Rose: Compressed, log-structured replication

Russell Sears
UC Berkeley
sears@cs.berkeley.edu

Mark Callaghan
Google
mcallaghan@google.com

Eric Brewer
UC Berkeley
brewer@cs.berkeley.edu

ABSTRACT

Rose[1] is a database storage engine for high-throughput replication. It targets seek-limited, write-intensive transaction processing workloads that perform near real-time decision support and analytical processing queries. Rose uses *log structured merge* (LSM) trees to create full database replicas using purely sequential I/O, allowing it to provide orders of magnitude more write throughput than B-tree based replicas. Also, LSM-trees cannot become fragmented and provide fast, predictable index scans.

Rose's write performance relies on replicas' ability to perform writes without looking up old values. LSM-tree lookups have performance comparable to B-tree lookups. If Rose read each value that it updated then its write throughput would also be comparable to a B-tree. Although we target replication, Rose provides high write throughput to any application that updates tuples without reading existing data, such as append-only, streaming and versioning databases.

We introduce a page compression format that takes advantage of LSM-tree's sequential, sorted data layout. It increases replication throughput by reducing sequential I/O, and enables efficient tree lookups by supporting small page sizes and doubling as an index of the values it stores. Any scheme that can compress data in a single pass and provide random access to compressed values could be used by Rose.

Replication environments have multiple readers but only one writer. This allows Rose to provide atomicity, consistency and isolation to concurrent transactions without resorting to rollback, blocking index requests or interfering with maintenance tasks.

Rose avoids random I/O during replication and scans, leaving more I/O capacity for queries than existing systems, and providing scalable, real-time replication of seek-bound workloads. Analytical models and experiments show that Rose provides orders of magnitude greater replication bandwidth over larger databases than conventional techniques.

1. INTRODUCTION

Rose is a database replication engine for workloads with high volumes of in-place updates. It is designed to provide high-throughput, general purpose transactional replication regardless of database size, query contention and update patterns. In particular, it is designed to run real-time decision support and analytical processing queries against some of today's largest TPC-C style online transaction processing applications.

Traditional database replication technologies provide acceptable performance if the application write set fits in RAM or if the storage system is able to update data in place quickly enough to keep up with the replication workload.

Transaction processing (OLTP) systems use fragmentation to optimize for small, low-latency reads and writes. They scale by adding memory and additional drives, increasing the number of I/O operations per second provided by the hardware. Data warehousing technologies introduce latency, giving them time to reorganize data for bulk insertion, and column stores optimize for scan performance at the expense of random access to individual tuples.

Rose combines the best features of the above approaches:

- High throughput writes, regardless of update patterns
- Scan performance comparable to bulk-loaded structures
- Low latency lookups and updates

We implemented Rose because existing replication technologies only met two of these three requirements. Rose achieves all three goals.

Rose is based on LSM-trees, which reflect updates immediately without performing disk seeks or resorting to fragmentation. This allows it to provide better write and scan throughput than B-trees. Unlike existing LSM-tree implementations, Rose makes use of compression, further increasing replication and scan performance.

Rose provides extremely high write throughput to applications that perform updates without performing reads, such as replication, append-only, streaming and versioning databases. Replication is a particularly attractive application of LSM-trees because it is common, well-understood and requires complete transactional semantics. Also, we know of no other scalable replication approach that provides real-time analytical queries over seek-bound transaction processing workloads.

[1] Replication Oriented Storage Engine

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

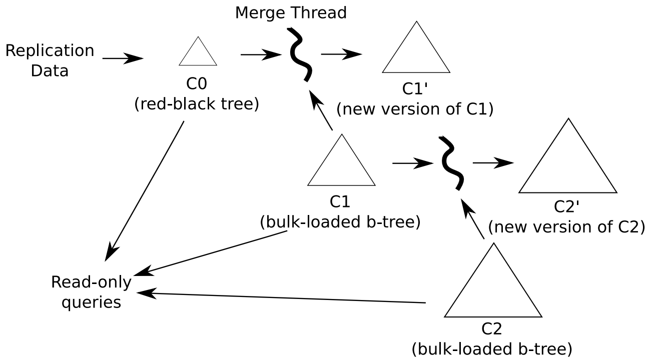

Figure 1: The structure of a Rose LSM-tree

2. SYSTEM OVERVIEW

A Rose replica takes a replication log as input and stores the changes it contains in a *log structured merge* (LSM) tree [11]. An LSM-tree is an index that consists of multiple sub-trees called *components* (Figure 1). The smallest component, $C0$, is a memory resident binary search tree that is updated in place. The next-smallest component, $C1$, is a bulk-loaded B-tree. $C0$ is merged with $C1$ as it grows. The merge process consists of index scans and produces a new bulk-loaded version of $C1$ that contains the updates from $C0$. LSM-trees can have arbitrarily many components, though three (two on-disk trees) is generally adequate. All other components are produced by repeated merges with the next smaller component. Therefore, LSM-trees are updated without using random disk I/O.

Lookups are performed by examining tree components, starting with the in-memory component and moving on to progressively larger and out-of-date trees until a match is found. This involves a maximum of two on-disk tree lookups unless a merge is in process, in which case three lookups may be required.

Rose's first contribution is to use compression to increase merge throughput. Compression reduces the amount of sequential I/O required by merges, trading surplus computational power for scarce storage bandwidth. Lookup performance is determined by page cache hit ratios and the number of seeks provided by the storage system. Rose's compression increases the effective size of the page cache.

The second contribution is the application of LSM-trees to database replication workloads. In order to take full advantage of LSM-trees' write throughput, the replication environment must not force Rose to obtain pre-images of overwritten values. Therefore, we assume that the replication log stores each transaction `begin`, `commit`, and `abort` performed by the master database, along with the pre- and post-images associated with each update. The ordering of these entries must match the order in which they are applied at the database master. Workloads that do not contain transactions or do not update existing data may omit some of the mechanisms described here.

Rose immediately applies updates to $C0$, making them available to queries that do not require a consistent view of the data. Rose supports transactional isolation by delaying the availability of new data for the duration of a few update transactions. It then produces a new, consistent snapshot that reflects the new data. Multiple snapshots can coexist, allowing read-only transactions to run without creating lock contention or being aborted due to concurrent updates.

Long-running transactions do not block Rose's maintenance tasks or index operations. However, long-running read-only queries delay the deallocation of stale data, and long-running updates introduce replication delay. Rose's concurrency control mechanisms are described in Section 2.5.

Rose merges tree components in background threads, allowing it to continuously process updates and service index lookup requests. Index lookups minimize the overhead of thread synchronization by latching entire tree components at a time. On-disk tree components are read-only so these latches only block reclamation of space used by deallocated tree components. $C0$ is updated in place, preventing inserts from occurring concurrently with lookups. However, operations on $C0$ are comparatively fast, reducing contention for $C0$'s latch.

Recovery, allocation and atomic updates to Rose's metadata are handled by Stasis [13], an extensible transactional storage system. Rose is implemented as a set of custom Stasis page formats and tree structures.

In order to provide inexpensive queries to clients Rose assumes the replication log is made durable by some other mechanism. Rather than commit each replicated transaction independently, it creates one Stasis transaction each time a tree component is created. These transactions generate negligible amounts of log entries.

Redo and undo information for tree component metadata and headers are written to the Stasis log. Data contained in tree components are never written to the log and tree component contents are sequentially forced to the page file at Stasis commit. During recovery, partially written tree components are deallocated and tree headers are brought to a consistent state.

After Stasis recovery completes, Rose is in a consistent state that existed at some point in the past. The replication log must be replayed from that point to finish recovery.

Rose determines where to begin replaying the log by consulting its metadata, which is updated each time a tree merge completes. This metadata contains the timestamp of the last update that is reflected in $C1$, which is simply the timestamp of the last tuple written to $C0$ before the merge began.

2.1 Tree merging

Rose's LSM-trees always consist of three components ($C0$, $C1$ and $C2$), as this provides a good balance between insertion throughput and lookup cost. Updates are applied directly to the in-memory tree, and repeated tree merges limit the size of $C0$. These tree merges produce a new version of $C1$ by combining tuples from $C0$ with tuples in the existing version of $C1$. When the merge completes $C1$ is atomically replaced with the new tree and $C0$ is atomically replaced with an empty tree. The process is eventually repeated when $C1$ and $C2$ are merged.

Replacing entire trees at once introduces a number of problems. It doubles the number of bytes used to store each component, which is important for $C0$, as it doubles memory requirements. It is also important for $C2$, as it doubles the amount of disk space used by Rose. Finally, ongoing merges force index probes to access both versions of $C1$, increasing random lookup times.

The original LSM-tree work proposes a more sophisticated scheme that addresses these issues by replacing one sub-tree at a time. This reduces peak storage and memory requirements but adds complexity by requiring in-place updates of tree components.

An alternative approach would partition Rose into multiple LSM-trees and merge a single partition at a time [8]. This would reduce the frequency of extra lookups caused by ongoing tree merges. Partitioning also improves write throughput when updates are skewed, as unchanged portions of the tree do not participate in merges and frequently changing partitions can be merged more often. We do not consider these optimizations in the discussion below; including them would complicate the analysis without providing any new insight.

2.2 Amortized insertion cost

This section provides an overview of LSM-tree performance. A more thorough analytical discussion [11], and comparisons between LSM-trees and a wide variety of other indexing techniques [8] are available elsewhere.

To compute the amortized LSM-tree insertion cost we consider the cost of comparing the inserted tuple with existing tuples. Each tuple insertion ultimately causes two rounds of I/O operations. One merges the tuple into $C1$; the other merges it into $C2$. Insertions do not initiate more I/O once they reach $C2$.

Write throughput is maximized when the ratio of the sizes of $C1$ to $C0$ is equal to the ratio between $C2$ and $C1$ [11]. This ratio is called R, and:

$$size\ of\ tree \approx\ R^2 * |C0|$$

Merges examine an average of R tuples from $C1$ for each tuple of $C0$ they consume. Similarly, R tuples from $C2$ are examined each time a tuple in $C1$ is consumed. Therefore:

$$insertion\ rate * R(t_{C2} + t_{C1}) \approx\ sequential\ i/o\ cost$$

Where t_{C1} and t_{C2} are the amount of time it takes to read from and write to C1 and C2.

2.3 Replication Throughput

LSM-trees have different asymptotic performance characteristics than conventional index structures. In particular, the amortized cost of insertion is $O(\sqrt{n}\ log\ n)$ in the size of the data and is proportional to the cost of sequential I/O. In a B-tree, this cost is $O(log\ n)$ but is proportional to the cost of random I/O. This section describes the impact of compression on B-tree and LSM-tree performance.

We use simplified models of each structure's performance characteristics. In particular, we assume that the leaf nodes do not fit in memory, that tuples are accessed randomly with equal probability, and that internal tree nodes fit in RAM. Without a skewed update distribution, reordering and batching I/O into sequential writes only helps if a significant fraction of the tree's data fits in RAM. Therefore, we do not consider B-tree I/O batching here.

In B-trees, each update reads a page and eventually writes it back:

$$cost_{Btree\ update} = 2\ cost_{random\ io}$$

In Rose, we have:

$$cost_{LSMtree\ update} = 2 * 2 * 2 * R * \frac{cost_{sequential\ io}}{compression\ ratio}$$

We multiply by $2R$ because each new tuple is eventually merged into both on-disk components, and each merge involves an average of R comparisons with existing tuples. The second factor of two reflects the fact that the merger must read existing tuples into memory before writing them back to disk.

An update of a tuple is handled as an insertion of the new tuple and a deletion of the old tuple. Deletion is an insertion of a special tombstone tuple that records the deletion of the old version of the tuple, leading to the third factor of two. Updates that do not modify primary key fields avoid this final factor of two. The delete and insert share the same primary key and snapshot number, so the insertion always supersedes the deletion. Therefore, there is no need to insert a tombstone.

The *compression ratio* is $\frac{uncompressed\ size}{compressed\ size}$, so improved compression leads to less expensive LSM-tree updates. For simplicity, we assume the compression ratio is the same throughout each component of the LSM-tree; Rose addresses this at runtime by reasoning in terms of the number of pages used by each component.

Our test hardware's hard drive is a 7200RPM, 750 GB Seagate Barracuda ES. Third party benchmarks [14] report random access times of 12.3/13.5 msec (read/write) and 44.3-78.5 megabytes/sec sustained throughput. Timing `dd if=/dev/zero of=file; sync` on an empty ext3 file system suggests our test hardware provides 57.5 megabytes/sec of storage bandwidth, but running a similar test via Stasis' buffer manager produces a multimodal distribution ranging from 22 to 47 megabytes/sec.

Assuming a fixed hardware configuration and measuring cost in disk time, we have:

$$cost_{sequential} = \frac{|tuple|}{78.5MB/s} = 12.7\ |tuple|\ \ nsec/tuple\ (min)$$

$$cost_{sequential} = \frac{|tuple|}{44.3MB/s} = 22.6\ |tuple|\ \ nsec/tuple\ (max)$$

and

$$cost_{random} = \frac{12.3 + 13.5}{2} = 12.9\ msec/tuple$$

Assuming $44.3mb/s$ sequential bandwidth:

$$2\ cost_{random} \approx 1,000,000 \frac{cost_{sequential}}{|tuple|}$$

yields:

$$\frac{cost_{LSMtree\ update}}{cost_{Btree\ update}} = \frac{2 * 2 * 2 * R * cost_{sequential}}{compression\ ratio * 2 * cost_{random}}$$

$$\approx \frac{R * |tuple|}{250,000 * compression\ ratio}$$

If tuples are 100 bytes and we assume a compression ratio of 4, which is lower than we expect to see in practice, but numerically convenient, then the LSM-tree outperforms the B-tree when:

$$R < \frac{250,000 * compression\ ratio}{|tuple|}$$

$$R < 10,000$$

A 750 GB tree with a 1GB $C0$ would have an R of $\sqrt{750} \approx 27$. If tuples are 100 bytes such a tree's sustained insertion throughput will be approximately 8000 tuples/sec or 800 kilobytes/sec.

Our hard drive's average access time allows the drive to deliver 83 I/O operations/sec. Therefore, we can expect an insertion throughput of 41.5 tuples/sec from a B-tree that does not cache leaf pages. With 1 GB of RAM, Rose should outperform the B-tree by more than two orders of magnitude. Increasing Rose's system memory to cache 10 GB of tuples would increase write performance by a factor of $\sqrt{10}$.

Increasing memory another ten fold to 100 GB would yield an LSM-tree with an R of $\sqrt{750/100} = 2.73$ and a throughput of 81,000 tuples/sec. In contrast, the B-tree would cache less than 100 GB of leaf pages in memory and would write fewer than $\frac{41.5}{1-(100/750)} = 47.9$ tuples/sec. Increasing memory further yields a system that is no longer disk bound.

Assuming CPUs are fast enough to allow Rose to make use of the bandwidth supplied by the disks, we conclude that Rose will provide significantly higher throughput than a seek-bound B-tree.

2.4 Indexing

Our analysis ignores the cost of bulk-loading and storing LSM-trees' internal nodes. Each time the merge process fills a page it inserts an entry into the rightmost position in the tree, allocating additional internal nodes if necessary. Our prototype does not compress internal tree nodes.

The space overhead of building these tree nodes depends on the number of tuples that fit in each page. If tuples take up a small fraction of each page then Rose gets good fan-out on internal nodes, reducing the fraction of storage used by those nodes. Therefore, as page size increases, tree overhead decreases. Similarly, as tuple sizes increase, the fraction of storage dedicated to internal tree nodes increases.

Rose pages are 4KB. Workloads with larger tuples would save disk and page cache space by using larger pages for internal nodes. For very large trees, larger internal tree nodes also avoid seeks during index lookups.

2.5 Isolation

Rose handles two types of transactions: updates from the master and read-only queries from clients. Rose has no control over the order or isolation of updates from the master database. Therefore, it must apply these updates in an order consistent with the master database in a way that isolates queries from concurrent modifications.

LSM-tree updates do not immediately remove pre-existing data from the tree. Instead, they provide a simple form of versioning where data from the newest component ($C0$) is always the most up-to-date. We extend this idea to provide snapshots. Each transaction is assigned to a snapshot, and no more than one version of each tuple exists within a snapshot.

To lookup a tuple as it existed at some point in time, we examine all snapshots that were taken before that point and return the most recent version of the tuple that we find. This is inexpensive because all versions of the same tuple are stored in adjacent positions in the LSM-tree. Furthermore, Rose's versioning is meant to be used for transactional consistency and not time travel, limiting the number of snapshots.

To insert a tuple, Rose first determines to which snapshot it should belong. At any given point in time up to two snapshots may accept new updates. One snapshot accepts new transactions. If the replication log may contain interleaved transactions, such as when there are multiple master databases, then a second, older snapshot waits for any pending transactions that it contains to complete. Once all of those transactions are complete, the older snapshot stops accepting updates, the first snapshot stops accepting new transactions, and a snapshot for new transactions is created.

Rose examines the timestamp and transaction ID associated with the tuple insertion and marks the tuple with the appropriate snapshot ID. It then inserts the tuple into $C0$, overwriting any tuple that matches on primary key and has the same snapshot ID.

The merge thread uses a list of currently accessible snapshots to decide which versions of the tuple in $C0$ and $C1$ are accessible to transactions. Such versions are the most recent copy of the tuple that existed before the end of an accessible snapshot. After finding all such versions of a tuple (one may exist for each accessible snapshot), the merge process writes them to the new version of $C1$. Any other versions of the tuple are discarded. If two matching tuples from the same snapshot are encountered then one must be from $C0$ and the other from $C1$. The version from $C0$ is more recent, so the merger discards the version from $C1$. Merges between $C1$ and $C2$ work the same way as merges between $C0$ and $C1$.

Rose handles tuple deletion by inserting a tombstone. Tombstones record the deletion event and are handled similarly to insertions. A tombstone will be kept if it is the newest version of a tuple in at least one accessible snapshot. Otherwise, the tuple was reinserted before next accessible snapshot was taken and the tombstone is deleted. Unlike insertions, tombstones in $C2$ are deleted if they are the oldest remaining reference to a tuple.

Rose's snapshots have minimal performance impact, and provide transactional concurrency control without rolling back transactions or blocking the merge and replication processes. However, long-running updates prevent queries from accessing the results of recent transactions, leading to stale results. Long-running queries increase Rose's disk footprint by increasing the number of accessible snapshots.

2.6 Parallelism

Rose operations are concurrent; readers and writers work independently, avoiding blocking, deadlock and livelock. Index probes must latch $C0$ to perform a lookup, but the more costly probes into $C1$ and $C2$ are against read-only trees. Beyond locating and pinning tree components against deallocation, probes of these components do not interact with the merge processes.

Each tree merge runs in a separate thread. This allows our prototype to exploit two to three processor cores while inserting and recompressing data during replication. Remaining cores could be exploited by range partitioning a replica into multiple LSM-trees, allowing more merge processes to run concurrently. Therefore, we expect the throughput of Rose replication to increase with memory size, compression ratios and I/O bandwidth for the foreseeable future.

3. ROW COMPRESSION

Rose stores tuples in a sorted, append-only format. This simplifies compression and provides a number of new oppor-

Table 1: Compression ratios and index overhead - five columns (20 bytes/column)

Format	Compression	Page count
PFOR	1.96x	2494
PFOR + tree	1.94x	+80
RLE	3.24x	1505
RLE + tree	3.22x	+21

Table 2: Compression ratios and index overhead - 100 columns (400 bytes/column)

Format	Compression	Page count
PFOR	1.37x	7143
PFOR + tree	1.17x	8335
RLE	1.75x	5591
RLE + tree	1.50x	6525

tunities for optimization. Compression reduces sequential I/O, which is Rose's primary bottleneck. It also increases the effective size of the buffer pool, allowing Rose to service larger read sets without resorting to random I/O.

Row-oriented database compression techniques must cope with random, in-place updates and provide efficient random access to compressed tuples. In contrast, compressed column-oriented database layouts focus on high-throughput sequential access, and do not provide in-place updates or efficient random access. Rose never updates data in place, allowing it to use append-only compression techniques from the column database literature. Also, Rose's tuples never span pages and are stored in sorted order. We modified column compression techniques to provide an index over each page's contents and efficient random access within pages.

Rose's compression format is straightforward. Each page is divided into a header, a set of compressed segments and a single exception section (Figure 2). There is one compressed segment per column and each such segment may be managed by a different compression algorithm, which we call a *compressor*. Some compressors cannot directly store all values that a column may take on. Instead, they store such values in the exception section. We call Rose's compressed page format the *multicolumn* format, and have implemented it to support efficient compression, decompression and random lookups by slot id and by value.

3.1 Compression algorithms

The Rose prototype provides three compressors. The first, *NOP*, simply stores uncompressed integers. The second, *RLE*, implements run length encoding, which stores values as a list of distinct values and repetition counts. The third, *PFOR*, or patched frame of reference, stores values as a single per-page base offset and an array of deltas [17]. The values are reconstructed by adding the offset and the delta or by traversing a pointer into the exception section of the page. Currently, Rose only supports integer data although its page formats could be easily extended with support for strings and other types. Also, the compression algorithms that we implemented would benefit from a technique known as *bit-packing*, which represents integers with lengths other than 8, 16, 32 and 64 bits. Bit-packing would increase Rose's compression ratios, and can be implemented with a modest performance overhead [17].

We chose these techniques because they are amenable to optimization; our implementation makes heavy use of C++ templates, static code generation and g++'s optimizer to keep Rose from becoming CPU-bound. By hardcoding table formats at compilation time, this implementation removes length and type checks, and allows the compiler to inline methods, remove temporary variables and optimize loops.

Rose includes a second, less efficient implementation that uses dynamic method dispatch to support runtime creation of new table schemas. A production-quality system could use runtime code generation to support creation of new schemas while taking full advantage of the compiler's optimizer.

3.2 Comparison with other approaches

Like Rose, the PAX [2] page format stores tuples in a way that never spans multiple pages and partitions data within pages by column. Partitioning pages by column improves processor cache locality for queries that do not need to examine every column within a page. In particular, many queries filter based on the value of a few attributes. If data is clustered into rows then each column from the page will be brought into cache, even if no tuples match. By clustering data into columns, PAX ensures that only the necessary columns are brought into cache.

Rose's use of lightweight compression algorithms is similar to approaches used in column databases [17]. Rose differs from that work in its focus on small, low-latency updates and efficient random reads on commodity hardware. Existing work targeted RAID and used page sizes ranging from 8-32MB. Rather than use large pages to get good sequential I/O performance, Rose relies upon automatic prefetch and write coalescing provided by Stasis' buffer manager and the underlying operating system. This reduces the amount of data handled by Rose during index probes.

3.3 Tuple lookups

Because we expect index lookups to be a frequent operation, our page format supports efficient lookup by tuple value. The original "patched" compression algorithms store exceptions in a linked list and perform scans of up to 128 tuples in order to materialize compressed tuples [17]. This reduces the number of branches required during compression and sequential decompression. We investigated a number of implementations of tuple lookup by value including per-column range scans and binary search.

The efficiency of random access within a page depends on the format used by column compressors. Rose compressors support two access methods. The first looks up a value by slot id. This operation is $O(1)$ for frame of reference columns and $O(log\ n)$ in the number of runs of identical values on the page for run length encoded columns.

The second operation is used to look up tuples by value and is based on the assumption that the the tuples are stored in the page in sorted order. The simplest way to provide access to tuples by value would be to perform a binary search, materializing each tuple that the search needs to examine. Doing so would materialize many extra column values, potentially performing additional binary searches.

To lookup a tuple by value, the second operation takes a range of slot ids and a value, and returns the offset of the first and last instance of the value within the range. This operation is $O(log\ n)$ in the number of slots in the range for

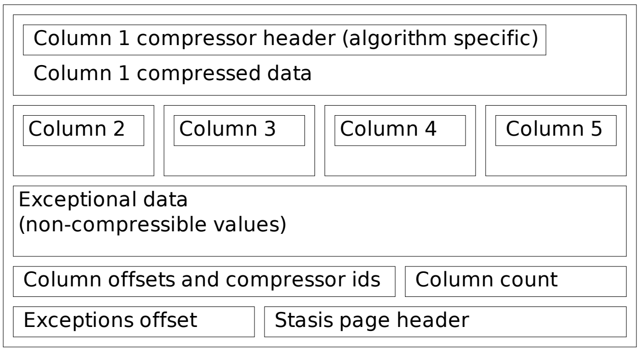

Figure 2: Multicolumn page format. Many compression algorithms can coexist on a single page. Tuples never span multiple pages.

Table 3: Compressor throughput - Random data Mean of 5 runs, $\sigma < 5\%$, except where noted

Format	Ratio	Compress	Decompress
PFOR - 1 column	3.96x	547 mb/s	2959 mb/s
PFOR - 10 column	3.86x	256	719
RLE - 1 column	48.83x	960	1493 (12%)
RLE - 10 column	47.60x	358 (9%)	659 (7%)

frame of reference columns and $O(log\ n)$ in the number of runs on the page for run length encoded columns. The multicolumn implementation uses this method to look up tuples by beginning with the entire page in range and calling each compressor's implementation to narrow the search until the correct tuple(s) are located or the range is empty. Partially matching tuples are only partially decompressed during the search, reducing the amount of data brought into processor cache.

3.4 Multicolumn computational overhead

Our multicolumn page format introduces additional computational overhead. Rose compresses each column in a separate buffer then uses `memcpy()` to gather this data into a single page buffer before writing it to disk. This happens once per page allocation.

Bit-packing is typically implemented as a post-processing technique that makes a pass over the compressed data [17]. Rose's gathering step can be performed for free by such a bit-packing implementation, so we do not see this as a significant disadvantage compared to other approaches.

Also, Rose needs to translate tuple insertions into calls to appropriate page formats and compression implementations. Unless we hardcode the Rose executable to support a predefined set of page formats and table schemas, each tuple compression and decompression operation must execute an extra `for` loop over the columns. The `for` loop's body contains a `switch` statement that chooses between column compressors, since each column can use a different compression algorithm and store a different data type.

This form of multicolumn support introduces significant overhead; these variants of our compression algorithms run significantly slower than versions hard-coded to work with single column data. Table 3 compares a fixed-format single column page layout with Rose's dynamically dispatched (not custom generated code) multicolumn format.

3.5 The `append()` operation

Rose's compressed pages provide a `tupleAppend()` operation that takes a tuple as input and returns `false` if the page does not have room for the new tuple. `tupleAppend()` consists of a dispatch routine that calls `append()` on each column in turn. `append()` has the following signature:

```
void append(COL_TYPE value, int* exception_offset,
    void* exceptions_base, void* column_base,
    int* freespace)
```

where `value` is the value to be appended to the column, `exception_offset` is a pointer to the offset of the first free byte in the exceptions region, and `exceptions_base` and `column_base` point to page-sized buffers used to store exceptions and column data as the page is being written. One copy of these buffers exists for each LSM-tree component; they do not significantly increase Rose's memory requirements.

`freespace` is a pointer to the number of free bytes remaining on the page. The multicolumn format initializes these values when the page is allocated. As `append()` implementations are called they update this data accordingly. Each time a column value is written to a page the column compressor must allocate space to store the new value. In a naive allocation approach the compressor would check `freespace` to decide if the page is full and return an error code otherwise. Then its caller would check the return value and behave appropriately. This would lead to two branches per column value, greatly decreasing compression throughput.

We avoid these branches by reserving a portion of the page approximately the size of a single incompressible tuple for speculative allocation. Instead of checking for freespace, compressors decrement the current freespace count and append data to the end of their segment. Once an entire tuple has been written to a page, it checks the value of freespace and decides if another tuple may be safely written. This also simplifies corner cases; if a compressor cannot store more tuples on a page, but has not run out of space[2] it simply sets `freespace` to -1 and returns.

The original PFOR implementation [17] assumes it has access to a buffer of uncompressed data and is able to make multiple passes over the data during compression. This allows it to remove branches from loop bodies, improving compression throughput. We opted to avoid this approach in Rose because it would increase the complexity of the `append()` interface and add a buffer to Rose's merge threads.

3.6 Buffer manager interface extensions

In the process of implementing Rose, we added a new API to Stasis' buffer manager implementation. It consists of four functions: `pageLoaded()`, `pageFlushed()`, `pageEvicted()`, and `cleanupPage()`.

Stasis supports custom page layouts. These layouts control the byte level format of pages and must register callbacks that will be invoked by Stasis at appropriate times. The first three are invoked by the buffer manager when it loads an existing page from disk, writes a page to disk, and evicts a page from memory.

The fourth is invoked by page allocation routines immediately before a page is reformatted to use a different layout.

[2] This can happen when a run length encoded page stores a very long run of identical tuples.

This allows the page's old layout's implementation to free any in-memory resources that it associated with the page during initialization or when `pageLoaded()` was called.

We register implementations for these functions because Stasis maintains background threads that control eviction of Rose's pages from memory. As we mentioned above, multicolumn pages are split into a number of temporary buffers while they are being created and are then packed into a contiguous buffer before being flushed. Multicolumn's `pageFlushed()` callback guarantees that this happens before the page is written to disk. `pageLoaded()` parses the page headers and associates statically generated compression code with each page as it is read into memory. `pageEvicted()` and `cleanupPage()` free memory that is allocated by `pageLoaded()`.

`pageFlushed()` could be safely executed in a background thread with minimal impact on system performance. However, the buffer manager was written under the assumption that the cost of in-memory operations is negligible. Therefore, it blocks all buffer management requests while `pageFlushed()` is being executed. In practice, calling `pack()` from `pageFlushed()` would block multiple Rose threads.

Also, `pack()` reduces Rose's memory utilization by freeing up temporary compression buffers. For these reasons, the merge threads explicitly invoke `pack()` instead of waiting for `pageFlushed()` to be called.

3.7 Storage overhead

The multicolumn page format is similar to the format of existing column-wise compression formats. The algorithms we implemented have page formats that can be divided into two sections. The first section is a header that contains an encoding of the size of the compressed region and perhaps a piece of uncompressed data. The second section typically contains the compressed data.

A multicolumn page contains this information in addition to metadata describing the position and type of each column. The type and number of columns could be encoded in the "page type" field or be explicitly represented using a few bytes per page column. If we use 16 bits to represent the page offset and 16 bits for the column compressor type then the additional overhead for each column is four bytes plus the size of the compression format headers.

A frame of reference column header consists of a single uncompressed value and two bytes to record the number of encoded rows. Run length encoding headers consist of a two byte count of compressed blocks. Therefore, in the worst case (frame of reference encoding 64-bit integers, and 4KB pages) our prototype's multicolumn format uses 14 bytes, or 0.35% of the page to store each column header. Bitpacking and storing a table that maps page types to lists of column and compressor types would reduce the size of the page headers.

In cases where the data does not compress well and tuples are large, additional storage is wasted because Rose does not split tuples across pages. Table 2 illustrates this; it was generated in the same way as Table 1, except that 400 byte tuples were used instead of 20 byte tuples. Larger pages would reduce the impact of this problem. We chose 4KB pages because they are large enough for the schemas we use to benchmark Rose, but small enough to illustrate the overheads of our page format.

Table 4: Weather data schema

Column Name	Compression Format	Key
Longitude	RLE	*
Latitude	RLE	*
Timestamp	PFOR	*
Weather conditions	RLE	
Station ID	RLE	
Elevation	RLE	
Temperature	PFOR	
Wind Direction	PFOR	
Wind Speed	PFOR	
Wind Gust Speed	RLE	

4. EVALUATION

4.1 Raw write throughput

To evaluate Rose's raw write throughput, we used it to index weather data. The data set ranges from May 1, 2007 to Nov 2, 2007 and contains readings from ground stations around the world [10]. Our implementation assumes these values will never be deleted or modified. Therefore, for this experiment the tree merging threads do not perform versioning or snapshotting. This data is approximately $1.3GB$ when stored in an uncompressed tab delimited file. We duplicated the data by changing the date fields to cover ranges from 2001 to 2009, producing a 12GB ASCII dataset that contains approximately 132 million tuples.

Duplicating the data should have a limited effect on Rose's compression ratios. We index on geographic position, placing all readings from a particular station in a contiguous range. We then index on date, separating duplicate versions of the same tuple from each other.

Rose only supports integer data types. We store ASCII columns for this benchmark by packing each character into 5 bits (the strings consist of A-Z, "+," "-" and "*") and storing them as integers. Floating point columns in the original data set are always represented with two digits of precision; we multiply them by 100, yielding an integer. The data source uses nonsensical readings such as -9999.00 to represent NULL. Our prototype does not understand NULL, so we leave these fields intact.

We represent each integer column as a 32-bit integer, even when we could use a 16-bit value. The "weather conditions" field is packed into a 64-bit integer. Table 4 lists the columns and compression algorithms we assigned to each column. The "Key" column indicates that the field was used as part of a B-tree primary key.

In this experiment, we randomized the order of the tuples and inserted them into the index. We compare Rose's performance with the MySQL InnoDB storage engine. We chose InnoDB because it has been tuned for bulk load performance, and avoid the overhead of SQL insert statements and MySQL transactions by using MySQL's bulk load interface. Data is loaded 100,000 tuples at a time so that MySQL inserts values into its tree throughout the run and does not sort and bulk load the data all at once.

InnoDB's buffer pool size was 2GB and its log file size was 1GB. We enabled InnoDB's doublewrite buffer, which writes a copy of each updated page to a sequential log. The doublewrite buffer increases the total amount of I/O performed, but decreases the frequency with which InnoDB

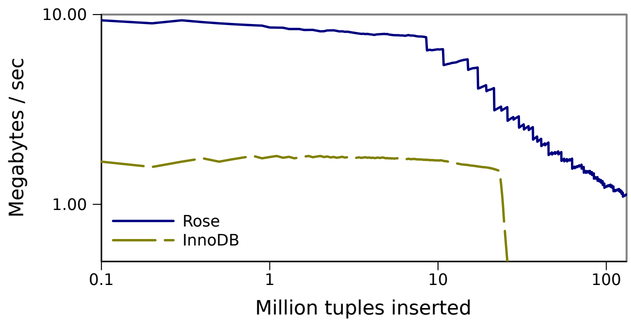

Figure 3: Mean insertion throughput (log-log). We truncated InnoDB's trend line; throughput was 20kb/s ($\frac{1}{57}$th of Rose's) after 50 million insertions.

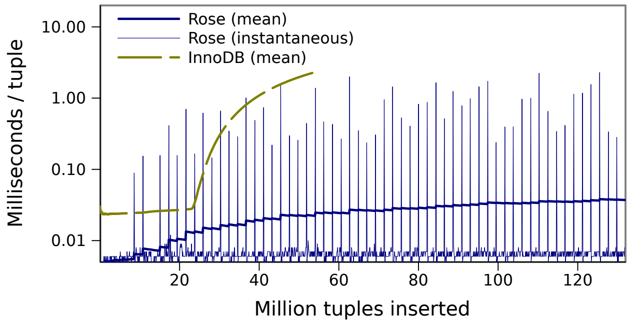

Figure 4: Tuple insertion time (log). "Instantaneous" is the mean over 100,000 tuples. Rose's cost increases with $\sqrt{n}$; InnoDB's increases linearly.

calls `fsync()` while writing dirty pages to disk. This increases replication throughput for this workload.

We compiled Rose's C components with "-O2", and the C++ components with "-O3". The later compiler flag is crucial, as compiler inlining and other optimizations improve Rose's compression throughput significantly. Rose was set to allocate $1GB$ to $C0$ and another $1GB$ to its buffer pool. In this experiment, Rose's buffer pool is essentially wasted once its page file size exceeds 1GB; Rose accesses the page file sequentially and evicts pages using LRU, leading to a cache hit rate near zero.

Our test hardware has two dual core 64-bit 3GHz Xeon processors with 2MB of cache (Linux reports 4 CPUs) and 8GB of RAM. We disabled the swap file and unnecessary system services. Datasets large enough to become disk bound on this system are unwieldy, so we `mlock()` 5.25GB of RAM to prevent it from being used by experiments. The remaining 750MB is used to cache binaries and to provide Linux with enough page cache to prevent it from unexpectedly evicting portions of the Rose binary. We monitored Rose throughout the experiment, confirming that its resident memory size was approximately 2GB.

All software used during our tests was compiled for 64-bit architectures. We used a 64-bit Ubuntu Gutsy (Linux 2.6.22-14-generic) installation and its prebuilt MySQL package (5.0.45-Debian_1ubuntu3). All page files and logs were stored on a dedicated drive. The operating system, executables and input files were stored on another drive.

4.2 Comparison with conventional techniques

Rose provides roughly 4.7 times more throughput than InnoDB at the beginning of the experiment (Figure 3). InnoDB's throughput remains constant until it starts to perform random I/O, which causes throughput to drop sharply.

Rose's performance begins to fall off earlier due to merging and because its page cache is half the size of InnoDB's. However, Rose does not fall back on random I/O and maintains significantly higher throughput than InnoDB throughout the run. InnoDB's peak write throughput was 1.8 mb/s and dropped to 20 kb/s after 50 million tuples were inserted. The InnoDB trend line in Figure 3 has been truncated for readability. In contrast, Rose maintained an average throughput of 1.13 mb/sec over the entire 132 million tuple dataset.

Rose merged $C0$ and $C1$ 59 times and merged $C1$ and $C2$ 15 times. At the end of the run (132 million tuple insertions) $C2$ took up 2.8GB and $C1$ was 250MB. The actual page file was 8.0GB, and the minimum possible size was 6GB. InnoDB used 5.3GB after 53 million tuple insertions.

4.3 Comparison with analytical model

In this section we measure update latency and compare measured write throughput with the analytical model's predicted throughput.

Figure 4 shows tuple insertion times for Rose and InnoDB. The "Rose (instantaneous)" line reports insertion times averaged over 100,000 insertions, while the other lines are averaged over the entire run. The periodic spikes in instantaneous tuple insertion times occur when an insertion blocks waiting for a tree merge to complete. This happens when one copy of $C0$ is full and the other one is being merged with $C1$. Admission control would provide consistent insertion times.

Figure 5 compares our prototype's performance to that of an ideal uncompressed LSM-tree. The cost of an insertion is $4R$ times the number of bytes written. We can approximate an optimal value for R by taking $\sqrt{\frac{|C2|}{|C0|}}$, where tree component size is measured in number of tuples. This factors out compression and memory overheads associated with $C0$.

We use this value to generate Figure 5 by multiplying Rose's replication throughput by $1 + 4R$. The additional 1 is the cost of reading the inserted tuple from $C1$ during the merge with $C2$.

Rose achieves 2x compression on the real data set. For comparison, we ran the experiment with compression disabled, and with run length encoded synthetic datasets that lead to 4x and 8x compression ratios. Initially, all runs exceed optimal throughput as they populate memory and produce the initial versions of $C1$ and $C2$. Eventually, each run converges to a constant effective disk utilization roughly proportional to its compression ratio. This shows that Rose continues to be I/O bound with higher compression ratios.

The analytical model predicts that, given the hard drive's 57.5 mb/s write bandwidth, an ideal Rose implementation would perform about twice as fast as our prototype. We believe the difference is largely due to Stasis' buffer manager, which delivers between 22 and 47 mb/s of sequential bandwidth on our test hardware. The remaining difference in throughput is probably due to Rose's imperfect overlapping of computation with I/O and coarse-grained control

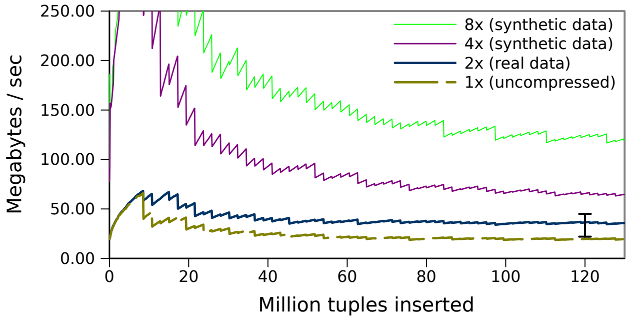

Figure 5: Disk bandwidth an uncompressed LSM-tree would require to match Rose's throughput. The buffer manager provides 22-45 mb/s.

over component sizes and R.

Despite these limitations, compression allows Rose to deliver significantly higher throughput than would be possible with an uncompressed LSM-tree. We expect Rose's throughput to increase with the size of RAM and storage bandwidth for the foreseeable future.

4.4 TPC-C / H

TPC-H is an analytical processing benchmark that targets periodically bulk-loaded data warehousing systems. In particular, compared to TPC-C, it de-emphasizes transaction processing and rollback and allows database vendors to permute the dataset off-line. In real-time database replication environments faithful reproduction of transaction processing schedules is important and there is no opportunity to sort data before making it available to queries. Therefore, we insert data in chronological order.

Our Rose prototype is far from a complete storage engine implementation. Rather than implement relational algebra operations and attempt to process and appropriately optimize SQL queries on top of Rose, we chose a small subset of the TPC-H and C benchmark queries and wrote custom code to invoke appropriate Rose tuple modifications, table scans and index lookup requests. For simplicity, updates and queries are performed by a single thread.

When modifying TPC-H and C for our experiments, we follow an existing approach [12, 7] and start with a precomputed join and projection of the TPC-H dataset. We use the schema described in Table 5, and populate the table by using a scale factor of 30 and following the random distributions dictated by the TPC-H specification. The schema for this experiment is designed to have poor update locality.

Updates from customers are grouped by order id, but the index is sorted by product and date. This forces the database to permute these updates into an order that would provide suppliers with inexpensive access to lists of orders to be filled and historical sales information for each product.

We generate a dataset containing a list of product orders, and insert tuples for each order (one for each part number in the order), then add a number of extra transactions. The following updates are applied in chronological order:

- Following TPC-C's lead, 1% of orders are immediately cancelled and rolled back. This is handled by inserting a tombstone for the order.

Table 5: TPC-C/H schema

Column Name	Compression Format	Data type
Part # + Supplier	RLE	int32
Year	RLE	int16
Week	NONE	int8
Day of week	NONE	int8
Order number	NONE	int64
Quantity	NONE	int8
Delivery Status	RLE	int8

- Remaining orders are delivered in full within the next 14 days. The order completion time is chosen uniformly at random between 0 and 14 days after the order was placed.
- The status of each line item is changed to "delivered" at a time chosen uniformly at random before the order completion time.

The following read-only transactions measure the performance of Rose's access methods:

- Every 100,000 orders we initiate a table scan over the entire data set. The per-order cost of this scan is proportional to the number of orders processed so far.
- 50% of orders are checked with order status queries. These are simply index probes that lookup a single line item (tuple) from the order. Orders that are checked with status queries are checked 1, 2 or 3 times with equal probability. Line items are chosen randomly with equal probability.
- Order status queries happen with a uniform random delay of 1.3 times the order processing time. For example, if an order is fully delivered 10 days after it is placed, then order status queries are timed uniformly at random within the 13 days after the order is placed.

The script that we used to generate our dataset is publicly available, along with Stasis' and Rose's source code (Section 7).

This dataset is not easily compressible using the algorithms provided by Rose. Many columns fit in a single byte, rendering Rose's version of PFOR useless. These fields change frequently enough to limit the effectiveness of run length encoding. Both of these issues would be addressed by bit packing. Also, occasionally re-evaluating and modifying compression strategies is known to improve compression of TPC-H data. TPC-H dates are clustered during weekdays, from 1995-2005, and around Mother's Day and the last few weeks of each year.

Order status queries have excellent temporal locality and generally succeed after accessing $C0$. These queries simply increase the amount of CPU time between tuple insertions and have minimal impact on replication throughput. Rose overlaps their processing with the asynchronous I/O performed by merges.

We force Rose to become seek bound by running a second set of experiments with a different version of the order status query. In one set of experiments, which we call "Lookup C0," the order status query only examines $C0$. In the other, which we call "Lookup all components," we force each order

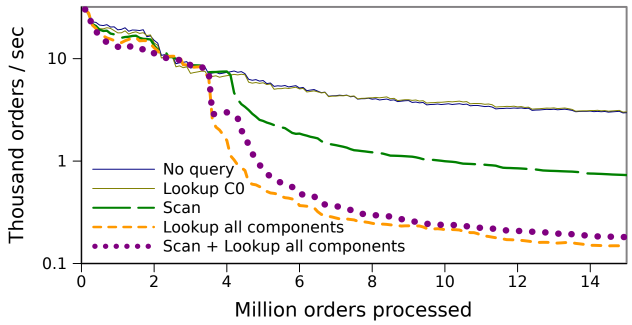

Figure 6: Rose TPC-C/H order throughput

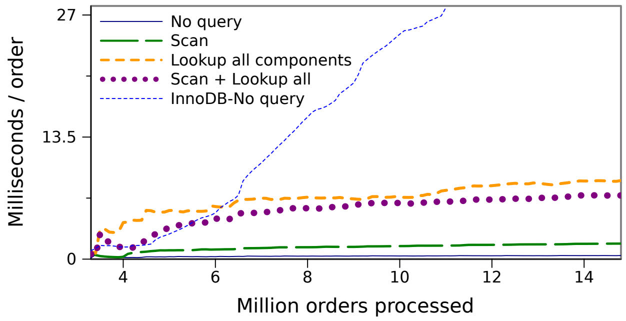

Figure 7: Rose and InnoDB TPC-C/H order times, averaged starting after 3.3 million orders. Drive access time is ∼13.5ms. Reading data during updates dominates the cost of queries.

status query to examine every tree component. This keeps Rose from exploiting the fact that most order status queries can be serviced from $C0$. Finally, Rose provides versioning for this test; though its garbage collection code is executed, it never collects overwritten or deleted tuples.

Figure 6 plots the number of orders processed by Rose per second against the total number of orders stored in the Rose replica. For this experiment, we configure Rose to reserve 1GB for the page cache and 2GB for $C0$. We `mlock()` 4.5GB of RAM, leaving 500MB for the kernel, system services, and Linux's page cache.

The cost of searching $C0$ is negligible, while randomly accessing the larger components is quite expensive. The overhead of index scans increases as the table increases in size, leading to a continuous downward slope throughout runs that perform scans.

Surprisingly, periodic table scans improve lookup performance for $C1$ and $C2$. The effect is most pronounced after 3 million orders are processed. That is approximately when Stasis' page file exceeds the size of the buffer pool, which is managed using LRU. After each merge, half the pages it read become obsolete. Index scans rapidly replace these pages with live data using sequential I/O. This increases the likelihood that index probes will be serviced from memory. A more sophisticated page replacement policy would further improve performance by evicting obsolete pages before accessible pages.

We use InnoDB to measure the cost of performing B-tree style updates. In this experiment, we limit InnoDB's page cache to $1GB$, and `mlock()` $6GB$ of RAM. InnoDB does not perform any order status queries or scans (Figure 7). Although Rose has a total of $3GB$ for this experiment, it only uses $1GB$ for queries, so this setup causes both systems to begin performing random I/O at the same time.

InnoDB's performance degrades rapidly once it begins using random I/O to update its B-tree. This is because the sort order of its B-tree does not match the ordering of the updates it processes. Ignoring repeated accesses to the same tuple within the same order, InnoDB performs two B-tree operations per line item (a read and a write), leading to many page accesses per order.

In contrast, Rose performs an index probe 50% of the time. Rose's index probes read $C1$ and $C2$, so it accesses one page per order on average. However, by the time the experiment concludes, pages in $C1$ are accessed R times more often (~ 6.6) than those in $C2$, and the page file is 3.9GB. This allows Rose to keep $C1$ cached in memory, so each order uses approximately half a disk seek. At larger scale factors, Rose's access time should double, but remain well below the time a B-tree would spend applying updates.

After terminating the InnoDB run, we allowed MySQL to quiesce, then performed an index scan over the table. At this point, it had processed 11.8 million orders, and index scans took 19 minutes and 20 seconds. Had we performed index scans during the InnoDB run this would translate to an 11.5 ms cost per order. The overhead of Rose scans at that point in the run was approximately 2.2 ms per order.

5. RELATED WORK

5.1 LSM-trees

The original LSM-tree work [11] provides a more detailed analytical model than the one presented above. It focuses on update intensive OLTP (TPC-A) workloads and hardware provisioning for steady state workloads.

LHAM is an adaptation of LSM-trees for hierarchical storage systems [9]. It stores higher numbered components on archival media such as CD-R or tape drives, and scales LSM-trees beyond the capacity of high-performance storage media. It also supports efficient time travel queries over archived, historical data [9].

Partitioned exponential files are similar to LSM-trees, except that they range partition data into smaller indices [8]. This solves a number of issues that are left unaddressed by Rose, most notably skewed update patterns and merge storage overhead.

Rose is optimized for uniform random insertion patterns and does not attempt to take advantage of skew in the replication workload. If most updates are to a particular partition then partitioned exponential files merge that partition frequently, skipping merges of unmodified partitions. Also, smaller merges duplicate smaller components, wasting less space. Multiple merges can run in parallel, improving concurrency. Partitioning a Rose replica into multiple LSM-trees would enable similar optimizations.

Partitioned exponential files avoid a few other pitfalls of LSM-tree implementations. Rose addresses these problems in different ways. One issue is page file fragmentation. Partitioned exponential files make use of maintenance tasks to move partitions, ensuring that there is enough contiguous space to hold a new partition. Rose avoids this problem by allowing tree components to become fragmented. It does this by using Stasis to allocate contiguous regions of pages that are long enough to guarantee good sequential scan performance. Rose always allocates regions of the same length, guaranteeing that Stasis can reuse all freed regions before extending the page file. This can waste nearly an entire region per component, which does not matter in Rose, but could be significant to systems with many small partitions.

Some LSM-tree implementations do not support concurrent insertions, merges and queries. This causes such implementations to block during merges that take $O(n)$ time in the tree size. Rose overlaps insertions and queries with tree merges. Therefore, admission control would provide predictable and uniform latencies. Partitioning can be used to limit the number of tree components. We have argued that allocating two unpartitioned on-disk components is adequate for Rose's target applications.

Reusing existing B-tree implementations as the underlying storage mechanism for LSM-trees has been proposed [5]. Many standard B-tree optimizations, such as prefix compression and bulk insertion, would benefit LSM-tree implementations. However, Rose's custom bulk-loaded tree implementation benefits compression. Unlike B-tree compression, Rose's compression does not need to support efficient, in-place updates of tree nodes.

Recent work optimizes B-trees for write intensive workloads by dynamically relocating regions of B-trees during writes [6]. This reduces index fragmentation but still relies upon random I/O in the worst case. In contrast, LSM-trees never use disk seeks to service write requests and produce perfectly laid out B-trees.

Online B-tree merging [15] is closely related to LSM-trees' merge process. B-tree merging addresses situations where the contents of a single table index have been split across two physical B-trees that now need to be reconciled. This situation arises, for example, during rebalancing of partitions within a cluster of database machines.

One B-tree merging approach lazily piggybacks merge operations on top of tree access requests. To service an index probe or range scan, the system must read leaf nodes from both B-trees. Rather than simply evicting the pages from cache, this approach merges the portion of the tree that has already been brought into memory.

LSM-trees can service delayed LSM-tree index scans without performing additional I/O. Queries that request table scans wait for the merge processes to make a pass over the index. By combining this idea with lazy merging an LSM-tree implementation could service range scans immediately without significantly increasing the amount of I/O performed by the system.

5.2 Row-based database compression

Row-oriented database compression techniques compress each tuple individually and sometimes ignore similarities between adjacent tuples. One such approach compresses low cardinality data by building a table-wide mapping between short identifier codes and longer string values. The mapping table is stored in memory for convenient compression and decompression. Other approaches include NULL suppression, which stores runs of NULL values as a single count and leading zero suppression which stores integers in a variable length format that does not store zeros before the first non-zero digit of each number. Row oriented compression schemes typically provide efficient random access to tuples, often by explicitly storing tuple offsets at the head of each page.

Another approach is to compress page data using a generic compression algorithm, such as gzip. The primary drawback of this approach is that the size of the compressed page is not known until after compression. Also, general purpose compression techniques typically do not provide random access within pages and are often more processor intensive than specialized database compression techniques [16].

5.3 Column-oriented database compression

Some column compression techniques are based on the observation that sorted columns of data are often easier to compress than sorted tuples. Each column contains a single data type, and sorting decreases the cardinality and range of data stored on each page. This increases the effectiveness of simple, special purpose, compression schemes.

PFOR was introduced as an extension to MonetDB [17], a column-oriented database, along with two other formats. PFOR-DELTA is similar to PFOR, but stores differences between values as deltas. PDICT encodes columns as keys and a dictionary that maps to the original values. We plan to add both these formats to Rose in the future. We chose to implement RLE and PFOR because they provide high compression and decompression bandwidth. Like MonetDB, each Rose table is supported by custom-generated code.

C-Store, another column oriented database, has relational operators that have been optimized to work directly on compressed data [1]. For example, when joining two run length encoded columns, it is unnecessary to explicitly represent each row during the join. This optimization would be useful in Rose, as its merge processes perform repeated joins over compressed data.

Search engines make use of similar techniques and apply column compression to conserve disk and bus bandwidth. Updates are performed by storing the index in partitions and replacing entire partitions at a time. Partitions are rebuilt offline [4].

A recent paper [7] provides a survey of database compression techniques and characterizes the interaction between compression algorithms, processing power and memory bus bandwidth. The formats within their classification scheme either split tuples across pages or group information from the same tuple in the same portion of the page.

Rose, which does not split tuples across pages, takes a different approach and stores each column separately within a page. Our column-oriented page layouts incur different types of per-page overhead and have fundamentally different processor cache behaviors and instruction-level parallelism properties than the schemes they consider.

In addition to supporting compression, column databases typically optimize for queries that project away columns during processing. They do this by precomputing the projection and potentially resorting and recompressing the data. This reduces the size of the uncompressed data and can improve compression ratios, reducing the amount of I/O performed by the query. Rose can support these optimizations by computing the projection of the data during replication. This provides Rose replicas optimized for a set of queries.

Unlike column-oriented databases, Rose provides for high throughput, low-latency, in-place updates and tuple lookups comparable to row-oriented storage.

5.4 Snapshot consistency

Rose relies upon the correctness of the master database's concurrency control algorithms to provide snapshot consistency to queries. Rose is compatible with most popular approaches to concurrency control in OLTP environments, including two-phase locking, optimistic concurrency control and multiversion concurrency control.

Rose only provides read-only queries. Therefore, its concurrency control algorithms need only address read-write conflicts. Well-understood techniques protect against such conflicts without causing requests to block, deadlock or livelock [3].

6. CONCLUSION

Compressed LSM-trees are practical on modern hardware. Hardware trends such as increased memory size and sequential disk bandwidth will further improve Rose's performance. In addition to developing new compression formats optimized for LSM-trees, we presented a new approach to database replication that leverages the strengths of LSM-trees by avoiding index probing during updates. We also introduced the idea of using snapshot consistency to provide concurrency control for LSM-trees.

Our implementation is a first cut at a working version of Rose. Our prototype's random read performance compares favorably to that of a production-quality B-tree, and we have bounded the increases in Rose's replication throughput that could be obtained without improving compression ratios. By avoiding disk seeks for all operations except random index probes, uncompressed LSM-trees can provide orders of magnitude more write throughput than B-trees. With real-world database compression ratios ranging from 5-20x, Rose can outperform B-tree based database replicas by an additional factor of ten.

7. AVAILABILITY

Rose's source code and the tools used to generate our TPC-C/H workload are available at:
`http://www.cs.berkeley.edu/∼sears/stasis`

8. ACKNOWLEDGEMENTS

We would like to thank Petros Maniatis, Tyson Condie, Jens Dittrich and the anonymous reviewers for their feedback. Portions of this work were performed at Intel Research, Berkeley.

9. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[4] E. A. Brewer. Combining systems and databases: A search engine retrospective. In J. M. Hellerstein and M. Stonebraker, editors, *Readings in Database Systems.* 4th edition, 2005.

[5] G. Graefe. Sorting and indexing with partitioned B-Trees. In *CIDR*, 2003.

[6] G. Graefe. B-Tree indexes for high update rates. *SIGMOD Rec.*, 35(1):39–44, 2006.

[7] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: Compression and bandwidth trade offs for database scans. In *SIGMOD*, 2007.

[8] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, 2007.

[9] P. Muth, P. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In *VLDB*, 1998.

[10] National Severe Storms Laboratory Historical Weather Data Archives, Norman, Oklahoma, from their Web site at http://data.nssl.noaa.gov.

[11] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[12] V. Raman and G. Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *VLDB*, 2006.

[13] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[14] StorageReview.com. Seagate barracuda 750es. http://www.storagereview.com/ST3750640NS.sr, 12 2006.

[15] X. Sun, R. Wang, B. Salzberg, and C. Zou. Online B-tree merging. In *SIGMOD*, 2005.

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

[17] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.