Performance Tradeoffs in Read-Optimized Databases

Stavros Harizopoulos Velen Liang Daniel J. Abadi Samuel Madden
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139
{stavros, vliang, dna, madden}@csail.mit.edu

ABSTRACT

Database systems have traditionally optimized performance for write-intensive workloads. Recently, there has been renewed interest in architectures that optimize read performance by using column-oriented data representation and light-weight compression. This previous work has shown that under certain broad classes of workloads, column-based systems can outperform row-based systems. Previous work, however, has not characterized the precise conditions under which a particular query workload can be expected to perform better on a column-oriented database.

In this paper we first identify the distinctive components of a read-optimized DBMS and describe our implementation of a high-performance query engine that can operate on both row and column-oriented data. We then use our prototype to perform an in-depth analysis of the tradeoffs between column and row-oriented architectures. We explore these tradeoffs in terms of disk bandwidth, CPU cache latency, and CPU cycles. We show that for most database workloads, a carefully designed column system can outperform a carefully designed row system, sometimes by an order of magnitude. We also present an analytical model to predict whether a given workload on a particular hardware configuration is likely to perform better on a row or column-based system.

1. INTRODUCTION

A number of recent papers [21][7] have investigated column oriented physical database designs (column stores), in which relational tables are stored by vertically partitioning them into single-column files. On first blush, the primary advantage of a column-oriented database is that it makes it possible to read just the subset of columns that are relevant to a query rather than requiring the database to read all of the data in a tuple, and the primary disadvantage is that it requires updates to write in a number of distinct locations on disk (separated by a seek) rather than just a single file.

Surprisingly, these initial observations turn out to not be trivially true. For example, to obtain a benefit over a row store when reading several columns in a table, a column store must ensure that it can read large sequential portions of each column, since the cost of seeking between two fields in the same tuple (stored separately on disk) can be prohibitively expensive. Similarly, writes can actually be made relatively inexpensive as long as many inserts are buffered together so they can be done sequentially. In this paper, we carefully study how column orientation affects the performance characteristics of a read-optimized database storage manager, using scan-mostly queries.

1.1 Read-Optimized DBMS Design

While column-oriented systems often have their own sets of column-wise operators that can provide additional performance benefit to column stores [1], we focus in this paper on the differences between column and row stores related solely to the way data is stored on disk. To this end, we implement both a row- and column-oriented storage manager from scratch in C++ and measure their performance with an identical set of relational operators. As data is brought into memory, normal row store tuples are created in both systems, and standard row store operations are performed on these tuples. This allows for a fixed query plan in our experiments and a more direct comparison of column and row systems from a data layout perspective.

Both of our systems are read-optimized, in the sense that the disk representation we use is tailored for read-only workloads rather than update-intensive workloads. This means, for example, that tuples on-disk are dense-packed on pages, rather than being placed into a slotted-tuple structure with a free-list per page. Figure 1 shows the basic components of a generalized read-optimized DBMS, upon which we base both of our systems (solid lines show what we have actually implemented for the purposes of this paper). In this design we assume a staging area (the "write optimized store") where updates are done, and a "read-optimized"

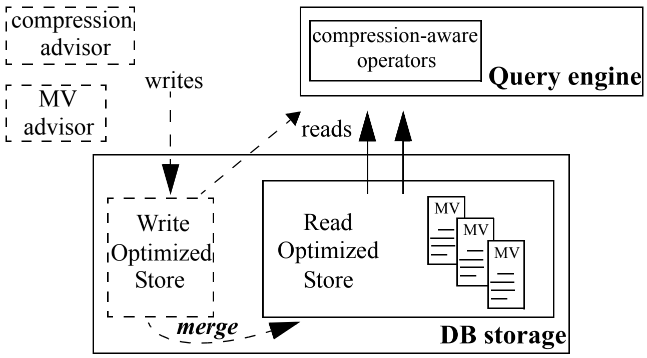

Figure 1. Basic components of read-optimized DBMS.

VLDB '06, September 12-15, 2006, Seoul, Korea.

store where tuples are permanently stored on disk. Tuples are periodically moved in bulk from the write store to the read store. A compression advisor and a materialized view (MV) advisor choose compression schemes and appropriate vertical partitioning depending on the workload characteristics. This design is loosely inspired by the design of C-Store [21], a column-oriented database that also maintains separate storage for writes and periodically merges new data into the read-only store.

1.2 Performance Study

We use our read-optimized systems to perform an in-depth analysis of the tradeoffs between column and row-oriented architectures, in terms of disk bandwidth, CPU cache latency, and CPU cycles. Our performance study is based on a set of scan-mostly queries (which represent the majority of queries in read-mostly databases like data warehouses), operating on a database schema derived from the TPC-H benchmark specification. The purpose of this paper is not to definitively settle the question of whether a column store is “better” than a row store, but to explore the situations under which one architecture outperforms the other. We focus in particular on answering the following questions:

1. As the number of columns accessed by a query increase, how does that affect the performance of a column store?
2. How is performance affected by the use of disk and L2 cache prefetching?
3. On a modern workstation, under what workloads are column and row stores I/O bound?
4. How do parameters such as selectivity, number of projected attributes, tuple width, and compression affect column store performance?
5. How are the relative performance tradeoffs of column and row stores affected by the presence of competition for I/O and memory bandwidth along with CPU cycles from competing queries?

In addition to a complete set of performance results comparing the column- and row-based systems, we present an analytical model that provides accurate estimates of CPU and I/O costs for a given hardware configuration and table layout. We note several things that we explicitly chose not to study in this paper. Because we are focused on the performance of queries in a read-optimized system (e.g., warehouses) we do not model updates. Our view is that updates in these systems are not done via OLTP-style queries but rather happen via a bulk-loading tool. Building an efficient bulk-loader is an interesting but orthogonal research question. As noted above, we also do not study the effects of column versus row orientation on different database operators — our query plans are identical above the disk access layer. This decision allows us to directly compare the two approaches, isolating implementation differences to the smallest possible part of the system.

1.3 Summary of Results

We find that, in general, on modern database hardware, column stores can be substantially more I/O efficient than row stores but that, under certain circumstances, they impose a CPU overhead that may exceed these I/O gains. Figure 2 summarizes these results (Section 5 explains how we construct the graph). In this contour plot, each color represents a speedup range achieved by a column system over a row system when performing a simple scan of a relation, selecting 10% of the tuples and projecting 50% of the tuple attributes. The x-axis is the tuple width of the relation in bytes (either compressed or uncompressed). The y-axis represents the system’s available resources in terms of CPU cycles per byte read sequentially from disk (*cpdb*). In Section 5 we show that typical configurations (varying the number of available CPUs, disks, query requirements, and also competing traffic to both the CPUs and the disks) range from 20 to 400 cpdb. As a reference, a modern single-disk, dual-processor desktop machine has a cpdb of about 108.

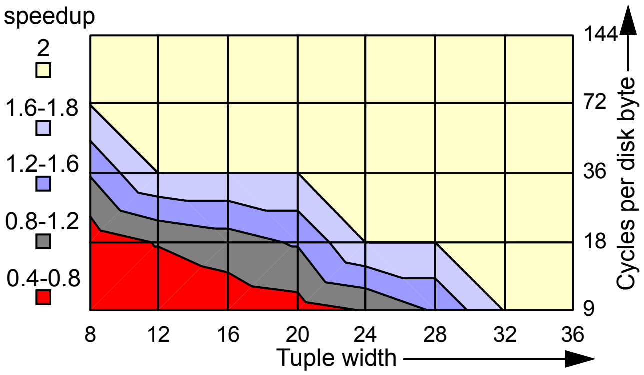

Figure 2. Average speedup of columns over rows (see text below for an explanation).

For the graph of Figure 2, we compute the *average* speedup by assuming the column system is accessing half of the columns (50% projection). For example, for a tuple width of 8 bytes, a column system reads only 4 bytes per tuple. Later we present results that show that the speedup of columns over rows converges to 1 when the query accesses all attributes, but it can be as high as *N* if the query only needs to access 1 / *N*th of the tuple attributes. As Figure 2 shows, row stores have a potential advantage over column stores only when a relation is lean (less than 20 bytes), and only for CPU-constrained environments (low values of cpdb). As tuples get wider, column stores maintain a sizeable relative performance advantage.

1.4 Contributions and Paper Organization

The main contribution of this paper is increased understanding of the tradeoffs between column and row stores in read-optimized settings. We focus on isolating the important factors that affect relative query performance in both column and row stores. Our performance study is based on a real implementation that strips the problem to the bare minimum by avoiding irrelevant details. This allows us to report results that are general and easy to reproduce. Furthermore, by capturing important system parameters in a few simple measures, we are able to provide a set of equations that can predict relative performance of column and row stores on a wide range of different configurations.

The rest of the paper is organized as follows. Section 2 discusses performance considerations in read-optimized databases (more specifically, the role of disk and main memory bandwidth), and then describes our implementation. Section 3 contains the experimental setup, while Section 4 carries out the performance study. We present our analysis in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. READ-OPTIMIZED DBMS DESIGN

In this section we first discuss performance considerations in read-optimized database architectures, and then present our implementation of a minimum-complexity, high-performance relational query engine that can operate on both row and column-oriented data.

2.1 Performance Considerations

Data warehousing and business intelligence applications rely on the efficiency a database system can provide when running complex queries on large data repositories. Most warehouses are bulk-loaded during periods of low activity (e.g., over night) with data acquired from a company's operation (such as sales, or clicks on web sites). Bulkloading is followed by periods when analysts issue long, read-only queries to mine the collected data. The database schema for these types of applications typically specifies a few (or, just one) *fact tables* that store the majority of the newly collected information (e.g., sales/clicks), and a number of smaller *dimension tables* for the rest of the data (e.g., customer/company info). In a typical schema, the fact tables store mostly foreign keys that reference the collection of dimension tables.

In such environments, typical SQL queries examine fact tables and compute aggregates, either directly from the fact table data, or by first joining them with dimension tables. Data sometimes needs to be corrected in a data warehouse, which can involve updates to fact tables. Often, however, compensating facts are used (e.g., a negative Sale amount). The majority of typical long-running data warehousing queries involve sequential scans on the fact tables. Speeding up the time it takes to read data off the disk and, through the cache hierarchies, into the CPU can have a significant performance impact. Exactly this type of optimization lies in the heart of a read-optimized DB design and comprises the focus of this paper. Other types of optimizations (such as materialized view selection or multi-query optimization) are orthogonal to scan-related performance improvements and are not examined in this paper.

An important goal of read-optimized databases is to minimize the number of bytes read from the disk when scanning a relation. For a given access plan, there are two ways to achieve this goal: (a) minimize extraneous data read (information that is not needed to evaluate a query), and (b) store data in a compressed form. To achieve (a), read-optimized systems improve the space utilization of a database page by packing attributes as densely as possible. Since updates occur in bulk, there is no reason to leave empty slots in pages to accommodate inserts of a few tuples. Column-oriented data can further reduce the number of bytes read, since only the columns relevant to the query need to be read. Compression schemes can apply to both column and row data, and they have been shown to speed up query execution [22][18][24][1]. Database-specific compression techniques differ from general-purpose compression algorithms in that they need to be "light-weight," so that the disk bandwidth savings do not exceed the decompression cost. In this paper we study three such compression techniques (described later in Section 2.2.1).

2.1.1 Disk-Related Considerations

As disk drives yield their peak data delivery bandwidth when data is read sequentially, database systems embrace sequential storing and accessing of relations. As a reference, inexpensive workstation configurations (e.g., plain SATA controllers and software RAID) can easily yield 100-300MB/sec of aggregate disk bandwidth. When a sequential disk access pattern breaks, the disks spend about 5-10 msec while the heads perform a seek to the new data location. To preserve high performance, the system needs to minimize the time spent on seeks. For multiple concurrent sequential requests, this is typically achieved by prefetching a large amount of data from one file before seeking to read from the next file. Modern database storage managers employ some form of prefetching. For example, the SQL Server Enterprise Edition's storage manager may prefetch up to 1024 pages (each page is 8KB).

Even when database systems have the option to use an index on a large relation, in many cases it is better to use a plain sequential scan. For instance, consider a query that can utilize a secondary, unclustered index. Typically, the query probes the index and constructs a list of record IDs (RIDs) to be retrieved from disk. The list of RIDs is then sorted to minimize disk head movement. If we were to assume a 5ms seek penalty and 300MB/sec disk bandwidth, then the query must exhibit less than 0.008% selectivity before it pays off to skip any data and seek directly to the next value (assuming 128-byte tuples and uniform value distribution).

When multiple concurrent queries scan the same table, often it pays off to employ a single scanner and deliver data to multiple queries off a single reading stream (*scan sharing*). Teradata, RedBrick, and SQL Server are among the commercial products that have been reported to employ this optimization, and, recently, it has also been demonstrated in the QPipe research prototype engine [13]. Such an optimization is orthogonal to data placement (columns versus rows), and therefore we do not examine it in this paper.

2.1.2 CPU Cache-Related Considerations

In recent years, a thread of database research has been investigating CPU cache-related delays in database workloads, and has proposed cache-conscious algorithms and data placement techniques (relevant references and additional background can be found in [3]). Recent studies have pointed out that commercial DBMS spend a significant amount of time on L2 cache misses [5][15]. The proposed methodologies calculate the time spent on L2 cache misses by measuring the actual number of misses (via CPU performance counters) and multiplying this number by the measured memory latency. While this methodology accurately described DBMS performance on the machines used at the time (Intel Pentium Pro / Pentium II), it no longer applies to today's CPUs (such as Pentium 4). The reason is that all major chip manufacturers now include both hardware and software data prefetching mechanisms[1]. Once the hardware detects that a memory region (such as a database page) is accessed sequentially, it will start prefetching data from main memory to the L2 cache. For example, Pentium 4 will prefetch two cache lines (128 bytes each) for every accessed cache line that is characterized by a predictable access pattern.

The implications of hardware cache prefetching are especially significant in data warehousing, as data pages are regularly accessed sequentially. The bottleneck now becomes the memory bus bandwidth, and not the cache latency, as previously thought. This effect makes main memory behave somewhat similarly to a

[1] Intel Corporation. "IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide." (*Order Number 253668*).

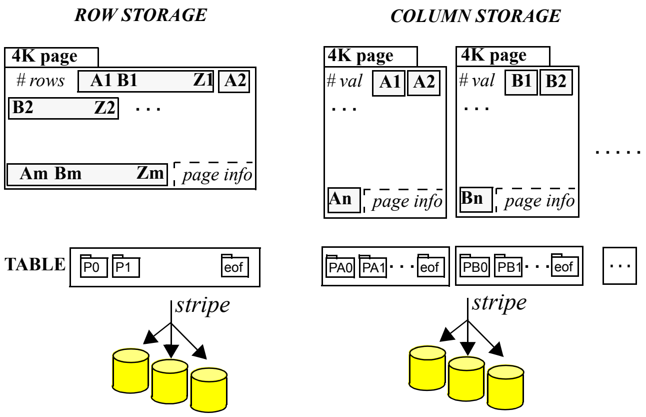

Figure 3.Page structure and disk storage layout for a single table for Row data (left) and Column data (right).

disk when it comes to sequential accesses: it allows the CPU to overlap computation with memory-to-cache transfers (on top of disk-to-memory transfers). We describe in Section 4.1 how we calculate memory-related delays when taking into consideration the hardware cache prefetcher. For configurations where memory bandwidth is the bottleneck, database architectures can speed-up query execution in the same way they deal with disk bandwidth: by avoiding reading extraneous data and by utilizing compressed forms of data [24].

2.2 Implementing a Read-Optimized Engine

In this section we describe our implementation of a minimum-complexity, read-only relational query engine that operates on both column and row data. We first describe page layouts and compression schemes (Section 2.2.1), then the column and row data scanners (Section 2.2.2), and, lastly, the overall architecture and the rest of the relational operators (Section 2.2.3).

2.2.1 Disk Storage for Columns and Rows

Since there are no real-time updates in a read-optimized DBMS, we forego the traditional slotted-page structure in favor of a tightly packed one (see Figure 3). For both row and column data, a page contains an array of values: entire tuples for row data and single-attribute values for column data (we discuss in Section 6 alternative row page structures such as PAX [4]). At the beginning of the page, an integer value stores the number of entries. We store additional page-specific information at the end of the page (using a fixed offset). Such information includes the page ID (which, in conjunction with a tuple's position in the page gives the Record ID), along with compression-specific data. For simplicity, we use fixed-length attributes throughout this paper. For variable-length attributes, additional offset values need to be placed between values.

Pages are stored adjacently in a disk file. For column data, a table is stored using one file per column. In our system, we stripe database files across all available disks in a disk array. The page size is a system parameter, and for all experiments in this paper we use a 4KB value. For the type of sequential accesses we study, the page size has no visible effect on performance (as long as the cost of crossing page boundaries is significantly lower than the cost of processing a page).

We compress data using three commonly used lightweight compression techniques: *Dictionary*, *Bit packing*, and *FOR-delta* (see [22][10][1][24] for additional background on these schemes). Compression schemes are typically chosen during physical design. These three techniques yield the same compression ratio for both row and column data. They also produce compressed values of fixed length. We refrain from using techniques that are better suited for column data (such as *run length encoding* [1]) to keep our performance study unbiased. We use bit-shifting instructions to pack compressed values inside a page, and the reverse procedure for decompressing at read time. Following is a brief description of the three compression schemes we use.

Bit packing (or Null suppression). This scheme stores each attribute using as many bits as are required to represent the maximum value in the domain. For example, if an integer attribute has a maximum value of 1000, then we need at most 10 bits to store each attribute.

Dictionary. When loading data we first create an array with all the distinct values of an attribute, and then store each attribute as an index number to that array. For example, to compress an attribute that takes the values "MALE" and "FEMALE," we store 0 or 1 for each attribute. We use Bit packing on top of Dictionary. At read time, we first retrieve the index value through bit-shifting, and then make a look-up in the corresponding array.

FOR-delta. FOR (Frame-Of-Reference) maintains a base value per block of values (in our case, per page), and then stores differences from that base value. For most of the experiments we use FOR-delta, which also maintains a base value per block but stores the difference of a value from the *previous* one (the first value in the block is the base value of that block). This scheme applies to integer-type attributes that exhibit value locality. For example, a sorted ID attribute (100, 101, 102, 103, etc.), will be stored as (0, 1, 2, 3, etc.) under plain FOR, and as (0, 1, 1, 1, etc.) under FOR-delta; in both cases the base value for that page will be 100. At read time, real values are calculated either from the base value (in FOR) or from the base value and all the previous readings in that page (in FOR-delta).

2.2.2 Row and Column Table Scanners

Scanners are responsible for applying predicates, performing projection and providing the output tuples to their parent operators. Both the row and column scanner produce their output in exactly the same format and therefore they are interchangeable inside the query engine (see Figure 4). Their major difference is that a row scanner reads from a single file, whereas the column scanner must read as many files as the columns specified by the query. Both scanners employ the same I/O interface to receive buffers containing database pages. We describe the I/O interface in the next sub-section (2.2.3). The row scanner is straightforward: it iterates over the pages contained inside an I/O buffer, and, for each page, it iterates over the tuples, applying the predicates. Tuples that qualify are projected according to the list of attributes selected by the query and are placed in a block of tuples. When that block fills up, the scanner returns it to the parent operator.

A column scanner consists of a series of pipelined scan nodes, as many as the columns selected by the query. The deepest scan node starts reading the column, creating {position, value} pairs for all qualified tuples. We use the same block-iterator model to pass data from one scanner to the next. Once the second-deepest

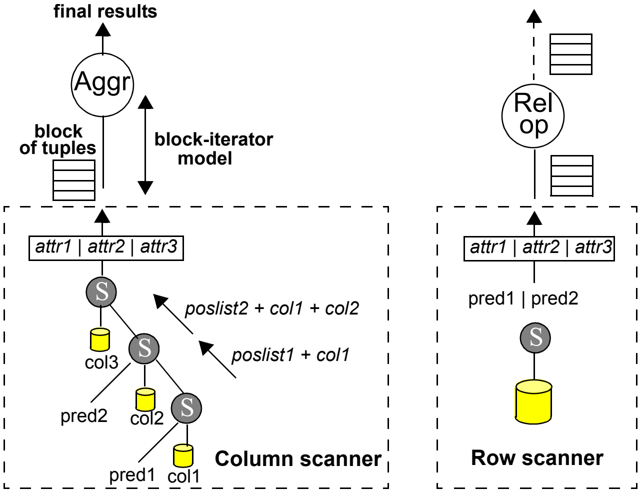

Figure 4. Query engine and scanner architecture.

scan node receives a block of tuples (containing position pairs), it uses the position information to drive the inner loop, examining values from the second column. That is, we only evaluate predicates corresponding to qualified records in the first column. If the second predicate is true, we copy both values along with the position to a new tuple that is passed on to the next scanner. When a scan node has no predicates, it simply attaches all values corresponding to the input positions, without re-writing the resulting tuples. At all times, the tuple blocks are reused between scanners (and operators), so there is no new memory allocation during query execution. As in most systems, we push scan nodes that yield few qualifying tuples as deep as possible.

2.2.3 Query Engine and I/O Architecture

Our query engine follows a pull-based, block-iterator model for all operators. Each relational operator calls *next* on its child (or children), and expects to receive a block (array) of tuples (Figure 4). Adopting a block-iterator model instead of a tuple-by-tuple one amortizes the cost of procedure calls and also removes any delays related to CPU instruction-cache performance [17][23]. We make the block size a tunable parameter and choose a value that makes the block fit in the L1 data cache to minimize L1 cache delays (on our machine, the L1 data cache is 16KB and we use blocks of 100 tuples). Relational operators are agnostic about the database schema, and operate on generic tuple structures. We have implemented the following operators: aggregation (sort-based and hash-based), merge join, and table scanners which can apply SARGable predicates.

In our implementation we took care to avoid overhead associated with extra functionality that was not necessary for the purposes of our study. We therefore omitted implementing a parser or optimizer; instead we use precompiled queries. With the same thinking, we opted for a single-threaded implementation. Evaluating a query in parallel is orthogonal to this study and our results trivially extend to multiple CPUs.

Initially, we used the BerkeleyDB[2] storage manager to manage disk files, but eventually opted for a custom-built I/O interface and file storage. BerkeleyDB proved to be a mismatch to our high-performance requirements since reading a page caused multiple copies between user and system space, and there was no built-in prefetching. DB engines commonly use a multithreaded storage manager to handle prefetching and also overlap I/O with computation. We take advantage of the Asynchronous I/O (AIO) interface in Linux 2.6 to implement a non-blocking prefetching mechanism within our single-threaded application. Our AIO interface performs reads at the granularity of an I/O *unit*. Throughout the paper we use I/O units of 128KB per disk. Every time we run the engine we also specify the *depth* of prefetching: how many I/O units are issued at once when reading a file. Disk transfers go directly to memory (i.e., using DMA) without involving the CPU, and without passing through the Linux file cache. There is no buffer pool (since it does not make a difference for sequential accesses). Instead, the AIO interface passes the query a pointer to a buffer containing data from an I/O unit.

Our I/O interface provides a minimum-overhead data path from disk to memory. This allows us to report results that are not biased by a non-efficient implementation. The code consists of about 2,500 lines of C++ (without including the code to create, load, and query data), compiled with gcc, on Linux 2.6, using the libaio library for asynchronous I/O. Our code is publicly available and it can be used as a benchmarking tool for measuring the performance limit of TPC-H style queries, on both row and column data[3].

3. EXPERIMENTAL SETUP

3.1 Workload

For all experiments we use two tables, LINEITEM and ORDERS, based on the TPC-H[4] benchmark specification. We populate the tables with data generated by the official TPC-H toolkit. Our only modifications to the original specifications were the following: First, we use four-byte integers to store all decimal types (we preserve the original values though). Second, for LINEITEM, we use a fixed text type instead of a variable one for the L_COMMENT field, to bring the tuple size to 150 bytes. Finally, for ORDERS, we drop two text fields, and change the size of another one, to bring the tuple size to 32 bytes. The schema of the two tables is shown in Figure 5. The row system uses 152 bytes to store a LINEITEM tuple (the extra 2 bytes are for padding purposes), and 32 bytes to store an ORDERS tuple (same as the tuple width). The column system packs attributes from each column contiguously without any storage overhead.

The rationale behind the choice for the tuple sizes was to construct a "wide" tuple that is larger than a cache line (the L2 cache line for Pentium 4 is 128 bytes), and a "narrow" one, that can fit multiple times within a cache line. Fixing the tuple size provides measurement consistency. TPC-H specifies a ratio of four LINEITEM lines for every ORDERS line. To be able to compare results between the two tables, we use scale 10 for LINEITEM (60M tuples, 9.5GB on disk) and scale 40 for ORDERS (60M tuples, 1.9 GB on disk). TPC-H generated data allow for substantial compression. The compressed versions of the LINEITEM and ORDERS tuples are shown in Figure 5. We apply all three compression techniques described in the previous section.

[2] http://www.sleepycat.com

[3] http://db.csail.mit.edu/projects/cstore/

[4] http://www.tpc.org/tpch/

LINEITEM (150 bytes)			LINEITEM-Z (52 bytes)	
1	L_PARTKEY	int	**1**	*non-compressed*
2	L_ORDERKEY	int	**2Z**	**delta**, 8 bits
3	L_SUPPKEY	int	**3**	*non-compressed*
4	L_LINENUMBER	int	**4Z**	**pack**, 3 bits
5	L_QUANTITY	int	**5Z**	**pack**, 6 bits
6	L_EXTENDEDPRICE	int	**6**	*non-compressed*
7	L_RETURNFLAG	text, 1	**7Z**	**dict**, 2 bits
8	L_LINESTATUS	text, 1	8	*non-compressed*
9	L_SHIPINSTRUCT	text, 25	**9Z**	**dict**, 2 bits
10	L_SHIPMODE	text, 10	**10Z**	**dict**, 3 bits
11	L_COMMENT	text, 69	**11Z**	**pack**, 28 bytes
12	L_DISCOUNT	int	**12Z**	**dict**, 4 bits
13	L_TAX	int	**13Z**	**dict**, 4 bits
14	L_SHIPDATE	int	**14Z**	**pack**, 2 bytes
15	L_COMMITDATE	int	**15Z**	**pack**, 2 bytes
16	L_RECEIPTDATE	int	**16Z**	**pack**, 2 bytes

ORDERS (32 bytes)			ORDERS-Z (12 bytes)	
1	O_ORDERDATE	int	**1Z**	**pack**, 14 bits
2	O_ORDERKEY	int	**2Z**	**delta**, 8 bits
3	O_CUSTKEY	int	**3**	*non-compressed*
4	O_ORDERSTATUS	text, 1	**4Z**	**dict**, 2 bits
5	O_ORDERPRIORITY	text, 11	**5Z**	**dict**, 3 bits
6	O_TOTALPRICE	int	**6**	*non-compressed*
7	O_SHIPPRIORITY	int	**7Z**	**pack**, 1 bit

Figure 5. Table schema. Compressed versions of the tables are shown on the right (compressed attributes are noted with the letter "Z" next to an attribute's number).

3.2 Hardware platform & measurement tools

All experiments are performed on a Pentium 4 3.2GHz, with 1MB L2 cache, 1GB RAM, running Linux 2.6. Unless otherwise noted, all data is read off a software RAID consisting of three SATA disk drives, capable of delivering 180MB/sec bandwidth (60MB/sec per disk). The default configuration of our system is a 128KB I/O unit with prefetch depth of 48 I/O units and tuple block sizes of 100 entries. There is no file-caching in our system, so all data requests are served from the disk.

We use the standard UNIX utility `iostat` to monitor disk activity. All measurements are collected through the PAPI[5] library [16] which provides access to the CPU performance counters, enabling us to measure events at the micro-architectural level (number of L2 cache misses, executed instructions, etc.). More specifically, we use `papiex`[6], which can monitor any binary and report performance counter events either in user mode or system (kernel) mode. The former are events that our application is responsible for, while in user space (executing the code we wrote), and the latter are events that occurred while the CPU executes system calls and OS code. The way we convert these measurements into time breakdowns is described in the next section, next to the results.

4. PERFORMANCE STUDY

There are several factors that impact query performance when comparing column and row stores. We study five such factors:

- Query workload
- Database physical design
- System parameters
- Capacity planning
- System load

We discuss next how each factor may affect performance.

(i) Query workload. The primary advantage of column over row stores is the ability to read from disk exactly the **attributes selected** by a query, thus saving on disk and main memory bandwidth, along with CPU time that is required to read pages from disk. We expect queries that select increasingly more attributes, up to the entire tuple, to exhibit a convergence in the performance of column and row stores. For queries with decreased **selectivity** (few qualifying tuples[7]), we expect the CPU time to decrease in both column and row stores. The end-to-end performance of a query depends on the **cost of other operators** involved in the query plan, not just the table scans. Given an identical operator implementation in column and row stores, we expect the relative end-to-end performance difference to decrease as the bulk of the query cost moves to operators other than scans (we discuss this in Section 5).

(ii) Database physical design. The **tuple width** in a table is specific to a database schema, but it can change (to be narrower) during the physical design phase, using vertical partitioning or materialized view selection. During the design phase, the database administrator (or an automatic tool) can also decide on whether **compression** schemes will be used and how. Both narrower (in the case of row stores) and compressed tuples remove pressure from disk and main memory bandwidth requirements. They also reduce CPU time spent executing disk read instructions (CPU system time); for compressed tuples, we expect the CPU user time to slightly increase due to extra CPU instructions required by decompression.

(iii) System parameters. A parameter that may differ across implementations is the **prefetch** unit size in large scans (how much data the storage manager reads ahead). As we show later, a sufficiently large prefetch buffer results into good performance for both column and row stores for sequential scans, whereas small values negatively affect both systems. A small prefetch buffer means that the disks spend more time seeking (moving the disk heads) to read from multiple tables than actually reading, resulting into poor disk utilization.

(iv) Capacity planning. The database administrator can specify **how many disks** a table will be striped over. When there are more than one CPUs available, the DBMS decides **how many CPUs** will be used to evaluate the query in parallel (a parameter known as "degree of parallelism" or DOP; the user may also specify what the DOP is for a given query). Different ratios of CPUs per disk for a given query may have different effects in column stores than in row stores, depending on what the bottleneck is (disk, CPU, or memory) for any given configuration.

[5] http://icl.cs.utk.edu/papi/

[6] http://icl.cs.utk.edu/~mucci/papiex/

[7] Throughout the paper we adopt the following convention: *decreased* or *lower* selectivity corresponds to a lower percentage of qualified tuples.

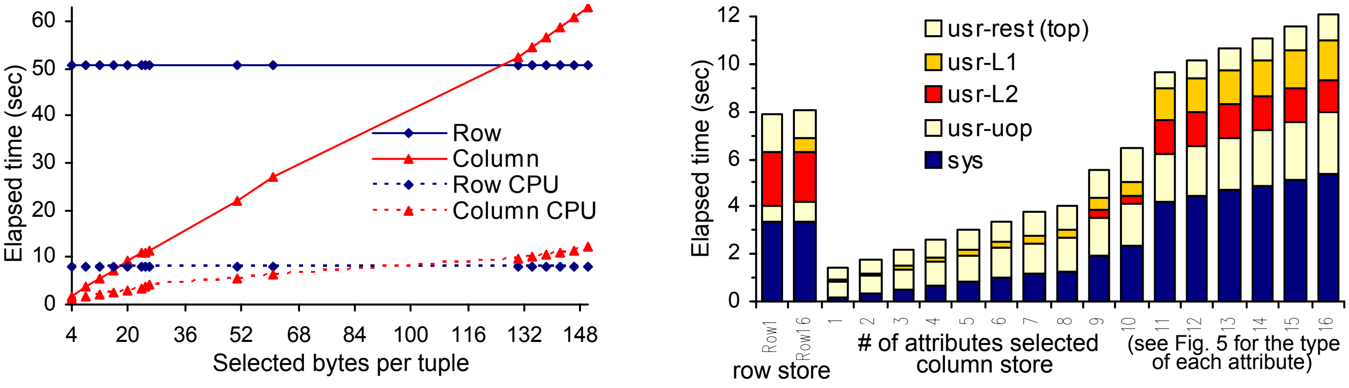

Figure 6. Baseline experiment (10% selectivity, LINEITEM). Left: Total elapsed time (solid lines) and CPU time (dashed lines) for column and row store. The total elapsed time is equal to I/O time since CPU time is overlapped. X-axis is spaced by the width of selected attributes. Right: CPU time breakdowns. The first two bars are for row store, the rest are for column store.

(v) System load. The system load (**disk utilization** and **CPU utilization**) can significantly affect a single query's response time. Obviously, a disk-bound query will see a big increase in the response time if it is competing with other disk-bound queries. Competing disk and CPU traffic may again have different effects in column stores than in row stores.

We base all experiments on a variant of the following query:

select A1, A2 … **from** TABLE
where *predicate* (A1) *yields* variable *selectivity*

Since the number of selected attributes per query is the most important factor, we vary that number on the X-axis through all experiments. Table 1 summarizes the parameters considered, what the expected effect is in terms of time spent on disk, memory bus and CPU, and which section discusses their effect.

Table 1: Expected performance trends in terms of elapsed disk, memory transfer, and CPU time (arrows facing up mean increased time), along with related experimentation sections.

parameter	Disk	Mem	CPU	section
selecting more attributes (column store **only**)	↑	↑	↑	**4.1**
decreased selectivity			↓	**4.2**
narrower tuples	↓	↓	↓	**4.3**
compression	↓	↓	↓↑	**4.4**
larger prefetch	↓			**4.5**
more disk traffic	↑			**4.5**
more CPUs / more Disks	↓		↓	**5**

4.1 Baseline experiment

select L1, L2 … **from** LINEITEM
where *predicate* (L1) *yields* 10% *selectivity*

As a reminder, the width of a LINEITEM tuple is 150 bytes, it contains 16 attributes, and the entire relation takes 9.5GB of space. Figure 6 shows the elapsed time for the above query for both row and column data. The graph on the left of the figure shows the total time (solid lines), and the CPU time separately (dashed lines) as we vary the number of selected attributes on the x-axis. Both systems are I/O-bound in our default configuration (1 CPU, 3 disks, no competition), and therefore the total time reflects the time it takes to retrieve data from disk. Both systems are designed to overlap I/O with computation (as discussed in Section 2). As expected, the row store is insensitive to projectivity (since it reads all data anyway), and therefore its curve remains flat. The column store, however, performs better most of the time, as it reads less data. Note that the x-axis is spaced by the width of the selected attributes (e.g., when selecting 8 attributes, the column store is reading 26 bytes per LINEITEM row, whereas for 9 attributes, it reads 51 bytes — see Figure 5 for detailed schema information).

The "crossover" point that the column store starts performing worse than the row store is when selecting more than 85% of a tuple's size. The reason it performs worse in that region is that it makes poorer utilization of the disk. A row store, for a single scan, enjoys a full sequential bandwidth. Column stores need to seek between columns. The more columns they select, the more time they spend seeking (in addition to the time they spend reading the columns). Our prefetch buffer (48 I/O units) amortizes some of the seek cost. A smaller prefetch buffer would lower the performance of the column store in this configuration, but additional disk activity from other processes would make the crossover point to move all the way to the right. We show these two scenarios later, in Section 4.5.

While this specific configuration is I/O-bound, it still makes sense to analyze CPU time (dashed lines in left graph of Figure 6), as it can affect performance in CPU-bound configurations, or when the relations are cached in main memory. The left graph of Figure 6 shows the total CPU time for both systems. We provide a time breakdown of CPU costs in the graph on the right part of Figure 6. The first two bars correspond to the row store, selecting 1 and 16 attributes (the two ends in our experiment). The rest of the bars belong to the column store, selecting from 1 to 16 attributes. The height of each stacked bar is the total CPU time in seconds. The bottom area (dark color), is the time (in sec) spent in system mode. This is CPU time spent while Linux was executing I/O requests and we do not provide any further break-down of that. For the row store, the system time is the same regardless of the number of selected attributes. For the column store it keeps

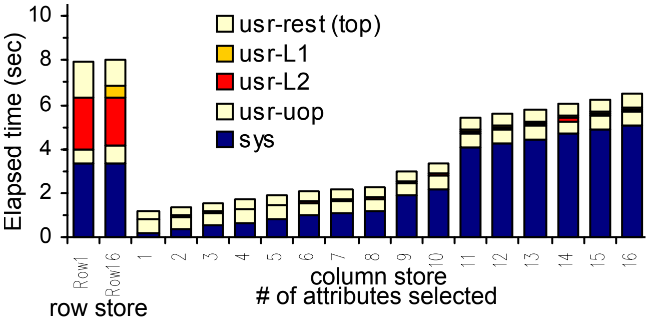

Figure 7. Changing selectivity to 0.1% (LINEITEM table).

increasing as the system performs more I/O. The jumps in bars #10, #11, #12, are due to the larger, string attribute types that are being added to the selection list. Note that the column store ends up spending more system time than the row store when selecting all attributes, due to more work needed by the Linux scheduler to handle read requests for multiple files.

The rest of the fields in the stacked bars are user time. Moving upwards, the next field (usr-uop) is the minimum time the CPU could have possibly spent executing our code. Pentium 4 can execute a maximum of 3 micro-operations (uops) per cycle. By measuring the actual number of uops (and dividing by 3), we compute the minimum time. In reality, CPUs never achieve this time, due to several stall factors [14], however, we can treat this time as a metric of the work assigned to a CPU, and how far off the CPU is from a perfect execution. The next two fields (usr-L2 and usr-L1), is the time it took for data to move from main memory to L2, and from L2 to L1 cache, correspondingly.

We carefully compute L2 delays for each access pattern, taking into consideration the hardware L2 prefetcher (see also discussion in Section 2.1.2). For sequential memory access patterns, the memory bus in our system can deliver 128 bytes (one L2 cache line) every 128 CPU cycles. The time spent fetching sequential data into L2 is overlapped with the usr-uop field (mentioned above). Whenever the CPU tries to access a non-prefetched cache line, the stall time is 380 cycles (measured time for random memory access time in our system). The usr-L2 area shown in the stacked bar of Figure 6 is the minimum time the CPU was stalled waiting for data to arrive in the L2, after subtracting any overlaps with actual CPU computation. The usr-L1 field, on the other hand, is the *maximum possible* time the CPU could have been stalled to L1 cache misses. In reality, that component is much smaller due to out-of-order execution of modern CPUs [14]. Finally, the remaining, light-colored field, is the amount of time the CPU was active in user mode. A number of other factors can contribute to the remaining time (branch mispredictions, functional unit stalls, etc.).

There are two observations to make for the CPU-time graph of Figure 6. Column stores require increasingly more CPU work than a row store as more attributes are selected from a relation. Their average performance, however, is marginally better due to less work in system mode, and less time spent in main-memory transfers in user mode. The second observation is that the type of selected attributes is crucial for column stores. Once the query expands enough to include the three strings of LINEITEM (#9, #10, #11), we suddenly see a significant L2/L1 component that will remain the same as the query goes back to adding integers (#12 - #16). These observations lead us to two questions and to our next experiments: what happens when the column store has less work for each additional column (decreased selectivity: Section 4.2), and what happens when both systems access narrower structures (Section 4.3 and Section 4.4).

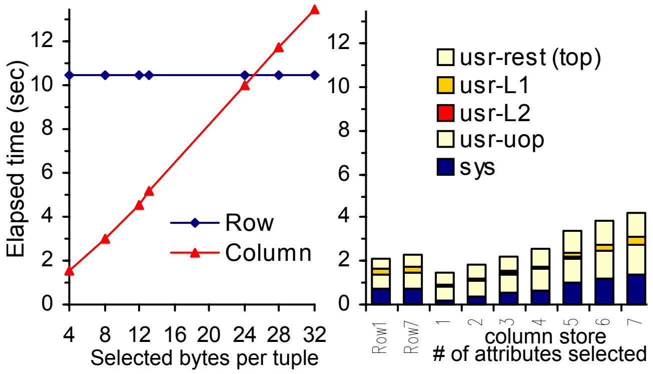

Figure 8. 10% selection query on ORDERS (narrow tuples).

4.2 Effect of selectivity

select L1, L2 … **from** LINEITEM
where *predicate* (L1) *yields* 0.1% *selectivity*

Selecting fewer tuples through a very selective filter has no effect on I/O performance. Figure 7 shows the CPU time breakdown in the same format as Figure 6. As expected, system time remains the same (since it depends on I/O). The row store also remains the same, since it has to examine every tuple in the relation to evaluate the predicate, and so, memory delays continue to dominate. The column store, however, behaves entirely different. Selecting additional attributes adds negligible CPU work, since the different scan nodes (other than the first) process on average only one out of every 1000 values. The memory delay of transferring large strings (attributes #9, #10, #11) is no longer an issue since their cost is much less than the cost of evaluating predicate L1.

In general, for decreased selectivity of the first attribute in the query list, the pure CPU computation time (usr-uop) of column stores is very close to a row store, regardless of the number of selected attributes. However, as selectivity increases towards 100%, each additional column scan node contributes an increasing CPU component, causing the crossover point to move towards the left. Note that this behavior stems directly from the pipelined column scanner architecture used in this paper (shown in Figure 4), where each scan node is driven by a separate value iterator. One optimization that a column system could employ is to utilize a non-pipelined, single-iterator scanner. The way such a scanner works is the following. It first fetches disk pages from all scanned columns into memory. Then, it uses memory offsets to access all attributes within the same row, iterating over entire rows, similarly to a row store. This architecture is similar to PAX [4] and MonetDB [7]. Since such an optimization is out of the scope of this performance study, we do not further discuss it.

4.3 Effect of narrow tuples

select O1, O2 … **from** ORDERS
where *predicate* (O1) *yields* 10% *selectivity*

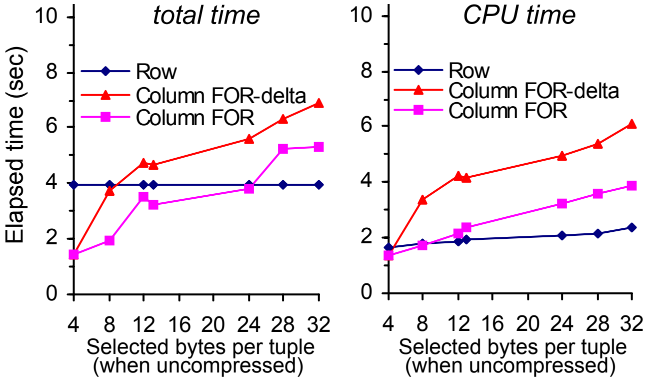

Figure 9. Selection query on ORDERS-Z (compressed), total time (left) and CPU time (right). The column system uses two different compression schemes for one of the attributes.

We switch tables and use ORDERS (32 bytes wide tuple, 7 attributes). Figure 8 contains the results (total time on the left, CPU breakdowns on the right). Both systems are still I/O-bound, and so we see the same behavior as before in total time. The CPU behavior, however, is different. First, the system time is a smaller percentage of the overall CPU time; the reason is that we still scan the same number of tuples as before (ORDERS have the same cardinality as LINEITEM), but perform less I/O per tuple. Second, the pure computation (usr-uop) is almost the same as before, however, the memory-related delays are no longer visible in either system: the main memory bandwidth surpasses the rate at which the CPU can process data. In a memory-resident dataset, for this query, column stores would perform worse than row stores no matter how many attributes they select. However, if we were to use decreased selectivity (see CPU usr-uop time for column data in Figure 7), both systems would perform similarly.

4.4 Effect of compression

select Oz1, Oz2 … **from** ORDERS-Z
where *predicate* (Oz1) *yields* 10% *selectivity*

In studying the effects of compression, we initially ran a selection query on LINEITEM-Z. However, the results for total time did not offer any new insights (the LINEITEM-Z tuple is 52 bytes, and we already saw the effect of a 32-byte wide tuple). We therefore only show results with the ORDERS-Z table (12 bytes wide) in Figure 9. The graph on the left shows the total time whereas the graph on the right shows CPU time. The x-axis is spaced on the uncompressed size of selected attributes. For this experiment we show results for two different compression schemes for attribute #2 of ORDERS, FOR and FOR-delta (see Section 2.2.1).

In this query, the column store becomes CPU-bound (the row store is still I/O-bound), and therefore the crossover point moves to the left. Note that the slightly irregular pattern for total time in the left graph of Figure 9 is due to imperfect overlap of CPU and I/O time. Both systems exhibit reduced system times. The row store, for the first time so far, shows a small increase in user CPU time when going from selecting one attribute to all seven (right graph in Figure 9). The reason for that is the cost of decompression. The column store exhibits a peculiar behavior in the CPU time of the FOR-delta curve, in that there is a sudden jump when selecting the second attribute. FOR-delta requires reading all values in the page to perform decompression. Plain FOR compression for that attribute (storing the difference from a base value instead of the previous attribute) requires more space (16 bits instead of 8), but is computationally less intensive. This is apparent in the right graph of Figure 9. This experiment does not necessarily suggest that plain FOR is a better compression scheme than FOR-delta. If our system was disk-constrained (for example, if we used one disk instead of three), then the I/O benefits would offset the CPU cost of that specific compression scheme.

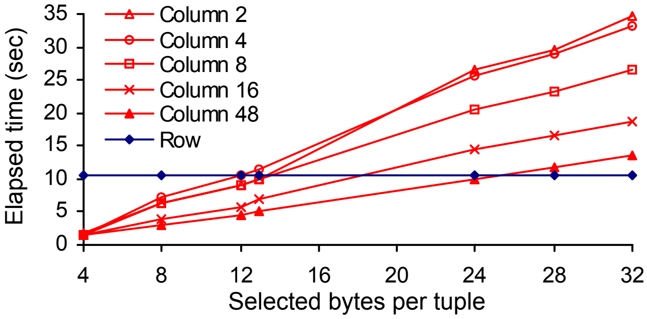

Figure 10. Varying the prefetch size when scanning ORDERS. The row system is not affected by prefetching in this scenario.

4.5 Effect of prefetch size, competing traffic

select O1, O2 … **from** ORDERS
where *predicate* (O1) *yields* 10% *selectivity*

We repeat the experiment from Figure 8, this time varying the prefetch depth. So far we have been using a prefetch depth of 48 I/O units (each unit is 128KB per disk). Figure 10 shows, in addition to 48 units, results for prefetch depth of 2, 4, 8, and 16 units. Since there is only a single scan in the system, prefetch depth does not affect the row system. The column system, however, performs increasingly worse as we reduce prefetching, since it spends more time seeking between columns on disk instead of reading. It therefore makes sense to aggressively use prefetching in a column system. However, the same is true in the general case for row systems as well, as we show next.

Figure 11 shows three graphs where both the row and column system use a prefetch size of 48, 8, and 2 correspondingly, in the presence of a concurrent competing scan. The competing scan is issued by a separate process that executes a row-system scan on a different file on disk (LINEITEM). In each scenario, we matched the prefetch size of the competing process to the prefetch size of the systems under measurement, to present the disk controller with a balanced load.

As Figure 11 shows, the column system (solid-triangle line) outperforms the row system in all configurations. This contradicts our expectations that a column system selecting all columns in a relation would perform the same as a row system in the presence of competing disk traffic (since both systems should spend the same time reading off disk and waiting on disk-head seeks). It turns out that a column system benefits from the fact it employs several scans. When the first data for column #1 arrives from disk, the CPU is moving on to column #2 while the disk still serves column #1. At that point, column #2 submits its disk request and the CPU is halted until all disk requests for column #1 are served and the first data from column #2 arrives. Being

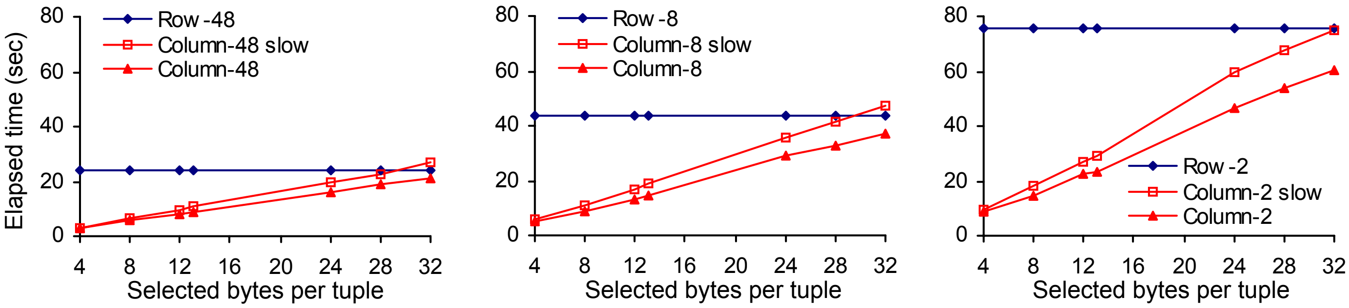

Figure 11.Repeating previous experiment for prefetch size of 48, 8, and 2 units, this time in the presence of another concurrent scan. See text for an explanation of "slow" curve.

one step ahead allows the column system to be more aggressive in its submission of disk requests, and, in our Linux system, to get favored by the disk controller. As a reference, we implemented a version of the column system that waits until a disk request from one column is served before submitting a request from another column. The results ("slow" line in Figure 11) for this system are now closer to our initial expectations.

5. ANALYSIS

The previous section gave us enough experimental input to guide the construction of a simple set of equations to predict relative performance differences between row and column stores under various configurations. We are primarily interested in predicting the rate of tuples/sec at which row and column systems will operate for a given query and given configuration. We summarize in Table 2 the parameters we are going to use in our analysis along with what different configurations these parameters can model.

Our analysis focuses on a setting where scan nodes continuously read input tuples which are then passed on to other relational operators in a pipelined fashion. We assume (as in our implementation) that CPU and I/O are well overlapped. For simplicity, we do not model disk seeks (we assume that the disks read sequentially the majority of the time). The rate (tuples/sec) at which a query can process its input is simply the minimum of the rate R the disks can provide data and the rate the CPUs can process these data:

$$R = \text{MIN}(R_{\text{DISK}}, R_{\text{CPU}}) \quad \textbf{(1)}$$

Table 2: Summary of parameters used in the analysis.

parameter	what can it model
SizeFile	various database schemas
TupleWidth	
MemBytesCycle	various speeds for the memory bus
f	number of attributes selected by a query (projection)
I	CPU work of each operator (can model various selectivities for scanners, or various decompression schemes)
cpdb	more/fewer disks more/fewer CPUs competing traffic for disk / CPU

Throughout the analysis, "DISK" and "CPU" refer to all disks and all CPUs made available for executing the query. Parallelizing a query is orthogonal to our analysis. If a query can run on three CPUs, for example, we will treat it as one that has three times the CPU bandwidth as a query that runs on a single CPU.

Disk analysis. We model the disk rate R_{DISK} (in tuples/sec) as the sum of rates of all files read, weighted by the size of a file (in bytes):

$$R_{\text{DISK}} = R^{\text{File1}} \cdot \frac{\text{SizeFile1}}{\text{SizeFileALL}} + R^{\text{File2}} \cdot \frac{\text{SizeFile2}}{\text{SizeFileALL}} + \ldots \quad \textbf{(2)}$$

For example, in the case of a merge-join, if File1 is 1GB and File2 is 10GB, then the disks process on average one byte from File1 for every ten bytes from File2. The individual file rates are defined as:

$$R^{\text{File}_N} = \frac{\text{DiskBW}}{\text{TupleWidth}_N}$$

DiskBW is simply the available bandwidth from the disks in bytes/sec, which we divide by the width of the tuples to obtain the rate in tuples/sec. DiskBW is always the full sequential bandwidth (we assume large prefetching buffers that minimize time spent in disk seeks, as shown in the previous section). Note that:

$$\text{SizeFile}_i = N_i \cdot \text{TupleWidth}_i$$

where N_i is the cardinality of relation i.

We can now rewrite (2) as:

$$R_{\text{DISK}} = \text{DiskBW} \cdot \frac{N_1 + N_2 + \ldots}{\text{SizeFileALL}} \quad \textbf{(3)}$$

For a column store, we can derive a similar equation to the one above:

$$R^{\text{Columns}}_{\text{DISK}} = \text{DiskBW} \cdot \frac{N_1 f_1 + N_2 f_2 + \ldots}{\text{SizeFileALL}} \quad \textbf{(4)}$$

where *f1, f2,* etc. are the factors by which a regular row tuple is larger than the size of the attributes needed by a query. For example, if a column system needs to read only two integers (8 bytes) from ORDERS (32 bytes), the factor *f* is 4 (= 32 / 8).

CPU analysis. To model CPU rate, we assume a cascaded connection of all relational operators and scanners. If an operator processes Op_1 tuples/sec, and is connected to another operator with rate Op_2, which in turn is connected to an operator with rate Op_3 and so on, the overall CPU rate R_{CPU} is:

$$\frac{1}{R_{\mathrm{CPU}}} = \frac{1}{\mathrm{Op}_1} + \frac{1}{\mathrm{Op}_2} + \frac{1}{\mathrm{Op}_3} + \ldots \quad \textbf{(5)}$$

The above formula resembles the formula for computing the *equivalent resistance* of a circuit where resistors are connected in parallel; we adopt the same notation (two parallel bars) and rewrite (5) (this time including the rates for the various scanners):

$$R_{\mathrm{CPU}} = \mathrm{Op}_1 \parallel \mathrm{Op}_2 \parallel \ldots \parallel \mathrm{Scan}_1 \parallel \mathrm{Scan}_2 \parallel \ldots \quad \textbf{(6)}$$

As an example, consider one operator processing 4 tuples/sec, connected to an operator that processes 6 tuples/sec. The overall rate of tuple production in the system is:

$$\mathrm{Op}_1 \parallel \mathrm{Op}_2 = \frac{\mathrm{Op}_1 \cdot \mathrm{Op}_2}{\mathrm{Op}_1 + \mathrm{Op}_2}$$

or 2.4 tuples/sec.

The rate of an operator is approximated as:

$$\mathrm{Op} = \frac{\mathrm{clock}}{\mathrm{I}_{\mathrm{Op}}} \quad \textbf{(7)}$$

where *clock* is the available cycles per second from the CPUs (e.g., for our single-CPU machine, *clock* is 3.2 billion cycles per second). I_{Op} is the total number of CPU instructions it takes the operator to process one tuple. Note that we approximate the rate by assuming 1 CPU cycle per instruction. If the actual ratio is available, we can replace it in the above formula.

To compute the rate of a scanner, we have to take into consideration the CPU-system time, the CPU-user time, and the time it takes the memory to deliver tuples to the L2 cache. We treat CPU-system and CPU-user as two different operators. Further, we compute CPU-user rate as the minimum of the pure computation rate and the rate the memory can provide tuples to the L2 cache. The latter is equal to the memory bandwidth divided by the tuple width. We compute memory bandwidth as: *clock* times how many bytes arrive per CPU cycle (*MemBytesCycle*).

Therefore we can write the rate *Scan* of a scanner as:

$$\mathrm{Scan} = \frac{\mathrm{clock}}{\mathrm{I}_{\mathrm{system}}} \parallel \mathrm{MIN}\left(\frac{\mathrm{clock}}{\mathrm{I}_{\mathrm{user}}}, \frac{\mathrm{clock} \cdot \mathrm{MemBytesCycle}}{\mathrm{TupleWidth}}\right) \quad \textbf{(8)}$$

Speedup of columns over rows. We are now ready to compute the speedup of columns over rows by dividing the corresponding rates (using (1), (3), (4), (6), (7) and (8)):

$$\textbf{Speedup} = \frac{MIN\left\{\frac{N_1 f_1 + N_2 f_2 + \ldots}{\mathrm{SizeFileALL}},\ \textbf{cpdb} \cdot \left(\frac{1}{I_{\mathrm{op}}} \parallel \frac{1}{I_{\mathrm{ScanC}}} \parallel \ldots\right)\right\}}{MIN\left\{\frac{N_1 + N_2 + \ldots}{\mathrm{SizeFileALL}},\ \textbf{cpdb} \cdot \left(\frac{1}{I_{\mathrm{op}}} \parallel \frac{1}{I_{\mathrm{ScanR}}} \parallel \ldots\right)\right\}}$$

To derive the above formula we divided all members by DiskBW, and replaced the following quantity:

$$\textbf{cpdb} = \frac{\mathrm{clock}}{\mathrm{DiskBW}}$$

cpdb (*cycles per disk byte*) combines into a single parameter the available disk and CPU resources for a given configuration. It shows how many (aggregate) CPU cycles elapse in the time it takes the disks to sequentially deliver a byte of information. For example, the machine used in this paper (one CPU, three disks) is rated at 18 cpdb. By operating on a single disk, cpdb rating jumps to 54.

Parameters such as competing traffic and number of disks/CPUs can be modeled through cpdb rating. Since competing CPU traffic fights for cycles, the cpdb rating for a given query drops. On the other hand, competing disk traffic causes cpdb to increase. Looking up trends in CPU [3] and disk[8] speed, we find that, for a single CPU over a single disk, cpdb has been slowly growing, from 10 in 1995, to 30 in 2005. With the advent of multicore chips, we expect cpdb to grow faster. When calculating the cpdb rating of an arbitrary configuration, note that disk bandwidth is limited by the maximum bandwidth of the disk controllers.

We use the speedup formula to predict relative performance of column systems over row systems for various configurations by changing the cpdb rating. In disk-bound systems (when the disk rate is lower than the CPU rate), column stores outperform row stores by the same ratio as the total bytes selected over the total size of files. In CPU-bound systems, either system can be faster, depending on the cost of the scanners. In the previous section we saw some conditions (low selectivity, narrow tuples) where row stores outperform column stores. Note that a high-cost relational operator lowers the CPU rate, and the difference between columns and rows in a CPU-bound system becomes less noticeable. The formula can also be used to examine whether a specific query on a specific configuration is likely to be I/O or CPU bound.

The graph shown in the introduction (Figure 2), is constructed from the speedup formula, filling up actual CPU rates from our experimental section. In that figure we use a 50% projection of the attributes in the relation, and 10% selectivity. The graph shows that row stores outperform column stores only in very limited settings.

6. RELATED WORK

The decomposition storage model (DSM) [8] was one of the first proposed models for a column oriented system designed to improve I/O by reducing the amount of data needed to be read off disk. It also improves cache performance by maximizing inter-record spatial locality [6]. DSM differs from the standard row-store N-ary storage model (NSM) by decomposing a relation with *n* attributes into *n* vertically partitioned tables, each containing one column from the original relation and a column containing TupleIDs used for tuple reconstruction. The DSM model attempts to improve tuple reconstruction performance by maintaining a clustered index on TupleID, however, complete tuple reconstruction remained slow. As a result, there have been a variety of optimizations and hybrid NSM/DSM schemes proposed in subsequent years: (a) partitioning of relations based on how often attributes appear together in a query [9], (b) using different storage models for different mirrors of a database, with queries that perform well on NSM data or DSM data being sent to the appropriate mirror [19], (c) optimizing the choice of horizontal and vertical partitioning given a database workload [2].

[8] http://www.hitachigst.com/hdd/technolo/overview/chart16.html

Recently there has been a reemergence of pure vertically partitioned column-oriented systems as modern computer architecture trends favor the I/O and memory bandwidth efficiency that column-stores have to offer [7][21]. PAX [4] proposes a column-based layout for the records within a database page, taking advantage of the increased spatial locality to improve cache performance, similarly to column-based stores. However, since PAX does not change the actual contents of the page, I/O performance is identical to that of a row-store. The Fates database system [20] organizes data on the disk in a column-based fashion and relies on clever data placement to minimize the seek and rotational delays involved in retrieving data from multiple columns.

While it is generally accepted that generic row-stores are preferred for OLTP workloads, in this paper we complement recent proposals by exploring the fundamental tradeoffs in column and row oriented DBMS on read-mostly workloads. Ongoing work [12] compares row- and column-based pages using the Shore storage manager. The authors examine 100% selectivity and focus on 100% projectivity, both of which are the least favorable workload for pipelined column scanners, as we showed earlier.

7. CONCLUSIONS

In this paper, we compare the performance of read-intensive column- and row-oriented database systems in a controlled implementation. We find that column stores, with appropriate prefetching, can almost always make better use of disk bandwidth than row-stores, but that under a limited number of situations, their CPU performance is not as good. In particular, they do less well when processing very narrow tuples, use long projection lists, and apply non-selective predicates. We use our implementation to derive an analytical model that predicts query performance with a particular disk and CPU configuration and find that current architectural trends suggest column stores, even without other advantages (such as the ability to operate directly on compressed data [1] or vectorized processing [7]) will become an even more attractive architecture with time. Hence, a column oriented database seems to be an attractive design for future read-oriented database systems.

8. ACKNOWLEDGMENTS

We thank David DeWitt, Mike Stonebraker, and the VLDB reviewers for their helpful comments. This work was supported by the National Science Foundation under Grants 0520032, 0448124, and 0325525.

9. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. “Integrating Compression and Execution in Column-Oriented Database Systems.” In *Proc. SIGMOD*, 2006.

[2] S. Agrawal, V. R. Narasayya, B. Yang. “Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design.” In *Proc. SIGMOD*, 2004.

[3] A. Ailamaki. “Database Architecture for New Hardware.” Tutorial. In *Proc. VLDB*, 2004.

[4] A. Ailamaki, D. J. DeWitt, et al. “Weaving Relations for Cache Performance.” In *Proc. VLDB*, 2001.

[5] A. Ailamaki, D. J. DeWitt, at al. “DBMSs on a modern processor: Where does time go?” In *Proc. VLDB*, 1999.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. “Database Architecture Optimized for the New Bottleneck: Memory Access.” In *Proc. VLDB*, 1999.

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

[8] A. Copeland and S. Khoshafian. “A Decomposition Storage Model.” In *Proc. SIGMOD*, 1985.

[9] D. W. Cornell and P. S. Yu. “An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases.” In *IEEE Transactions on Software Engineering 16(2): 248-258*, 1990.

[10] J. Goldstein, R. Ramakrishnan, and U. Shaft. “Compressing Relations and Indexes.” In *Proc. ICDE*, 1998.

[11] G. Graefe and L. D. Shapiro. “Data compression and database performance.” In *Proc. ACM/IEEE-CS Symposium on Applied Computing, Kansas City, MO*, 1991.

[12] A. Halverson, J. L. Beckmann, J. F. Naughton, and D. J. DeWitt. “A Comparison of C-Store and Row-Store in a Common Framework.” Technical Report, *University of Wisconsin-Madison, Department of Computer Sciences, TR1566*, 2006.

[13] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. “QPipe: A Simultaneously Pipelined Relational Query Engine.” In *Proc. SIGMOD*, 2005.

[14] J. L. Hennessy, D. A. Patterson. “Computer Architecture: A Quantitative Approach.” 2nd ed, Morgan-Kaufmann, 1996.

[15] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. “Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads.” In *Proc. ISCA-25*, 1998.

[16] P. J. Mucci, S. Browne, C. Deane, and G. Ho. “PAPI: A Portable Interface to Hardware Performance Counters.” In *Proc. Department of Defense HPCMP Users Group Conference*, Monterey, CA, June 1999.

[17] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. “Block Oriented Processing of Relational Database Operations in Modern Computer Architectures.” In *Proc. ICDE*, 2001.

[18] M. Poss and D. Potapov. “Data Compression in Oracle.” In *Proc. VLDB*, 2003.

[19] R. Ramamurthy, D. J. DeWitt, and Q. Su. “A Case for Fractured Mirrors.” In *Proc. VLDB*, 2002.

[20] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. “Clotho: Decoupling memory page layout from storage organization.” In *Proc. VLDB*, 2004.

[21] 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 *Proc. VLDB*, 2005.

[22] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. “The Implementation and Performance of Compressed Databases.” In *SIGMOD Rec., 29(3):55–67*, Sept. 2000.

[23] J. Zhou and K. A. Ross. “Buffering Database Operations for Enhanced Instruction Cache Performance.” In *Proc. SIGMOD*, 2004.

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