Read-Optimized Databases, In Depth

Allison L. Holloway and David J. DeWitt
University of Wisconsin – Madison
Computer Sciences Department
{ahollowa, dewitt}@cs.wisc.edu

ABSTRACT

Recently, a number of papers have been published showing the benefits of column stores over row stores. However, the research comparing the two in an "apples-to-apples" way has left a number of unresolved questions. In this paper, we first discuss the factors that can affect the relative performance of each paradigm. Then, we choose points within each of the factors to study further. Our study examines five tables with various characteristics and different query workloads in order to obtain a greater understanding and quantification of the relative performance of column stores and row stores. We then add materialized views to the analysis and see how much they can help the performance of row stores. Finally, we examine the performance of hash join operations in column stores and row stores.

1. INTRODUCTION

Recently there has been renewed interest in storing relational tables "vertically" on disk, in columns, instead of "horizontally", in rows. This interest stems from the changing hardware landscape, where processors have increased in speed at a much faster rate than disks, making disk bandwidth a more valuable resource.

The main advantage to vertical storage of tables is the decreased I/O demand, since I/O is an increasingly scarce resource and most queries do not reference all the columns of a table. Vertical data storage has other advantages, such as better cache behavior [8, 18] and reduced storage overhead. Some argue that column stores also compress better than row stores, enabling the columns to be stored in multiple sort orders and projections on disk for the same amount of space as a row store, which can further improve performance [18].

A vertical storage scheme for relational tables does come with some disadvantages, the main one being that the cost of stitching columns back together can offset the I/O benefits, potentially causing a longer response time than the same query on a row store. Inserting new rows or deleting rows when a table is stored vertically can also take longer. First, all the column files must be opened. Second, unless consecutive rows are deleted, each delete will incur a disk seek. Updates have similar problems: each attribute value that is modified will require a seek.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

Traditional row stores store the tuples on slotted pages [15], where each page has a slot array that specifies the offset of the tuple on the page. The advantages to this paradigm are that updates are easy, and queries that use most or all of the columns are fast. The disadvantage is that, since most queries do not use all columns, there can be a substantial amount of "wasted" I/O bandwidth. Slotted pages also result in poor cache behavior [6] and are less compressible, since each tuple is stored individually and has its own header.

However, several recent studies [16, 13] have demonstrated that row stores can be very tightly compressed if the tuples are stored dense-packed on the page without using slot arrays. While this format causes row stores to lose their advantage of easy updatability, the change can save substantial amounts of I/O bandwidth, which is where they often lose compared to column stores. [16, 13] also examine skewed datasets in row stores and use advanced compression techniques, such as canonical Huffman encoding [14], to achieve a degree of row store compression very close to that of the entropy for the table.

The confluence of these two ideas, column stores and advanced compression in row stores, brings us to the central question of this paper: How do the scan times for row and column stores compare when both tables are as tightly compressed as possible? In other words, can row stores compete with column stores for decision-support workloads that are dominated by read-only queries?

"Performance Tradeoffs in Read-Optimized Databases," was a first step toward answering this question [12]. This paper provides an initial performance comparison of row stores and column stores, using an optimized, low-overhead shared code base. Additionally, the tuples for both storage paradigms are stored dense-packed on disk, a necessity for truly obtaining an apples-to-apples performance comparison.

[12] studies both wide and narrow tables using the uniformly distributed TPC-H [19] *Lineitems* and *Orders* tables. The two formats are compared without compression for *Lineitems*, and both compressed and uncompressed for *Orders*. The *Orders* table is compressed using bit-packing, dictionary encoding and delta coding, where appropriate. All queries are of the form: "Select column_1, column_2, column_3, … from TABLE where predicate(column_1)" with a selectivity of 0.1% or 10%.

The results from [12] that are most relevant to this paper are that:

- At 10% selectivity, when half of the columns are returned, a row store will be faster than a column store only if the tuples are less than 20 bytes wide and the system is CPU-constrained.
- The CPU time required for a column store to execute a scan query is very sensitive to the number of columns returned and the selectivity factor of the query.
- The time required for a row store to process a scan query is relatively insensitive to the number of columns being returned or the selectivity of the predicate.
- Column stores can be sensitive to the amount of processing needed to decompress a column.

However, these results left a number of questions unresolved. First, the attribute values in each column of the TPC-H tables are uniformly distributed. The results presented in [16, 13] demonstrate that, if the distribution of values in a column is skewed, even higher degrees of compression can be obtained. Thus, in this paper we explore non-uniform value distributions and the impact of advanced compression techniques.

Additionally, [12] considered only a very limited set of queries. First, no queries with selectivity factors greater than 10% were studied. Also, all of the queries considered in [12] consisted of a single predicate that was always applied to the first column of the table and only the left-most columns of a table were returned. We believe that the relative performance of column and row stores is affected by (a) how many predicates the query has, (b) what columns the predicates are applied to, and (c) which columns are returned. A key focus of this paper is to explore each of these issues.

Finally, we noticed that the row store performed very poorly compared to the column store for the *Lineitem* table, which has a long string attribute. While the string does not impair the performance of a column store if it is not used in the query, it disproportionately hurts the row store since that one attribute is the same size as the sum of the others. We hypothesized that if the string had been stored in a separate table, the performance of the two paradigms would have been more comparable. Hence, our study will also include materialized views of the row store.

While [12] was a first step at an "apples-to-apples" comparison of row stores and column stores in a common software framework, we want to achieve a greater understanding of the issues raised above. The aim of this work is to systematically study scans for compressed row and column stores. Specifically, we explore a large space of possible tables, compression methods, and queries, in which the results of [12] would fall as points in this space. This paper's contributions are to:

- Provide a more in-depth discussion of the factors that affect the scan times for read-optimized row stores and column stores, and to choose points within those factors to study further.
- Quantify the effect of (a) more qualifying tuples, (b) additional predicates, and (c) tables that do not have uniform data distributions.
- Compare the performance of materialized views with the performance of column stores and row stores in this framework.
- Examine the effect a hybrid hash join has on column stores and row stores, when using early materialization of tuples in the join.

Section 2 discusses other related work. Sections 3 and 4 discuss the overall search space and how points within it were selected for evaluation. The remaining sections provide implementation details and results. We conclude in Section 7 with a short discussion of our work and how the results might be applied.

2. RELATED WORK

2.1 Vertical Storage

While this paper builds on [12], two of the authors of [12] have also recently published a follow-up [4] that compares C-Store, a column store, with a commercial row store running the Star Schema Benchmark. The authors implement a column store using a commercial relational database system by making each column its own table; the performance of this design is quite bad since every column must have its own row id. They also implement a row store within C-Store by storing the entire row as if it were a single column and evaluate three optimizations said to provide column stores superior performance: compression, late materialization and block-iteration. The workload is first run with all the optimizations. Then, each optimization is removed to detemine how much performance it contributes. They find that compression improves performance by a factor of two and late materialization improves performance by a factor of three. We believe these results are largely orthogonal to ours, since we heavily compress both the row store and column store and our workload does not lend itself to late materialization of tuples.

"Comparison of Row Stores and Column Stores in a Common Framework" also directly compares row stores and column stores [10]. Two contributions of this paper are the idea of a *super-tuple* and *column abstraction*. The authors note that one reason row stores do not traditionally compress as well as column stores is the use of the slotted-page format. The paper introduces the idea of *super-tuples*, which store all rows on a page with just one header, instead of one header per tuple and no slot-array. *Column-abstraction* avoids storing repeated attributes multiple times by adding information to the header. The paper compares results for 4-, 8-, 16-, and 32- column tables, however, it focuses on uniformly distributed data and examines trends within column stores, row stores and super-tuples when returning all columns and all rows. While those results are interesting, we are more interested in looking at the general case where a query can return any number of columns.

One of the first vertical storage models was the decomposition storage model (DSM) [9], which stored each column of a table as pairs of (tuple id, attribute values). Newer vertical storage systems include the MonetDB/X100 [8] and C-Store [18] systems. MonetDB operates on the columns as vectors in memory. C-Store differs from DSM in that it does not explicitly store the tuple id with the column. Another novel storage paradigm is PAX [7], which stores tuples column-wise on each disk page. This results in better L2 data cache behavior, but it does not reduce the I/O bandwidth required to process a query. Data Morphing further improves on PAX to give even better cache performance by dynamically adapting attribute groupings on the page [11].

2.2 Database Compression Techniques

Most research on compression in databases has assumed slotted-page row stores. Two papers that have not made this assumption, [13, 16], were mentioned above. Both papers also used more

processing-intensive techniques to achieve compression factors of up to 8-12 on row stores. "Superscalar RAM-CPU Cache Compression" [21] and "Integrating Compression and Execution in Column-Oriented Database Systems" [3] examine compression in column stores. Zukowski finds that compression can benefit both row stores and column stores in I/O-restricted systems and presents algorithms to optimize the effective use of modern processors. [3] considers various forms of compression and the effects of run lengths on the degree of compression achieved. The main conclusion related to this study is that RLE and dictionary encoding tend to be best for column stores.

3. FACTORS AFFECTING PERFORMANCE

In this paper, we consider four factors that can have a significant effect on the relative performance of row stores and columns stores for a given system:

1) The width of a table (i.e. number of columns), the cardinality of each column (i.e. the number of unique attribute values in the column), and the distribution of these values (uniform or skewed).
2) The compression techniques employed.
3) The query being executed, including the number of predicates and which columns the query returns.
4) The storage format (i.e. slotted pages or super-tuples).

These four factors combine to produce a very large search space. In the following sections, we describe each factor in additional detail. In Section 4, we explain how the search space was pruned in order to make our evaluation feasible while still obtaining a representative set of results.

3.1 Data Characteristics

Each relational table can have a range of characteristics. First, and foremost, the number of columns and rows in a table directly affects the time required to scan a table. Second, the type of each attribute column (i.e. VAR CHAR or INT) determines how the data would be stored when uncompressed, and the distribution of values within the column (such as uniform or skewed) affects how well the column's values can be compressed. The number of unique values stored within the column, or column cardinality, also affects the column's compressibility.

To illustrate, imagine a table with a single column of type INT which can only assume one of the four values: 1, 2, 3, or 4. The column cardinality for this column is 4. If the values are distributed equally within the column (i.e., each value has the same probability of occurring), then the column's distribution is termed uniform, otherwise, the distribution is termed skewed. For a row-store with a slotted page format, this column would most likely be stored as a 32-bit integer. However, only two bits are actually needed to represent the four possible values. The distribution of the values and whether or not the table is sorted on this column will determine if the column can be stored in, on average, less than two bits per row by using compression.

3.2 Compression

Many different types of compression exist in the literature, from light-weight dictionary coding, to heavy-weight Huffman coding. Since a full evaluation of compression techniques is outside the scope of this paper, based on the results in [13], we focus on those techniques that seem to generally be most cost-effective. These techniques are bit-packing, dictionary coding, delta coding, and run-length encoding (RLE).

As demonstrated in the example above, bit packing uses the minimum number of bits to store all the values in a column; for example, two bits per attribute value are used instead of the 32 normally needed for an INT. This space savings means that bit-packed attributes (whether in a column or row) will not generally be aligned on a word boundary. With the current generation of CPUs, that cost has been found to be negligible [3, 13].

Dictionary coding is another intuitive compression technique in which each attribute value is replaced with an index into a separate dictionary. For example, the months of the year ("January" to "December") could be replaced with the values 1 to 12, respectively, along with a 12-entry dictionary to translate the month number back to a name. Although dictionaries are often useful, they can sometimes hurt performance, for instance, if they do not fit into the processor's L2 data cache. Dictionaries should also not be used if the index into the dictionary would be bigger than the value it is replacing, or if the size of the un-encoded column is smaller than the size of the encoded column plus the size of the dictionary.

Delta coding stores the difference between the current value and the previous value. To delta encode a sequence, first the sequence is sorted and the lowest value is stored. Then, each difference should be stored in two parts: the difference itself starting at the first '1' in the bit representation of the difference, and the number of leading zeroes. For instance, consider the sequence 24, 25, 27, 32. The bit representation for each is 011000, 011001, 011011, 100000, respectively. The differences between subsequent values in the sequence are 000001, 000010, and 000101. Thus, the sequence to encode would be 24, (5, "1"), (4,"10"), (3,"101"), which would be 011000 101 1 100 10 011 101. The number of leading zeroes should be encoded as a fixed-width field, but the difference will generally be variable length. To simplify decoding, the number of leading zeroes should be stored before the difference. Delta coding can be performed both at a column level, and at a tuple level, provided that each column of the tuple has been bit packed. Delta coding should not be used on unsorted sequences or when the encoding for the number of leading zeroes plus the average size of the differences in bits is bigger than the non-delta-coded value.

Run-length encoding (RLE) transforms a sequence into a vector of <value, number of consecutive occurrences (runs)> pairs. For instance, the sequence 1, 1, 1, 1, 2, 2, 4, 4, 4 would become <1,4>, <2,2>, <4,3>. RLE compresses best if the sequence has first been sorted, and if there are many long runs.

Bit packing, dictionary coding and run-length encoding are all light-weight compression schemes. They can lead to substantial space and I/O savings while incurring very little extra processing. Delta coding can provide extremely good compression for the right type of data, but requires much more processing to decode [13].

Generally, the row store and column store versions of the same table should be compressed using different techniques. For

instance, RLE is often a good scheme for column stores but not for row stores, since it is rare to have multiple consecutive rows that are identical.

3.3 Query Characteristics

The third factor we considered was the characteristics of the queries to be used for evaluating the relative performance of the column store and row store configuration, including the number of predicates and the columns to which the predicates are applied, the number of output columns and which ones, and the selectivity of the selection predicate(s). To simplify the search space somewhat, we primarily considered scan queries. Row stores can benefit from indexes, but clustered indexes should have very similar behavior to sequentially scanning a subset of the table. Additionally, column stores will benefit from queries with aggregation, but this issue has been studied extensively by others [5], and we wanted to keep our queries similar to those in [12].

Each query materializes its output as un-encoded row-store tuples, but these tuples are not written back to disk. The number of predicates and the number of output columns affect the number of CPU cycles needed for the query. In this paper, a predicate is a Boolean expression of the form "Col_i >= Value." For queries with more than one predicate, predicates are "anded" together.

The type of each column can also have a significant effect on scan time, since different types can consume varying amounts of CPU and I/O time. A column of CHARs will take four times less I/O than a column of INTs, and a well-compressed RLE column of CHARs will take even less, while scanning a delta-coded column may consume a substantial amount of CPU time.

The output selectivity (i.e. number of tuples returned), also impacts the scan time since materializing result tuples requires, at the very least a copy.

3.4 Storage

The two main paradigms used by the current generation of commercial database system products for table storage are row stores with slotted pages and dense-packed column stores. However, there are other possibilities, such as PAX [7], DMG [11], and column abstraction [10]. Column abstraction tries to avoid storing duplicate leading attribute values multiple times. For instance, if four tuples in a row have '1' in their first attribute, that knowledge is encoded in the first header, and the subsequent tuples do not store the '1.' We will refer to dense-packed row stores as "super-tuples" [10].

4. NARROWING THE SEARCH SPACE AND GENERATING THE TABLES

Once the overall parameters were identified, they had to be narrowed down to a representative and insightful set. The parameter space can be viewed as two sets: the set of tables (affected by the data characteristics, storage format, and compression) and the set of queries.

4.1 Table Parameters

We elected not to use TPC-H tables since we wanted more control over all aspects of the tables. Instead, we devised five tables:

- **Narrow-E:** This table has ten integer columns and 60 million rows. The values in column i are drawn uniformly and at random from the range of values [1 to $2^{(2.7 * i)}$]: column 1 from the values [1 to 6]; column two from [1 to 42]; column three from [1 to 274]; column four from [1 to 1,782]; column five from [1 to 11,585]; column six from [1 to 75,281]; column seven from [1 to 489,178]; column eight from [1 to 3,178,688]; column nine from [1 to 20,655,176]; and column ten from [1 to 134,217,728].
- **Narrow-S:** This table is similar to the table in Narrow-E (ten integer columns and 60 million rows) but the values in each column are drawn from a Zipfian distribution with an exponent of 2 instead of uniformly at random from the given ranges.
- **Wide-E**: This table is similar to Narrow-E, except it has 50 columns instead of 10 and 12 million rows instead of 60 million. Thus, the uncompressed sizes of the two tables are the same. The values for column i are drawn uniformly and at random from the range [1 to $2^{4+(23/50*i)}$].
- **Narrow-U**: This table has ten integer columns and 60 million rows. The values in each column are drawn uniformly and at random from the range [1-100 million].
- **Strings:** This table has ten 20-byte string columns and 12 million rows. No compression is used on this table.

We initially studied a wider range of tables, but we found that these tables provide the most interesting and, in our opinion, representative results.

We limited our study to predominantly integer columns, since they are common and generalize nicely. However, we did decide to study one table comprised solely of strings. Strings tend not to compress as well as integers and at the same time are wider than integers, so they are an important point to study.

The tables are stored on disk in either row store or column store format, with both using super-tuples and no slot array. This storage method results in read-optimized tuples.

Since there are so many different compression techniques, it would be impossible to implement and test all of them in a timely manner. Thus, this study compresses each table once for each storage format (row and column) using the compression technique that optimizes the I/O and processor trade-offs. While Huffman encoding could provide better compression for the Narrow-S table, it often results in worse overall performance due to the extra CPU cycles it consumes decoding attribute values. How the tables are generated and compressed is presented in the next section.

4.2 Table Generation and Compression

First, an uncompressed row-store version of each of the five tables was generated. Then, each table was sorted on all columns starting with the left-most column in order to maximize the run lengths of each column. From this table, projections of each column were taken to form the column-store version of the table. The value of each attribute of each column is a function of a randomly generated value, the column cardinality, and the column's distribution. For the table Narrow-S, a traditional bucketed implementation of Zipfian took prohibitively long with column cardinalities greater than 1000 so a faster algorithm that produces an approximation of Zipfian results was used [1].

The compression technique(s) used for each configuration of each table is shown in Table 1. Dictionaries are "per page". That is, the dictionary for the attribute values (whether organized as rows or columns) on a page is stored on the same page.

Traditionally, RLE uses two integers: one for the value, and one for the run length. However, this can easily result in sub-optimal compression, particularly when the value range for the column does not need all 32 bits to represent it. As a further optimization, we store each RLE pair as a single 32-bit integer, with *n* bits used for the value and 32-*n* bits for the run length. For example, consider the last column of Narrow-E, whose values range from 1 to 134 million. In this case, 27 bits are needed for the value, leaving 5 bits to encode the run length. If the length of a run is longer than 31, the entire run cannot be encoded into one entry (32 bits), so multiple entries are used. This optimization allows us to store the RLE columns in half the space.

For the table Narrow-S, columns 7-10 are dictionary encoded for the row store. Each column is individually dictionary-encoded, but, in total, there can be no more than 2,048 (2^{11}) dictionary entries shared amongst the four columns. This requirement means that each of the attribute values is encoded to 11 bits, but allows the number of entries for each individual column's dictionary to vary from page to page.

To find a reasonable dictionary size, we estimated how many compressed tuples, t, would fit on a page, then we scanned the table to calculate the dictionary size needed to fit t tuples per page. The reasoning behind this approach is that the page size must be greater than or equal to the size of the dictionary and all of the tuples on that page. A 2,048 entry dictionary is 8KB, so for a 32KB page, there is room for 24KB of data. So, if the data are expected to compress to 8 bytes per tuple. (assuming each dictionary encoded column is 11 bits), 3K tuples will fit on the page. If, after scanning the data, it is found that a 2,048 entry dictionary would be too small, the dictionary size should be doubled and the analysis redone. For the best dictionary compression possible, analysis would be performed for every page, but we were satisfied with the compression we obtained using this simpler method.

For the column store version of Narrow-S, columns 7-10 are also dictionary encoded. Since each column is stored in its own file, the dictionary does not need to be shared. However, a larger dictionary is needed – in this case, we allow a maximum of 4,096 dictionary entries per page, which encodes to 12 bits per value.

4.3 Query Parameters

In order to understand the base effects of the different tables and queries, most of the queries we tested have one predicate. However, a few have three predicates in order to study the effect of queries with multiple predicates on response time. Each query is evaluated as a function of the number of columns included in the output table: 1, 25% of the columns, 50% of the columns, and all columns. The selectivity factor is varied from 0.1% to 50%. Runtimes for different selectivity factors and number of columns returned can be interpolated between these points.

Table 1. Type of compression used for each table.

Table	Row Store Compression	Column Store Compression
Narrow-E	All columns bit-packed Columns 1-5 delta encoded	Columns 1-3 RLE, Columns 4-10 bit-packed
Narrow-S	Columns 1-6 bit packed Columns 7-10 dictionary encoded	Columns 1-6 RLE, Columns 7-10 dictionary encoded
Wide-E	All columns bit-packed Columns 1-8 delta encoded	Columns 1-4 RLE, Columns 5-50 bit-packed
Narrow-U	All columns bit-packed	All columns bit-packed

4.4 Query Generation

The query generator takes four inputs: the schema definition for the input table (Narrow-E, Narrow-S, Narrow-U, Wide-E, Strings), the desired selectivity factor, the total number of columns referenced, *c*, and the number of predicates, *p*. Since each column has different processing and I/O requirements, which columns are used, and the order in which they are processed, affects the execution time of logically equivalent queries. Thus, for each configuration of input parameters, the query generator randomly selects the columns used to produce a set of "equivalent" queries. This differs from [12], where every query returned the columns in the same order, and the single predicate was always applied to the first column.

The query generator first randomly picks *c* columns to include in the output table. The first *p* of those columns have predicates generated for them. The column store's query is generated first. There is a different column scanner for each kind of compression used: an RLE scanner, a dictionary scanner, and a bit-packed scanner. The generator knows which scanner to choose based on the table's schema and column number (see Table 1). The inputs to the scanner are then picked, including any predicate values. The primary scanner inputs are the data file, that column's offsets within the output tuple, and the number of bits needed to represent the encoded input column value. After all the columns have been processed, the generator outputs C++ code for the column store variant of the query.

Then, using the same set of columns, the code for the row store's query is generated. Each of the five table types (see Section 4.1) must be scanned differently because of compression, and the generator outputs the minimum code necessary to decode the columns used in the query in order to save processing cycles.

Some desired output selectivities are difficult to obtain when queries are randomly generated. For example, assume that the input table is Narrow-E and that the query has a single predicate on the first column. In this case, it is only possible to obtain selectivity factors that are multiples of 0.1666 since the first column only has six values that are uniformly distributed. Since the selectivity factor affects performance, we needed a way to obtain finer control over the selectivity factor of the query's predicates. To do this, we modified how predicates are evaluated by also incorporating how many times the function itself has been called. For instance, if the desired selectivity is 10%, every tenth call to the predicate function will pass, regardless of whether the attribute value satisfies the predicate. Since selectivities are

multiplicative, when there are multiple predicates, each predicate has a selectivity of desired_selectivity$^{(1/p)}$, where p is the number of predicates.

5. IMPLEMENTATION DETAILS

The results presented in this paper are based on the same code-base as [12], which is provided online [2]. The scanners and page creation procedures were modified to allow for the different forms of compression. We first discuss why we chose this code-base, then the next two subsections discuss and summarize the salient features of the software. The last subsection gives the experimental methods.

5.1 Choice of Code-Base

We chose this code-base for a variety of reasons. First, our work follows on to that in [12]; thus, we use the same code so that a direct comparison can be made between our results and theirs. The code is also easy to understand and modify, and, most importantly, is minimal, so we can have more confidence that performance differences between the row store and the column store are fundamental, and not related to code overheads or quirks in the file system. Additionally, this experimental setup has passed the vetting process of reviewers.

The two main research column-store databases are C-Store and MonetDB [18, 8]. Both systems are available online, but they are heavier-weight, and we are trying to understand performance fundamentals. MonetDB uses a different processing model, where columns (or parts of columns) are stored in vectors in memory, whereas we assume the columns are disk resident. MonetDB proponents argue this main memory-based kernel can provide better performance than C-Store, but, to our knowledge, no direct comparison of the systems has been published.

5.2 Read-Optimized Pages

Both the row store and column store dense pack the table on the pages. The row store keeps tuples together, placed one after another, while the column store stores each column in a different file. The page size is 32 KB. In general, the different entries on the page are not aligned to byte or word boundaries in order to achieve better compression.

Each page begins with the number of entries on the page. The row or column entries themselves come next, followed by the compression dictionary (if one is required). The size of the compression dictionary is stored at the very end of the page, with the dictionary growing backwards from the end of the page towards the front. For the row store, the dictionaries for the dictionary-compressed columns are stored sequentially at the end of the page.

5.3 Query Engine, Scanners and I/O

The query engine and table scanners provide only a minimal set of functions. All queries are precompiled, and the query executor operates on the table an output block at a time. The scanners decode values, apply predicates and either project or combine the columns into a materialized tuple buffer.

The query scanner and I/O architecture are depicted in Figure 1. Since the row scanner is simpler to understand, it is explained first. The relational operator calls "next" on the row scanner to receive a block of tuples. The row scanner first reads data pages from disk into an I/O buffer, then iterates through each page in the buffer. The scanner always decodes the columns of the tuple that might be used, then applies the predicate(s). If the tuple passes the predicate(s), the uncompressed projection is written to the materialized tuple buffer. When the buffer is full, it is returned to the relational operator parent, which can print the tuples, write them to disk, or do nothing (for our experiments the parent operator simply tosses the output tuples). The relational operator then empties the buffer and returns it to the scanner.

The column scanner is similar to the row scanner, but must read multiple files – one for each column referenced by the query. Each column is read until the materialized output tuple buffer is full (this buffer is discussed shortly); at that point, the read requests for the next column are submitted. Since predicates are applied on a per-column basis, columns are processed by order of their selectivity, most selective (with the fewest qualifying tuples) to least selective (the most qualifying tuples). Placing the most selective predicate first allows the scanner to read more of the current file before having to switch to another file, since the output buffer fills up more slowly.

For each attribute value that satisfies its predicate, the value and its position in the input file are written into the output buffer. The passing positions (pos list in Figure 1) are then input into the next column's scanner, and that column is only examined at those positions.

The materialized tuple buffer holds 100 tuples. At this size, it can fit in the 32KB L1 data cache, even if there are dictionaries, for each of the five different tables. This buffer is used to reduce overhead. Instead of performing aggregate computation, outputting qualifying tuples, or switching from column to column for every individual tuple, these operations are batched. For our experiments, the qualifying tuples are simply materialized – they are not output to the screen or to disk. A buffer size that is too small can lead to unnecessary overhead and poor performance by, for instance, writing to disk as soon as one tuple has been found to qualify. On the other hand, if the buffer is too big, it will fall out of the data cache and increase processing time. We ran scans with multiple buffer sizes and found 100 gave the best performance.

All I/O is performed through Linux's Asynchronous I/O (AIO) library. The code's AIO interface reads I/O units of 128KB (a user-set parameter), and it allows non-blocking disk prefetching of up to *depth* units. Data is transferred from the disk to memory using DMA and does not use Linux's file cache. The code does not use a buffer pool, instead it writes the transferred data to a buffer pointed to by a program variable.

5.4 System and Experimental Setup

All results were run on a machine running RHEL 5 on a 2.4 GHz Intel Core 2 Duo processor and 2GB of RAM. The disk is a 320 GB 7200 RPM SATA Western Digital WD3200AAKS hard disk. We measured its bandwidth to be 70 MB/s. Column stores and row stores are affected by the amount of I/O and processing bandwidth available in the system; our experiments have less I/O bandwidth than those in [12], but the general shape of the graphs are similar. Runs are timed using the built-in Linux "time" function. For each combination of output selectivity and number

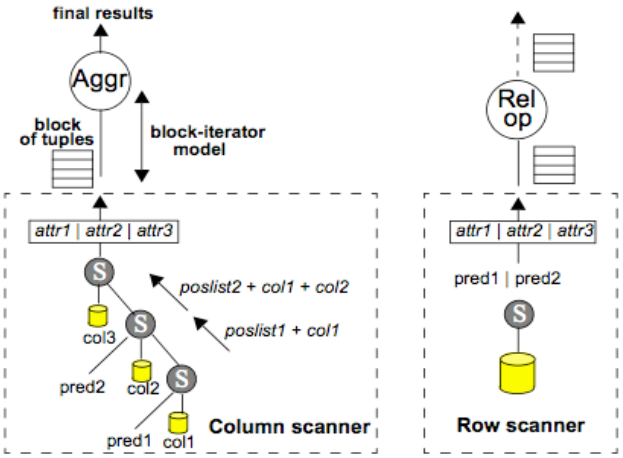

Figure 1. Query Engine Architecture [12].

of columns accessed, 50 equivalent queries were generated. The run times presented in Section 6 are the averages of those queries.

Sometimes we observed transient AIO errors, which caused a query to abort. When this happened, the run was terminated and restarted. We verified that runs that completed without errors returned the correct results. We also verified that running the same query multiple times in a row had negligible timing differences between runs.

The program has four user-defined variables: I/O depth, page size, I/O unit (scan buffer) size, and block (materialized tuple buffer) size. We use an I/O depth of 48; thus, each scan buffer can be prefetched up to 47 I/O units in advance. We use a larger page size than [12], but otherwise the program parameters are the same. We decided to use 32KB pages instead of 4KB pages since we think the larger size is more common in practice. However, [12] found that, as long as the page was not too small, the size did not significantly impact performance.

6. RESULTS

The results of our experiments are presented in this section, beginning with the amount of compression we were able to obtain.

6.1 Amount of compression

The compression methods used to encode each table are listed in Table 1. Table 2 presents their compressed sizes. The compression factors achieved range from 1 to 3 if just variable length, bit aligned attributes are used (column 3) and from 2 to almost 6 when compressed (column 4). While techniques that compress the tables even more could have been used (e.g. Huffman encoding), we think the techniques presented are a good tradeoff.

This final compressed size is almost exactly the same for both the row store and the column store, i.e. the size of the compressed row store file is the same as the sum of the column file sizes for the column store. This result is very important since column store proponents argue column stores compress better than row stores, so the columns can be saved in multiple sort orders and projections for the same space as one row store [18]. Thus, for read-optimized row stores, this assertion is not true, even with aggressive column store compression.

Table 2. Total tuple sizes with and without compression.

Table	Uncompressed	Bit-aligned	Compressed
Narrow-E	40 Bytes	153 bits (20 Bytes)	17 Bytes
Narrow-S	40 Bytes	153 bits (20 Bytes)	7 Bytes
Wide-E	200 Bytes	811 bits (102 Bytes)	100 Bytes
Narrow-U	40 Bytes	270 bits (34 Bytes)	34 Bytes

6.2 Effect of selectivity

In Figure 2 we explore the impact of selectivity factor as a function of the number of columns returned (presented in bytes) for table Narrow-E. The x-axis is the number of bytes returned, and the y-axis is the elapsed time. For both the column store (C-%) and row store (R-%), each line corresponds to a different output selectivity, with one, three, five or ten columns returned, and one predicate. Each data point is the average of 50 randomly generated queries, as described in Sections 4.4 and 5.4. The time to scan the uncompressed row store with 50% selectivity is also included (R-Uncomp) to illustrate how much time compression can save. Additionally, this graph presents error bars of plus and minus a standard deviation; however, the standard deviations are often quite small, so the error bars are difficult to see. For most of the graphs in this paper, all but the selectivity of 50% line for the row store tests (only) have been omitted to improve the clarity of graphs, since the response time was not significantly affected by the selectivity factor of the query. The C-0.1% and C-1% response times are also almost exactly the same.

The column store is faster than the row store when eight of the columns are returned with selectivity factors of 0.1% and 1%; when five of the columns are returned with a selectivity factor of 10%; when two of the columns are returned with a selectivity of 25%; and basically never at a selectivity of 50%. Further, for this table configuration, the best speedup of the column store over the compressed row store is about 2, while the best speedup for the compressed row store over the column store is about 5.

Next, we turn to investigating the factors underlying the performance of the two systems at a selectivity factor of 10%. Figure 3 presents the total CPU and elapsed times for both the row and column store. Both the CPU and elapsed times are constant for the row store as the number of columns returned is increased, since the run is primarily disk-bound. The results for the column store are substantially different, as the CPU time consumed increases as more columns are returned. This increase comes from two main sources: more values must be decoded, and more tuples must be stitched back together (materialized) using some form of a memory copy instruction. At a selectivity factor of 0.1%, the CPU cost of the column store is constant, since very few tuples must be materialized, and, at 50% selectivity, both the column store and the row store become CPU-bound. While the number of columns returned definitely has a large effect on column store scan times, the selectivity of the predicate is also an extremely important factor.

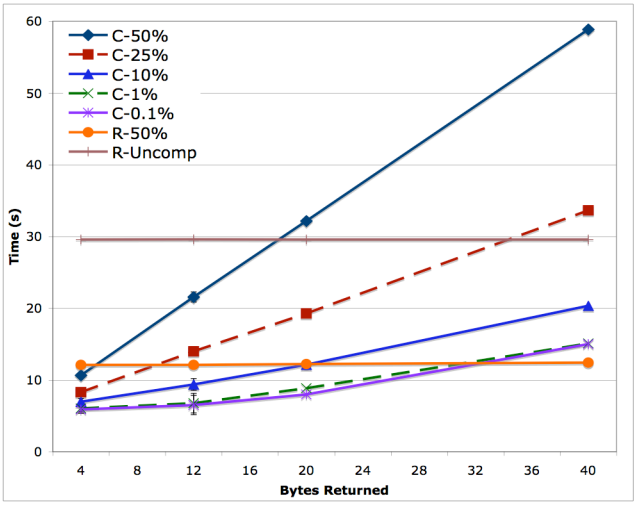

Figure 2. Elapsed time for Narrow-E with various selectivity factors.

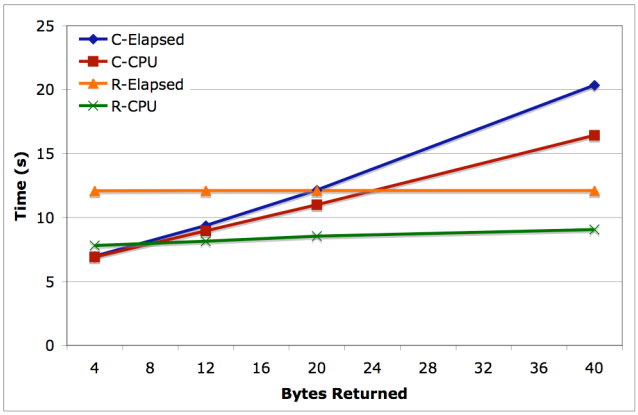

Figure 3. Elapsed time and CPU time for Narrow-E with 10% selectivity.

6.3 Effect of Skewed Data Distribution

To study the effect of a skewed data distribution, we repeated the experiments on table Narrow-S that we had performed on Narrow-E. Narrow-S allows more columns to be run-length (or delta) encoded, and made dictionary compression worthwhile for the columns with larger column cardinalities (number of different values in the column). Using these techniques, the row store went from averaging 17 bytes per tuple to 7 bytes per tuple (recall that the uncompressed tuple width is 40 bytes). The total size of the compressed column store tuple is the same as that for the row store tuple; each column compressed to less than two bytes per attribute, with the first attributes averaging just a few bits. The selectivity graph is presented in Figure 4. Error bars are also present on this graph, but the standard deviation is again small enough that they cannot be seen. Again, each data point represents the average of 50 different randomly generated queries in which both the columns returned and the column to which the predicate is applied are selected at random. The difference between the elapsed times for the column store runs with 0.1% and 1% selectivity factors are negligible. At 0.1% and 1% selectivity, the column store (C-0.1%,1%) is faster than the row store (R-0.1%,1%) when seven columns are returned, and at 10% selectivity, the column store is faster than the row store when three columns are returned. At a selectivity factor of 50%, the row

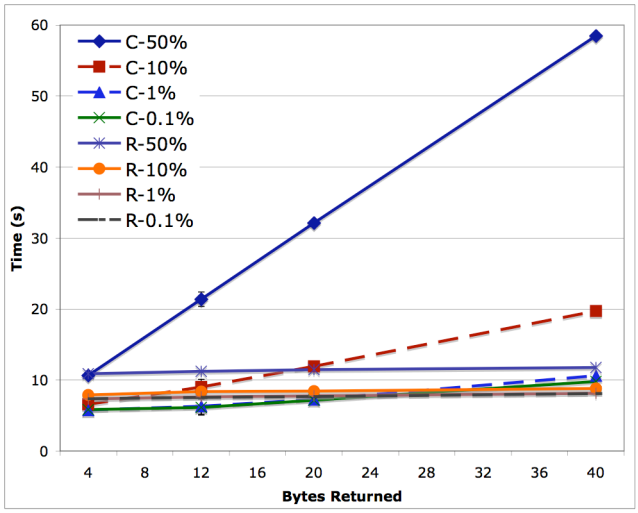

Figure 4. Elapsed time for Narrow-S at various selectivity factors.

store is always faster. For this table, the elapsed time for the row stores is affected by the selectivity factor since very little bandwidth is needed. Thus, the CPU time consumed dominates the overall execution time for the query for both column stores and row stores. Those queries with higher selectivity factors require more computation, so they have longer run times. For this table, the performance of the row store is very competitive with the column store largely due to the significant decrease in the amount of I/O it performs.

6.4 Wide Tables

Our next experiment was conducted using table Wide-E. Wide-E has fifty integer (4 byte) columns whose column cardinalities increase from left to right (see Table 1 for details). After bit compression, each tuple averages 100 bytes. To keep the total uncompressed table size the same as for Narrow-E, Wide-E has only 12 million rows (instead of 60 million). The time to scan the uncompressed table is shown in the R-Uncomp line. For the column store, the response times with either a 0.1% or 1% selectivity factor are essentially the same. The row store is faster than the column store when 85% of the columns are returned with 0.1% or 1% selectivity factors; when returning 66% of the columns at 10% selectivity, and with 25% of the columns at 50% selectivity. For the row store, the number of columns returned does not have an observable effect on the response time. As with Narrow-E, the elapsed time for the row store is dominated by its disk I/O component and the selectivity factor of the query; the CPU graph (not shown) is similar to Figure 3.

Overall, the graphs of Figure 5 and Figure 2 are very similar, however, the row store does not compress as well (and hence takes longer to scan), so there is a shift in the crossover points between the two systems. Because the slopes of the column store lines are not very steep, how well the row store compresses is a critical component in determining its performance relative to that of the column store. It should also be noted that many more columns are being returned than with the narrow table Narrow-E. If fewer than twelve columns are needed, the row store is always slower than the column store for this configuration.

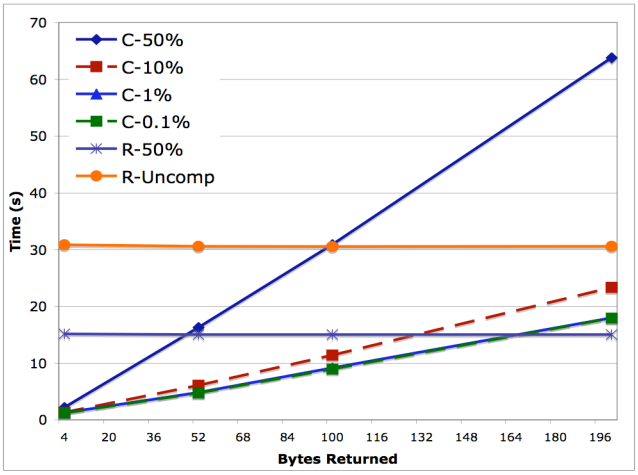

Figure 5. Elapsed time for Wide-E at various selectivity factors.

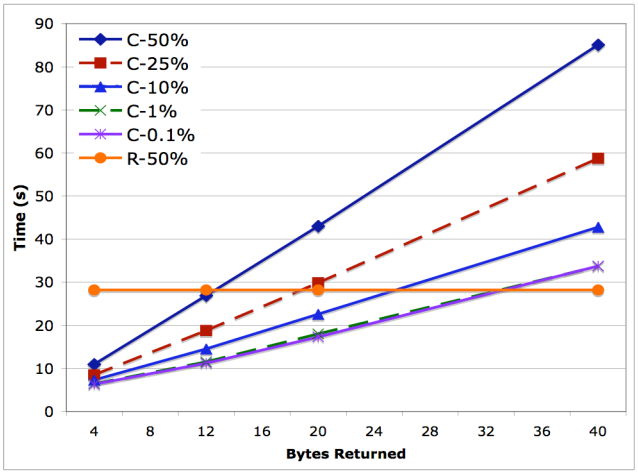

Figure 6. Elapsed time for Narrow-U with various selectivity factors.

6.5 Effect of Column Cardinality

Figure 6 presents the elapsed times for the experiments with Narrow-U. Like Narrow-E, this table has ten columns, but the values in each column are drawn uniformly and at random from a range of 1 to 100,000,000. The average compressed size of each tuple is 34 bytes, double that of the tuples in Narrow-E. The larger table and column sizes result in a significant increase in the response times that we measured for the two systems due to the additional I/O operations performed. The crossover points of the two systems are very similar to those for the wider and shorter table discussed in the previous section (see Figure 5).

6.6 Additional Predicates

The previous results have shown that the selectivity of the predicate can have a substantial effect on the relative performance of the row store and column store. We next consider how adding additional predicates affects the relative performance of the two systems by considering queries with three predicates (instead of one) and with selectivity factors of 0.1% and 10%. Table Narrow-E was used for these tests and the results are presented in Figure 7.

For each selectivity, we ran two sets of experiments. In the first set of experiments, the first, second, and third columns (x% left 3 preds) are used as the predicate columns (Figure 7 (top)). In the second, the columns for the three predicates were selected at random (x% random 3 preds) (Figure 7 (bottom)). The "C-x% 1 pred" results are taken from the experiments presented in Figure 2. Since the "1 predicate" point only uses one column, there is a predicate only on that column. For graph clarity, the row results are only shown for the 10% selectivity factor with one predicate, since it is representative of the other results.

We elected to draw predicates both from randomly selected columns and from the left three columns because we expected that the response time when the left hand columns were used would be better than when randomly selected columns were used. Since the table is sorted[1] on these columns, the runs in the left-most columns are longer. Hence, they compress better than the other columns. Our results verify that hypothesis. Our results also indicate that additional predicates can significantly affect the relative performance of the two systems. For instance, the 10% selectivity crossover is at five columns for the one-predicate case, but shifts to two columns when there are three predicates. The results are less stark for the 0.1% selectivity case since it requires so much less computation to begin with, but it still shifts the crossover from eight columns to seven.

These three-predicates results represent a worst-case scenario, as the selectivity is evenly divided between the columns. The results would be closer to the one predicate case if the first predicate had been highly selective. Thus, "real" workload results would probably fall somewhere in between the two points. However, the fact remains that increasing the number of predicates can have a significant impact on the performance of a column store.

6.7 Effect of "Wider" Attributes

We examine the impact of wider attributes by studying a table with ten 20-byte attributes. No compression was used for either the column store or the row store as we wanted to simulate the worst-case scenario for both systems. For this configuration, I/O is the dominant factor in the response times of both systems. Thus, the elapsed time of the column store is only slightly affected by the different selectivity factors, as can be seen in Figure 8.

To put this table's size in perspective, each width of column of this table is about the same as the compressed tuples in the Narrow-E table, and is about three times the size of the tuples in the Narrow-S table. Each tuple is twice as big as the compressed tuples in the Wide-E table.

6.8 Materialized Views

There is, of course, an intermediate form between row stores and column stores: materialized views. We implemented two sets of materialized views for the Narrow-E table. One set groups the ten attributes into five pairs: columns 1&2, 3&4, 5&6, 7&8, 9&10; the other groups the attributes into two groups of five: columns 1-5 and 6-10. We modified the column scanner to return an array of values instead of one value. Only columns used in the query are

[1] A major to minor sort is performed on the entire table from the left most column to the right most column.

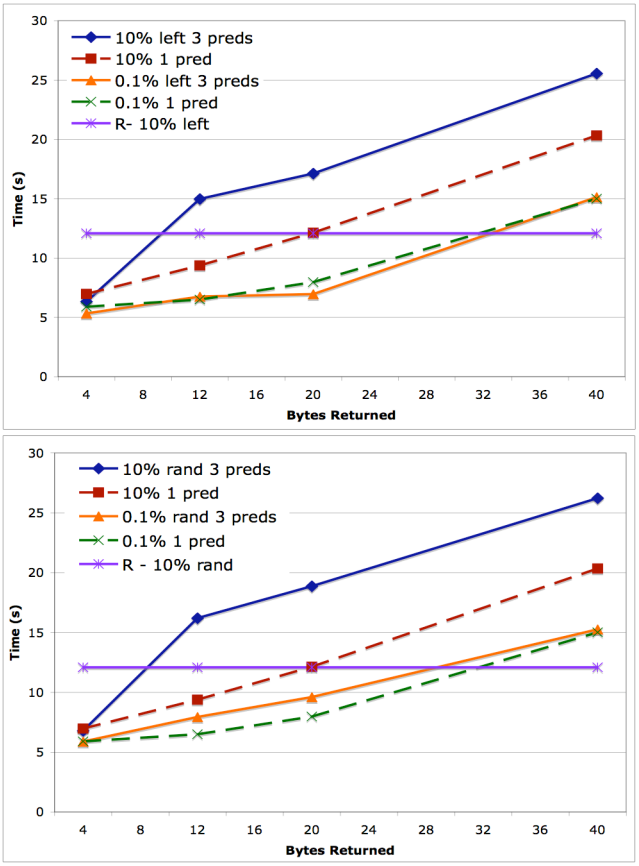

Figure 7. Three predicates on left-most columns (top) or random columns (bottom).

decoded. If more than one materialized view is needed, they are stitched together using the same mechanisms used to stitch columns together.

The benefit of materialized views depends greatly on the amount of correlation between columns exhibited by the query workload. Commercial products often provide guidance in forming views for row stores, and for column stores, Vertica has a Database Designer to aid in selecting which projections and sort orders would provide the best overall performance for a query workload [20]. However, since we are using a random query generator, we cannot rely on these automatic designers. Instead, we created the views, and then varied the correlation between the columns in randomly-generated workloads to find the benefit.

We looked at four different amounts of column correlation for each set of materialized views: 100%, 75%, 50% and none (independent). For the set of views where there are two views of five columns each, the query generator was set up so that one column was picked at random. Then, when there are five or fewer columns used, there is a *Correlation%* chance that there is only one view needed. For the set of five views of two columns each, first one column is drawn at random. Then, there is a *Correlation%* chance that the other column from that view is used. If the correlation test fails, the second column from that view is not used for that query, unless all five of the views are used and more columns are needed, and another column is randomly drawn. All columns are drawn without replacement. In

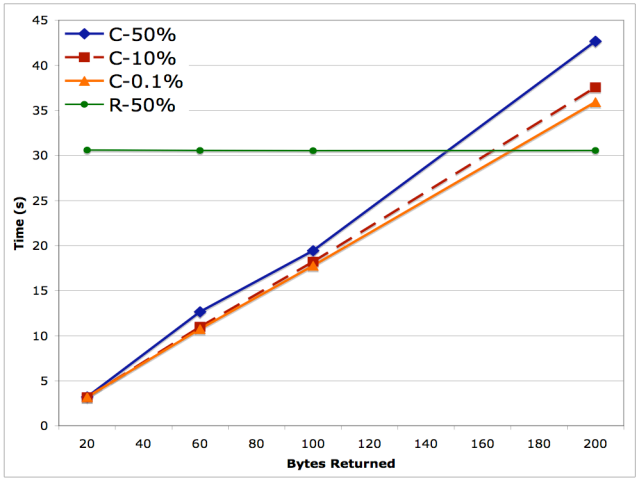

Figure 8. Elapsed time for String with various selectivity factors.

the "none" correlation case, all columns are drawn independently and at random, as in the earlier experiments.

Figure 9 provides the results for having two groups of five (left) or five groups of two (right) materialized views on the Narrow-E table. The queries have 1% selectivity, which is a selectivity where the column store's performance dominates. The row and column store results are those from earlier sections. We gathered materialized view results for returning up to six, then ten columns. We took more data points since the results are not smooth lines – they have stair-step performance due to the correlations between the columns. Error bars of plus one standard deviation are included for the random and 50% correlation cases, while minus error bars are included for correlations of 75% and 100%. Only one side is included for clarity. When the correlation between columns is high, the performance is similar to that of joining multiple small row stores. When there is no correlation between columns, the materialized views do not provide a large benefit for row stores compared to column stores, since multiple views must be joined, and reconstructing those tuples can be costly. In fact, the materialized views with five columns only outperform the row stores when three or fewer columns are returned, assuming no correlation. If more than five columns are used, the row store should be used due to the cost of stitching the two views together. Without correlation, the two column views outperform the row store when five or fewer columns are returned. However, with more correlation, the materialized views provide comparable performance to, and can occasionally beat, the column stores.

With the cost of storage essentially free, materialized views can easily be included with a row store, which can help the relative performance of the row store. The materialized views will also be affected by the selectivity of the query, in proportion to the number of columns in the view.

6.9 Joins

Finally, we also examined joins with different selectivities to see to what extent constructing result tuples interacts with executing the join (e.g., there may be more instruction cache misses due to switching between scanning and reconstructing tuples and performing the join). These experiments used a hybrid hash join

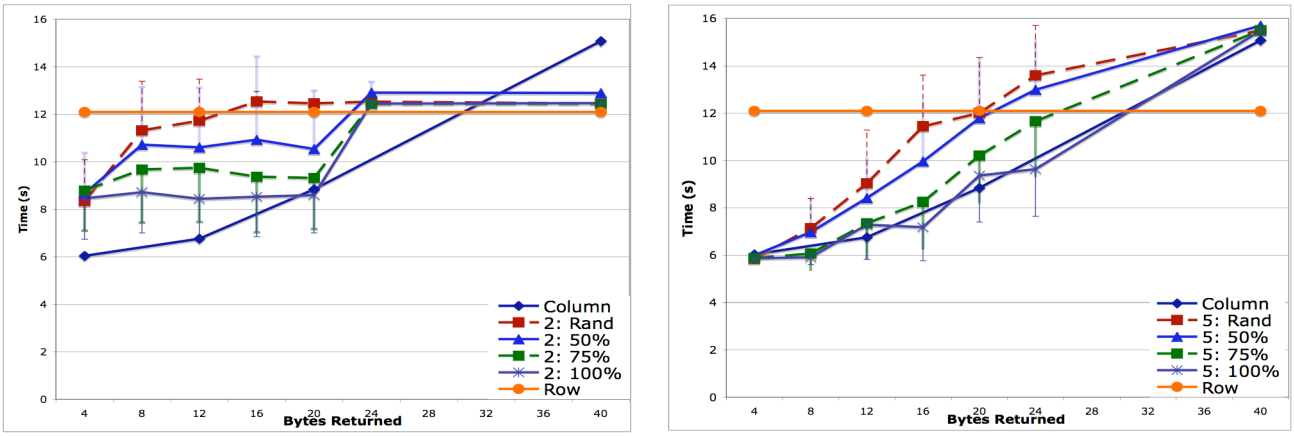

Figure 9. Materialized views for the Narrow-E table. The left figure presents the results for two materialized views with five columns in each view. The right figure presents the results for five materialized views with two columns each.

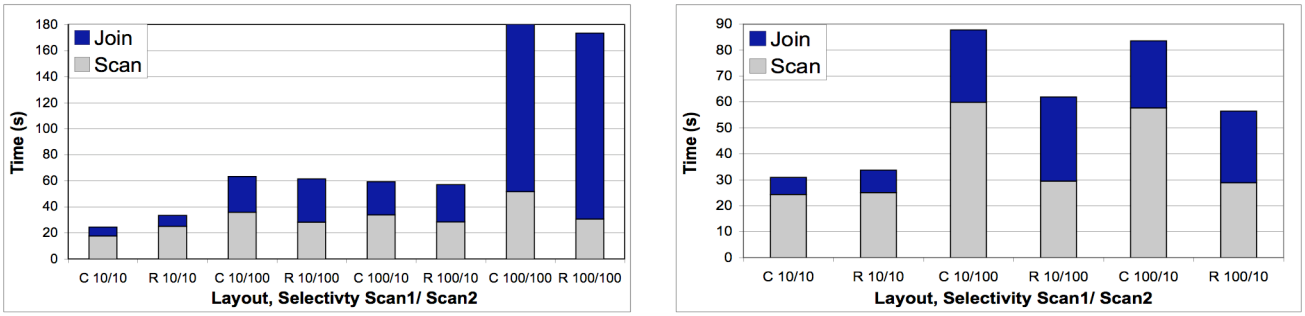

Figure 10. The left graph presents results for joining two two-column tables in a column store and row store. The x-axis labels give the storage paradigm and the scan selectivities. The right graph presents the results for joining two four-column tables.

[17] with enough pages allocated to ensure the inner table fits in memory. The two tables used in the join have either two columns or four columns accessed. The two-column case's SQL would be:

```
SELECT temp1.column2, temp2.column5
FROM temp1, temp2
WHERE temp1.column9 = temp2.column9 AND
temp1.column2>=x AND temp2.column5>=y;
```

Temp1 and Temp2 are both table Narrow-E for the row store. For the column store, Temp1 is column 2 and column 9 from Narrow-E; Temp2 is column 5 and column 9. For the four-column case, table Temp1 also includes columns 1 and 3, and table Temp2 also includes columns 7 and 10. Column 9 was chosen as the join attribute since each value occurs a small number of times within the table, and the number of tuples passing a join increases as a square of the number of duplicate matching join attribute values. The other columns were chosen at random so there would be no overlap in Temp1 and Temp2, besides column 9.

The variables x and y in the query are varied to return either 10% or 100% of the rows for the table, and the query is performed four times to get times for scan selectivity factors of 100/100, 100/10, 10/100 and 10/10. The number of resulting tuples is 234M, 22M, 28M and 2.6M, respectively.

We chose to use either two or four columns from the table for a variety of reasons. At five or more columns, the likelihood of running out of memory increases. Additionally, row stores outperform column stores as more columns are accessed. On the other hand, consistently using just the join attribute for both tables was unlikely in real workloads. Hence, we chose two and four columns to get two sets of results without severely impacting the column store's performance.

The inner table is the one with the fewest rows that pass the predicate. This table is scanned first, and its query is generated in the same way as the queries used for the results in the previous sections. However, once the tuple has been materialized, instead of being discarded, the join attribute is hashed, partitioned and inserted into the appropriate bucket page. After the first scan completes, the second scan begins, and probes for joins after the tuple has been materialized. If the bucket is wholly in memory, the resulting join tuple(s) is (are) materialized. If the bucket is not in memory, the tuple is written to a page and is processed after the scan is complete. This plan uses an early materialization strategy, as per the results of [5].

Figure 10-left presents the elapsed time for the join for both row stores and column stores for the two column tables, with the given scan selectivities. Figure 10-right presents the results for the four column tables, but does not include the 100/100 case in the results because the inner table does not fit in memory. Each bar presents the elapsed time for the join, and the part of that time it takes to just perform the two scans. The scan results are as expected, and the join component of the time is always roughly equivalent between the column store and row store. Thus, the paradigm with the smaller scan time will also have the smaller join time, and the join time is greatly affected by the number of joined tuples that must be materialized.

7. DISCUSSION

To begin the discussion, let us summarize the findings:

- Read-optimized row stores and column stores compress to within a few bits of each other.
- Regardless of the table size or table type, the selectivity of the predicate can substantially change the relative performance of row stores and column stores.
- Row stores perform better compared to column stores when the tuple is narrow.
- Adding predicates increases column store runtimes.
- Having more qualifying tuples increases column store runtime.
- Materialized views, which are essentially a hybrid of row stores and column stores, can out-perform both row stores and column stores, depending on the circumstances, but normally have performance somewhere between the two paradigms.
- Hybrid hash joins with early materialization do not change the relative performance of row stores and column stores.

While [12] reached some of the same conclusions, our results further quantify these findings and the extent to which they hold, and add some new results.

A rule of thumb is that a column store outperforms a row store when I/O is the dominating factor in the response time of a query, but a row store can outperform a column store when processing time is the dominating constraint. Our results show that I/O becomes less of a factor for row stores with compression, and CPU time is more of a factor for column stores in queries with more predicates, lower selectivities and more columns referenced.

Row stores on slotted pages will most likely never beat column stores for read-optimized workloads since their bandwidth requirements are much higher than for even the uncompressed bit-aligned tables. However, a read-optimized row store can clearly outperform a column store under some conditions.

Row store designers must seriously reconsider two points: compression and schema design. Using aggressive compression techniques is critical to reducing the overall scan time for a row store. In addition, our results along with those in [12] clearly demonstrate that, for the current generation of CPUs and disk drives, 20 bytes is a good average tuple size to aim for.

Additionally, some column store proponents have argued that, since column stores compress so much better than row stores, storing the data with multiple projections and sort orders is feasible and can provide even better speedups [18]. However, we have found that columns do not actually compress any better than read-optimized rows that employ bit compression and delta encoding. Since it is now feasible to store row stores in multiple projections and sort orders without a substantial storage overhead, developing techniques for selecting the best materialized views (keeping in mind the 20 bytes per view per row target) might prove to be beneficial, as our results from Section 6.8 show.

We have shown multiple ways to decrease the I/O requirements of a query workload. If the workload has many low-selectivity queries, or multiple predicates per query, the tuples could be even larger and still provide roughly the same performance as column stores. However, for workloads comprised of high selectivity queries that randomly select just one or two columns from a wide table that cannot be vertically partitioned in a non-column-store way, row stores cannot compete.

8. ACKNOWLEDGMENTS

The authors would like to thank David Lomet, Daniel Abadi, Sam Madden and the anonymous reviewers.

9. REFERENCES

[1] https://www.cs.hut.fi/Opinnot/T-106.290/K2005/Ohjeet/Zipf.html. Accessed November 8, 2007.

[2] http://db.lcs.mit.edu/projects/cstore/. Accessed November 8, 2007.

[3] Abadi, D. J., Madden, S. R., Ferreira, M. C. "Integrating Compression and Execution in Column-Oriented Database Systems." In *SIGMOD*, 2006.

[4] Abadi, D.J., Madeen, S. R., Hachem, N. "Column-Stores vs. Row-Stores: How Different Are They Really?" In *SIGMOD*, 2008.

[5] Abadi, D. J., Myers, D.S., DeWitt, D.J., Madden, S.R. "Materialization Strategies in a Column-Oriented DBMS." In *ICDE*, 2007.

[6] Ailamaki, A. *Architecture-Conscious Database System*s. Ph.D. Thesis, University of Wisconsin, Madison, WI, 2000.

[7] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. "Weaving Relations for Cache Performance." In *VLDB*, 2001.

[8] Boncz, P., Zukowski, M., and Nes, N. "MonetDB/X100: Hyper-Pipelining Query Execution." In *CIDR*, 2005.

[9] Copeland, A. and Khoshafian, S. "A Decomposition Storage Model." In *SIGMOD*, 1985.

[10] Halverson, A. J., Beckmann, J. L., Naughton, J. F., DeWitt, D. J. "A Comparison of C-Store and Row-Store in a Common Framework." *Technical Report, University of Wisconsin-Madison, Department of Computer Sciences*, T1666, 2006.

[11] Hankins, R. A., Patel, J. M. "Data Morphing: An Adaptive, Cache-Conscious Storage Technique." In *VLDB*, 2003.

[12] Harizopoulos, S., Liang, V., Abadi, D., and Madden, S. "Performance Tradeoffs in Read-Optimized Databases." In *VLDB*, 2006.

[13] Holloway, A. L., Raman, V., Swart, G. and DeWitt, D. J. "How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans." In *SIGMOD*, 2007.

[14] Huffman, D. "A Method for the Construction of Minimum-Redundancy Codes." In *Proceedings of the I. R. E.*, pages 1098-1102, 1952.

[15] Ramakrishnan, R. and Gehrke, J. *Database Management Systems.* McGraw-Hill, 3rd edition, 2003.

[16] Raman, V., Swart, G. "Entropy Compression of Relations and Querying of Compressed Relations." In *VLDB*, 2006.

[17] Shapiro, L. D. "Join processing in database systems with large main memories." *ACM Trans. Database Syst*. 11(3): 239-264 (1986).

[18] Stonebraker, M., et al. "C-Store: A Column-Oriented DBMS." In *VLDB*, 2005.

[19] T. P. P. Council. "TPC Benchmark H (Decision Support)," http://www.tpc.org/tpch/default.asp, August 2003.

[20] "The Vertica Database Technical Overview White Paper." Vertica, 2007.

[21] Zukowski, M., Heman, S., Nes, N., and Boncz, P. "Super-Scalar RAM-CPU Cache Compression." In *ICDE*, 2006.