value filters are dealt with much in the same way as equality predicates, with the exception that we alter the right hand side of the predicate for each new value. To provide support for this scenario, we simply repeat the procedure to find `Block`s that satisfy a predicate for each new value contained in the value filter `ValueBlock` stream.

5.4.3 Position Filters

Position filtering involves filtering a column for a set of positions. The set of positions desired is provided by a `PositionBlock` stream. At a high level, the implementation of position filters is simple to describe:

```
1. foreach (positionBlock in positionBlockFilter)
    (a) Find pages where the positions in positionBlock are coded.
    (b) Find blocks which contain the positions in positionBlock.
    (c) Remove positions from blocks that do not match positions in positionBlock.
2. Return these blocks.
```

Step 1.c may be a costly operation. In the case of delta coding schemes, removing a particular position in a `Block` requires decompression of both the `ValueBlocks`, to know what positions are contained, and the `PositionBlock` filter. The properties of `PositionBlocks` may allow us for some optimizations. In particular, if the `PositionBlock` stream is position sorted[4], then we can avoid decompressing the `PositionBlock` as it is just a range of positions. Additionally, if each `PositionBlock` is position sorted, and the coded column is position sorted then we can perform the equivalent of a merge equijoin between the column and position stream, where `ValueBlock.Pair.position=positionStream.position`.

A larger cost is involved in the case where the `PositionBlock`s we are using to filter are themselves not position sorted and the column itself is not position contiguous. In this case, the `Datasource` is forced to perform random I/O in order to transverse the column in the original sorted order, for each position in the filter. If the column contains `N` pages, and there are `k` positions in the position filter, then in the worst case we need to read $O(N * k)$ pages.

Naturally, we want to move away from such situations. Under CStore, there are currently no schemes where each `ValueBlock`s or `PositionBlock`s are not position sorted. Even in schemes whereby the physical layout of disk is not position sorted, there are always subsets of the column that are position sorted [5]. For each position sorted subset, we can perform the optimizations described above.

In conclusion, position filtering is an equijoin on position, where the predicate is `ValueBlock.Pair.position = PositionBlock.position` where the performance of this operation depends largely on the properties of the column to be filtered, and the stream of positions.

[4]Therefore, each `PositionBlock` is both position sorted and position contiguous.
[5]As is the case for `DeltaPos`, where we have position indexes for each subset

5.5 Lower Layer Primitives

In the discussion above we have largely ignored how we write the compression specific data structures introduced in Chapter 3. Each compression type has an `Encoder` and `Decoder` components that encode pages from `Block`s and that decode pages to `Block`s respectively. These components in turn leverage some components that provide the ability to read and write arbitrary number of bits from and to a buffer respectively. `Block`s and `Datasource`s use these components extensively, therefore, the performance of the entire executor can greatly depend on the performance of these components.

5.5.1 Encoding and Decoding

`Encoder`s and `Decoder`s provide the ability to translate `Block`s to disk pages and vice versa. `Encoder`s may write a single `Block` (as is the case for delta coding based compression algorithms) or many `Block`s per page. `Encoder`s also write per page header to indicate information necessary to decode, and encode any sub-columns required in the compression algorithm. A `PagePlacer` component then takes the series of coded pages and places them in BDB.

`Decoder`s provide iterator access to `Block`s contained on a page. Additionally, they provide methods for quick access to find the starting and ending values and positions contained in the `Block`s coded on that page. These methods are used by `Datasources` to determine whether a certain value or position could be contained on a page.

Encoders and decoders use a set of utility components to read and write an arbitrary number of bits to an in memory buffer, described in the next section.

5.5.2 Reading and Writing Bits

Many of the compression schemes described in chapter 3 depend on the ability to read and write bit width fields. This functionality is provided by the `BitReader` and `BitWriter` components. The `BitWriter` provides the ability to write a value in a variable number of bits to a buffer. It moves sequentially down the buffer eventually failing if it can no longer write a field of that size. Analogously, the `BitReader` class provides the ability to read an arbitrary number of bits sequentially from a buffer.

These two components although not complex, are used by many of the light weight coding schemes and have a significant impact on performance as can be noted in Chapter 6.

5.6 Final note on Software Architecture

In this section, we have discussed the design of most of the components in the executor. We began our discussion with a look at the central `Block` interface used throughout the executor. `Block`s provide the ability to access compressed data efficiently, and through a set of properties provide operators a set of guarantees on the structure of compressed data. Operators exploit this structure to increase performance. `Block`s are read from disk through `Decoder`s by `Datasource`s, who then pass them to operators in the query plan. `Datasource`s provide compression specific access methods with support for indexes and predicate push down on all compression schemes, although the performance of these features depends greatly on the coding scheme involved. In the next chapter, we present the results of the query executor and show the above architecture to be viable.

Chapter 6

Results

In this section we analyze the performance of the CStore query executor. Our benchmarking system is a 3.0 Ghz Pentium IV, running RedHat Linux, with 2 GB of memory and 750 GB of disk. We begin our analysis by first looking at the performance of our storage manager and light weight compression primitives. Subsequently, we compare the CStore executor with current commercial systems and finally we look at how compression is allowing using to improve the performance of queries.

6.1 Micro-Benchmarks: Developing a cost model for lower level operations

In this section, we look at results of micro benchmarks designed to provide us with a basic cost model for the CStore executor. We begin by measuring the performance of BDB indexes, followed by an analysis of the cost of decompression for lightweight schemes.

6.1.1 Performance of Berkeley DB indexes

To execute a query we must first retrieve data from BerkeleyDB, our storage manager. Our performance is dependent on how quickly we can perform sequential and index scans in BerkeleyDB. In the following micro benchmark, we stored 15000 64Kb pages with both primary and secondary indexes. These indexes were then accessed sequentially and randomly. A large file was copied between each experiment to ensure we have no buffered data. Figure 6-1 presents the time per page achieved in each test.

From Figure 6-1, we see that looking up a page in a secondary index requires approximately twice the time than lookups on the primary index. This fits with BDB's design, as a lookup on a secondary index returns a key for lookup in the primary index. Every lookup on a secondary index also requires a lookup

Access Pattern	Time Per Page		
	Sequential Scan	Lookup on Primary Index	Lookup on Secondary Index
Sequential	3.19ms	3.518ms	9.76ms
Random	N/A	12.38 ms	19.97ms

Figure 6-1: Time to Access Pages through BDB

on the primary index, thereby doubling the cost. Random I/O also costs on average 3 to 4 times more than sequential I/O.

6.1.2 Decompression costs

In the case of most light weight schemes, we rely on the ability to read an arbitrary number of bits from a memory buffer. Within CStore there are two classes that provide the methods to read values from a memory buffer: `BitReader` and `ByteReader`. As their names imply, `BitReader` provides the ability to read an arbitrary number of bits whereas `ByteReader` provides the ability to read an arbitrary number of bytes. In Figure 6-2, we measure the average CPU costs of reading fixed sized fields of various widths from memory.

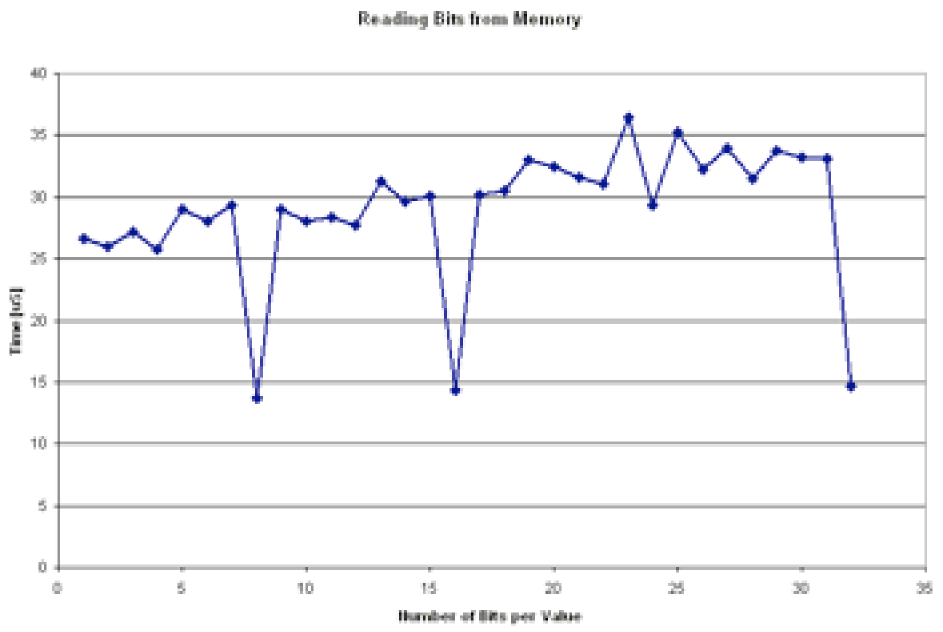


Figure 6-2: CPU costs of reading values in memory

The dips in the graph at 8,16 and 32 bits because processors are optimized to read bytes, shorts(2 bytes), and integers (4 bytes). We also note that the performance of `BitReader` deteriorates as the size of each value increases. This is because `BitReader` reads a 32 bit buffer each time. If the particular value it is reading is larger than the buffer, it reads the higher order bits from the current buffer and the lower order bits from the next 32 bit buffer. As values increase in size, the probability that `BitReader` has to read two 32 bit buffers increases and performance suffers.

In the context of CStore, to optimize performance we are interested in reducing the time required to read each value. In Figure 6-3, we consider reading a million values from disk. There are two parameters we alter. First we alter the way in which pages are read from disk. Are we reading the pages sequentially, through a sequential scan of a column, or are we reading random pages by reading each page from a different attribute. The random page I/O scenario is more likely under a column store system, as we have to read many different attributes in parallel to answer a query. The second parameter altered is the size of the fields on the page. The experiment runs by retrieving pages from disk and reading all the values on each page until we have read a million values.

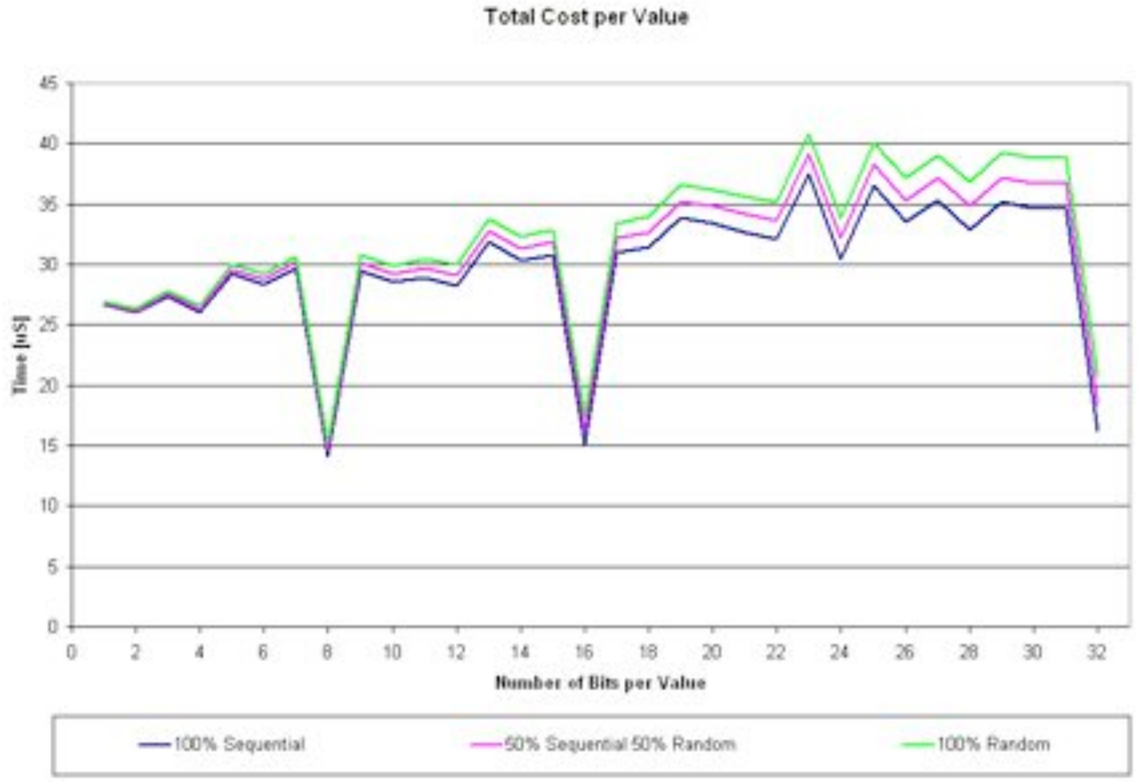


Figure 6-3: Total Cost of Reading Column Values

In Figure 6-3, we note that the total time per value depends more on the type of disk access as the field size increases. As the field size increases, we can place fewer values in a page and therefore are required to read more pages to retrieve a million values.

Figure 6-3 also shows that for performance reasons, we should always pick 8, 16 or 32 bit fields for light weight compression schemes. For example, in `DeltaPos` we maximize performance by having byte aligned deltas. This is not an intuitive result, as we note that we can fit eight times more 1 bit values than 8 bit values, but reading a bit is not eight times more expensive than reading a byte.

To clarify the reason for this difference, let us consider the total cost of reading 64K single bit values against reading 64K bytes. To read 64K one bit values we need to read a single page from disk, and from Figure 6-2 we see that reading bit sized fields requires twice the CPU time than reading byte sized values. Let us denote the time to load a page from disk as D, and the time to read a single byte from memory as C. For single bit values the time is equivalent to that of reading a page and reading in 64K single bit values (where each value read requires 2*C time). Therefore, we have:

$TotalTime = D + 2 * 65536 * C$

To read 64K byte sized values we need to read 8 pages, however the time to read each value is just C. Therefore the total time to read 64K byte-sized values is is:

$TotalTime = 8 * D + 65536 * C$

For reading one bit values to be faster than byte sized values, we require that

$D + 2 * 65536 * C < 8 * D + 2 * 65536 * C$

$65536 * C < 7 * D$

Now, C is around 14uS, so for reading bit values be faster than byte sized values then reading a page from disk would have to take:

$D > \frac{65536 * 14uS}{7}$

$D > 131072uS$

$D > 131$ ms

The conclusion is that disks are not slow enough to compensate non byte aligned data access.

6.2 CStore Performance compared with Commercial DBMSs

In data warehousing, the industry benchmark for performance is known as TPC-H. We use a simplified version of the TPC-H queries to compare our performance against two commercial systems, one row based and a second column based product. We implement the following tables:

lineitem < `orderID[int]`, `partkey[int]`, `suppkey[int]`, `linenumber[int]`, `quantity[int]`, `extendedprice[int]`, `returnflag[byte]`, `shipdate[int]` >

orders < `orderID[int]`, `orderDate[int]` >

customer < `custID[int]`, `nationID[int]` >

The TPC-H data generator, operated on scale 10, generates 1.8 GBs of data. Of the three tables, the lineitem table is the largest with 60 million rows. In our experiments, CStore operated on the following schema:

D1: <**lineitem**> sorted by `shipdate, suppkey`.
D2: <`orders.orderdate, lineitem.shipdate, lineitem.suppkey`> sorted by `orderdate, suppkey`.
D3: <`order.orderdate, order.custID, order.orderID`> sorted by `orderdate`.
D4: <`lineitem.returnflag, lineitem.extprice, customer.nationID`> sorted by `returnflag`.
D5: <`customer.custID, customer.nationID`> sorted by `custID`.

D2 and **D4** are materialized join views. A materialized join is created by storing the result of a join between two tables. **D2** is created by joining the `lineitem` and `order` tables with the predicate `lineitem.orderkey` = `orders.orderkey`. **D4** is created by joining **D2** and the `customers` table with the predicate `customer.custID` = `customers.custID`. We ran the following queries over this schema:

Query 1 Determine the total number of items shipped for each day after day D.

```
SELECT shipdate, COUNT (*)
FROM lineitem
WHERE shipdate > D
GROUP BY shipdate
```

Query 2 Determine the total number of items shipped for each supplier on day D.

```
SELECT suppkey, COUNT (*)
FROM lineitem
WHERE shipdate = D
GROUP BY suppkey
```

Query 3 Determine the total number of lineitems shipped for each supplier after day D.

```
SELECT suppkey, COUNT (*)
FROM lineitem
WHERE shipdate > D
GROUP BY suppkey
```

Query 4 For every day after D, determine the latest shipdate of all items ordered on that day.

```
SELECT orderdate, MAX (shipdate)
FROM lineitem, orders
WHERE lineitem.orderkey = orders.orderkey AND
orderdate > D
GROUP BY orderdate
```

Query 5 For each supplier, determine the latest shipdate of an item from an order that was made on some date, D.

```
SELECT suppkey, MAX (shipdate)
FROM lineitem, orders
WHERE lineitem.orderID = orders.orderID AND
orderdate = D
GROUP BY suppkey
```

Query 6 For each supplier, determine the latest shipdate of an item from an order made after some date, D.

```
SELECT suppkey, MAX (shipdate)
FROM lineitem, orders
WHERE lineitem.orderID = orders.orderID AND
orderdate > D
GROUP BY suppkey
```

Query 7 Return a list of identifiers for all nations represented by customers along with their total lost revenue for the parts they have returned. This is a simplified version of query 10 (Q10) of TPC-H.

```
SELECT nationID, sum(extendedprice)
FROM lineitem, orders, customer
WHERE lineitem.orderID=orders.orderID AND
customers.custID=customer.custID AND
returnflag='R'
GROUP BY nationID
```

The query plans for these seven queries are shown in Figure 6-4. We ran the same queries on two commercial databases, a row store and a column store. We ran the commercials systems on the same platform and disable locking and logging. In addition, we tuned any system specific parameters to what we believe to be optimal value for these queries. We presented the two commercial system in two scenarios. In the first scenario, these systems are space constrained and are to operate with a budget of 2.7GB. The row store was unable to operate in this scenario, so we expanded its budget to 4.5GB which was the minimum amount it required to store the tables and their indexes. In the second scenario, we removed the space constraint and gave both systems the CStore schema. The space usage by both systems is shown in Figure 6-5.

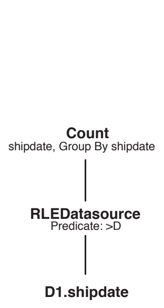
From Figure 6-5, we see that CStore space usage is remarkably smaller than either of the commercial systems. With the same schema, CStore uses 83% less disk space than the row store and 52% less than the column store.

CStore also excelled in performance. It outperformed both commercial systems under both scenarios. Figure 6-6 shows the performance of all systems, and in addition the performance of the CStore executor using only decompressed data.

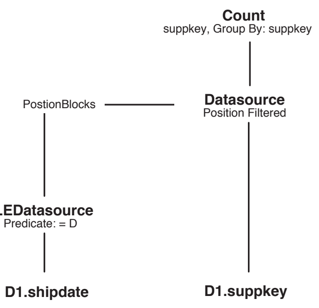
In subsequent sections, we provide more detailed analysis of the costs of these queries within CStore, beginning with a low level analysis of the disk access and decompression costs.

6.2.1 Analysis of Costs

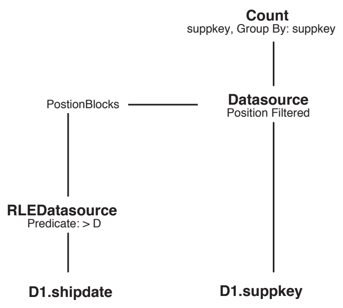
In this section we analyze the particular costs involved in each query. We measured disk cost by recording the user time, system time and real time before and after any calls to retrieve a page from disk. Additionally, to measure the relative costs of operations within the queries, we profiled each query using callgrind[1] a tool that uses the valgrind[18] framework estimate the number of cycles spent in each line of code. We present this analysis the next subsections.



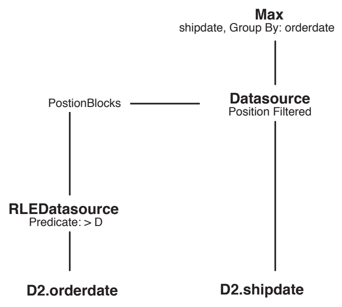
(a) Query 1 Plan



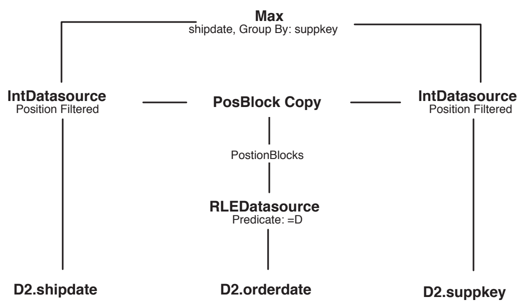
(b) Query 2 Plan



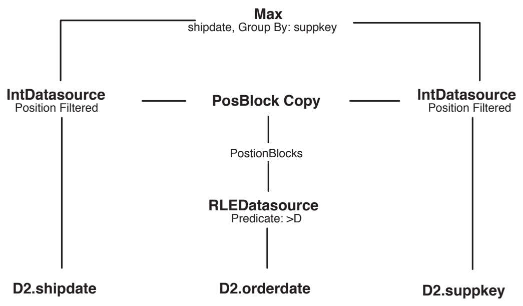
(c) Query 3 Plan



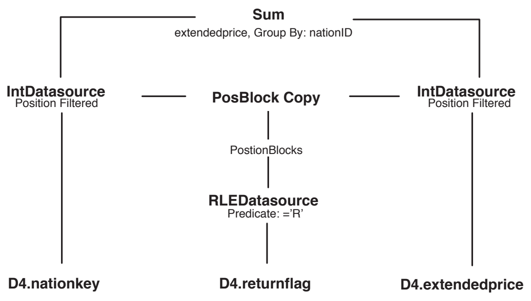
(d) Query 4 Plan



(e) Query 5 Plan



(f) Query 6 Plan



(g) Query 7 Plan

Figure 6-4: Query Plans

Database System	**Constrained Space Usage**	**Unconstrained Space Usage**
CStore	1.980GB	1.98GB
Row Store	4.48GB	11.90GB
Column Store	2.65GB	4.09GB

Figure 6-5: Disk Space Usage

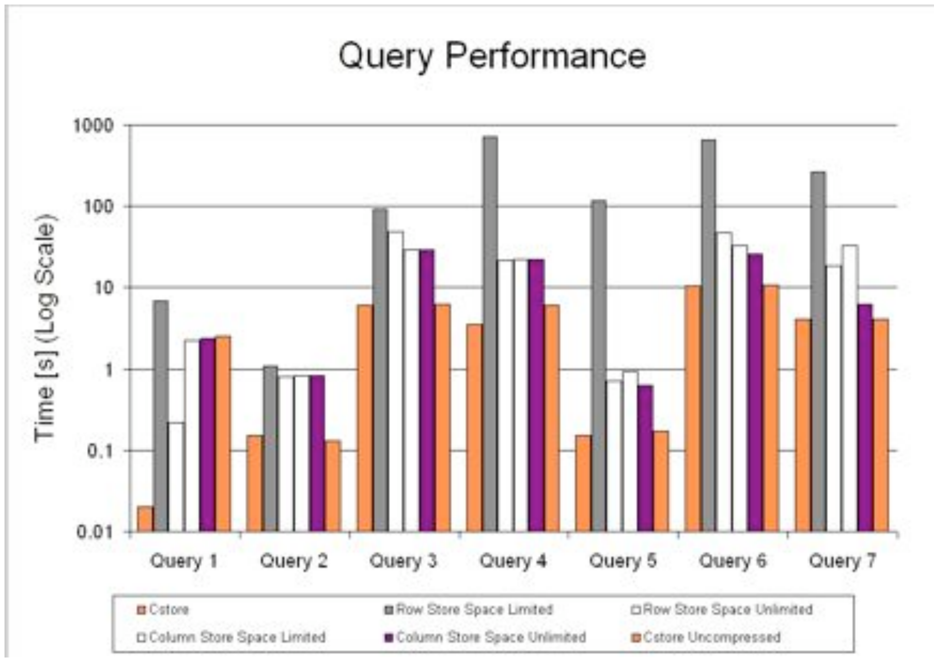


Figure 6-6: Query Results

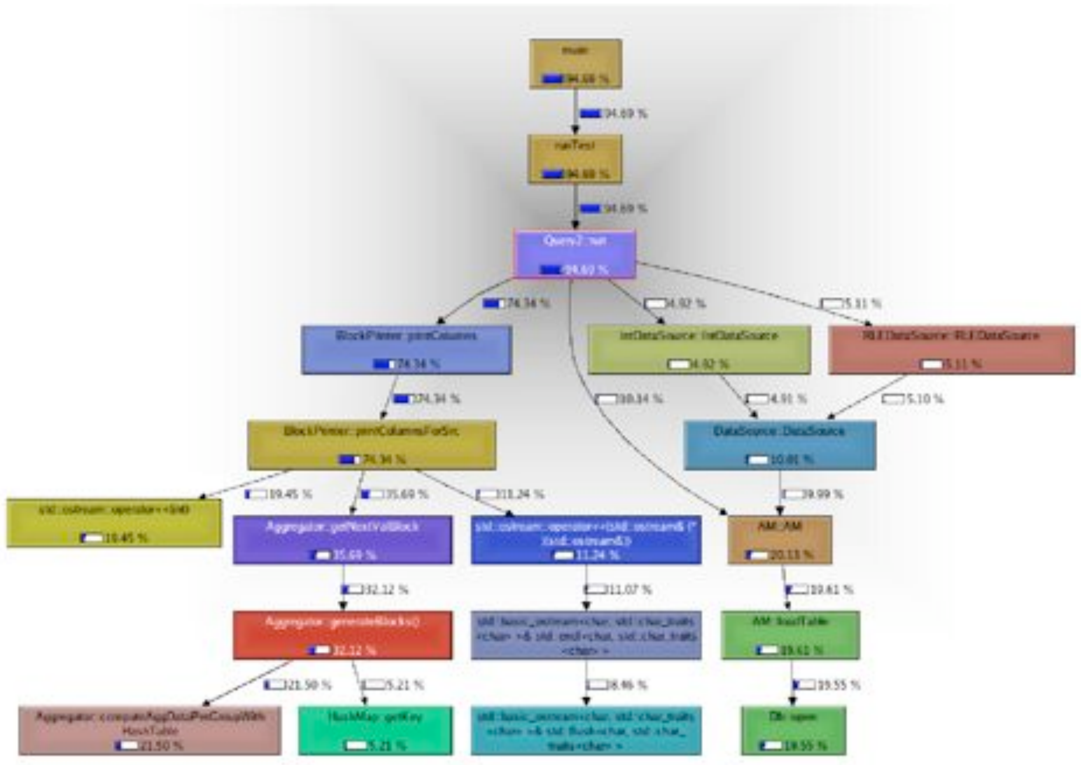
Queries 1, 2

Queries 1 and 2 are both short running queries. In these queries, disk I/O and query plan instantiation dominate the total query costs. Figure 6-7 shows the CPU and disk costs in each of these three queries, while Figure 6-8 shows the break down of CPU costs between various components of the code for Query2.

Query Number	CPU Time	Disk Time
1	50%	50%[1]
2	70%	30%

Figure 6-7: Results for Queries 1 and 2

We see from Figure 6-7 disk costs around 40% of the query time. From 6-8, we observe that main processor costs are in the final decompression to write the results to disk, with a total cost of 74.34%-35.69%=38.65%. Data computation requires 35.69% of the CPU time. The overhead for the instatiation of the `IntDataSource`, `RLEDataSource` and BDB access methods (`AM`) are the largest remaining costs accounting for around 20% of the processor time.



main
94.69 %
runTest
94.69 %
Query2::run
94.69 %
74.34 %
4.92 %
5.11 %
BlockPrinter::printColumns
74.34 %
IntDataSource::IntDataSource
4.92 %
RLEDataSource::RLEDataSource
5.11 %
10.34 %
4.91 %
5.10 %
BlockPrinter::printColumnsForSrc
74.34 %
DataSource::DataSource
10.01 %
19.45 %
35.69 %
11.24 %
9.99 %
std::ostream::operator<<(int)
19.45 %
Aggregator::getNextValBlock
35.69 %
std::ostream::operator<<(std::ostream& (*)(std::ostream&))
11.24 %
AM::AM
20.13 %
32.12 %
11.07 %
19.61 %
Aggregator::generateBlocks()
32.12 %
std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char> >
AM::loadTable
19.61 %
21.50 %
5.21 %
8.46 %
19.55 %
Aggregator::computeAggDataPerGroupWithHashTable
21.50 %
HashMap::getKey
5.21 %
std::basic_ostream<char, std::char_traits<char> >& std::flush<char, std::char_traits<char> >
Db::open
19.55 %

Figure 6-8: CPU Profiling for Query 2

Query Number	CPU Time	Disk Time	
3	73%	27%	
5	64%	46%	
6	76%	24%	
7	57%	43%	

Figure 6-9: Results for Queries 3,5,6 and 7

Query 3,5,6 and 7

Of this group of queries, Queries 3 and 6 are the most similar. Both queries are CPU limited. As shown in Figure 6-9 CPU time accounts for a three quarters of query time, with the remaining quarter of time used for disk I/O.

Both queries 3 and 6 have a predicate on an `RLE` encoded column. Query 3 position filters an uncompressed `Datasource`, while Query 6 filters two uncompressed `Datasource`s. Subsequently, they both aggregate, with a self aggregation in Query 3 and Query 6 aggregating on `orders.shipdate`. In both cases, the group by column is `suppkey`, which is an uncompressed and unsorted column.

As the `suppkey` column is unsorted, we perform hash based aggregation. The cost in both these queries is dominated by the aggregation operation and the `HashMap` data structure, as can be seen for Query 3 in Figure 6-10. In Query 3, the aggregation operator consumes a total of 98.6%-35.59%=63.01 % of CPU time. We also note that the performance of the query is dominated by operations on the uncompressed data. Filtering operations are fast as filtering `PositionBlock`s are compressed. Retrieving individual values from a memory buffer and placing them in `BasicBlock`s takes up a total of 32% of the CPU time. Optimizing this cost is an area of future work.

Query 5 is similar to Query 6, except that the equality predicate is far more selective. For this reason, it deals with less data and is a short running query. 50% of the costs are still involved in performing aggregation, with 30% involved in retrieving data and the remaining costs involved in instatiation and writing the results.

The query plan for Query 7 is identical to Query 5 except on different data. The equality predicate applied on the `RLEDataSource` is less selective than in Query 5, as the `returnflag` has fewer values than `orderdate`. For this reason, Query 7 requires more time, however, the relative costs of each operation in the query plan remain largely unaltered.

6.3 Operating Directly on Compressed Data

Of the 7 queries executed, only queries 1 and 4 operate directly on compressed data. All other queries filter a compressed column and subsequently use the results of that query to filter other columns. However, the executor is designed to generate compressed position vectors, therefore, a predicate applied on

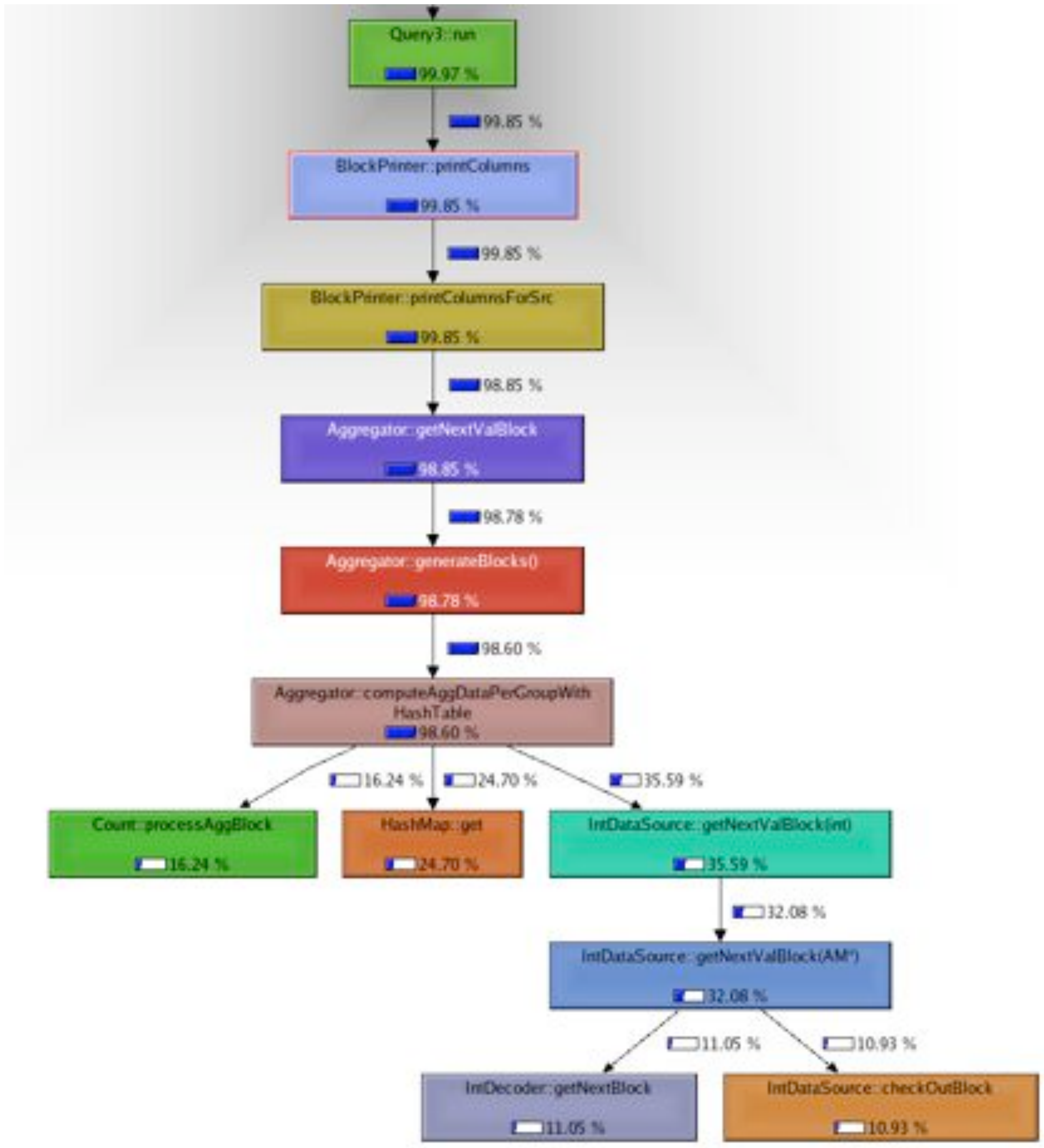

Query3::run
99.97 %
99.85 %
BlockPrinter::printColumns
99.85 %
99.85 %
BlockPrinter::printColumnsForSrc
99.85 %
98.85 %
Aggregator::getNextValBlock
98.85 %
98.78 %
Aggregator::generateBlocks()
98.78 %
98.60 %
Aggregator::computeAggDataPerGroupWith HashTable
98.60 %
16.24 %
24.70 %
35.59 %
Count::processAggBlock
16.24 %
HashMap::get
24.70 %
IntDataSource::getNextValBlock(int)
35.59 %
32.08 %
IntDataSource::getNextValBlock(AM*)
32.08 %
11.05 %
10.93 %
IntDecoder::getNextBlock
11.05 %
IntDataSource::checkOutBlock
10.93 %

Figure 6-10: CPU Profiling for Query 3

an `IntDataSource` on sorted data will produce a `PosRLEBlock` to represent the positions that match the predicate. For this reason, we see in Figure 6-6 the performance of the CStore executor on compressed or uncompressed data differs little for Queries 2,3,5,6 and 7.

Queries 1 and 4 do show great improvements when run on compressed versus on compressed data. In both these queries, we perform an aggregation grouped by an RLE encoded column. Query 1 shows a two orders of magnitude improvement in performance when operating on compressed data, while Query 4 shows only a factor of 2 improvement in performance. Query 4's benefit is more modest, as performance is dominated by the computation of the aggregation function on the uncompressed column.

6.4 Result Summary

To summarize we believe the following factors allowed us to achieve better performance:

- Column Architecture: we avoid reading unnecessary attributes
- Operation on Compressed Data: we avoid decompression costs and can perform operations faster.
- Better Compression Schemes: allow us to lower our disk space requirements.
- Compressed Intermediate Results: allow us the number of iterator calls and allow for further optimizations.

Chapter 7

Related Work

There is a long history of research into the use of compression in databases[9, 16, 3]. However, it was not until the 1990s that research first concentrated on how compression affects database performance[4, 7, 14, 5, 8]. Most work concluded that there were a limited number of compression algorithms that could be used as the CPU cost of decompression outweighs the I/O savings. Therefore, most systems concentrated on the use of light-weight compression techniques. Light-weight schemes sacrifice compression ratio in order to ensure low compression and decompression CPU requirements.

Following the initial studies, research turned to reducing the cost of decompression. In early systems, data was decompressed immediately after it was read from disk. Graefe and Shapiro [4] suggested that it was not always necessary to decompress data. In particular, it was possible to lazily decompress the data, if and when that data was used by the executor. In such systems, data is compressed on the attribute level and held compressed in memory. Data would only be decompressed when an operator were to access the data.

Others proposed hybrid schemes with transient decompression. Chen et. al. [2] 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.

In this thesis, we have described the development of a query executor that operates directly on compressed data. The idea of working on compressed data is not new. Graefe et al. [4] pointed that exact match comparisons and natural joins can be performed directly on compressed data. Additionally, they point out that projection and duplicate elimination can be performed on compressed data, as well as exact-match index lookups if an order-preserving (consistent) compression scheme is used.

Under CStore, we look at compression algorithms to provide us additional structure over the data. We introduce a novel architecture to pass compressed data between operators that minimizes operator code complexity while maximizing opportunities for direct operation on compressed data. [5, 19, 2] stress the importance of abstracting the executor from the details of a compression technique. They achieve this by decompressing the data before it is exposed to operators. Instead, we provide operators with a common interface to decompress data, to provide this abstraction. However, by exposing sufficient information about the compressed values, operators are able to operate directly on compressed data. Our executor's high performance is a direct consequence of operator's ability to do this.

Chapter 8

Future Work

There are many projects on going within CStore to make it a complete DBMS. Our work, in this thesis, has assumed many correct decisions will be made by the optimizer, so it is worthwhile discussing some of the challenges that will be faced in designing a query optimizer. In addition, in our implementation of the CStore executor we have identified several areas for future research. In the next two section, we outline some of our ideas for the future work in the optimizer and the executor.

8.1 Optimizer

Currently missing in our work is an analysis of how the CStore query optimizer will pick good query plans. The CStore executor introduces many new complexities to the optimizer. Storing columns redundantly under different schemes forces the optimizer to decide which columns to use for a particular query. To accomplish this, the optimizer is required to have a cost model for operations over different types of compressed, along with the traditional cost models for deciding which physical operators to use. The optimizer will also have to decide whether or not to filter `Datasource`s or perform sequential scans, depending on its estimate of the cardinality of the `PositionBlock` stream and the cost to apply this position filter to each specific `Datasource`. The metrics required by the optimizer and its structure are at this point undefined.

8.2 Executor

Within the executor, we believe there are two major areas of future work: one in the short term and a second in the long term. The first is a exhaustive performance study of the current architecture with the goal of minimizing the number of cache misses and improving performance. We estimate that a total of 18 to 23% of CPU time is due to cache misses. The second area is to relax the requirement on aggregation operators that their two input's positions be lined up.

From profiling the performance of the current system, we found that the performance of reading uncompressed data dominates our system. In our current design, we read a single value from the page and pass the value between all operators before reading the next value from the page. When we go to retrieve the next value, we find that the page itself (or whatever part of the page we reading at) has been evicted from the cache. This means to read each value we are required to read from main memory. The alternative is for `BasicBlock`s to contain more than one value. The optimal number of values contained by `Block` needs to be experimentally determined and is likely to be platform dependent.

In the long term, it would be interesting to relax the requirement that the two input streams to operators like the aggregation operators, `BAND`, `BOr` have to be position aligned. We currently require, in all of these operators, that the i^{th} position in the both the input streams is the same for all i. Relaxing this requirement means that these operators must keep some state, as operators must remember the positions (and in the case of aggregation, the value at that position) that they have not yet seen in the other input stream. Depending on the ordering of positions, the number of positions stored is O(n) where n is the largest input column. This seems unreasonable, as operators are forced to spill data to disk in order to hold so much state.

One option is to give some guarantees to operators as to how out of order columns are. Suppose that we have an unsorted column, and we decide to sort every 100 values while keeping track of each values original position. This is equivalent to sorting 100 adjacent `Pair`s on value. We can place these `Pair`s in a `Block`, where the successive values in the `Pair`s can be delta-encoded and positions in the `Pair`s can be offset-encoded[1]. We now have value-sorted blocks, allowing for greater performance in aggregation operators such as min and max and operators with predicates such as selects and joins. However, we have increased the cost and complexity of position operations, such as the group by operation in aggregation. The advantage is that we know that given 100 values we will have seen the the corresponding 100 adjacent positions, therefore, the amount of state that must be held is bounded.

[1]In offset encoding we take the median of the positions and encode each position as the difference from this median. This would require $log_2(50)$ bits plus one bit to encode the sign

Chapter 9

Conclusion

In this thesis, we showed how operating directly on compressed data can improve query performance. We saw a performance advantage of up to two orders of magnitude when we switched from uncompressed to compressed data. The key to performance is not only the reduction in disk I/O, but the executor's ability to operate directly on compressed data. Operating on compressed data not only eliminates the cost of decompression, but also increases operator performance as operators can take advantage of the structure of compressed data. In addition, the use of compressed intermediary results, such as `PositionBlock`s used for filtering, combined with column-oriented design allowed us to, on average, outperform commercial systems by an order of magnitude.

The CStore query executor provides a common interface to operate on compressed data, giving operators the ability to optimize based on a set of general properties. Introduction of new compression schemes requires no changes to operator code, as compression specific details are hidden from operators by compression specific code in `Block`s and `Datasource`s.

The encouraging performance numbers of our initial CStore implementation suggest the novel architecture proposed in this thesis has the potential to alter the way real data warehouse systems are built. The combination of compression and a column-oriented data model allow us to perform well and still have low disk storage costs.

Bibliography

[1] Callgrind. Callgrind. http://kcachegrind.sourceforge.net/cgi-bin/show.cgi.

[2] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 271–282. ACM Press, 2001.

[3] Gordon V. Cormack. Data compression on a database system. *Commun. ACM*, 28(12):1336–1342, 1985.

[4] G.Graefe and L.Shapiro. Data compression and database performance. In ACM/IEEE-CS Symp. On Applied Computing pages 22 -27, April 1991.

[5] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 370–379. IEEE Computer Society, 1998.

[6] http://www.sybase.com/products/databaseservers/sybaseiq.

[7] Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 695–704. Morgan Kaufmann Publishers Inc., 1994.

[8] Theodore Johnson. Performance measurements of compressed bitmap indices. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 278–289. Morgan Kaufmann Publishers Inc., 1999.

[9] Clifford A. Lynch and E. B. Brownrigg. Application of data compression to a large bibliographic data base. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 435–447. IEEE Computer Society, 1981.

[10] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Raisin, N. Tran and S. Zdonik. C-store: A column-oriented dbms. In *VLDB*, 2005.

[11] Markus F. X. J. Oberhummer. LZO: A real-time data compression library, 2002. http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[12] P. O'Neil and D. Quass. A universal algorithm for sequential data compression. In *IEEE Transactions on Information Theory, vol. 23, no. 3*, pages 337–343, 1977.

[13] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, 1997.

[14] Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, pages 0–, 1995.

[15] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, 1993.

[16] Dennis G. Severance. A practitioner's guide to data base compression - tutorial. *Inf. Syst.*, 8(1):51–62, 1983.

[17] Sleepycat. Berkeleydb. http://www.sleepycat.com/products/db.shtml.

[18] Valgrind. Valgrind. http://kcachegrind.sourceforge.net/cgi-bin/show.cgi.

[19] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, 2000.