Fast Database Restarts at Facebook

Aakash Goel,* Bhuwan Chopra, Ciprian Gerea, Dhrúv Mátáni,
Josh Metzler, Fahim Ul Haq, and Janet L. Wiener
Facebook, Inc.

ABSTRACT

Facebook engineers query multiple databases to monitor and analyze Facebook products and services. The fastest of these databases is Scuba, which achieves subsecond query response time by storing all of its data in memory across hundreds of servers. We are continually improving the code for Scuba and would like to push new software releases at least once a week. However, restarting a Scuba machine clears its memory. Recovering all of its data from disk — about 120 GB per machine — takes 2.5-3 hours to read and format the data per machine. Even 10 minutes is a long downtime for the critical applications that rely on Scuba, such as detecting user-facing errors. Restarting only 2% of the servers at a time mitigates the amount of unavailable data, but prolongs the restart duration to about 12 hours, during which users see only partial query results and one engineer needs to monitor the servers carefully. We need a faster, less engineer intensive, solution to enable frequent software upgrades.

In this paper, we show that using shared memory provides a simple, effective, *fast*, solution to upgrading servers. Our key observation is that we can decouple the memory lifetime from the process lifetime. When we shutdown a server for a planned upgrade, we know that the memory state is valid (unlike when a server shuts down unexpectedly). We can therefore use shared memory to preserve memory state from the old server process to the new process. Our solution does not increase the server memory footprint and allows recovery at memory speeds, about 2-3 minutes per server. This solution maximizes uptime and availability, which has led to much faster and more frequent rollouts of new features and improvements. Furthermore, this technique can be applied to the in-memory state of any database, even if the memory contains a cache of a much larger disk-resident data set, as in most databases.

*Aakash is a graduate student at Georgia Institute of Technology and was an intern at Facebook.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD 2014 Park City, UT USA
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2595642.

1. INTRODUCTION

Facebook engineers query multiple database systems to monitor and analyze Facebook products and services. Scuba[5] is a very fast, distributed, in-memory database used extensively for interactive, ad hoc, analysis queries. These queries typically run in under a second over GBs of data. Scuba processes almost a million queries per day for over 1500 Facebook employees. In addition, Scuba is the workhorse behind Facebook's code regression analysis, bug report monitoring, ads revenue monitoring, and performance debugging.

One significant source of downtime is software upgrades, yet upgrades are necessary to introduce new features and apply bug fixes. At Facebook, we are accustomed to the agility that comes with frequent code deployments. New code is rolled out to our web product multiple times each week [9]. The Facebook Android Alpha program also releases code multiple times a week [18, 17]. We would like to deploy new code to Scuba at least once a week as well.

However, any downtime on Scuba's part is a problem for the many tools and users that depend on it. When a server process is shut down, it loses all of the data in its heap memory. The new server process must then read all of its data from the backup copy Scuba keeps on a local disk. However, Scuba machines have 144 GB of RAM, most of which is filled with data. Reading about 120 GB of data from disk takes 20-25 minutes; reading that data in its disk format and translating it to its in-memory format takes 2.5-3 hours, a very long time — about 4 orders of magnitude longer than query response time.

Scuba can and does return partial query results when not all servers are available. We can mitigate the long downtime by restarting only a handful of servers at a time, usually 2% of them, to minimize the impact on query results. The entire system rollover then takes a lot longer, about 12 hours to restart the entire Scuba cluster with hundreds of machines. Furthermore, an engineer needs to monitor the rollover for its entire duration. This time-consuming procedure discourages frequent deployment of new features and bug fixes.

We needed to reduce the total downtime significantly, since it prevented us from upgrading Scuba software as often as we want. One possible solution keeps redundant copies of the data in memory on different servers. When one server is being upgraded, queries are routed exclusively to the other server. We discarded that solution as too expensive in two dimensions: first, it would require twice as many servers. The hardware cost of hundreds of servers with 144 GB of RAM is significant. Second, replication code can be tricky to get right: Which server should participate in which queries?

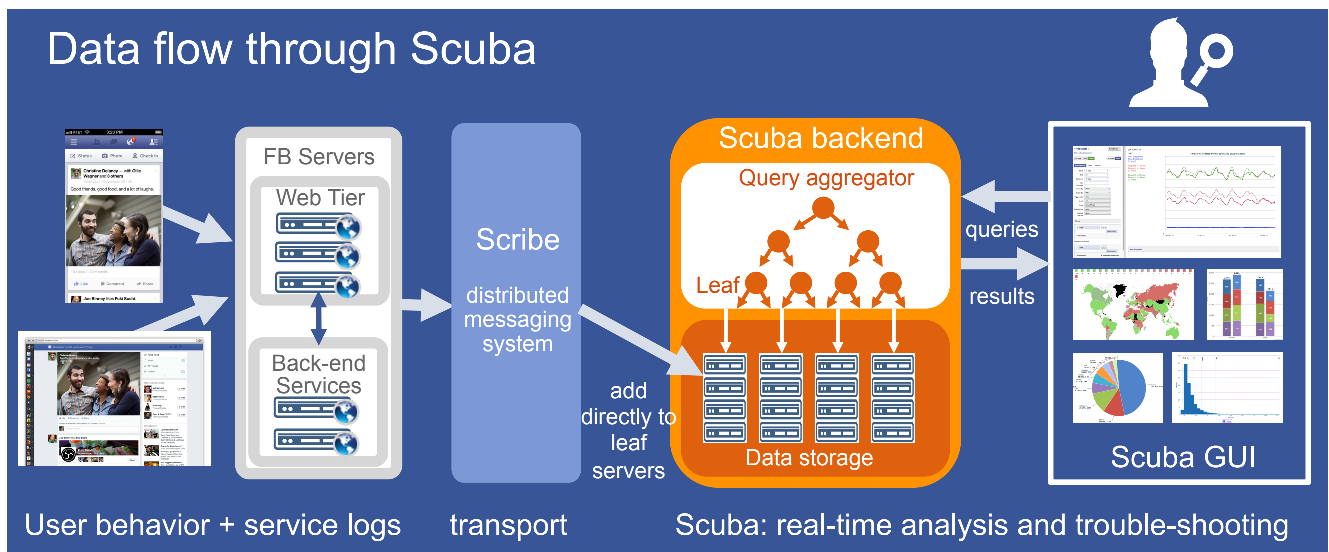

Figure 1: Scuba architecture: data flows from Facebook products and services through Scribe to Scuba. Users query Scuba and visualize the results in the Scuba GUI.

How should we keep pairs of servers synchronized with millions of row inserts per second?

Instead, we chose a different solution. We observed that when we shutdown a server for a planned upgrade, we know that the memory state is good (unlike when a server shuts down unexpectedly, which might or might not be due to memory corruption). We decided to decouple the memory's lifetime from the process's lifetime. In this paper, we describe how we use shared memory to persist data from one process to the next.

Using shared memory to store data provides a simple, effective solution to upgrading servers *fast*. We were inspired by two other big, distributed systems at Facebook that use shared memory to keep data alive across software upgrades: TAO [6] and Memcache [20]. In our solution, we made two key design decisions:

1. Scuba copies data from heap memory to shared memory at shutdown time and copies it back to the heap at startup.
2. During the copy, data structures are translated from their heap format to a (very similar but not the same) shared memory format.

Copying data between heap and shared memory avoids some of the pitfalls in writing a custom allocator in shared memory, such as fragmentation and problems with thread safety and scalability. It also allows us to modify the in-memory format (in heap memory) and rollover to the new format using shared memory. We describe how to copy all of the data to shared memory and back without increasing the memory footprint of the data.

Scuba's new upgrade path is about 2-3 minutes per server, rather than 2-3 hours. The entire cluster upgrade time is now under an hour, rather than lasting 12 hours. This path maximizes uptime and availability for Scuba users and minimizes monitoring time of the upgrade for our engineers. For example, instead of having 100% of the data available only 93% of the time with a 12 hour rollover once a week, Scuba is now fully available 99.5% of the time — and that hour of downtime can be during offpeak hours (after typical California office hours, when many Scuba users, i.e., Facebook engineers, are not working).

We are now able to deploy new features and improvements much more frequently. We believe this restart technique can be applied to the in-memory state of any database, even if the memory contains a cache of a much larger disk-resident data set, as in most databases.

In the next section, we describe Scuba's architecture. In Section 3, we show Scuba's data layout in shared memory and in Section 4 we describe the rollover procedure using shared memory. We consider related work in database recovery and using shared memory for fast system restarts in Section 5. Finally, we conclude in Section 6.

2. SCUBA ARCHITECTURE

Figure 1 shows Scuba's overall architecture. Data flows from log calls in Facebook products and services into Scribe [3]. Scuba "tailer" processes pull the data for each table out of Scribe and send it into Scuba.

Every N rows or t seconds, the tailer chooses a new Scuba leaf server and sends it a batch of rows. How does it choose a server? It picks two servers randomly and asks them both for their current state and how much free memory they have, as described previous [5]. If both are alive (see Figure 5(a)), it sends the data to the server with more free memory. If only one is alive, that server gets the data. If neither server is alive, the tailer will try two more servers until it finds one that is alive or (after enough tries) sends the data to a restarting server.

Each machine currently runs eight leaf servers and one aggregator server. The leaf servers store the data. Having eight servers allows for greater parallelism during query execution (without the complexity of multiple threads per query per server). More importantly for recovery, eight servers mean that we can restart the servers one at a time, while the other seven servers continue to execute queries. We there-

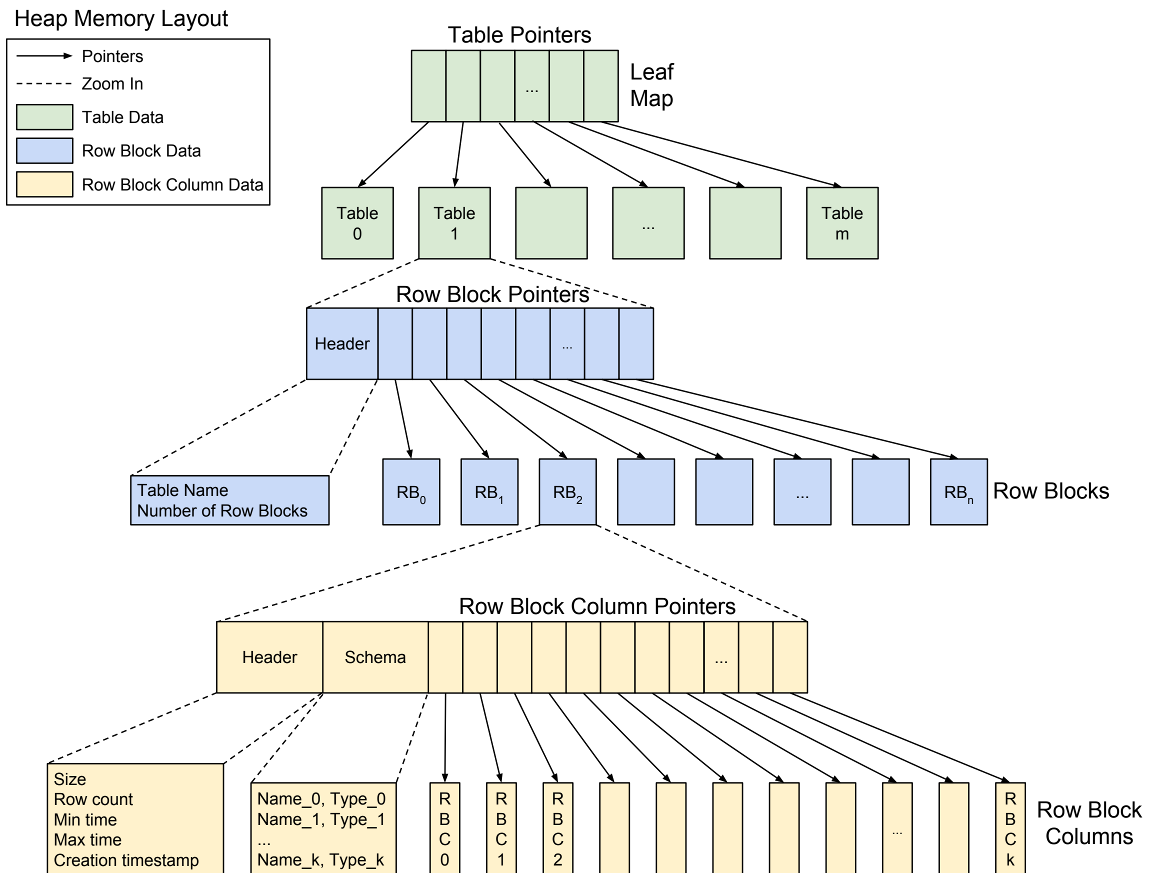

Figure 2: Heap memory layout for tables in Scuba. Each Table has a vector of Row Blocks. A Row Block contains all data for a set of rows. Each Row Block has a header, a schema, and a vector of Row Block Columns. Each Row Block Column contains the values for one column, for all rows in the Row Block.

fore maximize the number of disks in use for recovery while limiting the amount of offline data to 2% of the total. For example, suppose there are 100 machines. With one server per machine, we could restart only two servers. With a total of 800 leaf servers, we can restart 16 leaf servers on 16 machines at once and read from 16 disks. The full rollover thus takes much less time to complete. This technique also applies to parallelizing restarts using shared memory, although the critical resource is the memory bandwidth rather than the disk speed.

The leaf servers both add new data as it arrives and process queries over their current data. They also delete data as it expires due to either age or size limits.

The aggregator servers distribute a query to all leaves and then aggregate the results as they arrive from the leaves. Our previous work [5] describes query processing in more detail.

2.1 Storage layout

Within each leaf server, there is a fraction of most tables. Scuba's storage engine is a column store (a change since [5]. A column layout provides better compression of the data and enables faster query execution strategies, as described by others for C-Store [23] and Vertica [14], MonetDB [12], SAP Hana [22], Dremel [19], and Powerdrill [10].

Figure 2 depicts the memory layout of a leaf. There is a leaf map containing a vector of pointers, one pointer to each table. Each table has a vector of pointers to row blocks (RBs) plus a header. The table name and a count of the row blocks are in the table header. Each row block contains 65,536 rows that arrived consecutively. (The row block is capped at 1 GB, pre-compression, even if there are fewer than 65K rows.) Within each row block, the data is organized into a header, a schema, and row block columns. Each row block column contains all of the column values for one column, for every row in the row block.

The header describes general properties of the row block: its size in bytes, the number of rows in it (it may not be full), the minimum and maximum timestamps of rows it contains, and when the row block was first created. Every row in Scuba has a required column called "time" that contains a unix timestamp. These timestamps represent the time of the row-generating event. They are not unique, as many events happen on Facebook in the same second. Since rows flow into Scuba in roughly chronological order, the time column is close to an index for each table. Nearly all queries contain predicates on time; the minimum and maximum timestamps are used to decide whether to even look at a row block when processing a query.

The schema is a description of the columns in the row

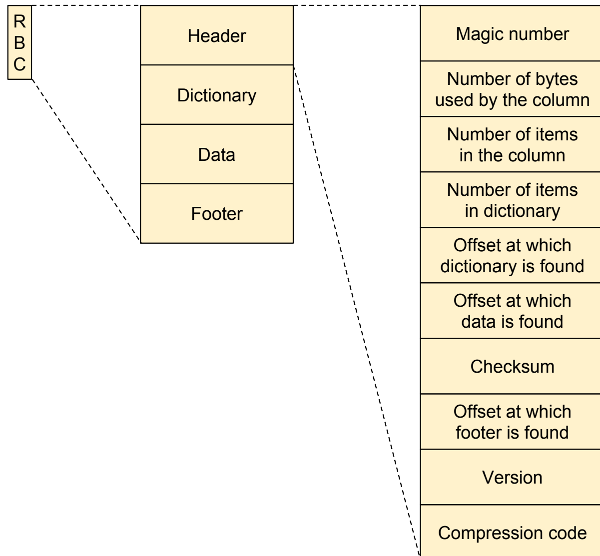

Figure 3: Row block column (RBC) layout for tables in Scuba.

block: their names and types. Different row blocks may have different schemas, although they usually have a large overlap in their columns.

Finally, Figure 3 shows the row block column layout. Each row block column contains a header, a dictionary if needed, the data (column values), and a footer. The header of the row block column starts at a base address. All other addresses in the row block column, such as the beginning of the dictionary, data, and footer, are offsets from this base address. BerkeleyDB [21] is another database that uses a base address plus offsets for its pointers. Using offsets enables us to copy the entire row block column between heap and shared memory in one memory copy operation. Only the address of the row block column itself (in the row block) needs to be changed for its new location.

The data in the row block column is stored in a compressed form. Compression reduces the size of the row block column by a factor of about 30, although compression results are outside the scope of this paper. Scuba's compression methods are a combination of dictionary encoding, bit packing, delta encoding, and lz4[7] compression, with at least two methods applied to each column.

3. SHARED MEMORY

Shared memory allows interprocess communication. For Scuba, shared memory allows a process to communicate with its replacement, even though the lifetimes of the two processes do not overlap. The first process writes to a location in physical memory and the second process reads from it. We use the Posix mmap (mmap, munmap, sync, mprotect) based API from Boost::Interprocess [4].

We considered two alternative methods of using shared memory:

1. Allocate all data in shared memory all of the time. This alternative requires writing a custom allocator to subdivide shared memory segments. To get thread safety and scalability in the allocator adds significant complexity.

2. Allocate data in heap memory during normal operation. Copy it to shared memory at shutdown and copy it back at start up. This method involves extra time for copying to and from shared memory, albeit at memory speeds. Copying also needs to be performed carefully, to ensure that there is enough memory.

At Facebook, our default heap memory allocator is jemalloc [8]. Jason Evans, the author of jemalloc, discussed writing a new shared memory allocator with us. jemalloc uses lazy allocation of backing pages for virtual memory to avoid fragmentation. Since Scuba is entirely memory-bound (rather than CPU-bound), using memory efficiently is very important. In shared memory, lazy allocation of backing pages is not possible. We worried that an allocator in shared memory would lead to increased fragmentation over time.

Therefore, we chose method 2. We describe how we copy to and from shared memory in the next section.

4. RESTART IMPLEMENTATION

We now describe the restart mechanism in Scuba. Scuba stores backups of all incoming data to disk, so it is always possible to recover from disk, even in the case of a software or hardware crash. When there is a clean shutdown, such as when we want to deploy a new Scuba binary, we can use shared memory rather than restarting by reading from disk. We do not use shared memory to recover from a crash; the crash may have been caused by memory corruption. We first outline recovery from disk and then describe how we can rollover from shared memory.

4.1 Restart from disk

There are two steps involved in a leaf restart: shutdown of the old server process and startup of the new server process.

1. Shutdown of a Scuba leaf server is straightforward. When it receives an API call to shutdown cleanly, the server stops accepting new data and new queries, finishes answering queries already in flight, finishes any pending synchronization with the data on disk, and exits. Although data synchronization to disk is a bottleneck, only the sections of data that have changed since the last synchronization point need to be updated. (During normal operation, disk writes are asynchronous.) If there is a crash rather than a clean shutdown, some new data may be lost. Since Scuba does not guarantee full query results, we consider losing a tiny amount of data (a few thousand rows out of millions of rows inserted per day) acceptable and it simplifies recovery greatly.

2. Starting a new Scuba server process is slower than shutting it down. All of the data for the server process needs to be read from the disk. While the new process starts answering queries as soon as it comes up, it only returns (gradually increasing) partial results to those queries until it completes recovery. The server also accepts new data as soon as it starts recovery, but the tailers will avoid adding data to servers in recovery if possible.

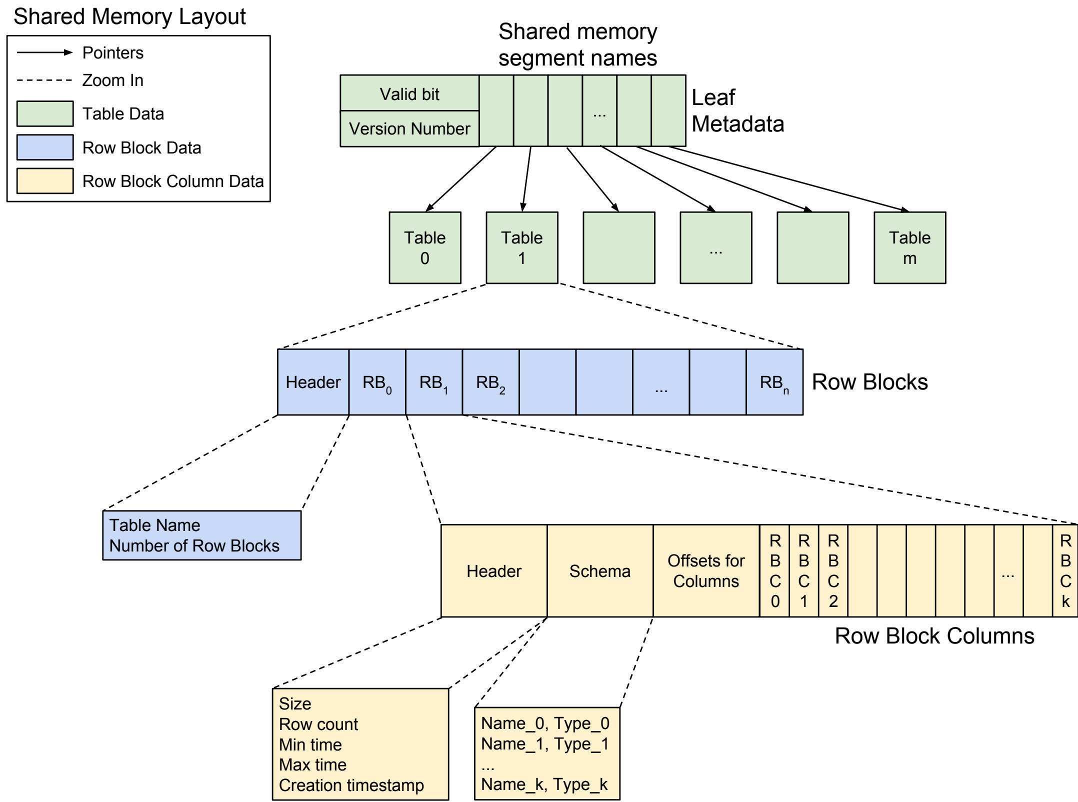

Figure 4: Shared memory layout for tables in Scuba. Shared memory layout is very similar to Heap memory layout. The primary difference is that Row Blocks and Row Block Columns can be laid out contiguously in memory, since the full set of them (and their sizes) is known when the memory is allocated. The shared memory layout therefore loses one level of indirection for both Row Blocks and Row Block Columns. Additionally, there is leaf metadata for every leaf server at a fixed location. This metadata says whether the shared memory is valid (usable for recovery) and identifies the shared memory segements being used.

Restart from disk is slow, but resilient to crashes and changes in memory layouts. Before we describe restarts from shared memory, we first present the memory layout of data in shared memory and contrast it to the heap memory layout.

4.2 Shared memory layout

Figure 4 shows the memory layout of tables, row blocks, and row block columns in shared memory. Figures 2 and 4 are very similar. Since the number and contents of row blocks and row blocks columns are known at allocation time in shared memory, we can eliminate one level of indirection and allocate them contiguously.

Additionally, there is leaf metadata for each of the eight leaf servers, although at most one of them will roll over using shared memory at a time. (Memory bandwidth for a machine is constant, no matter how many servers try to roll over, so it is much better to restart eight leaf servers on eight different machines in parallel than to restart all eight leaf servers on the same machine at once. See the example in Section 1 for a more detailed explanation.)

Each leaf has a unique hard coded location in shared memory for its metadata. In that location, the leaf stores a valid bit, a layout version number, and pointers to any shared memory segments it has allocated. There is one segment per table. The layout version number indicates whether the shared memory layout has changed; note that the heap memory layout can change independently of the shared memory layout.

4.3 Restart using shared memory

At all times, each leaf and table keeps track of its state. The state indicates whether the leaf and table are working on a restart and determines which actions are permissible: adding data, deleting (expired) data, evaluating queries, etc. Figure 5 illustrates the state machines for both leaves and tables.

Like restart from disk, restarting a leaf using shared memory also has two steps.

1. Shutdown involves copying all of the table data from heap memory to shared memory and setting a valid bit in shared memory before exiting. Figure 6 shows pseudocode for the shutdown procedure.

2. Starting a new server then first checks the valid bit in shared memory. If it is set, the server copies the data from shared memory back to the heap. If it is *not* set, the server reverts to recovering from disk (and frees any shared memory in use). Figure 7 shows pseudocode for the restart procedure.

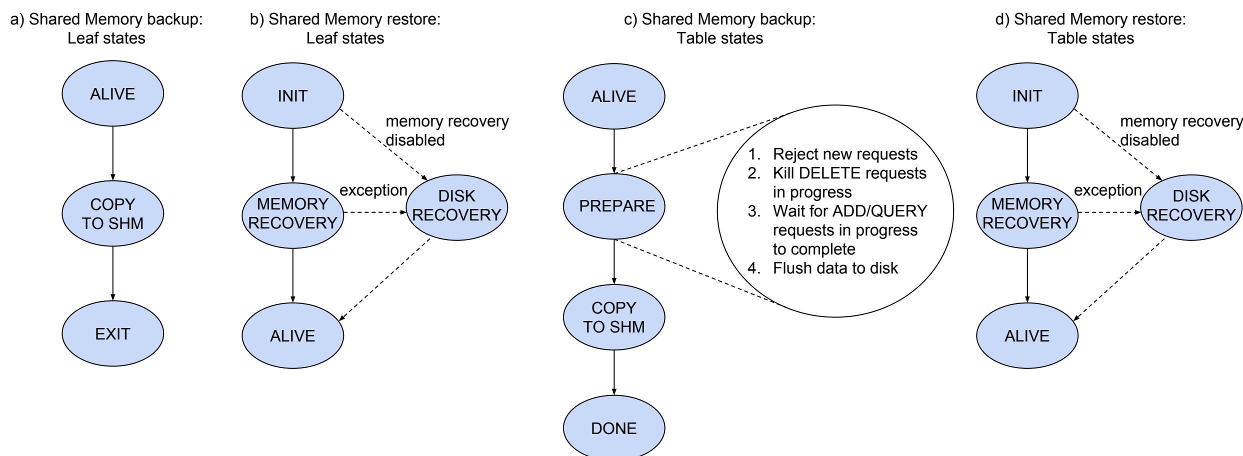

Figure 5: State machines for shutdown and restart in Scuba. (a) and (b) are the state machines for a leaf server. In (a), a leaf transitions from being alive, to being in "copy" mode, to exiting. In (b), a new leaf server transitions from initializing, to attempting memory recovery if it is enabled and disk recovery if not, to being alive. In (c), a table that is shutting down has one more state than a leaf: it transitions through a prepare state where it waits for some requests, kills delete requests, and rejects any new work. (Scuba stops deleting expired table data once shutdown starts. Any needed deletions are made after recovery.) In (d), the table restart state machine is identical to the leaf restart state machine.

```
create shared memory segment for leaf metadata
set valid bit to false

for each table
   estimate size of table
   create table shared memory segment
   add table segment to the leaf metadata

   for each row block
      grow the table segment in size if needed
      for each row block column
         copy data from heap to the table segment
         delete row block column from heap
      delete row block from heap
   delete table from heap

set valid bit to true
```

Figure 6: Shutdown pseudocode: backup all data to shared memory segments. The leaf metadata is at a known location, specified as a parameter to the leaf server.

```
if valid bit is false
   delete shared memory segments
   recover from disk
   return

set valid bit to false
for each table shared memory segment

   for each row block
      for each row block column
         allocate memory in heap
         copy data from table segment to heap

      truncate the table shared memory segment if needed
   delete the table shared memory segment

delete the metadata shared memory segment
```

Figure 7: Restart pseudocode: restore all data from shared memory segments. If this code path is interrupted, the valid bit will be false on the next restart and disk recovery will be executed.

The script that issues the shutdown command to each leaf then waits in a loop for the leaf server process to die. Usually, the leaf copies its data to shared memory and exits in 3-4 seconds. However, the loop ensures that we kill the leaf server if it has not shut down after 3 minutes. If the old leaf server is killed, the new leaf server will restart from disk.

During memory recovery, which takes a few seconds per leaf, no add data requests or queries are accepted. As we explain below, during a planned rollover, we keep most of the leaves alive at all times. The leaves that are alive accept the add requests (which can go to any leaf) and the query results are missing in only a tiny fraction of data. During disk recovery, which takes longer, both add and query requests are processed by each leaf.

4.4 Copying to and from shared memory

Even though one leaf server only contains 10-15 GB of data, there is still not enough physical memory free to allocate enough space for it in shared memory, copy it all, and then free it from the heap. Instead, we copy data gradually, allocating enough space for one row block column at a time in shared memory, copying it, and then freeing it from the heap. There are hundreds of tables (and thousands of

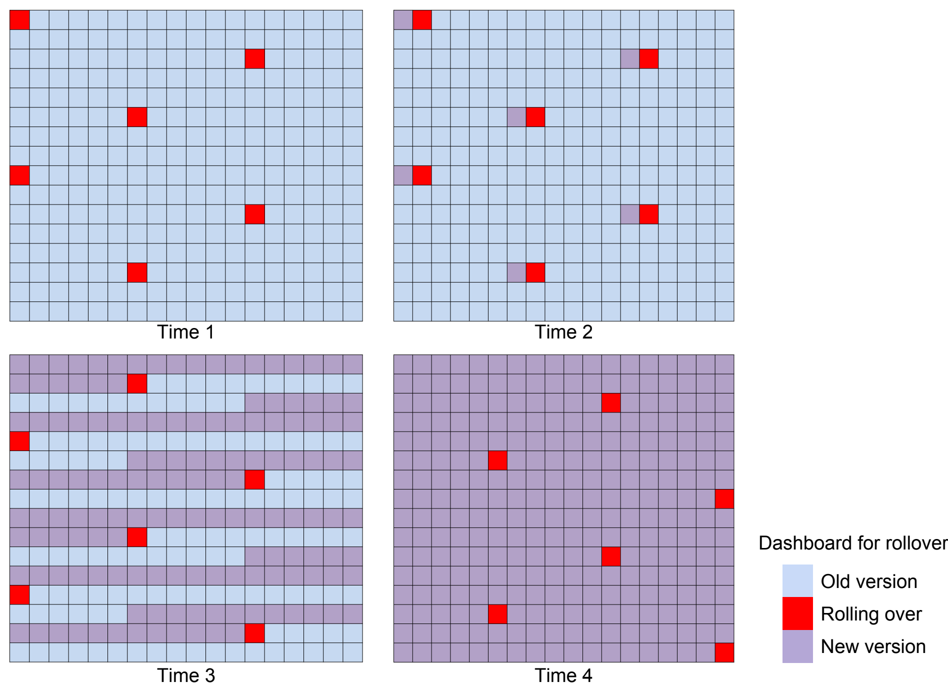

Figure 8: Dashboard shows progress of the restart. At time 1, about 2% of the leaf servers have started a rollover. 98% of the data is available to queries. At time 2, those leaf servers are now alive and another 2% are restarting. By time 3, about half of the servers are running the new version of the code, about half of the servers are running the old version, and a different 2% is restarting. At time 4, the restart is nearly complete.

row block columns, with a maximum size of 2 GB) per leaf servers, so this method keeps the total memory footprint of the leaf nearly unchanged during both shutdown and restart.

As explained in Section 2, since all pointers in a row block column are offsets from the start of the row block column, copying a row block column can be done in one call to *memcpy*. Therefore, copying a table only requires one call per row block column.

4.5 System-wide rollover

Shutting down and restarting many hundreds of leaf servers takes a long time. If all servers recover from disk at once, it takes 2.5-3 hours. If we plan a rollover, we keep most of the data available for queries. Typically, we restart 2% of the leaf servers at a time, and the entire rollover takes 10-12 hours to restart from disk. We therefore monitor the rollover process closely, to make sure it is making progress. Figure 8 shows an example dashboard depicting the progress of a rollover. Using shared memory is much faster, about 2-3 minutes per server (including the time to detect that a leaf is done with recovery and then initiate rollover for the next one).

5. RELATED WORK

In this section, we discuss database recovery and uses of shared memory in other types of distributed systems.

5.1 Database recovery

Most databases rely on recovery from disk (or sometimes solid state media). VoltDB [24], SAP Hana[22, 16], Hekaton [15], and TimesTen [13], are in memory databases that recover using a combination of checkpoints and write ahead logs.

Other database systems, such as SQLite [11], store the metadata required for restarts in shared memory. The metadata provides an index into the data files. For example, SQLite maintains a write-ahead-log index in shared memory. This technique restricts the amount of data kept in memory yet saves many disk accesses (for lookups) during recovery.

Finally, there are database systems that use shared memory to coordinate actions between concurrent server processes. eXtremeDB [2] is one such example. Since Scuba is essentially coordinating state between two *non-overlapping* server processes, coordinating their actions is not relevant. Also, different Scuba servers do not share any data, hence there is no need to coordinate between them.

5.2 Shared memory usage in other systems

At Facebook, two other big, distributed systems use shared memory to keep data alive across software upgrades: TAO [6] and Memcache [20]. The original inspiration to use shared memory for Scuba upgrades came from these systems.

Shared memory is also used for application checkpointing [1], where processes that need to coordinate to perform a checkpoint do so in shared memory. STLdb [25] stores C++ data structures in shared memory for persistence, much as Scuba uses shared memory for persistence beyond process lifetimes.

6. CONCLUSIONS

Using shared memory to store data between database server process lifetimes provides a fast rollover solution for Scuba. No extra memory or machines are needed, since we allocate, copy, and free data in chunks of one row block column (at most 1 GB) at a time. We can restart one Scuba machine in 2-3 minutes using shared memory versus 2-3 hours from disk. These numbers also apply to restarts of all of the machines at the same time.

Copying data between heap and shared memory has several advantages. Allocating and freeing heap memory during normal operation remains simple and uses well-tested code paths. The copying code is simple and, even though it is used infrequently, less likely to have bugs. Finally, separating the heap data structures from the shared memory data structures means that we can modify the heap data format and restart using shared memory.

Furthermore, this fast rollover path allows us to deploy experimental software builds on a handful of machines, which we could not do if took longer. We can add more logging, test bug fixes, and try new software designs — and then revert the changes if we wish. This use of shared memory rollovers as a software development tool is common in the Memcache and TAO teams at Facebook.

To maintain high availability of data without replication, we typically restart only 2% of Scuba servers at a time. By running N leaf servers on each machine (instead of only one leaf server), we increase the number of restarting servers by a factor of N. Restarting only one leaf server per machine at a time then means that N times as many machines are active in the rollover — and we get close to N times as much disk bandwidth (for disk recovery) and memory bandwidth (for shared memory recovery). We can restart the entire cluster of Scuba machines in under an hour by using shared memory, with 98% of data online and available to queries. In contrast, disk recovery takes about 12 hours. (The deployment software is responsible for about 40 minutes of overhead.)

One large overhead in Scuba's disk recovery is translating from the disk format to the heap memory format. This translation overhead is both time-consuming and CPU-intensive. We are planning to use the shared memory format described in this paper as the disk format, instead. We expect that the much simpler translation to heap memory format will speed up disk recovery significantly. We still need to recover from disk in case of software or hardware failures and hardware upgrades.

We also expect that replacing disks with solid state drives will speed up recovery from persistent storage, but writing to and reading back from memory will still be faster.

7. ACKNOWLEDGMENTS

Jay Parikh first suggested using shared memory for recovery. Jason Evans convinced us not to write a custom allocator in shared memory. Ryan McElroy and Nathan Bronson explained how Facebook's Memcache and TAO, respectively, use shared memory to make recovery faster.

8. REFERENCES

[1] Application checkpointing. http://en.wikipedia.org/wiki/Application_checkpointing.

[2] eXtremeDB Embedded In-Memory Database System. http://www.mcobject.com/standardedition.shtml.

[3] Scribe. https://github.com/facebook/scribe.

[4] Sharing memory between processes - 1.54.0. http://www.boost.org/doc/libs/1_54_0/, 2013.

[5] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, et al. Scuba: diving into data at facebook. In *VLDB*, pages 1057–1067, 2013.

[6] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *USENIX*, 2013.

[7] Y. Collet. Lz4: Extremely fast compression algorithm. code.google.com, 2013.

[8] J. Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan*, 2006.

[9] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[10] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, July 2012.

[11] D. R. Hipp. Sqlite: Write-ahead log. http://www.sqlite.org/draft/wal.html.

[12] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[13] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[14] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica Analytic Database: C-Store 7 Years Later . *PVLDB*, 5(12):1790–1801, 2012.

[15] P.-Å. Larson, M. Zwilling, and K. Farlee. The Hekaton Memory-Optimized OLTP Engine. *IEEE Data Eng. Bull.*, 36(2):34–40, 2013.

[16] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krueger, and M. Grund. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[17] C. Legnitto. 1m people try to help Facebook spruce up Android. http://news.cnet.com/8301-1023_3-57614540-93/1m-people-try-to-help-facebook-spruce-up-android/.

[18] C. Legnitto. Update on the Facebook for Android beta testing program. https://m.facebook.com/notes/facebook-engineering/update-on-the-facebook-for-android-beta-testing-program/10151729114953920.

[19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[20] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani.

Scaling Memcache at Facebook. In *NSDI*, pages 385–398. USENIX Association, 2013.

[21] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX*, pages 183–191, 1999.

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

[23] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564, 2005.

[24] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[25] B. Walters. STLdb. http://sourceforge.net/apps/trac/stldb/.