Rows: Compressed, log-structured replication

ABSTRACT

This paper describes Rows[1], a database storage engine designed for high-throughput replication. It targets applications with write-intensive (seek limited) transaction processing workloads and near-realtime decision support and analytical processing queries. Rows uses *log structured merge* (LSM) trees to create full database replicas using purely sequential I/O. It provides access to inconsistent data in realtime and consistent data with a few seconds delay. Rows was written to support micropayment transactions.

A Rows replica serves two purposes. First, by avoiding seeks, Rows reduces the load on the replicas' disks. This leaves surplus I/O capacity for read-only queries and allows inexpensive hardware to replicate workloads produced by expensive machines that are equipped with many disks. Affordable, read-only replication allows decision support and OLAP queries to scale linearly with the number of machines, regardless of lock contention and other bottlenecks associated with distributed transactional updates. Second, a group of Rows replicas provides a highly available copy of the database. In many Internet-scale environments, decision support queries are more important than update availability.

Rows' throughput is limited by sequential I/O bandwidth. We use column compression to reduce this bottleneck. Rather than reassemble rows from a column-oriented disk layout, we adapt existing column compression algorithms to a novel row-oriented data layout. This approach to database compression introduces negligible space overhead and can be applied to most single-pass, randomly accessible compression formats. Our prototype uses lightweight (superscalar) column compression algorithms.

Existing analytical models and our experiments reveal that, for disk and CPU-bound workloads, Rows provides orders of magnitude greater throughput than conventional replication techniques.

1. INTRODUCTION

[1][Clever acronym here]

ACM SIGMOD '08 Vancouver, BC, Canada

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

When faced with random access patterns, traditional database scalability is limited by the size of memory. If the system's working set does not fit in RAM, any attempt to update data in place is limited by the latency of hard disk seeks. This bottleneck can be alleviated by adding more drives, which increases cost and decreases reliability. Alternatively, the database can run on a cluster of machines, increasing the amount of available memory, CPUs and disk heads, but introducing the overhead and complexity of distributed transactions and partial failure.

These problems lead to large-scale database installations that partition their workloads across multiple servers, allowing linear scalability, but sacrificing consistency between data stored in different partitions. Fortunately, updates often deal with well-defined subsets of the data; with an appropriate partitioning scheme, one can achieve linear scalability for localized updates.

The cost of partitioning is that no globally coherent version of the data exists. In the best case, queries that rely on a global view of the data run against each master database instance, then attempt to reconcile inconsistencies in their results. If the queries are too expensive to run on master database instances they are delegated to data warehousing systems and produce stale results.

In order to address the needs of such queries, Rows gives up the ability to directly process SQL updates. In exchange, it is able to replicate conventional database instances at a small fraction of the cost of a general-purpose database server.

Like a data warehousing solution, this decreases the cost of large, read-only analytical processing and decision support queries, and scales to extremely large database instances with high-throughput updates. Unlike data warehousing solutions, Rows does this without introducing significant replication latency.

Conventional database replicas provide low-latency replication at a cost comparable to that of the master database. The expense associated with such systems prevents conventional database replicas from scaling. The additional

read throughput they provide is nearly as expensive as read throughput on the master. Because their performance is comparable to that of the master database, they are unable to consolidate multiple database instances for centralized processing.

Unlike existing systems, Rows provides inexpensive, low-latency, and scalable replication of write-intensive relational databases, regardless of workload contention, database size, or update patterns.

1.1 Fictional Rows deployment

Imagine a classic, disk-bound TPC-C installation. On modern hardware, such a system would have tens of disks, and would be seek limited. Consider the problem of producing a read-only, low-latency replica of the system for analytical processing, decision support, or some other expensive read-only workload. If the replica uses the same storage engine as the master, its hardware resources would be comparable to (certainly within an order of magnitude) those of the master database instances. Worse, a significant fraction of these resources would be devoted to replaying updates from the master. As we show below, the I/O cost of maintaining a Rows replica can be less than 1% of the cost of maintaining the master database.

Therefore, unless the replica's read-only query workload is seek limited, a Rows replica requires many fewer disks than the master database instance. If the replica must service seek-limited queries, it will likely need to run on a machine similar to the master database, but will use almost none of its (expensive) I/O capacity for replication, increasing the resources available to queries. Furthermore, Rows' indices are allocated sequentially, reducing the cost of index scans, and Rows' buffer pool stores compressed pages, increasing the effective size of system memory.

The primary drawback of this approach is that it roughly doubles the cost of each random index lookup. Therefore, the attractiveness of Rows hinges on two factors: the fraction of the workload devoted to random tuple lookups, and the premium one would have paid for a piece of specialized storage hardware that Rows replaces.

1.2 Paper structure

We begin by providing an overview of Rows' system design and then present a simplified analytical model of LSM-Tree I/O behavior. We apply this model to our test hardware, and predict that Rows will greatly outperform database replicas that store data in B-Trees. We proceed to present a row-oriented page layout that adapts most column-oriented compression schemes for use in Rows. Next, we evaluate Rows' replication performance on a real-world dataset, and demonstrate orders of magnitude improvement over a MySQL InnoDB B-Tree index. Our performance evaluations conclude with an analysis of our prototype's performance and shortcomings. We defer related work to the end of the paper, as recent research suggests a number of ways in which Rows could be improved.

2. SYSTEM OVERVIEW

A Rows replica takes a replication log as input, and stores the changes it contains in a *log structured merge* (LSM) tree[7].

An LSM-Tree is an index method that consists of multiple sub-trees (components). The smallest component, $C0$ is a memory resident binary search tree. The next smallest component, $C1$ is a bulk loaded B-Tree. Updates are applied directly to $C0$. As $C0$ grows, it is merged with $C1$. 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 components (two on-disk tress) are generally adequate. The memory-resident component, $C0$, is updated in place. All other components are produced by repeated merges with the next smaller component. Therefore, LSM-Trees are updated without resorting to random disk I/O.

Unlike the original LSM work, Rows compresses the data using techniques from column-oriented databases, and is designed exclusively for database replication. Merge throughput is bounded by sequential I/O bandwidth, and lookup performance is limited by the amount of available memory. Rows uses compression to trade surplus computational power for scarce storage resources.

The replication log should record each transaction `begin`, `commit`, and `abort` performed by the master database, along with the pre- and post-images associated with each tuple update. The ordering of these entries must match the order in which they are applied at the database master.

Upon receiving a log entry, Rows applies it to an in-memory tree, and the update is immediately available to queries that do not require a consistent view of the data. Rows provides snapshot consistency to readers that require transactional isolation. It does so in a lock-free manner; transactions' reads and writes are not tracked, and no Rows transaction can ever force another to block or abort. When given appropriate workloads, Rows provides extremely low-latency replication. Transactionally consistent data becomes available after a delay on the order of the duration of a few update transactions. The details of Rows' concurrency control mechanisms are provided in Section 2.5.

In order to look up a tuple stored in Rows, a query must examine all three tree components, typically starting with the in-memory (fastest, and most up-to-date) component, and then moving on to progressively larger and out-of-date trees. In order to perform a range scan, the query can iterate over the trees manually. Alternatively, it can wait until the next round of merging occurs, and apply the scan to tuples as the mergers examine them. By waiting until the tuples are due to be merged, the range-scan can occur with zero I/O cost, at the expense of significant delay.

Rows merges LSM-Tree components in background threads. This allows it to continuously process updates and service index lookup requests. In order to minimize the overhead of thread synchronization, index lookups lock entire tree components at a time. Because on-disk tree components are read-only, these latches only block tree deletion, allowing merges and lookups to occur concurrently. $C0$ is updated in place, preventing inserts from occurring concurrently with merges and lookups. However, operations on $C0$ are comparatively fast, reducing contention for $C0$'s latch.

Recovery, space management and atomic updates to Rows' metadata are handled by an existing transactional storage system. Rows is implemented as an extension to the transaction system and stores its data in a conventional database page file. Rows does not use the underlying transaction system to log changes to its tree components. Instead, it force writes tree components to disk after each merge completes, ensuring durability without significant logging overhead.

As far as we know, Rows is the first LSM-Tree implementation. This section provides an overview of LSM-Trees, and explains how we quantify the cost of tuple insertions. It then steps through a rough analysis of LSM-Tree performance on current hardware (we refer the reader to the original LSM work for a thorough analytical discussion of LSM performance). Finally, we explain how our implementation provides transactional isolation, exploits hardware parallelism, and supports crash recovery. These implementation specific details are an important contribution of this work; they explain how to adapt LSM-Trees to provide high performance database replication. We defer discussion of Rows' compression techniques to the next section.

2.1 Tree merging

For simplicity, this paper considers three component LSM-Trees. Component zero ($C0$) is an in-memory binary search tree. Components one and two ($C1$, $C2$) are read-only, bulk-loaded B-Trees. Each update is handled in three stages. In the first stage, the update is applied to the in-memory tree. Next, once enough updates have been applied, a tree merge is initiated, and the tuple is eventually merged with existing tuples in $C1$. The merge process performs a sequential scan over the in-memory tree and $C1$, producing a new version of $C1$.

When the merge is complete, $C1$ is atomically replaced with the new tree, and $C0$ is atomically replaced with an empty tree. The process is then eventually repeated when $C1$ and $C2$ are merged.

Although our prototype replaces entire trees at once, this approach introduces a number of performance problems. The original LSM work proposes a more sophisticated scheme that addresses some of these issues. Instead of replacing entire trees at once, it replaces one subtree at a time. This reduces peak storage and memory requirements.

Truly atomic replacement of portions of an LSM-Tree would cause ongoing merges to block insertions, and force the mergers to run in lock step. (This problem is mentioned in the LSM paper.) We address this issue by allowing data to be inserted into the new version of the smaller component before the merge completes. This forces Rows to check both versions of components $C0$ and $C1$ in order to look up each tuple, but it handles concurrency between merge steps without resorting to fine-grained latches. Applying this approach to subtrees would reduce the impact of these extra lookups, which could be filtered out with a range comparison in the common case.

2.2 Amortized insertion cost

In order to compute the amortized cost of insertion into an LSM-Tree, we need only consider the cost of comparing the inserted tuple with older tuples (otherwise, we would count the cost of each comparison twice). Therefore, we say that each tuple insertion ultimately causes two rounds of I/O operations; one for the merge into $C1$, and another to merge into $C2$. Once a tuple reaches $C2$ it does not contribute to the initiation of more I/O (For simplicity, we assume the LSM-Tree has reached a steady state).

In a populated LSM-Tree $C2$ is the largest component, and $C0$ is the smallest component. The original LSM-Tree work proves that throughput is maximized when the ratio of the sizes of $C1$ to $C0$ is equal to the ratio between $C2$ and $C1$. They call this ratio R. Note that (on average in a steady state) for every $C0$ tuple consumed by a merge, R tuples from $C1$ must be examined. Similarly, each time a tuple in $C1$ is consumed, R tuples from $C2$ are examined. Therefore, in a steady state, insertion rate times the sum of $R * cost_{read\ and\ write\ C2}$ and $R * cost_{read\ and\ write\ C1}$ cannot exceed the drive's sequential I/O bandwidth. Note that the total size of the tree is approximately $R^2 * |C0|$ (neglecting the data stored in $C0$ and $C1$)[2].

2.3 Replication Throughput

LSM-Trees have different asymptotic performance characteristics than conventional index structures. In particular, the amortized cost of insertion is $O(\sqrt{n})$ in the size of the data. This cost is $O(log\ n)$ for a B-Tree. The relative costs of sequential and random I/O determine whether or not Rows is able to outperform B-Trees in practice. This section describes the impact of compression on B-Tree and LSM-Tree performance using (intentionally simplistic) models of their performance characteristics.

Starting with the (more familiar) B-Tree case, in the steady state we can expect each index update to perform two random disk accesses (one evicts a page, the other reads a page). Tuple compression does not reduce the number of disk seeks:

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

(We assume that the upper levels of the B-Tree are memory resident.) If we assume uniform access patterns, 4 KB pages and 100 byte tuples, this means that an uncompressed B-Tree would keep $\sim 2.5\%$ of the tuples in memory. Prefix compression and a skewed update distribution would improve the situation significantly, but are not considered here. Without a skewed update distribution, batching I/O into sequential writes only helps if a significant fraction of the tree's data fits in RAM.

In Rows, we have:

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

where R is the ratio of adjacent tree component sizes ($R^2 = \frac{|tree|}{|mem|}$). We multiply by $2R$ because each new tuple is eventually merged into both of the larger components, and each merge involves R comparisons with existing tuples on average.

An update of a tuple is handled as a deletion of the old tuple (an insertion of a tombstone), and an insertion of the new tuple, leading to a second factor of two. The third reflects the fact that the merger must read existing tuples into memory before writing them back to disk.

The *compression ratio* is $\frac{uncompressed\ size}{compressed\ size}$, so improved compression leads to less expensive LSM-Tree updates. For simplicity, we assume that the compression ratio is the same throughout each component of the LSM-Tree; Rows addresses this at run-time 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[8] report random access times of 12.3/13.5 $msec$ and 44.3−78.5 MB/s sustained throughput. Timing `dd if=/dev/zero of=file;`

[2]The proof that keeping R constant across our three tree components follows from the fact that the mergers compete for I/O bandwidth and $x(1-x)$ is maximized when $x = 0.5$. The LSM-Tree paper proves the general case.

`sync` on an empty ext3 file system suggests our test hardware provides $57.5MB/s$ of storage bandwidth.

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

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

Pessimistically setting

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

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 (lower than we expect to see in practice, but numerically convenient), the LSM-Tree outperforms the B-Tree when:

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

$$R < 10,000$$

on a machine that can store 1 GB in an in-memory tree, this yields a maximum "interesting" tree size of $R^2 * 1GB = 100$ petabytes, well above the actual drive capacity of 750 GB. A 750 GB tree would have a $C2$ component 750 times larger than the 1GB $C0$ component. Therefore, it would have an R of $\sqrt{750} \approx 27$; we would expect such a tree to have a sustained insertion throughput of approximately 8000 tuples / second, or 800 kbyte/sec[3] given our 100 byte tuples.

Our hard drive's average access time tells us that we can expect the drive to deliver 83 I/O operations per second. Therefore, we can expect an insertion throughput of 41.5 tuples / sec from a B-Tree with a 18.5 GB buffer pool. With just $1GB$ of RAM, Rows should outperform the B-Tree by more than two orders of magnitude. Increasing Rows' system memory to cache $10GB$ of tuples would increase write performance by a factor of $\sqrt{10}$.

Increasing memory another ten fold to 100GB 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 could cache roughly 80GB of leaf pages in memory, and write approximately $\frac{41.5}{(1-(80/750)} = 46.5$ tuples/sec. Increasing memory further yields a system that is no longer disk bound.

Assuming that the CPUs are fast enough to allow Rows compression and merge routines to keep up with the bandwidth supplied by the disks, we conclude that Rows will provide significantly higher replication throughput for disk bound applications.

[3]It would take 11 days to overwrite every tuple on the drive in random order.

Table 1: Tree creation overhead - five column (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: Tree creation 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

2.4 Indexing

Our analysis ignores the cost of allocating and initializing our LSM-Trees' internal nodes. The compressed data constitutes the leaf pages of the tree. Each time the compression process fills a page, it inserts an entry into the leftmost entry in the tree, allocating additional internal nodes if necessary. Our prototype does not compress internal tree nodes[4], so it writes one tuple into the tree's internal nodes per compressed page. Rows inherits a default page size of 4KB from the transaction system we based it upon. Although 4KB is fairly small by modern standards, Rows is not particularly sensitive to page size; even with 4KB pages, Rows' per-page overheads are acceptable. Assuming tuples are 400 bytes, $\sim \frac{1}{10}$th of our pages are dedicated to the lowest level of tree nodes, with $\frac{1}{10}$th that number devoted to the next highest level, and so on. See Table 2 for a comparison of compression performance with and without tree creation enabled[5]. The data was generated by applying Rows' compressors to randomly generated five column, 1,000,000 row tables. Across five runs, in Table 1 RLE's page count had a standard deviation of $\sigma = 2.35$; the other values had $\sigma = 0$. In Table 2, $\sigma < 7.26$ pages.

As the size of the tuples increases, the number of compressed pages that each internal tree node points to decreases, increasing the overhead of tree creation. In such circumstances, internal tree node compression and larger pages should improve the situation.

2.5 Isolation

Rows combines replicated transactions into snapshots. Each transaction is assigned to a snapshot according to a timestamp; two snapshots are active at any given time. Rows assigns incoming transactions to the newer of the two active snapshots. Once all transactions in the older snapshot have completed, that snapshot is marked inactive, exposing its contents to new queries that request a consistent view of the data. At this point a new active snapshot is created,

[4]This is a limitation of our prototype; not our approach. Internal tree nodes are append-only and, at the very least, the page ID data is amenable to compression. Like B-Tree compression, this would decrease the memory used by lookups.

[5]Our analysis ignores page headers, per-column, and per-tuple overheads; these factors account for the additional indexing overhead.

and the process continues.

The timestamp is simply the snapshot number. In the case of a tie during merging (such as two tuples with the same primary key and timestamp), the version from the newer (lower numbered) component is taken.

This ensures that, within each snapshot, Rows applies all updates in the same order as the primary database. Across snapshots, concurrent transactions (which can write non-conflicting tuples in arbitrary orders) lead to reordering of updates. However, these updates are guaranteed to be applied in transaction order. The correctness of this scheme hinges on the correctness of the timestamps applied to each transaction.

If the master database provides snapshot isolation using multiversion concurrency control (as is becoming increasingly popular), we can simply reuse the timestamp it applies to each transaction. If the master uses two phase locking, the situation becomes more complex, as we have to use the commit time of each transaction[6]. Until the commit time is known, Rows stores the transaction id in the LSM-Tree. As transactions are committed, it records the mapping from transaction id to snapshot. Eventually, the merger translates transaction id's to snapshots, preventing the mapping from growing without bound.

New snapshots are created in two steps. First, all transactions in epoch $t-1$ must complete (commit or abort) so that they are guaranteed to never apply updates to the database again. In the second step, Rows' current snapshot number is incremented, new read-only transactions are assigned to snapshot $t-1$, and new updates are assigned to snapshot $t+1$. Each such transaction is granted a shared lock on the existence of the snapshot, protecting that version of the database from garbage collection. In order to ensure that new snapshots are created in a timely and predictable fashion, the time between them should be acceptably short, but still slightly longer than the longest running transaction.

2.5.1 Isolation performance impact

Although Rows' isolation mechanisms never block the execution of index operations, their performance degrades in the presence of long running transactions. Long running updates block the creation of new snapshots. Ideally, upon encountering such a transaction, Rows simply asks the master database to abort the offending update. It then waits until appropriate rollback (or perhaps commit) entries appear in the replication log, and creates the new snapshot. While waiting for the transactions to complete, Rows continues to process replication requests by extending snapshot t.

Of course, proactively aborting long running updates is simply an optimization. Without a surly database administrator to defend it against application developers, Rows does not send abort requests, but otherwise behaves identically. Read-only queries that are interested in transactional consistency continue to read from (the increasingly stale) snapshot $t-2$ until $t-1$'s long running updates commit.

Long running queries present a different set of challenges to Rows. Although Rows provides fairly efficient time-travel support, versioning databases are not our target application. Rows provides each new read-only query with guaranteed access to a consistent version of the database. Therefore, long-running queries force Rows to keep old versions of overwritten tuples around until the query completes. These tuples increase the size of Rows' LSM-Trees, increasing merge overhead. If the space consumed by old versions of the data is a serious issue, long running queries should be disallowed. Alternatively, historical, or long-running queries could be run against certain snapshots (every 1000th, or the first one of the day, for example), reducing the overhead of preserving old versions of frequently updated data.

[6]This assumes all transactions use transaction-duration write locks, and lock release and commit occur atomically. Transactions that obtain short write locks can be treated as a set of single action transactions.

2.5.2 Merging and Garbage collection

Rows merges components by iterating over them in order, garbage collecting obsolete and duplicate tuples and writing the rest into a new version of the largest component. Because Rows provides snapshot consistency to queries, it must be careful not to collect a version of a tuple that is visible to any outstanding (or future) queries. Because Rows never performs disk seeks to service writes, it handles deletions by inserting special tombstone tuples into the tree. The tombstone's purpose is to record the deletion event until all older versions of the tuple have been garbage collected. Sometime after that point, the tombstone is collected as well.

In order to determine whether or not a tuple can be collected, Rows compares the tuple's timestamp with any matching tombstones (or record creations, if the tuple is a tombstone), and with any tuples that match on primary key. Upon encountering such candidates for garbage collection, Rows compares their timestamps with the set of locked snapshots. If there are no snapshots between the tuple being examined and the updated version, then the tuple can be collected. Tombstone tuples can also be collected once they reach $C2$ and any older matching tuples have been removed.

Actual reclamation of pages is handled by the underlying transaction system; once Rows completes its scan over existing components (and registers new ones in their places), it tells the transaction system to reclaim the regions of the page file that stored the old components.

2.6 Parallelism

Rows provides ample opportunities for parallelism. All of its operations are lock-free; concurrent readers and writers work independently, avoiding blocking, deadlock and livelock. Index probes must latch $C0$ in order 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.

Our prototype exploits replication's pipelined parallelism by running each component's merge process in a separate thread. In practice, this allows our prototype to exploit two to three processor cores during replication. Remaining cores could be used by queries, or (as hardware designers increase the number of processor cores per package) by using data parallelism to split each merge across multiple threads.

Finally, Rows is capable of using standard database implementation techniques to overlap I/O requests with computation. Therefore, the I/O wait time of CPU bound workloads should be negligible, and I/O bound workloads should be able to take complete advantage of the disk's sequential I/O bandwidth. Therefore, given ample storage bandwidth, we expect the throughput of Rows replication to increase

with Moore's law for the foreseeable future.

2.7 Recovery

Like other log structured storage systems, Rows' recovery process is inexpensive and straightforward. However, Rows does not attempt to ensure that transactions are atomically committed to disk, and is not meant to replace the master database's recovery log.

Instead, recovery occurs in two steps. Whenever Rows writes a tree component to disk, it does so by beginning a new transaction in the underlying transaction system. Next, it allocates contiguous regions of disk pages (generating one log entry per region), and performs a B-Tree style bulk load of the new tree into these regions (this bulk load does not produce any log entries). Then, Rows forces the tree's regions to disk, and writes the list of regions used by the tree and the location of the tree's root to normal (write ahead logged) records. Finally, it commits the underlying transaction.

After the underlying transaction system completes recovery, Rows will have a set of intact and complete tree components. Space taken up by partially written trees was allocated by an aborted transaction, and has been reclaimed by the transaction system's recovery mechanism. After the underlying recovery mechanisms complete, Rows reads the last committed timestamp from the LSM-Tree header, and begins playback of the replication log at the appropriate position. Upon committing new components to disk, Rows allows the appropriate portion of the replication log to be truncated.

3. ROW COMPRESSION

Disk heads are the primary storage bottleneck for most OLTP environments, and disk capacity is of secondary concern. Therefore, database compression is generally performed to improve system performance, not capacity. In Rows, sequential I/O throughput is the primary replication bottleneck; and is proportional to the compression ratio. Furthermore, compression increases the effective size of the buffer pool, which is the primary bottleneck for Rows' random index lookups.

Although Rows targets row-oriented workloads, its compression routines are based upon column-oriented techniques and rely on the assumption that pages are indexed in an order that yields easily compressible columns. Rows' compression formats are based on our *multicolumn* compression format. In order to store data from an N column table, we divide the page into $N+1$ variable length regions. N of these regions each contain a compressed column. The remaining region contains "exceptional" column data (potentially from more than one column).

For example, a column might be encoded using the *frame of reference* (FOR) algorithm, which stores a column of integers as a single offset value and a list of deltas. When a value too different from the offset to be encoded as a delta is encountered, an offset into the exceptions region is stored. When applied to a page that stores data from a single column, the resulting algorithm is MonetDB's *patched frame of reference* (PFOR) [11].

Rows' multicolumn pages extend this idea by allowing multiple columns (each with its own compression algorithm) to coexist on each page. This reduces the cost of reconstructing tuples during index lookups, and yields a new approach to superscalar compression with a number of new, and potentially interesting properties.

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

Format (#col)	Ratio	Comp. mb/s	Decomp. mb/s
PFOR (1)	3.96x	547	2959
PFOR (10)	3.86x	256	719
RLE (1)	48.83x	960	1493 (12%)
RLE (10)	47.60x	358 (9%)	659 (7%)

We implemented two compression formats for Rows' multicolumn pages. The first is PFOR, the other is *run length encoding*, which stores values as a list of distinct values and repetition counts. This section discusses the computational and storage overhead of the multicolumn compression approach.

3.1 Multicolumn computational overhead

Rows builds upon compression algorithms that are amenable to superscalar optimization, and can achieve throughputs in excess of 1GB/s on current hardware.

Additional computational overhead is introduced in two areas. First, Rows 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 `memcpy()` occurs once per page allocation.

Second, we need a way to translate requests to write a tuple into calls to appropriate page formats and compression implementations. Unless we hardcode our Rows executable to support a predefined set of page formats (and table schemas), this invokes an extra `for` loop (over the columns) whose body contains a `switch` statement (in order to choose between column compressors) to each tuple compression request.

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 Rows' dynamically dispatched (not custom generated code) multicolumn format.

3.2 The `append()` operation

Rows' 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. Each column's `append()` routine secures storage space for the column value, or returns `false` if no space is available. `append()` has the following signature:

```
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 first free byte in the exceptions region, `exceptions_base` and `column_base` point to (page sized) buffers used to store exceptions and column data as the page is being written to. One copy of these buffers exists for each page that Rows is actively writing to (one per disk-resident LSM-Tree component); they do not significantly increase Rows' memory requirements. Finally,

`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. Initially, our multicolumn module managed these values and the exception space. This led to extra arithmetic operations and conditionals and did not significantly simplify the code. Note that, compared to techniques that store each tuple contiguously on the page, our format avoids encoding the (variable) length of each tuple; instead it encodes the length of each column.

The existing PFOR implementation assumes it has access to a buffer of uncompressed data and that it 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 Rows, as it would increase the complexity of the `append()` interface, and add a buffer to Rows' merge threads.

3.3 Static code generation

After evaluating the performance of a C implementation of Rows' compression routines, we decided to rewrite the compression routines as C++ templates. C++ template instantiation performs compile-time macro substitutions. We declare all functions `inline`, and place them in header files (rather than separate compilation units). This gives g++ the opportunity to perform optimizations such as cross-module constant propagation and branch elimination. It also allows us to write code that deals with integer data types instead of void pointers without duplicating code or breaking encapsulation.

Such optimizations are possible in C, but, because of limitations of the preprocessor, would be difficult to express or require separate code-generation utilities. We found that this set of optimizations improved compression and decompression performance by roughly an order of magnitude. Although compressor throughput varies with data distributions and type, optimizations yield a similar performance improvement across varied datasets and random data distributions.

We performed one additional set of optimizations. Rather than instantiate each compressor template once for each column type at compile time, we instantiate a multicolumn page format template for each page format we wish to support. This removes the `for` loop and `switch` statement that supporting multiple columns per page introduced, but hardcodes page schemas at compile time.

The two approaches could coexist in a single runtime environment, allowing the use of hardcoded implementations for performance critical tables, while falling back on slower, general purpose implementations for previously unseen table layouts.

3.4 Buffer manager interface extensions

Rows uses a preexisting, conventional database buffer manager. Each page contains an LSN (which is largely unused, as we bulk-load Rows' trees) and a page implementation number. This allows it to coexist with conventional write ahead logging mechanisms. As mentioned above, this greatly simplifies crash recovery without introducing significant logging overhead.

Memory resident pages are stored in a hashtable keyed by page number, and replaced using an LRU strategy[7].

In implementing Rows, we made use of a number of generally useful callbacks that are of particular interest to Rows and other database compression schemes. The first, `pageLoaded()` instantiates a new multicolumn page implementation when the page is first read into memory. The second, `pageFlushed()` informs our multicolumn implementation that the page is about to be written to disk, and the third `pageEvicted()` invokes the multicolumn destructor.

We need to register implementations for these functions because the transaction system maintains background threads that control eviction of Rows' pages from memory. Registering these callbacks provides an extra benefit; we parse the page headers, calculate offsets, and choose optimized compression routines when a page is read from disk instead of each time we access it.

As we mentioned above, pages are split into a number of temporary buffers while they are being written, and are then packed into a contiguous buffer before being flushed. Although this operation is expensive, it does present an opportunity for parallelism. Rows provides a per-page operation, `pack()` that performs the translation. We can register `pack()` as a `pageFlushed()` callback or we can explicitly call it during (or shortly after) compression.

`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, this causes multiple Rows threads to block on each `pack()`.

Also, `pack()` reduces Rows' memory utilization by freeing up temporary compression buffers. Delaying its execution for too long might allow this memory to be evicted from processor cache before the `memcpy()` can occur. For these reasons, the merge threads explicitly invoke `pack()` as soon as possible.

3.5 Storage overhead

The multicolumn page format is quite similar to the format of existing column-wise compression formats. The algorithms we implemented have page formats that can be (broadly speaking) 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 exemplar data (as in frame of reference compression). 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. Allocating 16 bits for the page offset and 16 bits for the column type compressor uses 4 bytes per column. Therefore, the additional overhead for an N column page's header is

$$(N-1) * (4 + |average\ compression\ format\ header|)$$

bytes. A frame of reference column header consists of 2 bytes to record the number of encoded rows and a single

[7]LRU is a particularly poor choice, given that Rows' I/O is dominated by large table scans. Eventually, we hope to add support for explicit eviction of pages read by the merge processes.

uncompressed value. Run length encoding headers consist of a 2 byte count of compressed blocks. Therefore, in the worst case (frame of reference encoding 64-bit integers, and Rows' 4KB pages) our prototype's multicolumn format uses $14/4096 \approx 0.35\%$ of the page to store each column header. If the data does not compress well, and tuples are large, additional storage may be wasted because Rows does not split tuples across pages. Tables 1 and 2, which draw column values from independent, identical distributions, show that Rows' compression ratio can be significantly impacted by large tuples.

Breaking pages into smaller compressed blocks changes the compression ratio in another way; the compressibility of the data varies with the size of each compressed block. For example, when frame of reference is applied to sorted data, incoming values eventually drift too far from the page offset, causing them to be stored as exceptional values. Therefore (neglecting header bytes), smaller frame of reference blocks provide higher compression ratios.

Of course, conventional compression algorithms are free to divide their compressed data into blocks to maximize compression ratios. Although Rows' smaller compressed block size benefits some compression implementations (and does not adversely impact either of the algorithms we implemented), it creates an additional constraint, and may interact poorly with some compression algorithms.

3.6 Supporting Random Access

The multicolumn page format is designed to allow efficient row-oriented access to data. The efficiency of random access within a page depends on the format used by individual compressors. Rows 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 (not columns) are stored in the page in sorted order. It 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 values in the range) for 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 in order to narrow the search until the correct tuple(s) are located or the range is empty. Note that partially-matching tuples are only partially examined during the search, and that our binary searches within a column should have better cache locality than searches of row-oriented page layouts.

We have not examined the tradeoffs between different implementations of tuple lookups. Currently, rather than using binary search to find the boundaries of each range, our compressors simply iterate over the compressed representation of the data in order to progressively narrow the range of tuples to be considered. It is possible that (because of expensive branch mispredictions and Rows' small pages) that our linear search implementation will outperform approaches based upon binary search.

4. EVALUATION

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	

4.1 The data set

In order to evaluate Rows' performance, we used it to index weather data. The data we used ranges from May 1, 2007 to Nov 2, 2007, and contains readings from ground stations around the world [6]. 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 122 million tuples.

Duplicating the data should have a limited effect on Rows' compression ratios. Although we index on geographic position, placing all readings from a particular station in a contiguous range, we then index on date. This separates most duplicate versions of the same tuple from each other.

Rows only supports integer data types. We encode the ASCII columns in the data by packing each character into 5 bits (the strings only contain the characters A-Z, "+," "-," and "*"). Floating point columns in the raw 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 column as a 32-bit integer (even when a 16-bit value would do), except current weather conditions, which is packed into a 64-bit integer. Table 4 lists the columns and compression algorithms we assigned to each column. The "Key" column refers to whether or not the field was used as part of a MySQL primary key. InnoDB performance tuning guides suggest limiting the length of the table's primary key. Rows does not support this optimization, so we indexed the Rows table on all columns.

Rows targets seek limited applications; we assign a (single) random order to the tuples, and insert them in this order. We compare Rows' performance with the MySQL InnoDB storage engine's bulk loader[8]. This avoids the overhead of SQL insert statements. To force InnoDB to update its B-Tree index in place, we break the dataset into 100,000 tuple chunks, and bulk load each one in succession.

If we did not do this, MySQL would simply sort the tuples, and then bulk load the index. This behavior is unacceptable in low-latency environments. Breaking the bulk load into multiple chunks forces MySQL to make intermediate results available as the bulk load proceeds[9].

[8]We also evaluated MySQL's MyISAM table format. Predictably, performance degraded quickly as the tree grew; ISAM indices do not support node splits.

[9]MySQL's `concurrent` keyword allows access to *existing*

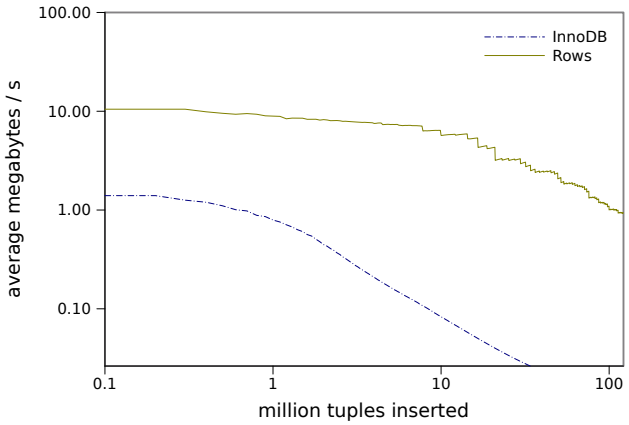

Figure 1: Insertion throughput (log-log, average over entire run).

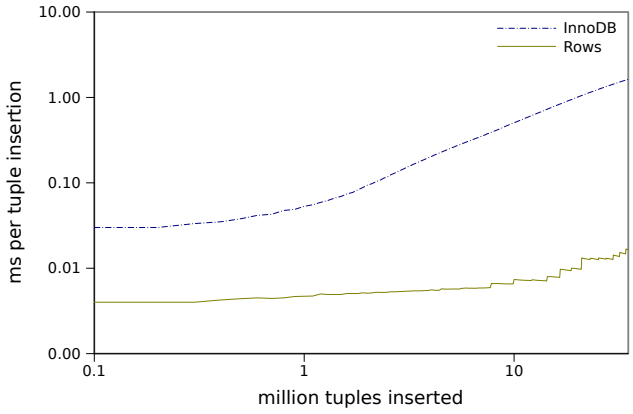

Figure 2: Tuple insertion time (log-log, average over entire run).

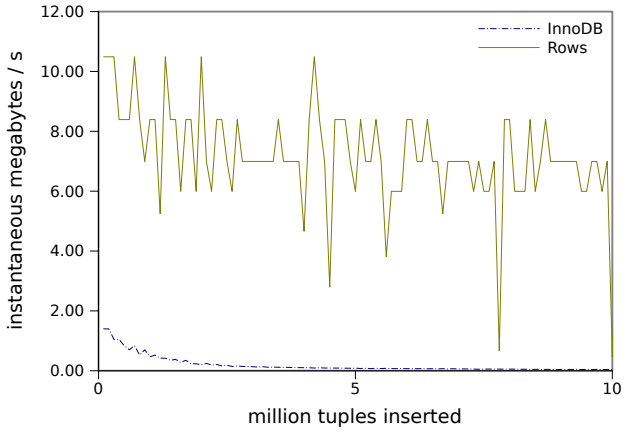

Figure 3: Instantaneous insertion throughput (average over 100,000 tuple windows).

We set InnoDB's buffer pool size to 1GB, MySQL's bulk insert buffer size to 900MB, the log buffer size to 100MB, and disabled InnoDB's double buffer, which writes a copy of each updated page to a sequential log. The double buffer increases the amount of I/O performed by InnoDB, but allows it to decrease the frequency with which it needs to fsync() the buffer pool to disk. Once the system reaches steady state, this would not save InnoDB from performing random I/O, but it would increase I/O overhead.

We compiled Rows' C components with "-O2", and the C++ components with "-O3". The later compiler flag is crucial, as compiler inlining and other optimizations improve Rows' compression throughput significantly. Rows was set to allocate $1GB$ to $C0$ and another $1GB$ to its buffer pool. The later memory is essentially wasted, given the buffer pool's LRU page replacement policy, and Rows' sequential I/O patterns.

Our test hardware has two dual core 64-bit 3GHz Xeon processors with 2MB of cache (Linux reports 4 CPUs) and 8GB of RAM. 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 the "5.0.45-Debian_1ubuntu3" build of MySQL.

4.2 Comparison with conventional techniques

As Figure 1 shows, on an empty tree Rows provides roughly 7.5 times more throughput than InnoDB. As the tree size increases, InnoDB's performance degrades rapidly. After 35 million tuple insertions, we terminated the InnoDB run, as Rows was providing nearly 100 times more throughput. We continued the Rows run until the dataset was exhausted; at this point, it was providing approximately $\frac{1}{10}$th its original throughput, and had a target R value of 7.1. Figure 2 suggests that InnoDB was not actually disk bound during our experiments; its worst-case average tuple insertion time was approximately $3.4ms$; well below the drive's average access time. Therefore, we believe that the operating system's page cache was insulating InnoDB from disk bottlenecks[10]. This problem with our experimental setup should work in InnoDB's favor.

4.3 Prototype evaluation

Rows outperforms B-Tree based solutions, as expected. However, the prior section says little about the overall quality of our prototype implementation. In this section, we measure update latency, and compare our implementation's performance with our simplified analytical model.

Figure 3 reports Rows' replication throughput averaged over windows of 100,000 tuple insertions. The large downward spikes occur periodically throughout our experimental run, though the figure is truncated to only show the first 10 million inserts. They occur for two reasons. First, $C0$ accepts insertions at a much greater rate than $C1$ or $C2$ can accept them. Over 100,000 tuples fit in memory, so multiple samples are taken before each new $C0$ component blocks on the disk bound mergers. Second, Rows merges entire trees at once, occasionally blocking smaller components for long periods of time while larger components complete a merge step. Both of these problems could be masked by rate limiting the updates presented to Rows. A better solution would

data during a bulk load; new data is still exposed atomically.

[10]In the process of running our experiments, we found that while Rows correctly handles 64-bit file offsets, and runs on 64-bit platforms, it crashes when given more than 2GB of RAM.

perform incremental tree merges instead of merging entire components at once.

This paper has mentioned a number of limitations in our prototype implementation. Figure 4 seeks to quantify the performance impact of these limitations. This figure uses our simplistic analytical model to calculate Rows' effective disk throughput utilization from Rows' reported value of R and instantaneous throughput. According to our model, we should expect an ideal, uncompressed version of Rows to perform about twice as fast as our prototype performed during our experiments. During our tests, Rows maintains a compression ratio of two. Therefore, our model suggests that the prototype is running at $\frac{1}{4}$th its ideal speed.

A number of factors contribute to the discrepancy between our model and our prototype's performance. First, the prototype's whole-tree-at-a-time approach to merging forces us to make extremely coarse and infrequent runtime adjustments to the ratios between tree components. This prevents Rows from reliably keeping the ratios near the current target value for R. Second, Rows currently synchronously forces tree components to disk. Given our large buffer pool, a significant fraction of each new tree component is in the buffer pool or operating system cache when the merge thread forces it to disk. This prevents Rows from overlapping I/O with computation. Finally, our analytical model neglects some minor sources of storage overhead.

One other factor significantly limits our prototype's performance. Atomically replacing $C0$ doubles Rows peak memory utilization, halving the effective size of $C0$. The balanced tree implementation that we use roughly doubles memory utilization again. Therefore, in our tests, the prototype was wasting approximately $750MB$ of the $1GB$ we allocated for $C0$.

Our performance figures show that Rows significantly outperforms a popular, production quality B-Tree implementation. Our experiments reveal a number of deficiencies in our prototype implementation, suggesting that further implementation efforts would improve its performance significantly. Finally, though our prototype could be improved, it already performs at roughly $\frac{1}{4}$th of its ideal throughput. Our analytical models suggest that it will significantly outperform any B-Tree implementation when applied to appropriate update workloads.

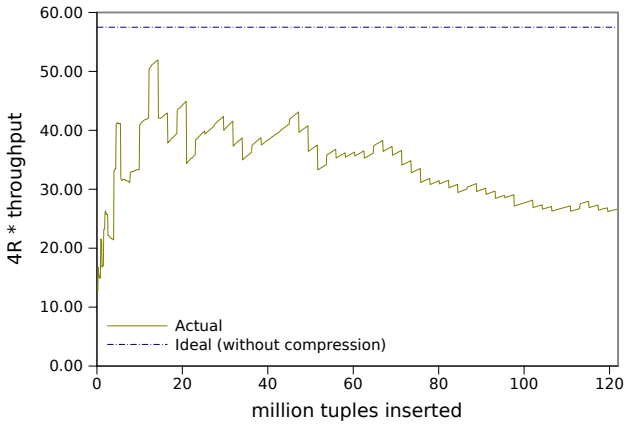

Figure 4: The Rows prototype's effective bandwidth utilization. Given infinite CPU time and a perfect implementation, our simplified model predicts $4R * throughput = sequential\ disk\ bandwidth * compression\ ratio$. (Our experimental setup never updates tuples in place)

5. RELATED WORK

5.1 LSM-Trees

The original LSM-Tree work[7] 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.

Later work proposes the reuse of existing B-Tree implementations as the underlying storage mechanism for LSM-Trees[3]. Many standard B-Tree operations (such as prefix compression and bulk insertion) would benefit LSM-Tree implementations. Rows uses a custom tree implementation so that it can take advantage of compression. Compression algorithms used in B-Tree implementations must provide for efficient, in place updates of tree nodes. The bulk-load update of Rows updates imposes fewer constraints upon our compression algorithms.

Recent work on optimizing B-Trees for write intensive updates dynamically relocates regions of B-Trees during writes [4]. 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.

The problem of *Online B-Tree merging* 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 particularly interesting approach lazily piggybacks merge operations on top of tree access requests. Upon servicing an index probe or range scan, the system must read leaf nodes from both B-Trees. Rather than simply evicting the pages from cache, lazy merging merges the portion of the tree that has already been brought into memory [9].

The original LSM-Tree paper proposes a mechanism that provides delayed LSM-Tree index scans with no additional I/O. The idea is to wait for the merge thread to make a pass over the index, and to supply the pages it produces to the index scan before evicting them from the buffer pool.

If one were to applying lazy merging to an LSM-Tree, it would 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 (in some cases) ignore similarities between adjacent data items. One such approach (for low cardinality data) builds a table-wide mapping from short identifier codes to 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 suppresses leading zeros. Row-based schemes typically allow for easy decompression of individual tuples. Therefore, they generally store the offset of each tuple explicitly at the head of each page.

Another approach is to compress page data using a generic compression algorithm, such as gzip. The primary drawback to this approach is that the size of the compressed page is not known until after compression. Also, general purpose compression techniques are typically more processor intensive than specialized database compression techniques [10].

5.3 Column-oriented database compression

Column-based compression is 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 (patched frame of reference) was introduced as an extension to the MonetDB[11] column-oriented database, along with two other formats (PFOR-delta, which is similar to PFOR, but stores values as deltas, and PDICT, which encodes columns as keys and a dictionary that maps to the original values). We plan to add both these formats to Rows in the future. We chose these formats as a starting point because they are amenable to superscalar optimization, and compression is Rows' primary CPU bottleneck. Like MonetDB, each Rows 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 particularly useful in Rows, as its merge processes perform repeated joins over compressed data. Our prototype does not make use of these optimizations, though they would likely improve performance for CPU-bound workloads.

A recent paper provides a survey of database compression techniques and characterizes the interaction between compression algorithms, processing power and memory bus bandwidth. To the extent that multiple columns from the same tuple are stored within the same page, all formats within their classification scheme group information from the same tuple together [5].

Rows, 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 amount of data on the disk and the amount of I/O performed by the query. In a column store, such optimizations happen off-line, leading to high-latency inserts. Rows can support such optimizations by producing multiple LSM-Trees for a single table.

Unlike read-optimized column-oriented databases, Rows is optimized for write throughput, and provides low-latency, in-place updates. This property does not come without cost; compared to a column store, Rows must merge replicated data more often, achieves lower compression ratios, and performs index lookups that are roughly twice as expensive as a B-Tree lookup.

5.4 Snapshot consistency

Rows relies upon the correctness of the master database's concurrency control algorithms to provide snapshot consistency to queries. Rows is compatible with the two most popular approaches to concurrency control in OLTP environments: two-phase locking and timestamps (multiversion concurrency control).

Rows only processes read-only transactions. Therefore, its concurrency control algorithms need only address read-write conflicts. Well-understood techniques protect against read-write conflicts without causing requests to block, deadlock or livelock [2].

5.5 Log shipping

Log shipping mechanisms are largely outside the scope of this paper; any protocol that provides Rows replicas with up-to-date, intact copies of the replication log will do. Depending on the desired level of durability, a commit protocol could be used to ensure that the Rows replica receives updates before the master commits. Because Rows is already bound by sequential I/O throughput, and because the replication log might not be appropriate for database recovery, large deployments would probably opt to store recovery and logs on machines that are not used for replication.

6. CONCLUSION

Compressed LSM trees are practical on modern hardware. As CPU resources increase, increasingly sophisticated compression schemes will become practical. Improved compression ratios improve Rows' throughput by decreasing its sequential I/O requirements. In addition to applying compression to LSM-Trees, we presented a new approach to database replication that leverages the strengths of LSM-Tree indices by avoiding index probing during updates. We also introduced the idea of using snapshot consistency to provide concurrency control for LSM-Trees. Our prototype's LSM-Tree recovery mechanism is extremely straightforward, and makes use of a simple latching mechanism to maintain our LSM-Trees' consistency. It can easily be extended to more sophisticated LSM-Tree implementations that perform incremental tree merging.

Our implementation is a first cut at a working version of Rows; we have mentioned a number of potential improvements throughout this paper. We have characterized the performance of our prototype, and bounded the performance gain we can expect to achieve via continued optimization of our prototype. Without compression, LSM-Trees can outperform B-Tree based indices by at least 2 orders of magnitude. With real-world database compression ratios ranging from 5-20x, we expect Rows database replicas to outperform B-Tree based database replicas by an additional factor of ten.

We implemented Rows to address scalability issues faced by large scale database installations. Rows addresses seek-limited applications that require near-realtime analytical and decision support queries over extremely large, frequently updated data sets. We know of no other database technology capable of addressing this class of application. As automated financial transactions, and other real-time data acquisition applications are deployed, applications with these requirements are becoming increasingly common.

7. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, New York, NY, USA, 2006. ACM.

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

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

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

[5] 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 '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 389–400, New York, NY, USA, 2007. ACM.

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

[7] 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.

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

[9] X. Sun, R. Wang, B. Salzberg, and C. Zou. Online b-tree merging. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 335–346, New York, NY, USA, 2005. ACM.

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

[11] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 59, Washington, DC, USA, 2006. IEEE Computer Society.