Integrating Compression and Execution in Column-Oriented Database Systems

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

Samuel R. Madden
MIT
madden@csail.mit.edu

Miguel C. Ferreira
MIT
mferreira@alum.mit.edu

ABSTRACT

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

In this paper, we discuss how we extended C-Store (a column-oriented DBMS) with a compression sub-system. We show how compression schemes not traditionally used in row-oriented DBMSs can be applied to column-oriented systems. We then evaluate a set of compression schemes and show that the best scheme depends not only on the properties of the data but also on the nature of the query workload.

1. INTRODUCTION

Compression in traditional database systems is known to improve performance significantly [13, 16, 25, 14, 17, 37]: it reduces the size of the data and improves I/O performance by reducing seek times (the data are stored nearer to each other), reducing transfer times (there is less data to transfer), and increasing buffer hit rate (a larger fraction of the DBMS fits in buffer pool). For queries that are I/O limited, the CPU overhead of decompression is often compensated for by the I/O improvements.

In this paper, we revisit this literature on compression in the context of column-oriented database systems [28, 9, 10, 18, 21, 1, 19]. A column-oriented database system (or "column-store") is one in which each attribute is stored in a separate column, such that successive values of that attribute are stored consecutively on disk. This is in contrast to most common database systems (e.g. Oracle, IBM DB2, Microsoft SQL Server) that store relations in rows ("row-stores") where values of different attributes from the same tuple are stored consecutively. Since there has been significant recent interest in column-oriented databases in both the research community [28, 9, 10, 18, 24, 34] and in the commercial arena [21, 1, 19], we believe the time is right to systematically revisit the topic of compression in the context of these systems, particularly given that one of the oft-cited advantages of column-stores is their compressibility.

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

Compression techniques for row-stores often employ dictionary schemes where a dictionary is used to code wide values in the attribute domain into smaller codes. For example, a simple dictionary for a string-typed column of colors might map "blue" to 0, "yellow" to 1, "green" to 2, and so on [13, 26, 11, 37]. Sometimes these schemes employ prefix-coding based on symbol frequencies (e.g., Huffman encoding [15]) or express values as small differences from some frame of reference and remove leading nulls from them (e.g., [29, 14, 26, 37]). In addition to these traditional techniques, column-stores are also well-suited to compression schemes that compress values from more than one row at a time. This allows for a larger variety of viable compression algorithms. For example, run-length encoding (RLE), where repeats of the same element are expressed as (value, run-length) pairs, is an attractive approach for compressing sorted data in a column-store. Similarly, improvements to traditional compression algorithms that allow basic *symbols* to span more than one column entry are also possible in a column-store.

Compression ratios are also generally higher in column-stores because consecutive entries in a column are often quite similar to each other, whereas adjacent attributes in a tuple are not [21]. Further, the CPU overhead of iterating through a page of column values tends to be less than that of iterating through a page of tuples (especially when all values in a column are the same size), allowing for increased decompression speed by using vectorized code that takes advantage of the super-scalar properties of modern CPUs [10, 37]. Finally, column-stores can store different columns in different sort-orders [28], further increasing the potential for compression, since sorted data is usually quite compressible.

Column-oriented compression schemes also improve CPU performance by allowing database operators to operate directly on compressed data. This is particularly true for compression schemes like run length encoding that refer to multiple entries with the same value in a single record. For ex-

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

ample, if a run-length encoded column says the value "42" appears 1000 times consecutively in a particular column for which we are computing a SUM aggregate, the operator can simply take the product of the value and run-length as the SUM, without having to decompress.

In this paper we study a number of alternative compression schemes that are especially well-suited to column stores, and show how these schemes can easily be integrated into C-Store, an open-source column-oriented database [3, 28].

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

- We overview commonly used DBMS compression algorithms and show how they can be applied in column-store systems. We compare this traditional set of algorithms with compression algorithms especially suited for column-store systems.
- Through experiments, we explore the trade-offs between these algorithms, varying the characteristics of the data set and the query workload. We use results from these experiments to create a decision tree to aid the database designer to decide how to compress a particular column.
- We introduce an architecture for a query executor that allows for direct operation on compressed data while minimizing the complexity of adding new compression algorithms. We experimentally show the benefits of operating directly on compressed data.
- We illustrate the importance of introducing as much order as possible into columns and demonstrate the value of having secondary and tertiary sort orders.

As a caveat, we note that the purpose of this paper is not to propose fundamental new compression schemes. Many of the approaches we employ have been investigated in isolation in the context of row-oriented databases, and all are known in the data compression literature (we only propose slight variations to these schemes). Our purpose is to explore the performance and architectural implications of integrating a wide range of compression schemes into a column-oriented database. Our focus in this paper is in using compression to maximize query performance, not to minimize storage sizes.

2. RELATED WORK

While research in database compression has been around nearly as long as there has been research in databases [20, 27, 12], compression methods were not commonly used in DBMSs until the 1990s. This is perhaps because much of the early work concentrated on reducing the size of the stored data, and it was not until the 90s when researchers began to concentrate on how compression affects database performance [13, 16, 25, 14, 17]. This research observed that while compression does reduce I/O, if the CPU cost of compressing/decompressing the data outweighs this savings, then the overall performance of the database is reduced. As improvements in CPU speed continue to outpace improvements in memory and disk access [8], this trade-off becomes more favorable for compression. In order to keep CPU costs down, most papers focus on light-weight techniques (in the sense that they are not CPU-intensive) that result in sub-optimal compression but that have low CPU overhead so that performance is improved when taking into consideration all relevant costs.

One way that the CPU overhead of compression has been reduced over the years is by integrating knowledge about compression into the query executor and allowing some amount of operation directly on compressed data. In early databases, data would be compressed on disk and then eagerly decompressed upon being read into memory. This had the disadvantage that everything read into memory had to be decompressed whether or not it was actually used. Graefe and Shapiro [13] (and later Goldstein et. al. [14], and Westmann et. al [29], and in the context of column-oriented DBMSs, MonetDB/X100 [37]) cite the virtues of lazy decompression, where data is compressed on the attribute level and held compressed in memory, and data is decompressed only if needed to be operated on. This has the advantage that some operations such as a hybrid hash join see improved performance by being able to keep a higher percentage of the table in memory, reducing the number of spills to disk. Chen et. al. [11] note that some operators can decompress transiently, decompressing to perform operations such as applying a predicate, but keeping a copy of the compressed data and returning the compressed data if the predicate succeeds.

The idea of decreasing CPU costs by operating directly on compressed data was introduced by Graefe and Shapiro [13]. They pointed out that exact-match comparisons and natural joins can be performed directly on compressed data if the constant portion of the predicate is compressed in the same way as the data. Also exact-match index lookups are possible on compressed data if an order-preserving (consistent) compression scheme is used. Further, projection and duplicate elimination can be performed on compressed data. However, this is the extent of research on direct operation on compressed data. In particular, to the best of our knowledge, there has been no attempt to take advantage of some compression algorithms' ability to represent multiple values in a single field to simultaneously apply an operation on these many values at once. In essence, previous work has viewed each tuple as compressed or uncompressed, and when operations cannot simply compare compressed values, they must be performed on decompressed tuples. Our work shows that column-oriented compression schemes provide further opportunity for direct operation on compressed data.

Our work also introduces a novel architecture for passing compressed data between operators that minimizes operator code complexity while maximizing opportunities for direct operation on compressed data. Previous work [14, 29, 11] also stresses the importance of insulating the higher levels of the DBMS code from the details of the compression technique. In general, this is accomplished by decompressing the data before it reaches the operators (unless dictionary compression is used and the data can be processed directly). However, in some cases increased performance can be obtained in query processing if operators can operate directly on compressed data (beyond simple dictionary schemes) and our work is the first to propose a solution to profit from these potential optimizations while keeping the higher levels of the DBMS as insulated as possible.

In summary, in this paper we revisit much of this related work on compression in the context of column-oriented database systems and we differ from other work on compression in column-oriented DBMSs (Zukowski et. al [37] on

MonetDB/X100) in that we focus on column-oriented compression algorithms and direct operation on compressed data (whereas [37] focuses on improving CPU/cache performance of standard row-based light-weight techniques).

3. C-STORE ARCHITECTURE

For this study on compression in column-oriented DBMSs, we chose to extend the C-Store system [28] since the C-Store architecture is designed to maximize the ability to achieve good compression ratios. We now present a brief overview of the salient parts of the C-Store architecture.

C-Store provides a relational interface on top of a column-store. Logically, users interact with tables in SQL. Each table is physically represented as a collection of *projections*. Each projection consists of a set of columns, each stored column-wise, along with a common sort order for those columns. Every column of each table is represented in at least one projection, and columns are allowed to be stored in multiple projections – this allows the query optimizer to choose from one of several available sort orders for a given column. Columns within a projection can be secondarily or tertiarily sorted; e.g. an example C-Store projection with four columns taken from TPC-H could be:

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

indicating that the projection is sorted by shipdate, secondarily sorted by quantity, and tertiarily sorted by return flag in the example above. These secondary levels of sorting increase the locality of the data, improving the performance of most of the compression algorithms (for example, RLE compression can now be used on quantity and return flag). Projections in C-Store typically have few columns and multiple secondary sort orders, which allows most columns to compress quite well. Thus, with a given space budget, it is often possible to store the same column in multiple projections, each with a different sort order.

Projections in C-Store are related to each other via *join indices* [28], which are simply permutations that map tuples in one projection to the corresponding tuples in another projection from the same source relation.

We extended C-Store such that each column is compressed using one of the methods described in Section 4. As the results in Section 6 show, different types of data are best represented with different compressions schemes. For example, a column of sorted numerical data is likely best compressed with RLE compression, whereas a column of unsorted data from a smaller domain is likely best compressed using our dictionary compression method. An interesting direction for future research could be to use these results to develop a set of tools that automatically select the best partitions and compression schemes for a given logical table.

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

- Selection operators produce bit-columns that can be efficiently combined. A special "mask" operator is used to materialize a subset of values from a column and a bitmap.
- A special permute operator is used to reorder a column using a join index.
- Projection is free since it requires no changes to the data, and two projections in the same order can be concatenated for free as well.
- Joins produce *positions* rather than values. A complete discussion of this distinction is given in Section 5.2

4. COMPRESSION SCHEMES

In this section we briefly describe the compression schemes that we implemented and experimented with in our column-oriented DBMS. For each scheme, we first give a brief description of the traditional version of the scheme as previously used in row store systems (due to lack of space we do not give complete descriptions, but cite papers that provide more detail when possible). We then describe how the algorithm is used in the context of column-oriented databases.

4.1 Null Suppression

There are many variations on the null compression technique (see [26, 29] for some examples), but the fundamental idea is that consecutive zeros or blanks in the data are deleted and replaced with a description of how many there were and where they existed. Generally, this technique performs well on data sets where zeros or blanks appear frequently. We chose to implement a column-oriented version of the scheme described in [29]. Specifically, we allow field sizes to be variable and encode the number of bytes needed to store each field in a field prefix. This allows us to omit leading nulls needed to pad the data to a fixed size. For example, for integer types, rather than using the full 4 bytes to store the integer, we encoded the exact number of bytes needed using two bits (1, 2, 3, or 4 bytes) and placed these two bits before the integer. To stay byte-aligned (see Section 4.2 for a discussion on why we do this), we combined these bits with the bits for three other integers (to make a full byte's worth of length information) and used a table to decode this length quickly as in [29].

4.2 Dictionary Encoding

Dictionary compression schemes are perhaps the most prevalent compression schemes found in databases today. These schemes replace frequent patterns with smaller codes for them. One example of such a scheme is the color-mapping given in the introduction. Other examples can be found in [13, 26, 11, 37].

We implemented a column-optimized version of dictionary encoding. All of the row-oriented dictionary schemes cited above have the limitation that they can only map attribute values from a single tuple to dictionary entries. This is because row-stores fundamentally are incapable of mixing attributes from more than one tuple in a single entry if other attributes of the tuples are not also included in the same entry (by definition of "row-store" – this statement does not hold for PAX-like [4] techniques that columnize blocks).

Our dictionary encoding algorithm first calculates the number of bits, X, needed to encode a single attribute of the column (which can be calculated directly from the number of unique values of the attribute). It then calculates how many of these X-bit encoded values can fit in 1, 2, 3, or 4 bytes. For example, if an attribute has 32 values, it can be encoded in 5 bits, so 1 of these values can fit in 1 byte, 3 in 2 bytes, 4 in 3 bytes, or 6 in 4 bytes. We choose one of these four options using the algorithm described in Section

4.2.1. Suppose that the 3-value/2-byte option was chosen. In that case, a mapping is created between every possible set of 3 5-bit values and the original 3 values. For example, if the value 1 is encoded by the 5 bits: 00000; the value 25 is encoded by the 5 bits: 00001; and the value 31 is encoded by the 5 bits 00010; then the dictionary would have the entry (read entries right-to-left)

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

where the X indicates an unused "wasted" bit. The decoding algorithm for this example is then straight-forward: read in 2-bytes and lookup entry in dictionary to get 3 values back at once. Our decision to keep data byte-aligned might be considered surprising in light of recent work that has shown that bit-shifting in the processor is relatively cheap. However our experiments show that column stores are so I/O efficient that even a small amount of compression is enough to make queries on that column become CPU-limited (Zukowski et. al observe a similar result [37]) so the I/O savings one obtains by not wasting the extra space are not important. Thus, we have found that it is worth byte-aligning dictionary entries to obtain even modest CPU savings.

4.2.1 Cache-Conscious Optimization

The decision as to whether values should be packed into 1, 2, 3, or 4 bytes is decided by requiring the dictionary to fit in the L2 cache. In the above example, we fit each entry into 2 bytes and the number of dictionary entries is $32^3 = 32768$. Therefore the size of the dictionary is 524288 bytes which is half of the L2 cache on our machine (1MB). Note that for cache sizes on current architectures, the 1 or 2 byte options will be used exclusively.

4.2.2 Parsing Into Single Values

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

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

These dictionary values in many cases can be operated on directly (as described in Section 5) and lazily decompressed at the top of the query-plan tree.

We chose not to use an order preserving dictionary encoding scheme such as ALM [7] or ZIL [33] since these schemes typically have variable-length dictionary entries and we prefer the performance advantages of having fixed length dictionary entries.

4.3 Run-length Encoding

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

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

4.4 Bit-Vector Encoding

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

1 1 3 2 2 3 1

would be represented as three bit-strings:

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

Since an extended version of this scheme can be used to index row-stores (so-called bit-map indices [23]), there has been much work on further compressing these bit-maps and the implications of this further compression on query performance [22, 5, 17, 31, 30, 32, 6]; however, the most recent work in this area [31, 32] indicates that one needs the bit-maps to be fairly sparse (on the order of 1 bit in 1000) in order for query performance to not be hindered by this further compression, and since we only use this scheme when the column cardinality is low, our bit-maps are relatively dense and we choose not to perform further compression.

4.5 Heavyweight Compression Schemes

Lempel-Ziv Encoding. Lempel-Ziv ([35, 36]) compression is the most widely used technique for lossless file compression. This is the algorithm upon which the UNIX command gzip is based. Lempel-Ziv takes variable sized patterns and replaces them with fixed length codes. This is in contrast to Huffman encoding which produces variable sized codes. Lempel-Ziv encoding does not require knowledge about pattern frequencies in advance; it builds the pattern table dynamically as it encodes the data. The basic idea is to parse the input sequence into non-overlapping blocks of different lengths while constructing a dictionary of blocks seen thus far. Subsequent appearances of these blocks are replaced by a pointer to an earlier occurrence of the same block. We refer the reader to [35, 36] for more details.

For our experiments, we used a freely available version of the Lempel-Ziv algorithm [2] that is optimized for decompression performance (we found it to be much faster than UNIX gzip).

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

5. COMPRESSED QUERY EXECUTION

In this section we describe how we integrate the compression schemes discussed above into the C-Store query execu-

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

Table 1: Compressed Block API

tor in a way that allows for direct operation on compressed data while minimizing the complexity of adding new compression algorithms to the system.

5.1 Query Executor Architecture

We extended C-Store to handle a variety of column compression techniques by adding two classes to the source code for each new compression technique. The first class encapsulates an intermediate representation for compressed data called a *compression block*. A compression block contains a buffer of the column data in compressed format and provides an API that allows the buffer to be accessed in several ways. Table 1 lists the salient methods of the compression block API.

The methods listed in the properties column of Table 1 will be discussed in Section 5.2 and are a way for operators to facilitate operating directly on compressed data instead of having to decompress and iterate through it. For the cases where decompression cannot be avoided, there are two ways to iterate through block data. First is through repeated use of the getNext() method which will progress through the compressed buffer, transiently decompressing the next value and returning that value along with the position (a position is the ordinal offset of a value in a column) that the value was located at in the original column. Second is through the asArray() method which decompresses the entire buffer and returns a pointer to an array of data in the uncompressed column type.

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

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

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

5.2 Compression-Aware Optimizations

We will show in Section 6 that there are clear performance advantages to operating directly on compressed data, but these advantages come at a cost: query executor complexity. Every time a new compression scheme is added to the system, all operators that operate directly on this type of data have to be supplemented to handle the new scheme. Without careful engineering, there would end up being n versions of each operator – one for each type of compression scheme that can be input to the operator. Operators that take two inputs (like joins) would need n^2 versions. This clearly causes the code to become very complex very quickly.

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

42		38		1	2	
36	$\bowtie$	42	$=$	3	2	
42		46		5	1	
44						
38						

An outline for the code for this operator is shown Figure 1 (assume that the join predicate is an equality predicate on one attribute from each relation).

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

So while many optimizations are possible if operators are allowed to work directly on compressed data, the example

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

Figure 1: Pseudocode for NLJoin

shows that the code becomes complex fairly quickly, since an if statement and an appropriate block of code is needed for each possible combination of compression types.

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

Hence, we enhanced each compression block with methods that make it possible for operators to determine the properties of each block of data. The properties we have added thus far are shown in the Properties column of Table 1. isOneValue() returns whether or not the block contains just one value (and many positions for that value). isValueSorted() returns whether or not the block's values are sorted (blocks with one value are trivially sorted). isPosContig() returns whether the block contains a consecutive subset of a column (i.e. for a given position range within a column, the block contains all values located in that range). Properties are usually fixed for each compression scheme but could in principle be set on a per-block basis by the DataSource operator.

The table below gives the value of these properties for various encoding schemes. Note that there are many variations of each scheme. For example, we experimented with three versions of dictionary encoding before settling on the one described in this paper; in one version there was a single dictionary entry per row value – i.e., a standard row-based dictionary scheme; another version was a pure column-based scheme that did not gracefully degenerate into single values as in the current scheme. In most cases, each variation of the same scheme will have the same block properties in the table below. A no/yes entry in the table indicates that the compression scheme is agnostic to the property and the value is determined by the data.

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

When an operator cannot operate on compressed data (if, for example, it cannot make any optimizations based on the block properties), it repeatedly accesses the block through an iterator, as described in Section 5.1. If, however, the operator can operate on compressed data, it can use the block information methods described in Section 5.1 to take shortcuts in operation. For example, the pseudocode for a Count aggregator is shown in Figure 2. Here, the passed in column is used for grouping (e.g., in a query of the form `SELECT c1, COUNT(*) FROM t GROUP BY c1`). (Note: this code is simplified from the actual aggregation code for ease of exposition).

```
Count(Column c1)
 b = GET NEXT COMPRESSED BLOCK FROM c1
 WHILE b IS NOT NULL
   IF b.isOneValue()
     x = FETCH CURRENT COUNT FOR b.getStartVal()
     x = x + b.getSize()
   ELSE
     a = b.asArray()
     FOR EACH ELEMENT i IN a
       x = FETCH CURRENT COUNT FOR i
       x = x + 1
   b = GET NEXT COMPRESSED BLOCK FROM c1
```

Figure 2: Pseudocode for Simple Count Aggregation

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

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

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

6. EXPERIMENTAL RESULTS

We ran experiments on our extended version of the C-Store system with two primary goals. First, we wanted to

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

Figure 3: Optimizations on Compressed Data

identify situations in which the encoding types described in Section 4 perform well. Second we wanted to demonstrate the benefits of operating directly on compressed data.

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

We begin by presenting results from a simple aggregation query on a single column of data encoded with each of the six encoding schemes described in Section 4. We used generated data so that we could carefully vary the data characteristics. We ran three variations of this experiment. In the first variation, we required the column to be decompressed as it was brought off disk. In the second variation, we lazily decompressed the data and allowed operators to apply optimizations to compressed data. In the third variation, queries ran with competition for CPU cycles. In these experiments, we observe that the number of distinct values and sorted run lengths are the primary determinant of query performance; we use these metrics to predict performance on TPC-H data.

We also present results from more complicated queries to further illustrate the benefits of different compression schemes and the interaction of these schemes with each other in multi-column queries. Section 7 summarizes our results.

6.1 Eager Decompression

In this experiment, we ran a simple aggregation on a single column of data encoded with each of the six encoding schemes described in Section 4. We ran on generated data and required that the column be decompressed as it was brought off disk. The query that we ran was simply:

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

The column that we are aggregating has 100 million 32-bit integer values. Since most columns in C-Store projections have some kind of order (see section 3), we assume sorted runs of size X (we vary X). For example, if column C is tertiarily sorted and the first column in the projection has 500 unique values and the second column in the projection has 1000 unique values then C will have average sorted runs of size 100000000/(500*1000)=200. If C itself has 10 unique values, then within each of these sorted runs, each value would appear 20 times. Since bit-vector compression is only designed to be able to run on columns with few distinct values, in our first set of experiments, we allowed the number of distinct values in C to vary between 2 and 40 (so that we could directly compare all the introduced compression techniques). Also, in most data-warehousing environments, there are a large number of columns with few distinct values; for example, in the TPC-H lineitem fact table, 25% of the columns have fewer than 50 distinct values. We experiment with columns with a higher number of distinct values in Section 6.3.

We experimented with 4 sorted run lengths in C: 50, 100, 500, and 1000. We compressed the data in each of the following six ways: Null suppression, Lempel-Ziv, RLE, bit-vector, dictionary, and no compression. The sizes of the compressed columns are shown in Figures 4(a) and 4(b) for different cardinalities of C (here, we use *cardinality* to mean the number of distinct values). We omit the plots for the 100 and 500 sorted runs cases as they follow the trends observed in Figure 4. In these experiments, dictionary and LZ compression consistently get the highest compression ratios, with RLE also performing well for low-cardinalities (this is because RLE performs better with large runs of repeated values and the average run-length of a point on these graphs can be calculated directly by dividing the sorted run-length by the number of unique values). Interestingly, dictionary does a slightly better job compressing the data than the heavy-weight LZ scheme at low column cardinalities. The compression ratio for bit-vector is linear in the number of unique values in the column. Since we do not further compress the bit-vectors, as soon as the column cardinality is more than 32, type-2 compression is no longer more compressed than the original 32-bit data.

The performance of the aggregation query on these same compressed columns is shown in Figures 5(a) and 5(b) (again we do not show the plots for sorted runs of 100 and 500 since we have limited space and they follow the trends between these two graphs).

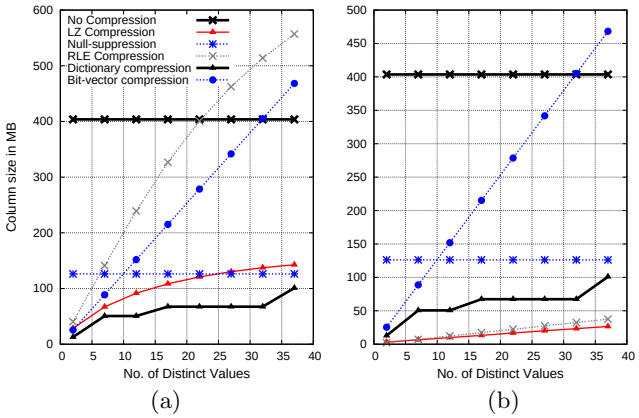

Figure 4: Compressed column sizes for varied compression schemes on column with sorted runs of size 50 (a) and 1000 (b)

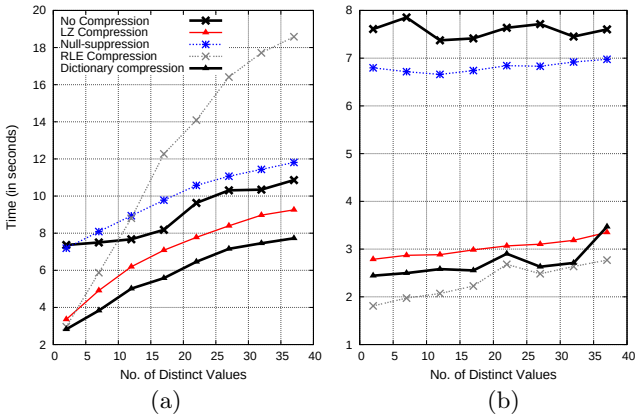

Figure 5: Query Performance With Eager Decompression on column with sorted runs of size 50 (a) and 1000 (b)

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

Bit-vector encoding was by far the slowest decompression scheme. To completely decompress a bit-vector encoded column, one must read in parallel and merge each bit-vector (one for each distinct value in the column). RLE and NS performed worse than dictionary and LZ (though RLE performed better as the average run-length of the column improved). This can be attributed to the fact that RLE and NS require if-then-else statements in the decompression code which makes loop pipelining difficult and results in code that does not take advantage of the super-scalar properties of modern CPUs (this was also observed in Monet DB/X100 [37]).

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

6.2 Operating Directly on Compressed Data

We ran the same experiments as in the previous section, without eager decompression. Operators were allowed to operate directly on compressed data. Since LZ and NS cannot operate on encoded data, their performance for these experiments was identical (and we omit them from some of our graphs). However, since there are two alternative ways for operating directly on our dictionary compression scheme for this aggregation query, there are two lines on each graph corresponding to dictionary compression. The first approach, called *dictionary single-value*, simply extracts each individual dictionary symbol from a 32-bit dictionary-compressed record (as described in section 4.2.2), performs a count group-by aggregation on these symbols, then decompresses each symbol to its original value and multiplies this original value by the counts to get a sum. For example, if value 2 maps to symbol 000, value 4 maps to 001, and value 8 maps to 002 and the aggregator receives the following input:

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

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

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

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

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

In addition to the dictionary optimizations, the aggregator also operates directly on RLE and bit-vector data as described in Section 5.2. The results are shown in Figures 6(a) and 6(b). We see that substantial performance gains are possible when data is not eagerly decompressed. On the data with 1000-record sorted runs, the performance improvement for RLE was 3.3X on average, for bit-vector it was 10.3X, and for dictionary it was 3.94X and 1.1X with and without the group-by-self optimization respectively.

To show the importance of operating directly on compressed data, we reran the same experiments with contention for CPU cycles (this was done by running C-Store at the same time as another process that infinitely accessed, processed, and wrote data to a large array). The bar graph in Figure 6(c) shows the average increase in query time caused by CPU contention compared with the results in Figures 6(a) and (b) for each compression technique.

We reran the experiment with performance counters to find out whether the contention was for CPU cycles or for cache lines and found that the competition for cache lines accounted for less than 2% of the increase in query time. Thus contention for CPU cycles is the dominant reason for the increase in query time, and queries that were CPU limited take longer.

NS and LZ perform the worst (relative to their previous values) since the aggregator does not operate directly on this data. Similarly for RLE (for small average run lengths) and the value-at-a-time dictionary scheme (although the dictionary data does not need to be completely decompressed, the aggregator must still iterate through all values and dictionary entries must be bit-shifted into integers). However, for the schemes on which the aggregator can take shortcuts, the performance hit of CPU contention is significantly smaller. This is because the column-oriented nature of these schemes allow the aggregator to aggregate multiple values at once; the CPU cost of the aggregation is proportional to n,

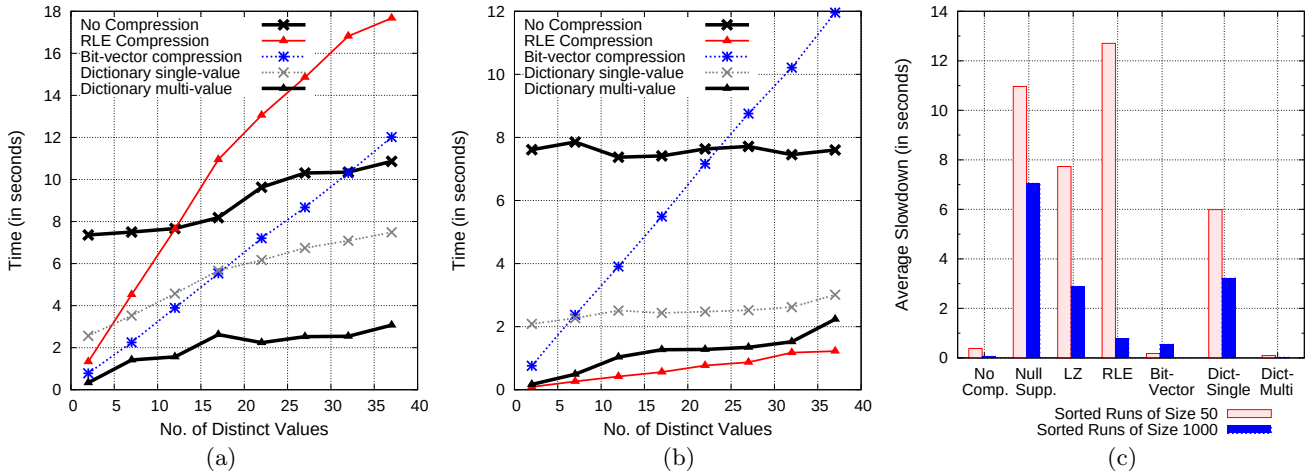

Figure 6: Query performance with direct operation on compressed data on column with sorted runs of size 50 (a) and 1000 (b). Figure (c) shows the average increase in query time relative to the query times in (a) and (b) when contention for CPU cycles is introduced.

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

6.3 Higher column cardinalities

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

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

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

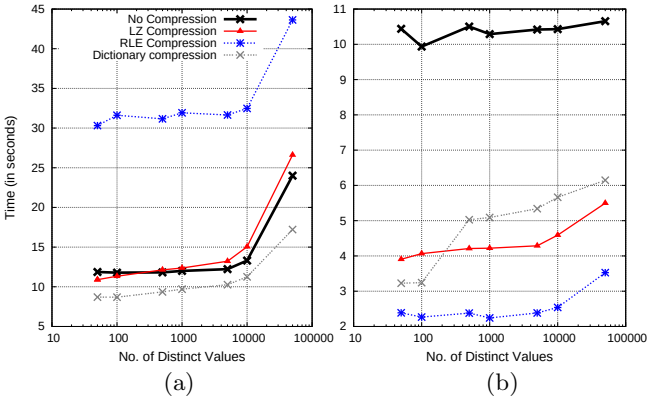

Figure 7: Aggregation Query on High Cardinality Data with Avg. Run Lengths of 1 (a) and 14 (b)

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

This table shows that for RLE and LZ, run-length is a better indicator of performance than cardinality. As soon as the data has moderate sized runs, performance improves dramatically. This correlation between run-length and performance is less significant for the latter three techniques. As explained in Section 6.1, all techniques see some improvement with longer run-lengths.

6.4 Generated vs. TPC-H Data

To verify that our results on our generated data set match the results on more general data sets, we compared our query performance on our generated data to query performance on TPC-H data. For this set of experiments, we used the shipdate, supplier key, extended price, linenumber, quantity,

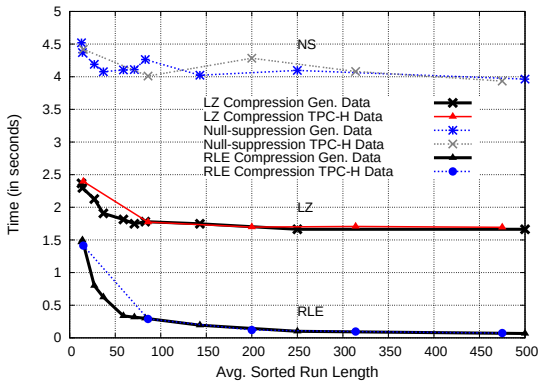

Figure 8: Comparison of query performance on TPC-H and generated data

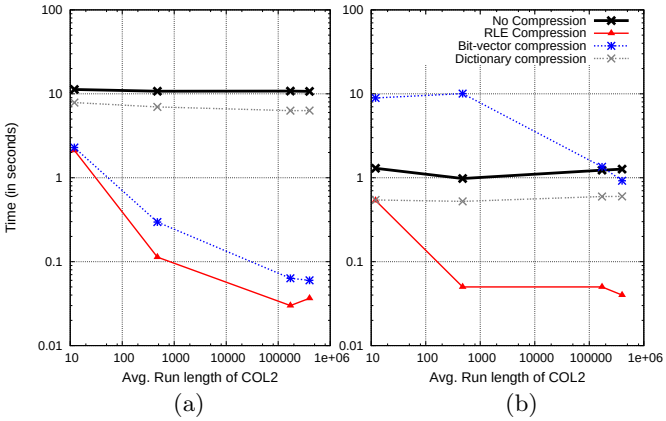

Figure 9: (a) Predicate on the variably compressed column, position filter on the RLE column and (b) Predicate on the RLE column, position filter on the variably compressed column. Note log-log scale.

extended price, and return flag columns from the TPC-H lineitem fact table and created the following projections:

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

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

6.5 Other Query Types

In this section we experiment with different types of queries to observe how compressing one column affects access to other columns in a query and also to observe further advantages of operating directly on compressed data.

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

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

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

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

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

In the next experiment we ran the same query; however, we switched the role of the two columns in the query. So

now the predicate is on COL1 and we position filter COL2 (which is again encoded using the same four compression techniques as in the previous query). The results of this experiment are shown in Figure 9(b). Bit-vector performs much more poorly (note the log scale). This is because the query requires the values of the bit-vector column in position order which forces decompression which has already been shown to be slow (at very high run-lengths bit-vector encoding starts to see entire pages of '1's and '0's which causes it to optimize its operation, which is why it starts to perform well in the final two points in the graph). This difference in performance between Figures 9(a) and 9(b) illustrates that the proper choice of encoding type for a column depends not just on data characteristics, but also on the expected query workload. This observation supports a major future research goal of exploring the interaction between physical database design, optimization, and compression. It also indicates that redundantly storing the same column in the same sort order using different compression schemes might be a good idea.

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

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

The algorithm for performing joins in C-Store was described in Section 5.1. Assume for this query that CSTORE_P1 is a projection from the fact table and that CSTORE_P2 is a projection from a dimension table that contains its primary key (which is the common join case in star schema queries). Hence, L.COL2 is a foreign key into CSTORE_P2 (S.COL1 is the key). This query applies a predicate to each table before the join, does a foreign-primary key join, and then uses the position list result from the join to filter and aggregate a column from CSTORE_P2.

Again, we started with CSTORE_P1 being the lineitem fact table from TPC-H. The join attribute is the supplier foreign key. We assume the projections are sorted on S.COL2 and L.COL1 (this is the common case since the C-Store optimizer will have a choice as to what projections to use for a query and will choose projections that are sorted by predicated columns) and are therefore RLE encoded. We allowed L.COL2 (suppkey) to be secondarily sorted and encoded it with the same four coding algorithms as the previous (select) queries. In order to show results for the bit-vector case, we reduced the number of unique supplier keys in the fact table to just 50 values in one of our experiments (we allowed 50000 values in the other experiment). The results of performing this join are shown in the table below (times are in seconds).

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

The techniques for operating directly on RLE and bit-vector data have been discussed previously, for the join part of this query in Section 5.1 and for the resulting position filtering in the previous query in this section. To operate directly on dictionary data, the dimension table join column had to be recoded using the fact table's dictionary at the beginning of the query (this is included in the query time.)

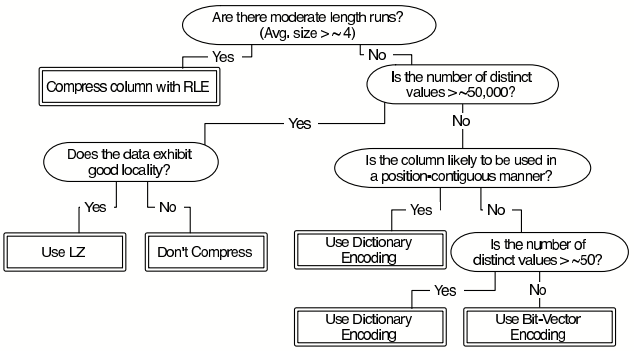

Figure 10: Decision tree summarizing our results regarding the proper selection of compression scheme.

7. CONCLUSION

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

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

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

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

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

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

In summary, this paper shows that significant database performance gains can be had by implementing light-weight compression schemes and operators that work directly on compressed data. By classifying compression schemes according to a set of basic properties, we were able to extend C-Store to perform this direct operation without requiring

new operator code for each compression scheme. Furthermore, our focus on column-oriented compression allowed us to demonstrate that the performance benefits of operating directly on compressed data in column-oriented schemes is much greater than the benefit in operating directly on row-oriented schemes. Hence, we see this work as an important step in understanding the substantial performance benefits of column-oriented database designs.

8. ACKNOWLEDEMENTS & REFERENCES

We would like to thank Michael Stonebraker, David DeWitt, Pat O'Neil, Stavros Harizopoulos, and Alex Rasin for their helpful feedback and ideas.

This work was supported by the National Science Foundation under NSF Grant number IIS-0325525 and by an NSF Graduate Research Fellowship.

[1] `http://www.addamark.com/products/sls.htm`.

[2] `http://www.lzop.org`.

[3] C-Store code release under bsd license. `http://db.csail.mit.edu/projects/cstore/`, 2005.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[5] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *VLDB*, pages 329–338, 2000.

[6] G. Antoshenkov. Byte-aligned data compression. U.S. Patent Number 5,363,098.

[7] G. Antoshenkov, D. B. Lomet, and J. Murray. Order preserving compression. In *ICDE '96*, pages 655–663. IEEE Computer Society, 1996.

[8] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[9] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal: Very Large Data Bases*, 8(2):101–119, 1999.

[10] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[11] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD '01*, pages 271–282, 2001.

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

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

[14] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE '98*, pages 370–379, 1998.

[15] D. Huffman. A method for the construction of minimum-redundancy codes. Proc. IRE, 40(9):1098-1101, September 1952.

[16] B. R. Iyer and D. Wilhite. Data compression support in databases. In *VLDB '94*, pages 695–704, 1994.

[17] T. Johnson. Performance measurements of compressed bitmap indices. In *VLDB*, pages 278–289, 1999.

[18] S. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *ICDE*, pages 636–643. IEEE Computer Society, 1987.

[19] Kx Sytems, Inc. Faster database platforms for the real-time enterprise: How to get the speed you need to break through business intelligence bottlenecks in financial institutions. `http://library.theserverside.com/data/document.do?res_id=1072792428_967`, 2003.

[20] C. A. Lynch and E. B. Brownrigg. Application of data compression to a large bibliographic data base. In *VLDB '81, Cannes, France*, pages 435–447, 1981.

[21] R. MacNicol and B. French. Sybase IQ multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.

[22] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1):1–9, 1994.

[23] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[24] R. Ramamurthy, D. Dewitt, and Q. Su. A case for fractured mirrors. In *VLDB*, pages 89 – 101, 2002.

[25] G. Ray, J. R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, 1995.

[26] M. A. Roth and S. J. V. Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, 1993.

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

[28] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[29] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, 2000.

[30] K. Wu, E. Otoo, and A. Shoshani. Compressed bitmap indices for efficient query processing. Technical Report LBNL-47807, 2001.

[31] K. Wu, E. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In SSDBM'02, pages 99–108, 2002. LBNL-49627., 2002.

[32] K. Wu, E. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, 2001.

[33] A. Zandi, B. R. Iyer, and G. G. Langdon Jr. Sort order preserving data compression for extended alphabets. In *Data Compression Conference*, pages 330–339, 1993.

[34] J. Zhou and K. Ross. A multi-resolution block storage model for database design. In *Proceedings of the 2003 IDEAS Conference*, pages 22–33, 2003.

[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[36] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[37] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.