Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases

Craig Chasseur
University Of Wisconsin
chasseur@cs.wisc.edu

Jignesh M. Patel
University Of Wisconsin
jignesh@cs.wisc.edu

ABSTRACT

Existing main memory data processing systems employ a variety of storage organizations and make a number of storage-related design choices. The focus of this paper is on systematically evaluating a number of these key storage design choices for main memory analytical (i.e. read-optimized) database settings. Our evaluation produces a number of key insights: First, it is always beneficial to organize data into self-contained memory blocks rather than large files. Second, both column-stores and row-stores display performance advantages for different types of queries, and for high performance both should be implemented as options for the tuple-storage layout. Third, cache-sensitive B+-tree indices can play a major role in accelerating query performance, especially when used in a block-oriented organization. Finally, compression can also play a role in accelerating query performance depending on data distribution and query selectivity.

Source Code: At `https://www.cs.wisc.edu/quickstep`

1. INTRODUCTION

Dropping DRAM prices and increasing memory densities have now made it possible to economically build high performance analytics systems that keep their data in main memory all (or nearly all) the time. There is now a surge of interest in main memory database management systems (DBMSs), with various research and commercial projects that target this setting [7, 8, 11, 13, 17, 22, 25, 33, 35].

In this paper, we present results from an experimental survey/evaluation of various storage organization techniques for a main memory analytical data processing engine, i.e. we focus on a read-optimized database setting. We note that the focus of this paper is on *experimental evaluation* of a number of existing storage organization techniques that have been used in a variety of DBMS settings (many in traditional disk-based settings), but the central contribution of this paper is to investigate these techniques for main memory analytic data processing. To the best of our knowledge such a study has not been conducted. In addition this paper also makes available an experimental platform for the community to use to design and evaluate other storage organization techniques for read-optimized main memory databases.

Designers of read-optimized main memory database storage engines have to make a number of design decisions for the data storage organization. Some systems use column-stores for the data representation (e.g. [11]), whereas others use row-stores (e.g. [25]). The use of indexing is not generally well understood or characterized in this setting. Other questions such as to whether it is advantageous to store the data for a given relation in large (main memory) "files" or segments, or to break up the data into traditional pages (as is done for disk-based systems), or to choose some intermediate data-partitioning design, are also not well characterized. In this paper we use an experimental approach to consider and evaluate various storage-related design tradeoffs.

Our approach for this research is to build a prototype main memory storage engine with a flexible architecture, which then allows us to study the cross section of storage organization designs that is produced by examining the following dimensions: a) large memory "files" vs. self-contained memory blocks, b) row-store vs. column-store, c) indexing vs. no indexing, d) compression vs. no compression. We have conducted a comprehensive study of the storage design space along these axes and arrived at a number of interesting findings. For example, we find that a novel block-based storage organization outperforms file-based organization at both query time and load time, and that cache-sensitive index structures have a major role to play in accelerating query performance in this main memory setting.

Overall, the key contributions of this paper are as follows: First, this paper systematically characterizes the design space for main memory database physical storage organizations using the four dimensions listed above.

Second, we present a flexible storage manager design that facilitates comparing the different storage organization choices. This storage manager has an interesting block-based design that allows for a very flexible internal organization. This framework allows direct comparison of a number of storage alternatives, and we hope it serves as a platform for other researchers to improve on our methods and perhaps design and evaluate other storage organization techniques.

Third, we conduct a rigorous, comprehensive experimental evaluation to characterize the performance of selection-based analytic queries across the design space, providing a complete view of the various storage organization alternatives for main-memory data processing engines. Our study reveals several interesting experimental findings, including that both row-stores and column-stores have performance sweet spots in this setting, and can coexist in our storage manager design. We also find that indexing and compression continue to play an important role for main memory storage, and are often required for high performance.

Finally, this paper allows designers of main memory storage engines to make informed decisions about tradeoffs associated with different storage organizations, such as figuring out what they leave on the table if they use a pure column-store vs. supporting both row-stores and column-stores.

We note that to keep this work focused on the core storage management aspect (as opposed to query evaluation algorithms, query optimization, etc.), our evaluation largely covers single relational access plans, which form the building blocks for more complex query evaluation methods. Furthermore, previous work on main memory analytical databases has emphasized the importance of these simple scan-based access methods [5, 11, 31]. This approach allows us to focus on key storage organization issues. We recognize that more complex query processing mechanisms can interact with the storage engine in interesting ways, but they still need fast support for basic selection operations, and currently there isn't a clear consensus on how to best build a main memory query processing engine; for example, even single join methods are being rethought in this setting [3, 4, 6, 18].

The remainder of this paper is organized as follows: In Section 2, we describe the Quickstep Storage Manager, our platform for read-optimized main memory storage experiments. In Section 3, we describe the experimental setup that is used in this paper, and Section 4 presents the experimental results. Related work is discussed in Section 5, and Section 6 contains our concluding remarks.

2. QUICKSTEP STORAGE MANAGER

Quickstep is a new SQL-based main memory relational database management system (DBMS) that we are currently developing. Quickstep aims to deliver high database performance on modern and future hardware by exploiting large main memories, fast on-die CPU caches, and highly parallel multi-core CPUs. The Quickstep Storage Manager (SM) is used as the experimental platform in this paper to systematically evaluate storage organizations for read-optimized main memory DBMSs. This Storage Manager has a flexible architecture that naturally allows direct comparisons of various storage organization choices within the same framework.

2.1 The Quickstep SM Architecture

The basic unit of organization in the Quickstep SM is a *storage block*. A table is typically stored across many blocks, though each block belongs to one and only one table.

From the perspective of the rest of the system, blocks are opaque units of storage that support a select operation. A select operation is specified as a projection list, a predicate, and a destination. The destination is some (temporary) block(s) where the results should be materialized. Blocks are independent and self-contained, and each one can decide for itself, based on its own internal organization, how to most efficiently evaluate the predicate and perform the selection. Blocks support update and delete operations that are also described logically and performed entirely within the storage system.

Like System R [29], the Quickstep SM allows pushing down single-table predicates to the SM, but Quickstep goes beyond System R in that Quickstep's expression and predicate framework allows *any* arbitrary predicate to be evaluated entirely inside the storage system, so long as it only references attributes of a single table (i.e. we are not limited to a narrow definition of "sargable" predicates). Like MonetDB [7] and Vectorwise [35], the Quickstep SM does away with the traditional tuple-at-a-time cursor interface, and materializes the complete result of a selection-projection operation on a block all at once into other in-memory block(s).

The Quickstep SM organizes the main memory into a large pool of memory "slots" that are occupied by storage blocks. A given table's tuples are stored in a number of blocks, and blocks may be different sizes (modulo the size of a slot), and have different physical layouts internally. The storage manager is responsible for creating and deleting blocks, and staging blocks to persistent storage (SSD, disk, etc.).

Internally, a storage block consists of a single *tuple-storage sub-block* and any number of *index sub-blocks*. The index sub-blocks contain index-related data for tuples in that block are are self-contained in the storage block. The index sub-blocks can be viewed as partitioned indices [12].

The block-oriented design of Quickstep's storage system offers a tremendous amount of flexibility with regards to physical database organization. Many different tuple-storage sub-block and index sub-block implementations are possible and can be combined freely (we describe the particular sub-block types we studied in detail in Section 3.3). Not only is it possible for different tables to have different physical layouts, it is possible for different blocks *within* a table to have different layouts. For instance, the bulk of a table could be stored in a sorted column-store format which is highly amenable to large-scale analytic queries, while a small "hot" group of indexed row-store blocks can support a live transactional workload.

Storage blocks are a natural unit of parallelism for multithreaded query execution, and temporary blocks are units of data flow between operators in complex queries.

3. EXPERIMENTS

As discussed in the Introduction, the focus of this study is on core storage manager performance for read-optimized main memory settings in which selection-based queries are common. For the workload we build on the work of [16] that had a similar goal, but in a disk-based setting. Specifically, we use the tables and queries that are described below.

3.1 Tables

We use the following set of four tables:

- **Narrow-U:** This table has ten 32-bit integer columns and 750 million rows. The values for each column are randomly generated in the range [1 to 100,000,000].
- **Narrow-E:** This table has ten 32-bit integer columns and 750 million rows. The values for column i are randomly generated in the range from [1 to $2^{2.7*i}$].
- **Wide-E:** A table similar to Narrow-E, except that it has 50 columns instead of 10 and 150 million rows instead of 750 million. Thus, the total size of the generated data is the same. The values for column i are randomly generated in the range from [1 to $2^{4+(23/50)*i}$].
- **Strings:** This table has ten 20-byte string columns and 150 million rows. Each string is a random sequence of mixed-case letters, digits, spaces, and periods.

As in [16], we primarily study integer columns, which are common and generalize nicely. We also consider a table composed entirely of strings, as their storage requires more space, and they are a commonly encountered data type.

3.2 Queries

Our workload primarily consists of queries of the form:

`SELECT COL_A,COL_B,... FROM TBL WHERE COL_A >= X;`

The projected columns `COL_A ...` are randomly chosen for each experiment. Each query has a predicate on a single column `COL_A`. The literal value `X` is chosen based on the range of values for `COL_A` to achieve a desired selectivity. In our experiments, we consider predicates with selectivity 0.1%, 1%, 10%, 50%, and 100%.

We also vary the number of columns that are projected. For the Narrow-U, Narrow-E, and Strings tables, which each have ten columns, we conduct experiments projecting 1, 3, 5, and 10 columns. For Wide-E, which contains 50 columns, we conduct experiments projecting 1, 15, 25, and 50 columns. For each query, a random subset of the desired number of columns is chosen to project. For each table, we also run experiments that only measure the time to evaluate a predicate without projecting any values or materializing any output (we report this as a projection of zero attributes).

With the exception of zero-projection queries (which produce no output), we materialize the output of each query in-memory in a simple non-indexed row-store format.

We also conduct experiments with more complex predicates to determine whether our observations for simple predicates hold. As in the previous study [16], the complex predicates which we study are conjunctions of three single-column predicates ANDed together. We again vary the selectivity and number of columns projected.

Finally, we conduct experiments with an aggregate query to gain some insight into how the storage organizations which we study affect the performance of higher-level query-processing operations. The query we experiment with is directly based on the aggregate queries Q21 and Q24 (`MIN` with 100 partitions) from the Wisconsin Benchmark [10], and has the following form:

```
SELECT ONEPERCENT, MIN(COL_A) FROM TBL GROUP BY
ONEPERCENT HAVING MIN(COL_A) < X;
```

We slightly modified the schema of the Narrow-U table for the aggregate experiment, replacing one of the standard columns with a `ONEPERCENT` column which has 100 unique values and is used as the group-by attribute.

As in previous work [1, 15, 16], the performance metric that we use for evaluation is single query response time, which has a large impact on overall system performance. Individual query response time is also an important metric in interactive analysis environments.

3.3 Physical Database Organization

In this paper, we explore a number of key dimensions related to physical organizations for read-optimized main memory DBMSs. These dimension are described below.

3.3.1 Files vs. Blocks

In traditional DBMSs, tuples are stored in large files which are subdivided into pages (disk or memory pages). Pages are the units of buffer management and disk I/O. Non-clustered indices are separate files, external to the base table files which they reference. Modern column-store DBMSs store projections of distinct columns as separate column-striped files, which nonetheless fit into the large-file paradigm [19, 32]. Even where modern main memory DBMSs have abandoned page-based buffer management and I/O, they typically still organize data into large file-like memory segments.

As described in Section 2.1, Quickstep's SM is built around the concept of "blocks" as self-contained, self-describing units of storage. A single table's tuples are horizontally partitioned across many blocks. Internally, blocks have a tuple-storage sub-block which stores complete tuples, and any number of index sub-blocks that index the tuples in the tuple-storage sub-block. Multiple implementations of both types of sub-block, with different physical layouts, are possible. The choice of sub-blocks is represented by additional dimensions in our experiments below.

The first major dimension of our experiments is a comparison between a traditional large-file layout and a block-oriented layout where tuples are divided amongst blocks and indices, if any, are co-located with data inside blocks.

Block Size & Parallelism An important consideration for the block-oriented layout is the size of blocks (Quickstep allows differently-sized blocks). We conduct a sub-experiment where we vary the size of blocks by powers of 2 from 128 KB to 256 MB and measure the response times of queries of the form described in Section 3.2 on the Narrow-U table. We also vary the number of worker threads used to process queries in this experiment, using 1, 2, 5, or 10 threads pinned to individual CPU cores, and 20 threads with one thread pinned to each hardware thread of a 10-core hyperthreading-enabled CPU. We report the results of this experiment in detail in Section 4.1. In summary, we found that a block size of 16 MB with 20 hyperthreading-enabled worker threads resulted in good performance across the mix of queries and other physical organization parameters that we tested, so we fix the number of worker threads at 20 and the block size at 16 MB for our other experiments.

Partitioning The block-oriented design also allows us to consider partitioning tuples into different groups of blocks within a table depending on the value of some column(s). A number of strategies for assigning tuples to partitions is possible, analogous to the choices for horizontal partitioning in a clustered parallel database system, e.g. [21,27]. For our queries, which involve a range-based predicate on a single column, we experiment with range-based partitioning based on the value of a single column. We use 16 partitions evenly divided by value range in these experiments[1]. For all but the 100% selectivity queries, this has the effect of assigning all relevant tuples for a predicate on the partition-column to a limited subset of blocks.

3.3.2 Row-Stores vs. Column-Stores

The relative benefits of row-store and column-store organization for read-optimized databases in disk-based settings have been extensively studied [1, 15, 16]. In this paper, we consider the impact in main memory settings. We evaluate both a conventional unsorted row-store and a column-store sorted on a single primary column in our experiments. Both layouts are evaluated in a large-file and block-oriented context, with and without secondary indices.

[1] The choice of 16 partitions is somewhat arbitrary, and is meant to illustrate the potential benefits of partitioning. A full study of partitioning strategies (including handling of skew) is beyond the scope of this paper.

3.3.3 Secondary Indices

Secondary indices can often speed evaluation of predicates when compared to scans of a base table. We have implemented a cache-sensitive B+-tree [26] index in the Quickstep SM with a node size of 64 bytes (equal to the cache-line size of our test system). We measure the response time of our test queries when evaluating predicates via a CSB+-tree on the appropriate column's values for both row-store and column-store layouts, with both the large-file and block-oriented designs. We compare this to predicate evaluation using scans and a binary search of the sorted column for the column-store organization.

3.3.4 Compression

Compression is often effective in disk-based read-optimized database settings [16]. An important consideration for main memory read-optimized databases is whether the advantages of compression still apply for data that is entirely memory-resident. We implemented dictionary-coding and bit-packing[2] in the Quickstep storage manager. Compression is available for both row-stores and column-stores, and if a CSB+-Tree index is built on a compressed column, it will also store compressed codes. All queries that we consider in our experiments can work directly on compressed codes. Our compression implementation builds up sorted dictionaries for each column as a block is being built and, if space can be saved by compressing that column, stores compressed codes (which still compare in the same order) instead of native values. Dictionaries for compressed columns are stored inside blocks. For integer columns whose values are in a limited range, we can use bit-packing without a compression dictionary by simply truncating these values down to their lower-order bits.

We measure the difference in performance by using compression in block-oriented organization on the Narrow-E table, which contains several compressible columns. We combine compression with both row-store and column-store layouts, with and without indexing.

3.4 Experimental Setup

We run our experiments on a four-processor Intel Xeon E7-4850 server with 256 GB of SDRAM in a NUMA configuration running Scientific Linux 6. Each processor has 10 cores and is clocked at 2.0 GHz. Each core has dedicated 32 KB L1 instruction and data caches and a dedicated 256 KB L2 cache, while each processor has a shared 24 MB L3 cache. The cache-line size is 64 bytes. Because our experiments are focused on the performance impact of data organization, we run most experiments on a single CPU socket (with up to 20 threads) using locally attached memory (64 GB). We do, however, conduct some scale-up experiments that use all four CPU sockets.

We created a test-driver executable which generates a table from Section 3.1 in memory with a specified storage organization, and then proceeds to run a series of experiments, varying query selectivity and projection width as described in Section 3.2. For each combination of physical organization, table, and query parameters, 10 query runs are performed and the total response time for each is measured. The mean and standard deviation of the execution times are reported. The total number of L2 and L3 cache misses incurred during each query run are also measured using hardware performance counters on the CPU. Quickstep is compiled with GCC 4.8.0 using `-march=native` and optimization level `-O3`.

In our initial block-size experiments, we found that using 20 worker threads with hyperthreading enabled tended to produce the best performance, so we run all subsequent queries using 20 worker threads. Queries are run one-at-a-time, with individual worker threads operating on different blocks for intra-query parallelism. Since the focus of this paper is on read-optimized databases we have not enabled any sophisticated transactional concurrency control or recovery mechanisms (queries simply grab table-level locks).

4. RESULTS

Most of the results reported in this section are based on queries on the Narrow-U table. We discuss how our findings are affected by the presence of wide attributes (as in the Strings table) in Section 4.8, and wide rows (as in the Wide-E table) in Section 4.9.

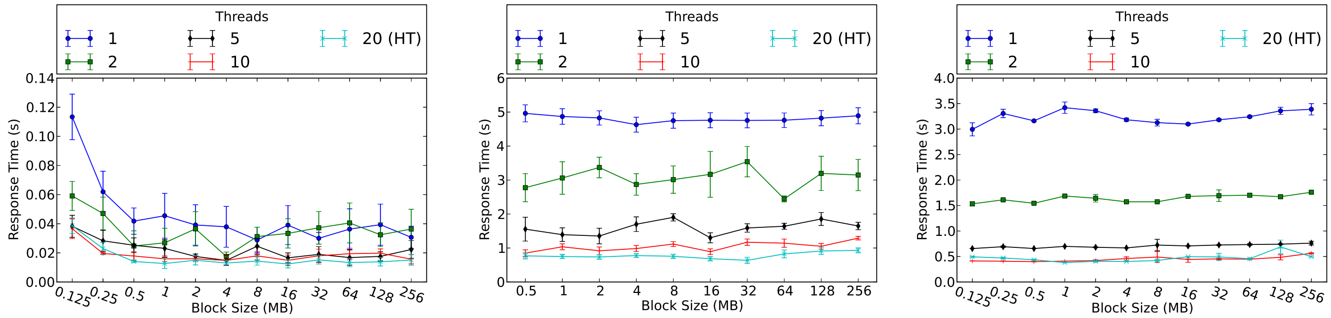
From Figure 2 onward, we show a series of graphs that compare the performance of various options in the large space of storage organizations that we studied. All graphs show total query response time on the vertical axis (lower is always better) and vary either the selectivity or the projection width of test queries on the horizontal axis. Error bars show the variation in query response time across 10 experiment runs. To make these graphs easier to read, each graph is accompanied by a table that identifies what region of the design space is being shown (the organization, tuple layout, and indexing), with the dimension that is being examined shown in **bold text**. The selectivity and projection width of queries are also indicated (for any given graph, one will be fixed and the other varied along the horizontal axis).

4.1 Block Size & Threading Experiment

We conducted an experiment to determine how we should tune the size of blocks in the block-oriented layout, and the number of worker threads used for intra-query parallel processing. For this experiment, we run the entire set of queries described in Section 3.2 on a smaller version of the Narrow-U table consisting of 100 million tuples. We vary the size of blocks by powers of two from 128 KB to 256 MB, and the number of threads from 1 to 20 (in the 1-10 thread cases, the threads are pinned to individual CPU cores, while with 20 threads, the threads are pinned to individual hardware threads with 2 threads per hyperthreading-enabled core). In the interest of space, we present three queries below that represent the key behavior that was observed across the entire set of queries that we ran.

In Figure 1(a) we show a query with a 0.1% selectivity predicate on a column store's sort column. Here we observe that optimal block sizes are in the range of 16-32 MB, and that adding threads for intra-query parallelism results in substantial speedup. We also observe that hyperthreading is useful in speeding up query execution. This is because predicate evaluation in this layout involves a binary search of the sorted column, a random-access pattern of memory accesses which is not well-handled by prefetching. When one thread incurs a cache miss, it is often possible for the other thread on the same core to immediately take over and do some work, effectively "hiding" the cost of many cache

[2] Our bit-packing implementation pads individual codes to 1, 2, or 4 bytes, as we found that the cost to reconstruct codes that span across more than one word significantly diminished performance.



(a) 0.1% selectivity, 3 attributes projected, predicate on sort column of Column Store.

(b) 10% selectivity, 3 attributes projected, Row Store with CSB+-Tree index.

(c) 1% selectivity, 3 attributes projected, Row Store (scan).

Figure 1: Block size vs. response time.

misses. In the example shown, at 16 MB block size, enabling hyperthreading reduces response time by 17.3% over the 10 thread case, even though the average total number of L3 cache misses is similar (35790 without hyperthreading and 37202 with hyperthreading). The 16-32 MB range of block sizes is well-tuned to allow individual worker threads to often hit cache lines that had already been faulted or prefetched as part of previous “random” accesses.

Next, in Figure 1(b), we show the result for a 10% selectivity query using a row store with a CSB+-Tree index. Similar to the column store case above, the optimal block size is approximately 16-32 MB. Again, adding threads for intra-query parallelism improves the query performance. Hyperthreading again improves performance because it is able to “hide” the cost of cache misses arising from random access (using a non-clustered index produces a sequence of matching tuples in an order other than their physical order in the base row-store, so tuples are accessed in a random order when the projection is performed). In the example shown, at 16 MB, hyperthreading reduces overall response time by 23.1%, even though the number of L3 cache misses incurred is actually slightly greater (18.4 million without hyperthreading and 19.0 million with hyperthreading). We see similar results when using a CSB+-Tree to evaluate a predicate on a non-sorted column of a column store.

Observation 1. *A block size of 16-32 MB with hyperthreading enabled and all hardware threads in use provides optimal or near-optimal performance for combinations of storage formats and queries that involve random access.*

The last query for this experiment is a simple scan query. This result is shown in Figure 1(c). Scan-based queries have a linear access pattern which works well with cache prefetching. Hence, we find little variation in response time for block sizes below 64 MB (most cache misses are avoided by prefetching), and cache misses are infrequent enough that hyperthreading does not significantly improve performance. The results are similar when scanning a column store.

Observation 2. *Scan-based queries are less sensitive to block size and perform well for all block sizes below 64 MB. Hyperthreading is not beneficial (but also not harmful) for scan-based queries, and query performance increases with additional threads up to the number of physical CPU cores.*

Based on these observations, we fix the block size at 16 MB and the number of worker threads at 20 for the rest of our experiments.

4.1.1 *Relationship to Cache Size & Number of Cores*

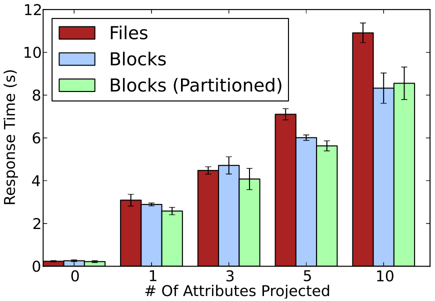
Optimal block size is related both to the CPU’s cache size and its number of cores (since multiple cores share a unified L3 cache). To test this, we also conduct block-size experiments on a machine with a Core i7-3610QM CPU (4 cores/8 threads with 6 MB of L3 cache), and one with a Xeon X5650 CPU (6 cores/12 threads with 12 MB of L3 cache). We find that the best-performing block size is in the range of 8-16 MB for the Core i7 machine, and about 16 MB for the Xeon X5650 machine. This suggests a rule of thumb for tuning block size: the optimal block size is approximately 5X to 10X times the L3 cache size divided by the number of cores. Cache locality is very important for performance, but the optimal block size actually somewhat overuses the L3 cache, because prefetching can help avoid cache misses (especially when scanning), and hyperthreading can mitigate the performance impact of cache misses when they occur.

4.1.2 *Multi-Socket NUMA Scale-Up*

We ran queries similar to those illustrated in Figures 1(a)-1(c) with a larger 3 billion row dataset using all four sockets in our multi-socket NUMA server. Increasing the number of cores used on a single socket yielded good, near-linear speedup (10 cores were 6.5X-9X as fast as one core, depending on the query). However, using 20 cores on 2 sockets was never more than 20% faster than using 10 cores on a single socket, and going to 3 or 4 sockets actually caused response time to increase. This is because our experimental code-base is NUMA-oblivious, and neither the creation of blocks nor the assignment of blocks to worker threads is done with awareness of the different access costs for memory attached to different sockets or the contention for bandwidth on inter-socket links, causing worker threads to frequently stall waiting for data from non-local memory. Our results here confirm the need for NUMA-aware data placement and processing observed in previous research [3, 34], which is an active area of research in the community, and an important area of future research with Quickstep.

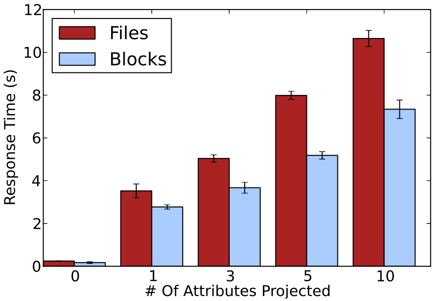
4.2 Files vs. Blocks

We compared the traditional large-file organization with Quickstep’s block-oriented organization across the other dimensions of our experiments. So as not to penalize the file-based organization for a lack of parallelism, we statically partition our test tables into 20 equally-sized files for 20 worker threads to operate on.



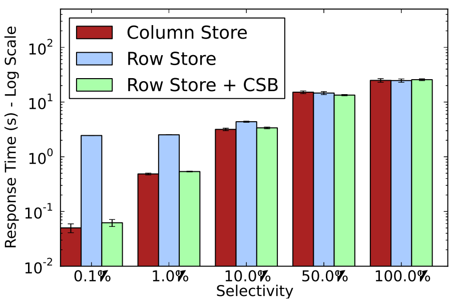
Organization	File vs. Blocks
Tuple Storage Layout	Column-Store
Indexing	None
Predicate	10% Selectivity On Sorted Column
Projection Width	Varies

Figure 2: Files vs. Blocks – Sorted Column



Organization	File vs. Blocks
Tuple Storage Layout	Row-Store
Indexing	CSB+-Tree
Predicate	10% Selectivity On Indexed Column
Projection Width	Varies

Figure 3: Files vs. Blocks – Non-Sorted Column



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None/CSB+-Tree
Predicate	On Sorted/Indexed Column
Projection Width	1 Column

Figure 4: Column-Store vs. Row-Store (+Index) – Narrow Projection

In Figure 2, we show the difference in performance for queries at 10% selectivity where the predicate is on the sorted column of a column-store (for partitioned blocks, this is also the column whose value the blocks are partitioned on). The performance of files is similar to the performance of blocks for narrow projections. When projecting 5 or 10 columns, blocks outperform files due to improved locality of access and caching behavior when accessing several column stripes within a relatively small block to reassemble tuples (in the example shown, file-based organization incurs 77.4 million total L3 cache misses and 1.53 billion L2 misses, while block-based organization incurs only 73.6 million L3 misses and 1.14 billion L2 misses). This same pattern of results occurs at the other selectivity factors that we tested.

Observation 3. *For queries with a predicate on the sorted column of a column-store, using a file or a block organization makes little difference in performance, except for wide projections, where blocks tend to perform better.*

In Figure 3, we show the difference in performance for queries at 10% selectivity where the predicate is on a non-sorted column of a row-store, and a CSB+-Tree index is used for predicate evaluation. Across all projection widths, the block-oriented organization outperforms the file-based organization. This is thanks to improved locality of access in relatively small blocks, (in the example shown, when projecting 5 attributes, file-based organization incurred 166 million L3 cache misses and 1.63 billion L2 misses, while block-based organization incurred 177 million L3 misses but only 673 million L2 misses). This result holds across all of the selectivity factors that we test. For projections of one or more attributes, reduction in response time for blocks vs. files ranges from 2.1% (projecting 1 attribute at 1% selectivity) to 37.7% (projecting 10 attributes at 50% selectivity). We see similar reductions in response times for blocks vs. files when evaluating a predicate on an unsorted column of a column-store with a CSB+-Tree index, and when evaluating predicates via a scan on a row-store or column-store.

Observation 4. *For queries with a predicate on a non-sorted column, a block-based organization outperforms a file-based organization. This result holds for both row stores and column stores, with and without indexing.*

The block-based organization in Quickstep adds very little storage overhead compared to files (there is less than 100 bytes of additional metadata per block). The total memory footprint of the Narrow-U table without indices is 28610 MB for files and 28624 MB for blocks (0.05% additional storage for blocks). With a secondary CSB+-Tree index, the total memory footprint is 35763 MB for files and 35776 MB for blocks (0.04% additional storage for blocks).

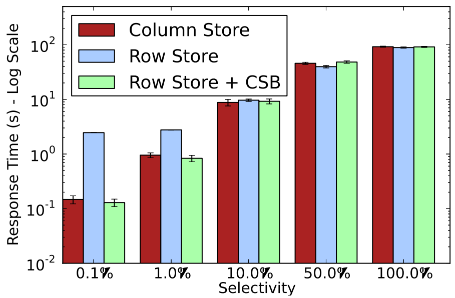
4.2.1 Column-Store Load Cost

It should be noted that the cost of building large sorted column-store files is higher than the cost of building many individual sorted column-store blocks. In our experiments, it takes 159.8 seconds to sort the 750 million tuple Narrow-U table into 20 partitioned column-store files (using 20 worker threads, 1 per partition). Building sorted 16 MB column-store blocks, on the other hand, takes only 87.8 seconds (only 55% as much time, again using 20 worker threads). The sort algorithm used in both cases is introsort [23]. As we saw in Figure 2, the read-performance advantages of a sorted column-store are maintained in the block-based layout, but the initial load time is smaller.

Observation 5. *The build time for a sorted column-store in a block-based organization is smaller than that for a file-based organization, but gives the same or better read query performance.*

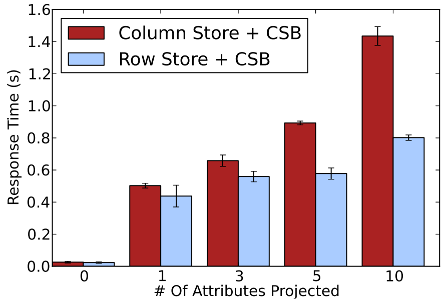
4.3 Row-Store vs. Column-Store

Comparing row-store and column-store tuple storage layouts in blocks, we find that, when selecting via a predicate on the sorted column of the column-store, the column-store outperforms the row-store for narrow projections, whether or not a secondary CSB+-Tree index was built on the row-store to speed predicate evaluation (we discuss indexing further in Section 4.4). As one might expect, predicate evaluation is fast for the column-store, merely requiring a binary search on the sorted column. After this search, performing the actual projection involves linear access to a contiguous region of a few column stripes. The column-store's performance advantage for narrow projections is illustrated in Figure 4. For wider projections (5 or 10 attributes), the performance of the column-store is nearly equal to that of the row-store with a secondary index, as seen in Figure 5. Although predicate evaluation is more complicated when using an index, and results in a random access pattern for tuples, the row store makes up for this by storing all the column



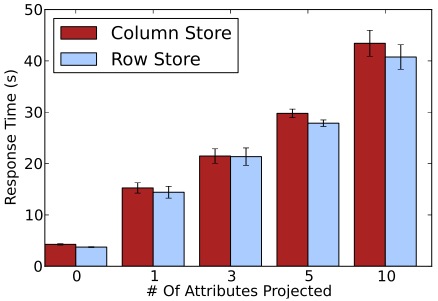
Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None vs. CSB+-Tree
Predicate	On Sorted/Indexed Column
Projection Width	10 Columns

Figure 5: Column-Store vs. Row-Store (+Index) – Wide Projection



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	CSB+-Tree
Predicate	1% Selectivity on Indexed (Non-Sorted) Column
Projection Width	Varies

Figure 6: Column-Store vs. Row-Store – Indices



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	None
Predicate	50% Selectivity on Non-Sorted Column
Projection Width	Varies

Figure 7: Column-Store vs. Row-Store – Scanning

values needed for a projection on one tuple in one or two contiguous cache lines instead of several disjoint column stripes. We saw similar results when comparing column-stores and row-stores in the large-file organization (in fact, the performance advantage for column-stores doing narrow projections was more pronounced).

Observation 6. *For queries with a predicate on a column-store's sorted column, a column-store outperforms a row-store (even when using an index on the appropriate column of the row-store) for narrow projections. For wider projections, performance of column-stores and row-stores (with an applicable CSB+-Tree index) is nearly equal.*

When testing queries that include a predicate on a column other than the column-store's sorted column, we find that, when using blocks, row-stores outperform column-stores for predicates that select a small number of tuples and benefit from using an index, as seen in Figure 6. This is because, when accessing column values to perform the projection, all values lie on one or two adjacent cache lines in the row store, whereas the each column value in a tuple is on a different (non-contiguous) cache line in the column store. The performance advantage for row-stores is larger for wider projections because column stores incur more cache misses for each additional column in the projection, while row stores do not. In the example shown, when projecting all 10 columns, 97.6 million L3 cache misses are incurred when using a column store, but only 22.0 million when using a row store.

For predicates that select a large number of tuples and are better evaluated with a scan, the performance of row stores and column stores is equal, as seen in Figure 7. When scanning, the access pattern is linear and prefetching is effective at avoiding cache misses (both for the single tuple-storage region in the row-store and the densely-packed values in column stripes in the column store).

Observation 7. *For queries with a predicate on a non-sorted column, with a* ***block-based*** *organization, row-stores outperform column-stores when using indexing. This result holds for various selectivity factors and is more pronounced for wider projections. For scan-based queries, row stores and column stores have similar performance.*[3]

When we test these same queries (with predicates on a non-sorted column) using large-file organization, we find that for queries with 0.1% and 1% selectivity, a row-store with a CSB+-Tree index outperforms a column-store with a CSB+-Tree index for projections of 3 or more attributes. For narrower projections, and for all queries at 10% selectivity, the column store and the row store (both with CSB+-Tree index) have the same performance. Just as in the block-based organization, predicate evaluation using an index takes the same amount of time in either case, but row-stores benefit from the fact that all column values for a projection lie on one or two adjacent cache lines when projecting more than one attribute. For queries with 50% and 100% selectivity (where indices were no longer useful) we find that the column-store slightly outperforms the row-store for projections of 1, 3, or 5 attributes, and that the row-store outperforms the column store for projections of 10 attributes. This is because, when scanning, column stripes are accessed in a prefetching-friendly linear pattern, and when selecting 50% or 100% of tuples, there tend to be several matching attribute values on every single cache-line accessed (8 values/line on average at 50% selectivity). Row stores do better when projecting all 10 attributes because a single contiguous row can be directly copied rather than being reassembled from several disjoint column stripes.

Observation 8. *For queries with a predicate on a non-sorted column, with a* ***file-based*** *organization, whether row-stores or column-stores perform best is situational and dependent on selectivity, projection width, and indexing.*

As a point of comparison, we also test scans of column-stores vs. row-stores in a leading commercial main-memory analytical database, which we call DB-X. DB-X supports both types of storage. The relative performance of column-stores and row-stores in DB-X is very similar to what we observe in files in Quickstep (i.e. column-stores slightly outperform row-stores when scanning).

We also test queries comparing row-store and column-store performance at larger block sizes (64 MB and 256 MB) to see if, at large block sizes, blocks start to behave more like files. 64 MB blocks have results similar to Observation 7.

[3] Note that, at 50% selectivity (where it is more efficient to evaluate predicates with a scan than index), column-stores outperform row-stores for wide columns (Section 4.8) and for narrow projections of wide rows (Section 4.9).

For 256 MB blocks, when using an index at lower selectivities, column-stores perform similar to row-stores for narrow projections, while row-stores perform best with wider projections. When scanning at large selectivity factors, column-stores perform best when projecting 1 attribute, while row-stores perform best when projecting all 10 attributes (performance is nearly equal when projecting 3 or 5 attributes). The relative performance of row-stores and column-stores in very large blocks (256 MB) is similar to that in files.

Row-stores and column-stores have essentially identical memory footprints (there is exactly the same amount of data, the only difference being whether it is organized in row-major or column-major order). The Narrow-U table (without indices) takes up 28610 MB in files and 28624 MB in blocks for both column-store and row-store layouts.

4.4 Effect Of Indices

As noted in Section 4.3 and illustrated in Figures 4 and 5, when evaluating a predicate that selects based on the value of the sort column in a column-store, a column-store tends to slightly outperform indexed access. In this section, we explore the effect of indexing on row-stores, and when evaluating a predicate on a non-sorted column of a column-store.

In Figure 8, we show the effect of using a CSB+-Tree index sub-block to evaluate a predicate where the underlying tuple-storage sub-block is a row-store or column-store. At 0.1%, 1%, and 10% selectivity, using the index is faster than scanning the tuple-storage sub-block. At 50% selectivity, using the index is roughly equal in performance to a scan. We do not show results for 100% selectivity, as queries that match all the tuples automatically skip any indices. We see similar results for other projection widths.

In Figure 9, we similarly show the effect of using a CSB+-Tree index in the large-file layout. Using an index again reduces query run time at 0.1%, 1%, and 10% selectivity, whether the base table file is a row-store or column-store. At 50% selectivity, an index is no longer useful, and for column stores it is faster to simply scan the base table.

Using an index outperforms scanning at lower selectivity factors, because predicate evaluation with an index requires only a logarithmic-time traversal of the CSB+-Tree structure, after which only matching tuples are accessed to perform the projection. Scanning requires accessing every tuple in the table individually to check the predicate (a linear-time procedure). In the example shown in Figure 8, at 0.1% selectivity, the total number of L3 cache misses is 1.38 million when using an index with a row store vs. 83.1 million when scanning the base row store). At very large selectivity factors, most tuples must be accessed anyway, so the advantage of the index is muted. Additionally, scans access data in a purely linear pattern, which allows prefetching to be very effective at avoiding cache misses. In the same example, at 50% selectivity, using an index incurs 682 million L3 cache misses, while scanning incurs only 148 million L3 misses (an index still performs competetively despite this cache-miss disadvantage because the predicate does not need to be individually checked for every tuple).

Observation 9. *For queries with a predicate on a non-sorted column, using a CSB+-Tree index improves query performance for selectivity factors below 50%. This result holds for blocks and files, for both column-store and row-store tuple-storage layouts, and across all projection widths.*

Our results here suggest that the conventional wisdom regarding query optimization in the presence of indices, namely that indices should be used when selectivity is estimated to be below 10%, is mistuned for the main memory environment. Our data indicate that the appropriate cutoff, which balances the more efficient predicate evaluation and sparse data access of the index approach against the prefetching-friendly linear access pattern of the scan approach, is somewhere above 10% selectivity and below 50% selectivity.

Our results also demonstrate that secondary indices can play a major role in accelerating query performance in a main memory analytic database. The scan-only approach of systems like Blink [5] greatly simplifies query optimization, but excluding indices from the system can unnecessarily penalize the performance of queries with lower selectivity factors. We also note that Quickstep's block-oriented storage allows the global query optimizer to remain index-oblivious, with the decision of whether to use an index being made independently on a per-block basis within the storage system.

Adding a secondary index necessitates using some additional storage. The additional memory footprint of a CSB+-Tree index on a single column of the Narrow-U table is 7152 MB for blocks and 7152.56 MB for files (25% of the storage used for the base table in either case).

4.4.1 Ordering and Cache Behavior

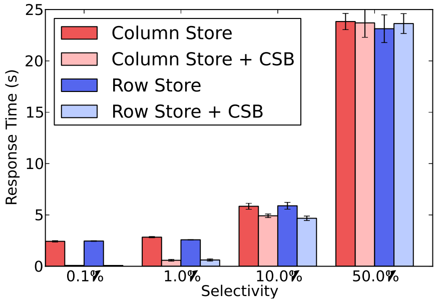
As noted above, at large selectivity factors, using an index can actually increase the number of cache misses compared to simply scanning the base table. This is because the order of matches in the index is, in general, different from the order of tuples in the base row-store or column-store, resulting in a random access pattern when accessing tuples to perform the projection. It is natural to ask whether it is possible to improve index performance by first using an index generate a list of tuple-IDs matching a predicate, then sorting this list into ascending order before accessing values in the base table to perform the projection. Tuples in the base table are then accessed in their physical order, effectively changing a random access pattern into a linear one, helping to avoid cache misses and take advantage of prefetching.

We conducted experiments to determine the effectiveness of sorting the matches (i.e. whether improvements in cache behavior are worth the additional time required to sort the matches). We find that the sorting optimization performs best in combination with a column-store in a file-based organization. For example, when projecting 10 columns at a selectivity factor of 10%, sorting matches reduces the total number of L3 cache misses from 824 million to 430 million. Despite this improvement, the overall query response time remains nearly the same (nearly all of the benefit of ordered access is negated by the cost of sorting). When the table is in a row-store format, or when using block-based organization instead of files, sorting was not as effective at avoiding cache misses, and as a result the overall response time actually increased due to the additional cost of sorting.[4]

4.4.2 Index Build Cost

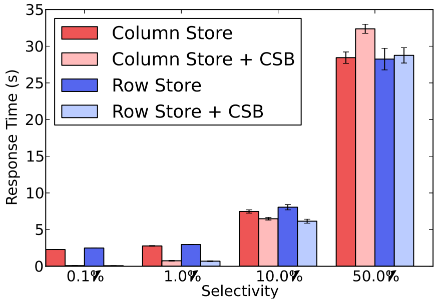
We measure the time to bulk-load CSB+-Tree indices for both the file and the block organizations, with both row-store and column-store tuple-storage layouts. We used 20

[4] For this experiment on a Xeon X5650 server, we found significant performance improvement from sorting for column-stores in files at 10% selectivity and above. Otherwise, the results were similar.



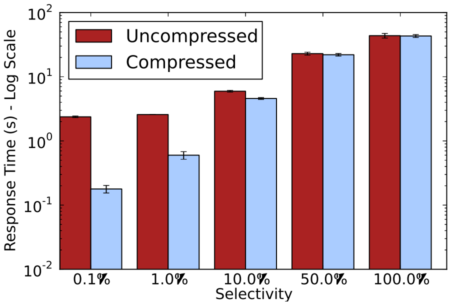
Organization	Blocks
Tuple Storage Layout	Column-Store & Row-Store
Indexing	None vs. CSB+-Tree
Predicate	Varying Selectivity on Indexed (Non-Sorted) Column
Projection Width	3 Columns

Figure 8: Effect Of Indices (Blocks)



Organization	File
Tuple Storage Layout	Column-Store & Row-Store
Indexing	None vs. CSB+-Tree
Predicate	Varying Selectivity on Indexed (Non-Sorted) Column
Projection Width	3 Columns

Figure 9: Effect Of Indices (Files)



Organization	Blocks
Tuple Storage Layout	Column-Store
Indexing	None
Predicate	Varying Selectivity on Non-Sorted Column
Projection Width	3 Columns

Figure 10: Effect Of Compression (Column Store Scan)

worker threads in all cases, with each thread working on one of the static partitions in the file organization, and working on individual blocks in the block organization. For files, the index build time is 111.6 seconds with a row-store and 87.3 seconds with a column-store. For blocks, the index build time is 61.6 seconds with row-stores and 33.6 seconds with column-stores. Index build time is only 38-55% as long for blocks as for files, and 55-78% as long when the base table is a column-store as a row-store. Note that, when using indices on a non-sorted column, performance is better with blocks than with files (see Figure 3). Build times are smaller with blocks because the tuple values in relatively small blocks constitute much smaller "runs" to sort into order and build a shallower tree from. Build-times are smaller for column-stores because all of the values that are accessed to build the index are in contiguous regions of memory, efficiently packed with 16 values per cache line.

Observation 10. *The build time for CSB+-Tree indices is much smaller for blocks than for files, even though block-based indices give better read query performance.*

4.5 Effect of Compression

We enable bit-packing and dictionary-coding compression techniques described in Section 3.3.4 in combination with row-stores, column-stores, and CSB+-Tree indices and test our compression techniques on the Narrow-E table. In general, columns 1 and 2 can be represented by single-byte codes, and columns 3, 4, and 5 can be represented by 2-byte codes. Compression reduces the size of the Narrow-E table in blocks from 28624 MB to 20032 MB (compression ratio is highly sensitive to data distribution, and is likely to be quite different for different tables). We test predicates on a compressed column (column 5), and randomly vary the projected attributes as in other experiments.

Generally, compression improves performance at selectivity 10% and below, and worsens performance at selectivity 50% and above. This is true for column-stores (with predicates on sorted or unsorted columns) and row-stores, with and without indexing, for all projection widths. Compression speeds up predicate evaluation, as predicates can be evaluated directly on compressed codes and all storage formats are more densely packed in memory, leading to more efficient usage of caches and memory bandwidth. However, compression also increases the cost of performing projections, as codes must be decompressed before they are written to (uncompressed) output blocks. When selectivity factors are low, the cost of predicate evaluation is dominant and compression improves performance. When selectivity factors are high, the cost of decompression begins to overwhelm the advantage from faster predicate evaluation.

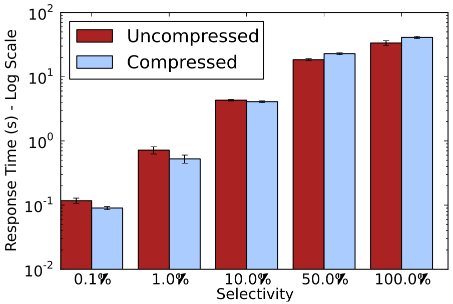
We illustrate the effect of compression when scanning a non-sorted compressed column of a column-store in Figure 10. Here, the performance improvement for compression is most dramatic. Scans can go through small, very dense compressed column stripes, avoiding most cache misses thanks to prefetching. In the example shown, for a predicate with 0.1% selectivity, enabling compression on the column store reduced the total number of L3 cache misses from 5.98 million to 3.32 million. We also illustrate the effect of compression when using a CSB+-Tree index with a row-store in Figure 11. These results are more typical of the benefit seen from compression (we see similar patterns for scans of row-stores, and when evaluating predicates on a column-store's sort column, or on a column-store with an index), which modestly improves performance at 10% selectivity and below, and slightly worsens performance at 50% and above.

Observation 11. *Dictionary-coding and bit-packing compression improve performance when selecting via a predicate on a compressed column at selectivity 10% or below. Compression makes performance slightly worse at selectivity 50% and above. These results hold for column-stores and row-stores, with and without indexing. Performance improvements are most pronounced when scanning column stores.*

4.6 Complex Predicates

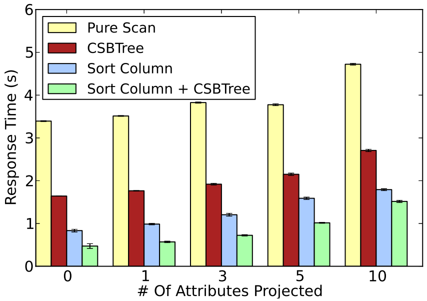
We now examine how conjunctive predicates affect the performance of different storage organizations. The predicates in these queries are conjunctions of three single-column predicates, and the projected columns are randomly chosen for each run as in other experiments.

There are a number of strategies possible for evaluating complex predicates, depending on the storage organization. We evaluate four simple predicate evaluation strategies across the space of storage organizations which we study (note that not all of these strategies are possible in every organization). A pure scan, where values for each predicate



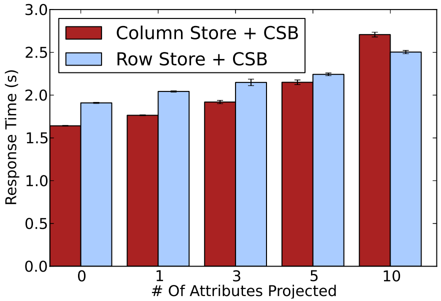
Organization	Blocks
Tuple Storage Layout	Row-Store
Indexing	CSB+-Tree
Predicate	Varying Selectivity on Indexed Column
Projection Width	3 Columns

Figure 11: Effect Of Compression (Row Store With Index)



Organization	Blocks
Tuple Storage Layout	Column-Store
Indexing	**None vs. CSB+-Tree**
Predicate	1% Selectivity (3 Columns)
Projection Width	Varies

Figure 12: Conjunction – Evaluation With Column-Store & Index



Organization	Blocks
Tuple Storage Layout	**Column-Store vs. Row-Store**
Indexing	CSB+-Tree
Predicate	1% Selectivity (3 Columns)
Projection Width	Varies

Figure 13: Conjunction – Column Store vs. Row Store – Indices

column are explicitly read and checked, remains the simplest strategy for both column-stores and row-stores. For a column-store where one of the predicates is on the sort column, a binary search can quickly evaluate that predicate, and the matching tuples can then be scanned and filtered by the remaining predicates on other columns (i.e. a scan, but with a simpler predicate over a smaller range of tuples). When an index is present on one of the columns in the conjunction, the index can be used to evaluate that predicate, and values from matching tuples can be fetched to explicitly check the remaining predicates (again, evaluating a simpler predicate over a smaller number of tuples, but in this case in random order). Synergies between these specialized methods are possible. As a fourth strategy, we evaluate the case where one predicate is on the sort column of a column-store and another is on an indexed column. We determine the range of tuples that match the first predicate (using a binary search), then use the index to evaluate the second predicate and automatically skip over any tuples which aren't in the range for the first. Finally, we fetch column values for any tuples which we know match the first two predicates to evaluate the third predicate.

We illustrate the performance of these strategies in Figure 12. Here we see that a pure scan performs worst, and using either a CSB+-Tree index or a search on the column-store's sort column is effective at reducing query response time. When projecting three columns, the pure scan incurs 38.7 million L3 cache misses. Using a CSB+-Tree index actually increases the number of L3 cache misses to 163 million (the random access pattern is less amenable to prefetching), but it makes up for this by reducing the number of tuples that predicates must be explicitly checked for by 78% (at 1% selectivity), and also reducing the number of predicates that must be explicitly checked from 3 to 2. Doing a binary search on the sort column has all the advantages of using an index, but also has the additional advantage of reducing the number of L3 cache misses to 16.3 million.

The virtuous effects of the index and the sorted-column search compound each other when they are used in combination to evaluate different parts of a conjunctive predicate, significantly outperforming either used alone. The combination of both techniques reduces the number of tuples that must have predicates explicitly checked to just 4.6% of the tuples in the table (at 1% selectivity), and only one predicate needs to be explicitly checked. The number of L3 cache misses incurred when using the combined technique is 39.8 million (about the same as a scan, and worse than just the binary search, but overall performance is better because there is only one remaining predicate, and it needs to be checked for far fewer tuples). We see similar results for conjunctions at other high selectivity factors.

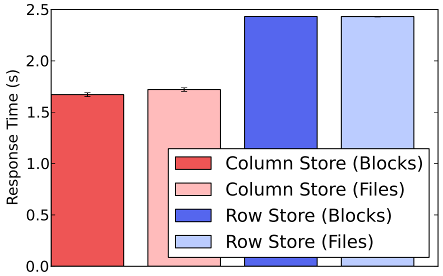
Observation 12. *For conjunctive predicates with overall high (<10%) selectivity, using a sorted-column search and a CSB+-Tree index in combination outperforms either technique used on its own.*

For most conjunctive queries, we found that our observations regarding the relative performance of different storage organizations hold. One exception to this was Observation 7. When evaluating a conjunctive predicate with a low selectivity factor (0.1% or 1%) using an index on a single column in the predicate, column stores slightly outperform row-stores for narrow projections. This is illustrated in Figure 13. This is because, after matches are obtained from the index, the rest of the predicate must be checked by fetching values from two other columns in the base table, and these additional values are more densely packed in cache lines in a column-store (in the example shown, when projecting 1 column, the number of L3 cache misses is 202 million for the row-store, but only 148 million for the column-store).

4.7 Aggregation

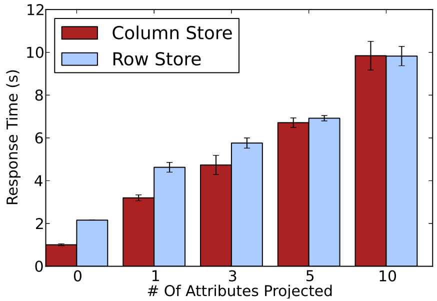
As a final experiment, we run an aggregate query with a `GROUP BY` clause (with 100 partitions) against the different storage organizations which we studied. We use a multithreaded hash-based implementation of `GROUP BY`, where each worker thread maintains its own hash-table keyed on the value of the grouping attribute, with a payload which is the aggregate's "handle" (in this case the running value for `MIN()`). The final step in query execution is to merge the per-thread hash tables together into a single global result and apply the `HAVING` condition as a filter. We show the response time for aggregate queries in Figure 14.

In general, the performance of aggregation very closely tracks the performance of a scan with a two-column predicate and no projected output. For every tuple, two columns values are read: the group-by column, which is used to probe



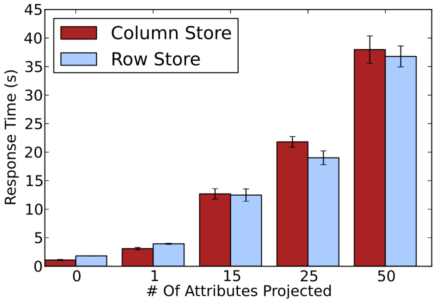
Organization	Blocks vs. Files
Tuple Storage Layout	Column-Store vs. Row-Store
Indexing	None
Predicate	None
Projection Width	N/A

Figure 14: Aggregate – MIN with 100 partitions



Organization	Blocks
Tuple Storage Layout	Column-Store vs. Row-Store
Indexing	None
Predicate	50% Selectivity
Projection Width	Varies

Figure 15: Wide Columns (Strings Table) – Scanning



Organization	Blocks
Tuple Storage Layout	Column-Store vs. Row-Store
Indexing	None
Predicate	50% Selectivity
Projection Width	Varies

Figure 16: Wide Rows (Wide-E Table) – Scanning

the hash table, and the aggregated column, which is compared with the running minimum for a given partition and possibly replaces it. The hash tables for this 100-partition query are quite small and easily fit in the L1 data cache, so the dominant cost is the cost of reading all the values from two columns of the base table. Prefetching is quite effective for this linear-scan access pattern, and the dense packing of values in the column store causes it to incur fewer cache-misses and outperform the row store in this case (the block-based row-store incurred 172 million L3 cache misses, but the block-based column-store incurred only 10.8 million).

Efficiently computing aggregates in-memory is an active area of research, particularly when data cubes and advanced holistic aggregates are involved [24,28]. A full study of main-memory aggregation is beyond the storage-engine focus of this paper, but our results do indicate that for simple aggregate operations, aggregate performance is closely related to scan performance.

4.8 Effect of Wide Columns

In order to study the effect of wide columns, we repeat our experiments from previous sections using the Strings table, which has ten 20-byte wide columns and 1/5 the number of tuples as the Narrow-U table. Most of the queries tested experience a reduction in runtime on the order of 3X to 5X compared with the equivalent queries on the Narrow-U table, which shows that table cardinality is a dominant linear factor in response time.

With the overall reduction in query response time, results for the Strings table are remarkably consistent with those for Narrow-U. The previously identified observations hold, with the following caveat: in the block-based organization, column-stores are more competitive with row-stores for queries with a predicate on a non-sorted column. At 0.1%, 1%, and 10% selectivities, when using an index, row-stores still outperform column-stores when projecting more than one attribute, but the difference is less pronounced. At 50% selectivity, when using a scan, column-stores outperform row-stores for narrow projections. This result is illustrated in Figure 15. This behavior is because tuples in the strings table are 200 bytes wide and span over 4 or 5 cache lines in a row store, which mutes the advantage row stores have when projecting small values for several columns which were often on the same cache line in Narrow-U.

4.9 Effect of Wide Rows

In order to study the effect of wide rows, we repeat our experiments from previous sections using the Wide-E table, which has 50 integer columns and 1/5 the number of tuples as the Narrow-U table. Most of the queries we test have a runtime similar to the equivalent queries on Narrow-U when producing the same volume of output (i.e. 1/5 as many tuples but 5 times as many columns). Results for the Wide-E table were also very consistent with those for Narrow-U. All previously identified observations hold for wide rows, except that in the block-based organization, at selectivity factor 50% (at which point indices are not useful, and we evaluate predicates via a scan), a column-store is faster for narrow projections, while a row-store is faster for wide projections. This is illustrated in Figure 16. Again, we can partially chalk this up to the fact that wide 200-byte rows are spread across 4 or 5 cache lines in a row store, and it is necessary to project a substantial portion of the columns to see a benefit from locality of access due to several projected attributes from a matching row lying on the same cache line. Additionally, a column store will typically store values from several matching tuples together on the same cache line for a query with high (∼50%) selectivity, so that a cache line which is fetched to perform a projection for one tuple will typically remain resident and also be used when projecting the same attribute for the next few tuples. For example, in the figure shown, when projecting one column, the row-store incurs 241 million L3 caches misses, but the column-store incurs only 3.72 million. Note that for smaller selectivity factors, when using an index, a row-store still outperforms a column-store across the board in the block organization.

4.10 Summary

Our experimental observations can be distilled to the following insights for physical organization of data in a read-optimized main memory database:

1. Block-based organization should always be used over file-based organization. Performance for blocks is always at least as good as files, and often better. Additionally, blocks have cheaper load costs, both for sorting column-stores and building CSB+-Tree indices.

2. Both column-store and row-store organization should be available as options for tuple-storage layout. Column-stores perform best when there is a single dominant (i.e. sorted) column which predicates select on, or when the selectivity factor is large (∼50%) and the columns are wide, or rows are wide and projections are narrow. Row-stores perform best when selecting via predicates on various different columns, with or without an index, except for edge cases at large (∼50%) selectivity factors noted previously.

3. CSB+-Tree indices, co-located with data inside blocks, are useful in speeding query evaluation with predicates on a non-sorted column. This is true whether the tuple-storage format is a row-store or column-store, for selectivity factors smaller than ∼50%. At larger selectivity factors, it is better to simply scan the base table.

4. Compression can improve query performance across the other dimensions of storage organization studied. When a column's data is amenable to compression, we find that compression consistently improves performance for queries at 10% selectivity or below, especially when scanning an unsorted column of a column store.

5. RELATED WORK

The design of high-performance main memory databases to support analytical workloads has been a vibrant area of research for over a decade. An early pioneer in this area (and also in the field of column-stores) was MonetDB [7], and subsequently Vectorwise [35]. Quickstep applies lessons from MonetDB and Vectorwise, particularly in that it materializes results of intermediate selection operations in-memory all at once on the scale of storage blocks, using tight execution loops that make good use of L1 instruction caches and tend to keep CPU pipelines full (see Section 2 for details).

There are a number of commercial products that target the main memory data analytics market, including SAP HANA [11], IBM Blink [25], and Oracle Exalytics [22], as well as academic projects like HyPer [17] and HYRISE [13]. Both row-store and column-store data organizations have found use in main memory analytical databases. Blink [5,25] is a row-store based main memory analytical database, while SAP HANA [11] is a primarily column-store based system.

The space of possible data organizations is not limited to row-stores and column-stores, and includes other alternatives such as PAX [2] and data morphing [14]. The HYRISE project [13] has adapted data morphing techniques to the main memory environment. To control the scope of this project, we limited ourselves to studying the widely-used row-store and column-store layouts in this paper, but we do intend to expand our scope to other alternatives in the future. Note that the highly modular and flexible Quickstep storage system (see Section 2.1) makes it easy to integrate new storage formats into Quickstep.

6. CONCLUSION AND FUTURE WORK

In this paper, we have identified and evaluated key parts of the design space of storage organizations for main memory read-optimized databases that can have a major impact on the performance of analytic queries. Our empirical evaluation has found that block-based organization performs better than file-based organization, that column-stores and row-stores each have advantages for certain classes of queries (and both options should be considered for high-performance), that CSB+-Tree indices, co-located with data inside blocks, can play a major role in accelerating query performance, and that where data is amenable to compression and selectivity factors are sufficiently small, compression can also improve query performance.

The design space for storage organizations is quite large, and while we have explored the key dimensions of the design space in this paper, we fully expect that there are open portions in the design space that may be worth exploring. The Quickstep storage manager that we have produced as part of this work could be used to explore portions of the design space that are not covered in this study.

7. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? *SIGMOD*, pages 967–980, 2008.

[2] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB*, pages 198–215, 2002.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB*, pages 1064–1075, 2012.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Oszu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. *ICDE*, 2013.

[5] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business analytics in (a) blink. *ICDE*, pages 9–14, 2012.

[6] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. *SIGMOD*, pages 37–48, 2011.

[7] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, Dec. 2008.

[8] D. Campbell. Breakthrough performance with in-memory technologies. `https://blogs.technet.com/b/dataplatforminsider/archive/2012/11/08/breakthrough-performance-with-in-memory-technologies.aspx`, Nov. 2012.

[9] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116 – 127, March 2004.

[10] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. *The Benchmark Handbook for Database and Transaction Systems. Morgan Kaufmann*, 1993.

[11] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *SIGMOD*, pages 45–51, 2011.

[12] G. Graefe. Sorting and indexing with partitioned b-trees. *CIDR*, 2003.

[13] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *VLDB*, pages 105–116, 2010.

[14] R. A. Hankins and J. M. Patel. Data morphing: an adaptive, cache-conscious storage technique. *VLDB*, pages 417–428, 2003.

[15] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. *VLDB*, pages 487–498, 2006.

[16] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *VLDB*, pages 502–513, 2008.

[17] A. Kemper and T. Neumann. Hyper: A hybrid oltp & olap main memory database system based on virtual memory snapshots. *ICDE*, pages 195–206, 2011.

[18] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *VLDB*, pages 1378–1389, 2009.

[19] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *VLDB*, pages 1790–1801, 2012.

[20] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *VLDB*, pages 1648–1653, 2009.

[21] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB*, pages 53–72, 1997.

[22] B. Murthy, M. Goel, A. Lee, D. Granholm, and S. Cheung. Oracle exalytics in-memory machine: A brief introduction. `http://www.oracle.com/us/solutions/ent-performance-bi/business-intelligence/exalytics-bi-machine/overview/exalytics-introduction-1372418.pdf`, October 2011.

[23] D. R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, 1997.

[24] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, pages 183–194, 2011.

[25] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. *ICDE*, pages 60–69, 2008.

[26] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. *SIGMOD*, pages 475–486, 2000.

[27] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. *SIGMOD*, pages 558–569, 2002.

[28] K. A. Ross and K. A. Zaman. Serving datacube tuples from main memory. In *Scientific and Statistical Database Management*, pages 182–195. IEEE, 2000.

[29] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. *SIGMOD*, pages 23–34, 1979.

[30] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the vertica analytic database: Lessons learned. *ICDE*, 2013.

[31] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. *SIGMOD*, pages 731–742, 2012.

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

[33] VoltDB Inc. VoltDB Technical Overview. `http://voltdb.com/resources/whitepapers`, June 2011.

[34] K. M. Wilson and B. B. Aglietti. Dynamic page placement to improve locality in cc-numa multiprocessors for tpc-c. In *ACM/IEEE Conference on Supercomputing*, pages 33–33, 2001.

[35] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. *ICDE*, pages 1349–1350, 2012.

APPENDIX

A. JOIN PROCESSING

Processing of join queries in a main-memory engine is a very active area of research. Well-known join algorithms such as sort-merge join [3,18] and hash-join [4,6,9] are being adapted and tuned to highly parallel multi-core CPUs in the main memory environment. Joins are a higher-level query processing operation that, in general, are beyond the single-relation access plan-centric focus of this paper. Nevertheless, it is known that widely-used join algorithms do interact with the storage manager in a predictable way, and storage manager performance affects join performance. For instance, the previous study which we base our experiments on showed that hash-join performance is closely related to the scan performance of base tables [16], since both building a hash table and scanning the outer table to probe it involve a linear scan of the base tables in the join.

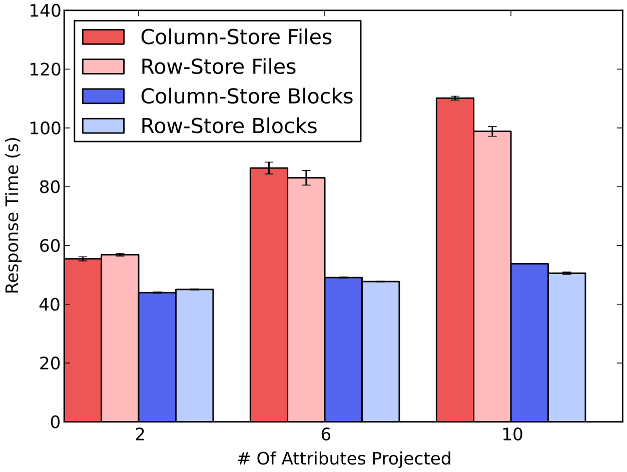
We evaluated a hash join in Quickstep, with both row-store and column-store data layout in both blocks and files. We use an outer table consisting of 750 million tuples based on the Narrow-U schema, but replace one column with a unique key. The inner table has a similar schema with 75 million tuples (1/10 as many, the same ratio of table sizes used in the join queries **Q10-Q17** of the Wisconsin benchmark [10]). Queries are of the form:

```
SELECT outer.col_a, ..., inner.col_a, ...  FROM
outer, inner WHERE outer.join_key = inner.join_key;
```

We vary the number of "payload columns" projected from each table and materialized in the output. As with our other experiments, we use 20 worker threads operating in parallel (for both the build and probe phases of the hash-join).

In all cases, we build a single global hash-table on the join key of the inner table (using a single global hash-table has been shown to outperform using smaller partitioned hash-tables when joining [6]). We use a concurrent hash-table implementation which allows us to build the global hash table in parallel, with multiple threads scanning different blocks or file partitions of the inner table and inserting entries in the hash table.

We show results for the actual join query in Figure 17. The query executes in two stages. In the first, worker threads scan the inner table in parallel and insert entries into the global hash table. In the second, worker threads scan the outer table in parallel and probe the hash table for a match on the join key. When a match is found, the projected columns are fetched from the inner and outer tables and materialized as a new row in the output (we do not claim that the projection method here is optimal, and acknowledge that aspects such as early vs. late materialization is an active an ongoing area of research [30], and beyond the scope of this paper). During the build stage, access on the inner table follows a linear scan pattern, while insertions into the hash table follow a random-access pattern. During the join stage, access on the outer table follows a linear scan pattern, and access on the hash table and the inner table follows a random-access pattern. The memory-access pattern of a hash-join therefore has some characteristics of a scan (scanning the inner table when building the hash table and the outer table when joining) and some which are similar to index-access (random-access probing of the hash table and fetching inner-table tuple values for projection). For all queries, the hash-build stage took about 7.5 seconds with files and 8.5 seconds with blocks[5], with the remaining bulk of the execution time (and most of the difference in performance) spent in the probe-and-project stage.



Organization	Blocks vs. Files
Tuple Storage Layout	Column-Store vs. Row Store
Indexing	None
Predicate	JOIN
Projection Width	Varies

Figure 17: Hash Join Performance

Block-based organization consistently outperforms file-based organization, due to improved locality of access in smaller blocks resulting in better cache behavior (consistent with our results for selection queries). For instance, when projecting 10 columns (5 from each table), row-stores in files incur 5.38 billion L3 cache misses and 14.4 billion L2 cache misses, but row stores in blocks incur only 4.58 billion L3 misses and 5.83 billion L2 misses. We also see a slight performance advantage for row-stores over column-stores when projecting several columns, since the random access to column values in the inner-table hits all the desired values on one or two contiguous cache lines in a row store, whereas they are in several disjoint cache lines in different column stripes in a column store. For example, when projecting 10 attributes in file-based organization, column-stores incur 6.41 billion L3 cache misses, but row-stores incur only 5.38 billion.

Observation 13. *For hash-join queries, block-based organization consistently outperforms file-based organization. For wider projections, row-stores slightly outperform column stores.*

[5] The slightly reduced performance for blocks in the build phase suggests that the block-based organization might actually be *too* parallelization-friendly relative to files during the build phase, since building the global hash table requires threads to sometimes synchronize when accessing shared hash-table buckets. Tuning the number of threads used in the hash-build phase below the number used in the probe-and-project phase is an interesting possible optimization, but general hash-join algorithm tuning is a broad area of research beyond the scope of this paper.

It should be noted that, in many analytics applications, there is a "star schema" where a large central fact table is connected to several smaller dimension tables by primary key-foreign key relationships. Queries on joins of the fact table with one or more dimension tables are extremely common. A widely-used optimization is to denormalize the star schema by pre-joining the fact table with the dimension tables and storing the result as a materialized view which single-table selection-projection and aggregate-grouping queries are run on. Some systems (for instance Vectorwise [20]) automatically apply this optimization by introducing a "join index" data structure, which is effectively a materialized view which the DBMS automatically creates based on the primary-foreign key relationships in the database schema.

Overall, results with Quickstep show that performance of in-memory hash-joins is very closely related to scan performance of the underlying tables, as well as the random-access performance of the inner table. Applying the common technique of pre-joining tables also effectively turns many join queries into scan queries.