SquirrelDB

Nick Alereza
Stephen Calabrese
Nick Clarke
Ethan Frame
Grant Frame
Luke Plewa

***Abstract*—The purpose of a database is to store large amounts of information for retrieval and usage in the future. Most database implementations store data on disk. Our database implementation will use an alternative approach to storing data. Data will be stored in main memory. This move will improve the overall speed of the database, at the sacrifice of overall storage space. To add more storage we will be implementing a distributed database; utilizing a collection of nodes to store the data and thus increase the overall capacity. We will consider one node to be the master while designating all the other nodes as slaves. Each slave node will only be able to communicate with the master, creating a hub and spoke pattern. In order to remove the bottleneck of message passing, the master node will be mostly filled with metadata concerning the data contained in the slave nodes. Another salient feature in our architecture is the lookup table contained in the master node. Each entry will be comprised of a (key, (node, memLoc), (node, memLoc), dataLen) tuple. From this table, the master will be able to relate each (key, value) to the correct node holding the data, the memory location of that data on that node, and the length of the data. Such design decisions have been made to increase the speed of retrieving data from the database, at the cost of sacrificing consistency and overall memory size. Mechanisms have also been implemented to ensure continued functionality in case the master node crashes.**

I. INTRODUCTION

SquirrelDB is a key-value database in memory using a slave/master multi-node distribution written in C. It will use a query API consisting of four functions: get, put, update and delete. Although the entire database will operate in main memory, it will be periodically flushed to the hard disk to create a backup. The nodes will be divided such that there is one master node and one or more slave nodes. This master node will do the computations, store the databases index table, and track the amount of free space on each slave node. There will be a series of slave nodes connected exclusively to the master. These slave nodes will store the actual data. The communication between nodes will utilize the Open MPI library.[1]

Here is a quick outline of the rest of the paper. Section II discusses some background information on databases and distributed systems. Related work is detailed in section III. Our design and implementation is explained in section IV. Testing and validation is outlined in section V. The performance of our implementation is detailed in section VI. A conclusion is given in section VII and some future work is outlined in section VIII.

II. BACKGROUND

The most basic database is a straightforward list of items. The next expansion is to add an index onto each item within the list. This index can be considered the key which corresponds to the items value. Thus forms a system of (key, value) pairs.

A (key, value) database is advantageous in that it is simple and fast but has drawbacks in the large memory space requirements. Each object is associated with a single key but the object can contain varying amounts of data. This amount of information can quickly become very large and thus the database must be able to handle such an influx. This is usually solved by exploiting the considerable expanse of disk space at hand. However, this will tend to slow down the system whenever extensive reads and writes are involved.[2] Our design follows another option. We store all the database information in main memory, interacting with the disk as a safety backup. Leveraging a distributed model, allows our system to seamlessly expand its storage space by including more nodes. These new nodes do not process any logic; they only provide additional primary memory.

Because we plan on following a distributed model, a NoSQL data management system is the best fit for SquirrelDB. For a cloud or a database consisting of a variable number of nodes, SQL systems are less efficient due to the large number of messages required over a network to ensure the needed two-phase transaction commitment protocol. NoSQL systems can also allow for additional computational nodes to be added to existing ones easily, while SQL systems are inflexible in their design.[3]

"It is common practice for NoSQL systems to make explicit tradeoffs with respect to desirable properties."[4] The CAP theorem is used to justify the trade offs of a non traditional database system. The CAP theorem states that no distributed system can simultaneously ensure consistency, availability and partition tolerance. For the system to be consistent, all nodes must contain consistent data at any time. This means that all nodes have the same data at any time. For the system to be available, if any node fails the remaining nodes will continue to operate. Finally, for a system to be partition tolerance, if the "system splits up into disconnected groups each group of nodes will continue to operate." [3]

III. RELATED WORKS

There were four main database management systems that we felt were comparable to the system that we were producing. They were analyzed over the areas of storage method, replication strategy, and distribution technique. By borrowing successful mechanisms from each, the system we present attempt to maximize the benefits while eliminating as many downsides as possible.

The first is MemcacheDB. MemcacheDB is a simple key-value database that stores all information within RAM.[5] It uses hash methods to index the data. When full, MemcacheDB will use an LRU algorithm to drop the least recently used piece of data. The hash-table format that MemcacheDB utilizes lends itself easily to being distributed across multiple machines. Repcached is an add-on that can be used with MemcacheDB to introduce replication.[6] This allows for data to be to duplicated from one machine to another, leading to full recovery in the case one of those machines crashes. From this summary, it can be seen that MemcacheDB is very close to the system, we are trying to implement. Yet, it is not identical. Using a hash-table based system means that MemcacheDB does not fully employ all its memory space. There will be gaps of empty memory in areas that are never reached by the output of the hash function when the input is not perfectly spread among values. Troubles can also arrive from repeated collisions. Our system will borrow several aspects of replication and distributed while trying to improve on the methods of storage.

The second DBMS that exhibits similarity to our work is Redis. Redis stores values in memory and and maps to them with a series of keys.[7] The keys are inputted with the data and thus are not generated using any hash function. Redis accepts any form of value entry instead of restricting to exclusively strings. Redis solves replication and distribution together using a master-slave tree structure. Each node in the tree is a slave to the nodes above it and a master to the nodes below. This lets information and messages to be passed and replicated throughout subtrees in addition to distributing the actual data storage around the nodes. This method does not scale well over a great many writes but is sufficient when expecting mostly reads. Our system will leverage a master-slave architecture that is slightly different in that it interacts as a hub-and-spoke. We will also generate keys automatically instead of accepting user input but plan to accept any value type.

The next system we investigated was VoltDB.[8] Released around 2010, VoltDB uses in-memory storage on an architecture that is relational and shared-nothing. It utilizes NewSQL which is an attempt to provide higher speed and throughput while simultaneously maintaining the assurances traditionally provided by an ACID DBMS. It has the capability for whole database replication or replication across a cluster. When executing commands, the procedures are sent to nodes that run in parallel.[9] Although VoltDB has several advantages in speed and accuracy, many of the techniques will not be applied to the system we are proposing.

Finally, we investigated Gemfire.[10] This DBMS manages data by distributing it over the memory of several nodes. As data comes in, it is replicated and partitioned out to multiple nodes. The highly distributed architecture leads to high throughput and data flow. The nodes act in parallel making the entire system highly scalable and greatly increasing the availability of the data. However, when values are updated, the changes have to ripple through every location that the data has been replicated within the system, slowing response times. Our system will borrow similar strategies in confronting distribution and replication. Implementing a master-slave formation will affect how and where data is replicated in addition to the partitioning and message passing.

By considering the advantages and noticing the disadvantages that are embedded in each of these DBMS, the system we propose will be able to leverage or overcome each element to provide the best system achievable.

IV. Design and Implementation

SquirrelDB uses a master/slave architecture for data storage. There is always only one master node in the system, which the slave nodes receive messages from. The database uses a client-server model to accept input from a user. The master node acts as the server in our implementation. The architecture is show in Figure 11 (included at the end of the document). Below each aspect of the database is described in detail.

A. Storage

There are two levels of storage in SquirrelDB, master storage and slave storage. Since SquirrelDB is a key/value store and has a master/slave architecture, we have split what data resides on the master and what data resides on a slave. The master node stores the keys, while a slave node stores values. Since the size of the values will be much larger than the size of the keys, it is reasonable to devote only one master node to store keys and slave nodes to store values. Think of a library, where there is a directory or card catalog, which keeps information about where each book is stored, and then there is the actual bookshelf where all the books are stored. This is the model for SquirrelDB.

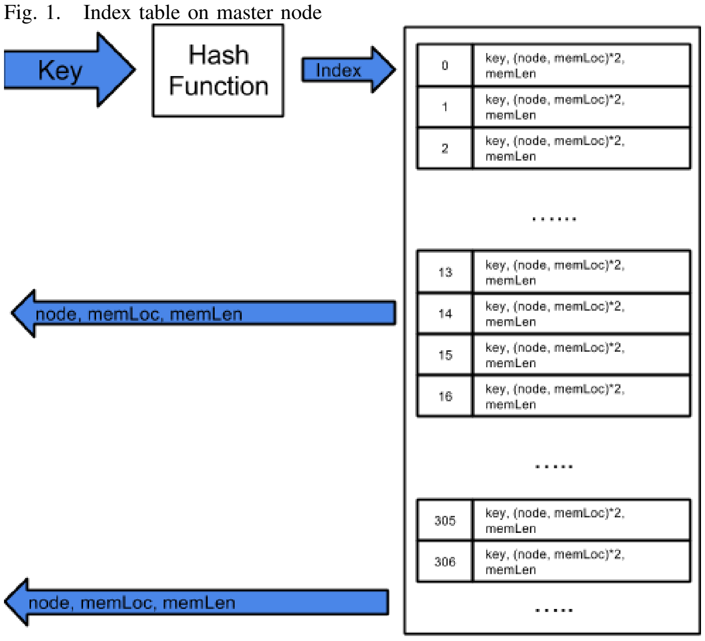
Storage in SquirrelDB is done in memory. All tables and data are stored in memory while the database is running. The only time SquirrelDB interacts with disk is when flushing the data, in order for the system to be somewhat fault tolerant. All of these tables are flushed to disk periodically in order to recover from crashes.[11] This is detailed more in the crash recovery section below.

1) Master Storage: Master storage has two tables it uses for data storage and recovery: an index table and a free memory table.

a) Index Table: First is the index table, seen in Figure 1. The index table is a hash table, where the key is used as the input to the hash function to obtain the index in the table. [11] We have used MurmurHash3 to hash our keys to produce an index. At each location in the index table three pieces of information are stored: the key, the storage locations, and the length of the value. Each key should map to an index, however there may be collisions, thus the key needs to be stored to make sure that when an operation is performed, the correct index table data is used. The second item stored in the index table is an array whose size is the number of replication nodes used by the system. At each index in the array the following are stored: the node which is storing the value, and the beginning memory location where the value is stored for that node. So for example if the number of replication nodes is two and the value is stored on nodes 2 and 3, at locations 500 and 2 respectively, the array would look like [{2,500},{3,2}]. This way the master can easily query the slave node for the value it is looking for. The last piece of data stored at the each index is the length of the value. This was done so that the master can tell the slave holding the data how much data should be returned.

When searching for a key in the hash table we leveraged the fact that if a entry had data length as 0 then the hash table at that index doesn't contain data. This helps when searching for a key that is not stored in the case a collision happens. If a key is hashed and it turns out that the index it would have been stored already contains another keys data, then we move to the next index. If at any point a value in the hash table has data length as 0 then we know that the index table cannot contain the key we are looking for, as it would have been placed in the first open space. This allows us to avoid searching the entire index table, unless every position is filled.

Fig. 1. Index table on master node

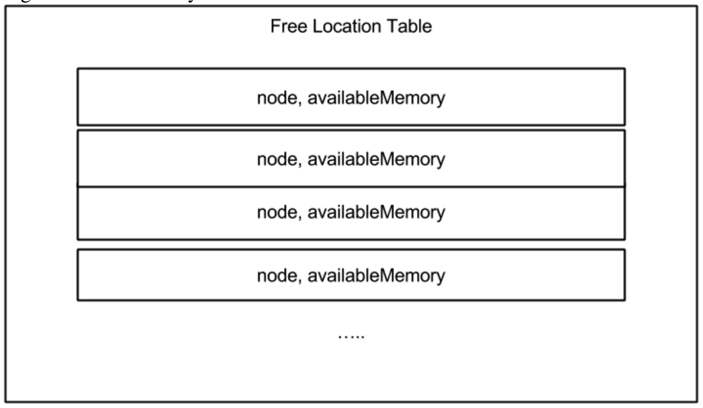


b) Free Memory Table: The second table that resides on the master node is the free memory table, seen in Figure 2. This table helps the master node keep track of how much memory each slave node has free. This table is used for load balancing. When a put request comes to the master node, the master node checks and finds which nodes have the most amount of storage space available, and uses those nodes to store the value. This way, no node will become disproportionately overused in comparison to the others.

2) Slave Storage: Slave storage has two tables it uses for data storage.

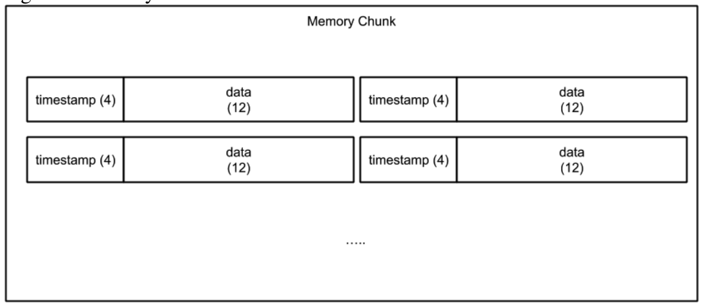
a) Memory Blocks: On each slave node a chunk of memory is allocated to store data. This chunk is further broken into blocks, to facilitate quick access and stop fragmentation from occurring, seen in Figure 3. Each block has two components: the header and the payload. The payload stores the actual value or a piece of the value. In the header one of two things is stored, either a timestamp for when the data was inserted, or an address to the next block where a piece of the

Fig. 2. Free memory table on master node



value is stored. The time stamp will only be present in the header if it is the last block that is used to store the value, otherwise an address will be present. So when a value comes in to be stored, if it is too large to be stored in a block, the first `block_size` bytes are stored, and the header is set to the next free block. Then a recursion happens on the value, minus the first `block_size` bytes. Eventually there will be a free block that can hold the rest of the value. In that case, the header is set to the current time. So a value stored in this database is not a continuous piece of memory, it is a fragmented linked list of pieces of the value. The next free block is chosen from the free block table, described below. Notice that the slaves do not store the key or the length of the data for the value being stored in the block. Thus if head block is lost, the value would be lost as well.

Fig. 3. Memory chunk with blocks on slave nodes



b) Free Block Table: The free block table is a linked list, where each node holds the number of a free block. This linked list acts like a stack. When a free block is needed, the first node is popped off and returned. When a block is freed, when a delete happens, a new node is created for the block and added to the front of the list. With this structure, data fragmentation is controlled and maintained. Without it, to fix data fragmentation, reallocation and data movement would have to occur. This would cause the master storage to be modified as well as the slave, potentially leading to long wait times while the database realigned itself.

B. Distribution

SquirrelDB uses a master/slave distributed architecture. To facilitate communication between the master and slaves, OpenMPI was used. OpenMPI is a widely used library for starting up and running distributed processes. A single command is given, which starts the process on multiple nodes. Each node is given an id, which are used as the destination or source when passing or receiving messages.

Distributed systems must be partition tolerant but then have the choice of being available or consistent. As our design is a key/value in memory store it made more sense to design an available system. We sought to be as fast as possible, however we also wanted to be able to recover from a slave or even a master node crash. This is why we periodically flush to disk.

1) Fault Tolerance: For the system to be fault tolerant, it must recognize that a node has gone down, and react to that. SquirrelDB uses heartbeats to check if nodes are alive or dead. The master node runs a loop. At the beginning of the loop it sets up a timer for t amount of seconds. Either the master will receive a request from the client in those t seconds or not. If the master receives a request in those t seconds, it will cancel the timer, send out a heartbeat, get responses from all active slaves and then process the request for the client. After processing the request the loop will start over again. If the no request comes in, then the timer will go off, the master will send out a heartbeat, get responses and schedule another timer for t seconds. The slaves also run a loop. At the beginning of the loop they set up a timer for s amount of seconds. If they do not hear from the master within those s seconds, they assume the master has gone down, and begin the recovery process described below. Using heartbeats, the master or the slaves can detect if the other has gone down. The low level details of detecting if a nodes has gone down first requires an understanding of how OpenMPI handles sends and receives.

When sending or receiving with OpenMPI the user specifies a buffer, the size of the buffer, the type of the buffer, the destination/source id, a tag (ignored for this project), a communication world (basically a collection of all the nodes in the system), and a request object. These sends/receives are non-blocking, meaning that code execution will continue, even if the message has not completed sending or been received. The request object can be used to test if the message has been sent or received. OpenMPI provides a function, which takes the request and returns a flag, true for success and false for failure depending on whether or not the operation has completed yet. Using these functions, a system has been created to detect if a node is down.

First the user makes a send/receive request. The non-blocking send/receive OpenMPI function is called, and the request object is created. Then the request object along with the destination/source id is stored in a linked list called the pending request list. This list can be added to multiple times from different sends/receives. The user then calls a function which checks all the pending request to see if they have completed. This function starts a timer, then checks each request, if a request has completed, it is removed from the pending request table. If the request is not completed, the function moves on to the next request, in the hopes that it can remove pending requests later in the list. Eventually either the list will be empty, at which point the timer will be canceled and the function will return knowing that every request was fulfilled, or the timer will go off. If the timer goes off, the pending request list is traversed, and each node present in the list is marked as inactive. No further communication with that node is permitted, and the system acts as if that node never existed. Using these methods, the system remains fault tolerant.

C. Crash Recovery

For SquirrelDB to be fault tolerant it must be able to recover from node loses. There are two types of node loses that SquirrelDB must handle: slave node loss and master node loss.

1) Slave Node Crash: When a slave node goes down, the master will recognize that it has gone down using the heartbeat or sending/receiving system described above. When a slave node is recognized as having gone down, the master marks that node as inactive, and never tries to communicate with it again. No other measures are taken when a slave node goes down. Because the data was being replicated there should still exist a copy of the data on at least one node. However, if that second node goes down, the data could be lost. Such behavior is acceptable for us because SquirrelDB is an available, not consistent, system. We note that the likelihood of both slave nodes holding the same data crashing is quite low.

2) Master Node Crash: When the master node goes down, the slave nodes will recognize that it has gone down using the heartbeat timer. At that point the slaves will identify the next lowest active id number, and declare that node the new master. For example if the slaves are numbered 0 to 4, with 0 as the master, when the master goes down, 1 becomes the new master. After selecting the new master node, all the slaves exit their current loop, and re-instantiate, just like if the database was starting up for the first time. Each node will try and read the old flushed files and re-instantiate or start empty if the files do not exist. Then the system continues as normal. This means that all transactions after the last flush to disk will be lost, but the system will continue to work without the user needing to restart the whole database.

3) Flushing: Data on each node is flushed to disk, in order to provide the system with a way of recovering when the master crashes. All flushing is from main memory to disk. Each slave node flushes its data blocks and free memory table. This way when a rollback occurs, these files can be read, and the system can be re-instantiated. The master node flushes the index table to disk, so that it can rollback as well. The available memory table does not need to be flushed, because the slave nodes tell the master how much memory they have when the system starts up. The master sends the its index table to each slave node, so that if the master goes down, and a new master is selected, the new master can use the old masters index table to run the system.

D. User Connection

SquirrelDB uses a server/client model to allow a user to connect to the database. Only one user can connect at a time, and only one operation can be performed at a time. A operation must be completed in full before the next operation can be given by the user. The server is running on the master node and is set up to bind to any connection. If a user closes their connection then the database waits until someone else connects and gives it instructions.

E. API

SquirrelDB, being a noSQL database and a key/value store, only has a few user facing operations. Those operations follow the CRUD model: create - put, retrieve - get, update - update and delete - delete.

There is one thing common across all these operations. A key is passed to all of them. The key must be unique over the database. The key can be at most 16 bytes long. The last byte of the key must be '\0'. A value is passed to put and update. The value is whatever the user wants to store, as long as it will fit in the database. A response buffer is passed to get. It must be large enough to store the data that is returned. If the buffer is set to NULL then the buffer is calloc'd to be the correct size.

1) Put: A put operation requires three variables be passed to the database. First the key for the value. Second is the value. The last variable passed to the database is the length of the data. This is necessary for knowing how much space is required to store the value.

Given the key, the master node performs a hash and finds the index where the key will be stored in the index table. The master then looks at the available memory table to find the nodes that have the most memory. The master then sends the value to those nodes for storage. The slave nodes designated to store the value receives the value and its length, and starts popping free blocks of the free block linked list and storing pieces of the value in them. Once the value is completely stored, the index of the first block storing the data is sent back to the master. The master stores this index with the slave node id in the index table. Finally the master sends back a success message to the user. If a failure occurs anywhere, a failure notice is sent back to the user.

2) Get: A get operation requires only the key be passed to the database.

Using the key the master node locates the index in the index table which has the information about where the value is stored. The master then requests the value from each node given the starting memory location on that node. The slave node starts reading from memory blocks, until it reaches the memory size of the value. The slave node then sends this value back to the master node, along with the timestamp for the value. The master node receives the values from the slave nodes, checks the timestamps, and uses the value with the most recent timestamp. The master sends a success message, the size of the value, and the value back to the user. If a failure occurs anywhere, a failure notice is sent back to the user.

3) Update: An update operation requires three variables be passed to the database. First is the key. Second is the new value to store, and third is length of the new value. To address load balancing, an update is treated as a delete and then put.

Using the key the master node locates the current location of the value. The master then performs a delete on the old value. After the old value is deleted, the master node performs a put on the new value for the key. This may cause the new value to reside on different slave nodes, compared to the old value. This is done because the size of the database may have changed, and the size of the value may have changed, thus storing the new value on the old nodes may not be the most efficient place for the new value to be stored. The master sends a success message back to the user when the update is complete. If a failure occurs anywhere, a failure notice is sent back to the user.

4) Delete: A delete operation requires only the key be passed to the database.

Using the key, the master node locates the index in the index table which has the information about where the value is stored. The master then sends a delete request to the slave node, giving the starting memory location. The slave node starts freeing blocks and adding them to the free memory linked list. When the last block is freed the slave node sends back success. The master receives success messages back from the slave nodes and then sends success to the client. If a failure occurs anywhere, a failure notice is sent back to the user.

V. Testing and Validation

To test and validate our system we performed the following tests.

A. Test 1: Correct Functionality Tests

For each test the below master node printed out which node had been given the value, or which node the value was being retrieved from. Timestamps for retrieved data were printed so that one could verify the most recent data was returned. The non empty master index table entries were printed and the available memory table was printed as well. The following tests were run (the query used is listed as well):

- Put test - put(key, value)
- Get test - get(key)
- Update to larger value test - update(key, new much much longer value)
- Get updated data test - get(key)
- Update to smaller value test - update(key, smaller value)
- Get updated data test - get(key)
- Delete test - delete(key)
- Put existing key test - put(key, value), then put(key, new value)
- Update non existent key - update(unknown key, value)
- Delete non existent key - delete(unknown key, value)
- Clearing Database Step - Delete(key, value)
- Get non existent data - get(key)
- key too long test - put(This key is really really far too long, test)
- Too much data test - put(long data, a * 500000001)

B. Test 2: Slave Node Crash Recovery

The same things were printed as above. Also printed was a message when the master node realized that one of the slave nodes went down. The following operations were run:

- put(key_1, value)
- put(key_2, value)
- shutdown node 1
- get(key_1)
- get(key_2)

C. Test 3: Master Node Crash Recovery

Beforehand we ran the database and called put(key_1, value) through put(key_50 value). We ran the database long enough for the logs and master data tables to be flushed to all slave nodes. We started up the database and restored to that save position. Then we ran commands (shown below). We then stopped the master node's termination before the data could be flushed to the slave nodes. After the master termination, one of the slave nodes was chosen as new master and restored to last saved point. We showed that data between flushes had been lost, but all previous data was still there by performing get(key_1) through get(key_52) requests.

- put(key_51, value)
- put(key_52, value)
- update(key_34, new value)
- update(key_11, this is the new value for key_11)
- delete(key_16)
- get(key_1) through
- get(key_52)

D. Test 4: Shell Test

The database was started up in shell mode. The user gave a file with commands, one per line, for the shell to read in and perform. All commands were successfully executed.

VI. Performance Evaluation

To test the performance we will be altering a series of variables, such as memory on each node, size of blocks used to store data, type of data being stored, and interval of disk flush. For all tests debug printing will be turned off, as that feature slows down run times. As our database does not support non-sequential query requests we cannot perform a test where we up the amount of requests per second to see the impacted performance of the database. When performing the tests we will fill the database with some data, to better simulate average running times. For each of our test we will get the average times for get, put, update, and delete. We note here that when we tried to run the small and medium block size tests for the ∼500MB per slave node, the master couldn't physically store the index table. This is the drawback of having our implementation be in memory. Those tests as a result have been omitted.

The first round will test how the amount of storage allotted to each slave node affects its performance. For each of the tests that follow we will run them with a small, ∼5MB per slave node, and large, ∼500MB per slave node, amount of memory allocated.

Secondly, we will test block size. Data is stored on the slave nodes in blocks, so that everything is aligned. As such we will also be altering the size of the blocks to be small, 2 bytes of data, medium, 8 bytes of data, and large, 16 bytes of data. Clearly each affects the amount of data that can be stored, but this can be calculated, so we wont be running tests to see how much data we can store. Instead we will be testing how this affects the put, get, update, and delete times of different size data.

Thirdly we will see how the size of the data affects the speed of the system. We will be timing put, get, update and delete for data comprised of all numeric values, very long strings, ∼500 characters, medium length strings, ∼50 characters, and a mix of all three. Numeric values are produced using the `rand()` function. Strings with random characters, using `rand()` and some manipulation to get a random character, were used during testing.

Our last test will mess with the rate at which data is flushed to disk. Currently there is a counter tied to number of heartbeats sent and when it reaches a predefined number the data is flushed. Heartbeats are sent every time an operation is performed or every 2 seconds when a timer goes off. We will alter that number so that flushes happen often, every 100 operations or ∼200 seconds if no operations are performed, or rarely, every 7500 operations or ∼250 minutes if no operations are performed.

Combining all our tests would give us a grand total of 48 tests. However as we mentioned before we were not able to run some of them. We instead have results from only 32 tests, 24 for ∼5MB per slave and 8 for ∼500MB per slave. The results of those tests are detailed below.

A. Query Function Results

Looking at all the tests one will note that the flush often configuration affected the running time of all the operations, not just the one on which it performed the flush. This performance difference was consistent across all tests and was ∼500 microseconds for put, get, and delete and ∼800 microseconds for update. We don't know where this delay is coming from or why it is greater for update. The only difference between flush rarely and flush often tests is that the heartbeat counter tied to flushing has to reach and be compared to a higher number before passing an if statement. There is no difference between how update runs on flush often and flush rarely test.

Timing results for the get operation across all tests run are shown in Figure 4. The get operation across all tests took ∼.08 seconds. No change in configuration, at least none for the tests that were run, produced a drastically quicker or slower get time. One will note that the get operation takes roughly twice as long as any of the other operations. This is because during the get operation the master waits for both slave nodes holding the data to respond with the value. It does this sequentially. Even though update appears to run get, the master only checks the index table to see if the data is stored in the database. The master doesn't actually ask the slave nodes for the data. Put and delete run quicker because the slave nodes only return success or failure, which is read by the master much quicker than a stored value.

Timing results for the put operation across all tests run are shown in Figure 5. The put operation across all tests took ∼.04 seconds. The put operation spends most of its time reading data from the client and then sending that data to the slave nodes for storage. Sending data to multiple slave nodes can be done at the same time, unlike for the get operation, which has to read data from multiple slave nodes. This is why it is quicker than the get operation.

Timing results for the update operation across all tests run are shown in Figure 6. The update operation across all tests took ∼.04 seconds. The update operation first checks the index table to see if the key passed in actually has been seen before. If it was then a delete is run, followed by a put. Since there is barely any time difference between the running times of update and delete we have concluded that the majority of the time spent performing an operation is during communication with the client, who sends the database data and receives data back.

Timing results for the delete operation across all tests run are shown in Figure 7. The delete operation across all tests took ∼.04 seconds. The delete operation first checks the index table to see if the key passed in actually has been seen before. If it was then a delete is run. It appears that this operation should be quicker than the update operation but it is not significantly quicker. We think this is because when an operation is received the master first performs sends out a heartbeat message and then has to wait for every slave to respond. This was done so that all slave nodes know that the master is still alive, but slows everything down as a result.

B. Flush Impact

The affect of flushing can be seen in Figure 8 and Figure 9. Flushing is done after a predetermined number of heartbeats occur. Since a heartbeat is sent whenever a query is performed this means that the query where the flush is performed is significantly affected. The figures show the difference in running a delete when flushing versus not flushing. This pattern was similar for the other query operations.

In Figure 8 one will notice that the smaller the `block_size` the larger the flush time. This is because when flushing, each block is flushed sequentially, thus a smaller `block_size` causes more calls to `write()`. Also a smaller `block_size` increases the number of entries in the index table, meaning more data is actually flushed, which slows down performance as well.

The most drastic affects are seen in Figure 9. It might be hard to see, but all configurations do have nonzero values in the graph. The flush often values are so much larger than the flush rarely values, which is why the flush rarely values appear to be 0.

C. Overall Time Comparison

Figure 10 shows how the timing for each operation relate to each other within the same database configuration. It is clear that put, update, and delete all take approximately the same time, while get takes twice as long as the others.

VII. Future work

A. Accommodating for Scale

Right now, SquirrelDB utilizes one master and four slave nodes. In a larger system which takes advantage of more virtual machines, it may not be feasible to only have one master node. Adding more slaves to the same master node would create a huge bottleneck in such a system.

One solution would promote multiple virtual machines to master nodes and have the master nodes communicate with each other, similarly to how other distributed databases work. Each master node would have their own set of slaves to account for. A perfect ratio would have to be found in order to balance the amount of workload, possibly similar to the existing 1:4 master-to-slave ratio SquirrelDB supports today. The master nodes would have to share backups so that when a master node dies, another master can pick up the slaves that were orphaned.

Another solution would be the creation of a tier of masters. One node would control all queries, but the management of slave nodes would be deferred to a middle-management master node. The top-tier master node would handle all middle-tier nodes similarly to how a master manages its slaves in the current implementation of SquirrelDB. The top-tier node would have to be backed-up on its middle-tier nodes. If the top-tier node fails, a middle-tier node would have to assume the its position.

B. Version Control

Currently, SquirrelDB does backups all on one version. What a user may find useful is the implementation of multiple versions of backups. This would allow a user to backup important states before a potentially dangerous change. With the current scheme of SquirrelDB, such a transition would be simply done with the naming and specified loading of the files stored to disk.

VIII. Conclusion

We have presented the architectural design for a relational database that stores all data in memory. By storing data in memory, our system exhibits increased performance in data retrieval while limiting the storage size. The database solves this by using a series of slave nodes communicating solely with a single prominent master node. The master node stores the metadata composed of both a look-up table holding the locations of each piece of data and the free table which holds the available memory associated with each node. Each slave node is broken into memory blocks. Each memory block contains a header and data where the header either holds a time stamp value or points to the next block that continues the sequence of data. Message passing is handled using Open MPI. Images of the nodes are saved to disk to create a backup and increase fault tolerance.

We used a series of tests to validate each aspect of our design. The get operation was the slowest. This was on account of the lag associated with passing the value back to the client.

The other three operations (put, delete, and update) all took approximately the same amount of time. We showed that the variance between large and small loads was insignificant, demonstrating successful scalability. Flush frequency plays a large role as flushing more often slows down the system considerably. Overall, we have shown that the outlined database management system produces high performance results while maintaining data availability and partition tolerance.

References

[1] Open mpi: Open source high performance computing. http://www.open-mpi.org/. Accessed: 2014-09-28.

[2] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: a holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[3] SD Kuznetsov and AV Poskonin. Nosql data management systems. *Programming and Computer Software*, 40(6):323–332, 2014.

[4] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[5] What is memcached? http://memcached.org/. Accessed: 2014-10-14.

[6] repcached. http://repcached.lab.klab.org/. Accessed: 2014-10-14.

[7] Redis. http://redis.io/. Accessed: 2014-10-14.

[8] Voltdb, the fastest in-memory operational database. http://voltdb.com/. Accessed: 2014-10-14.

[9] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[10] Pivotal gemfire. http://www.pivotal.io/big-data/pivotal-gemfiredetails. Accessed: 2014-10-14.

[11] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[12] squirreldb. https://github.com/CSC560/Mothership/tree/squirrelDB. Accessed: 2014-9-28.

Fig. 4.   Get timing results

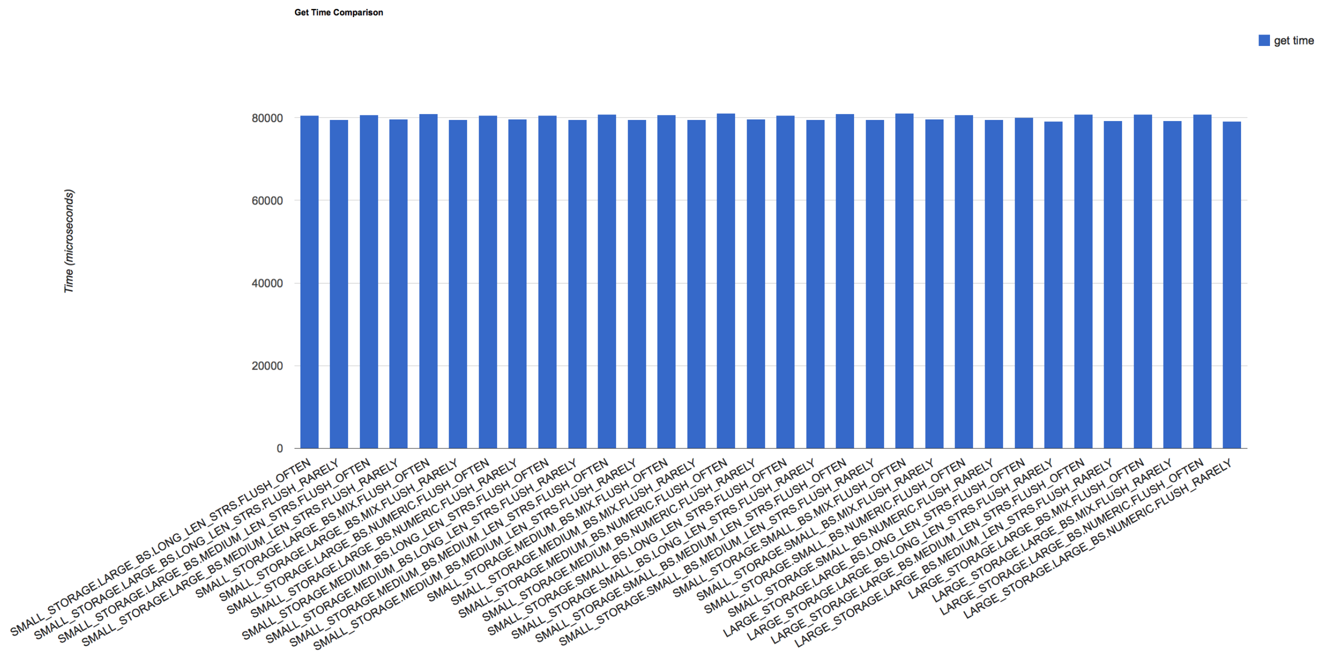


Fig. 5. Put timing results

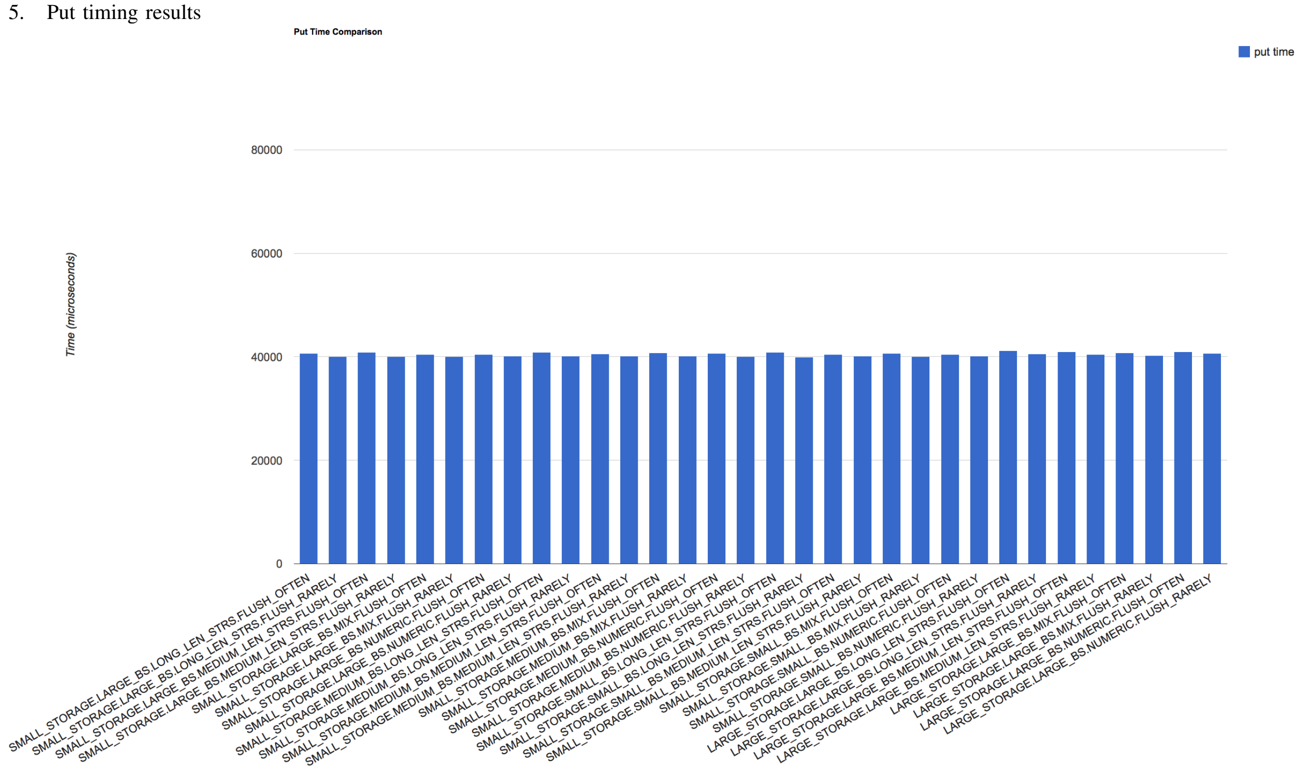


Fig. 6. Update timing results

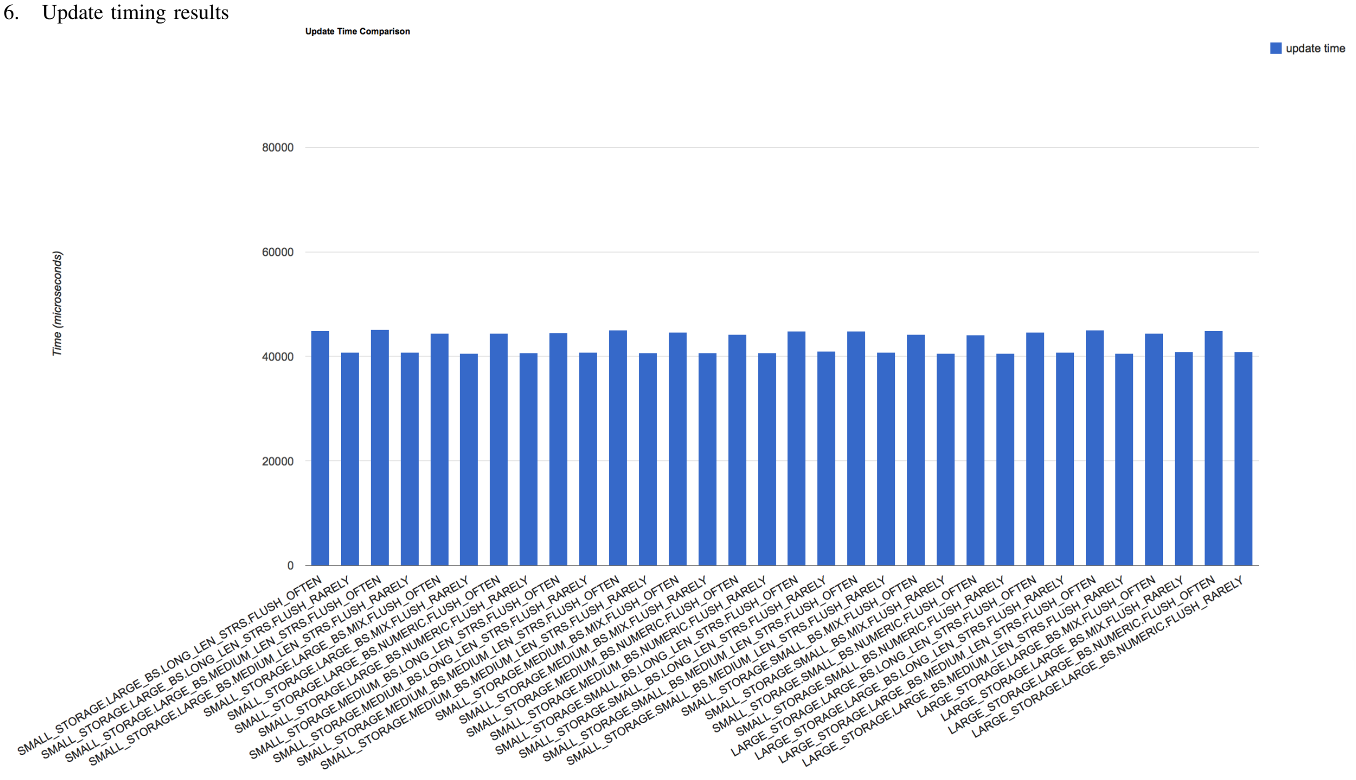


Fig. 7. Delete timing results

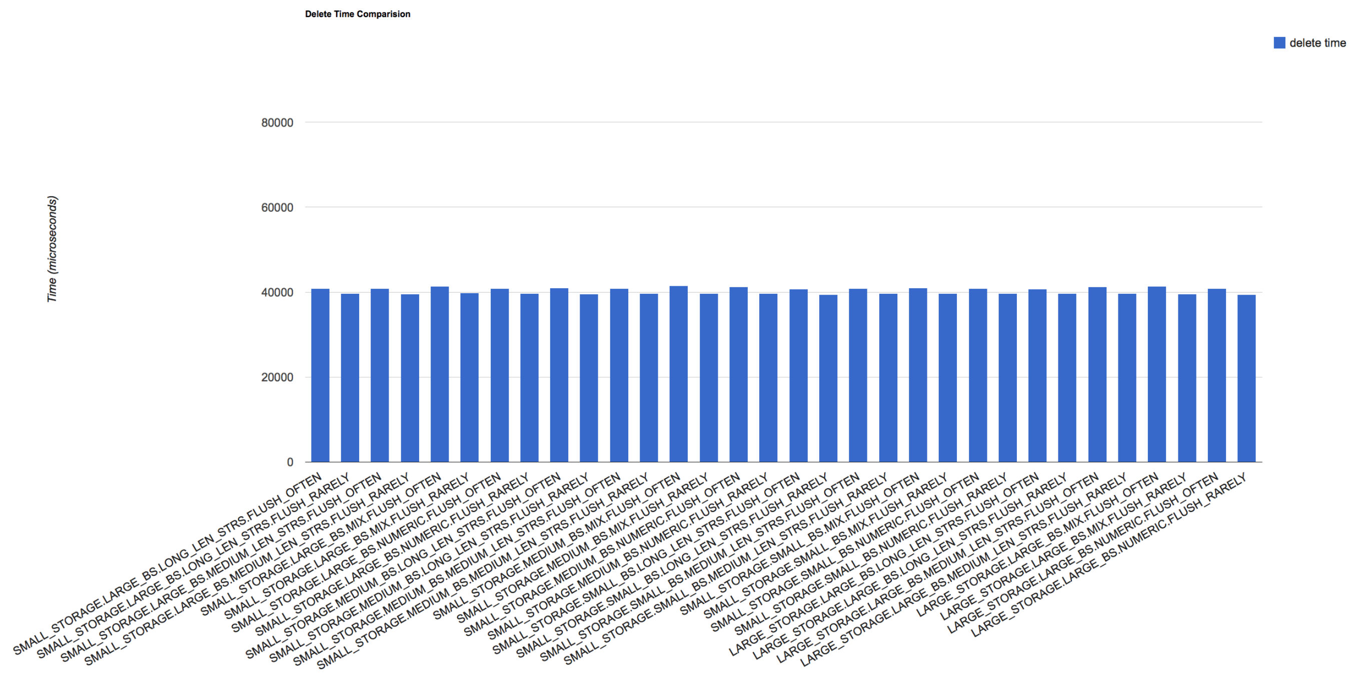


Fig. 8. Small Storage flush timing results

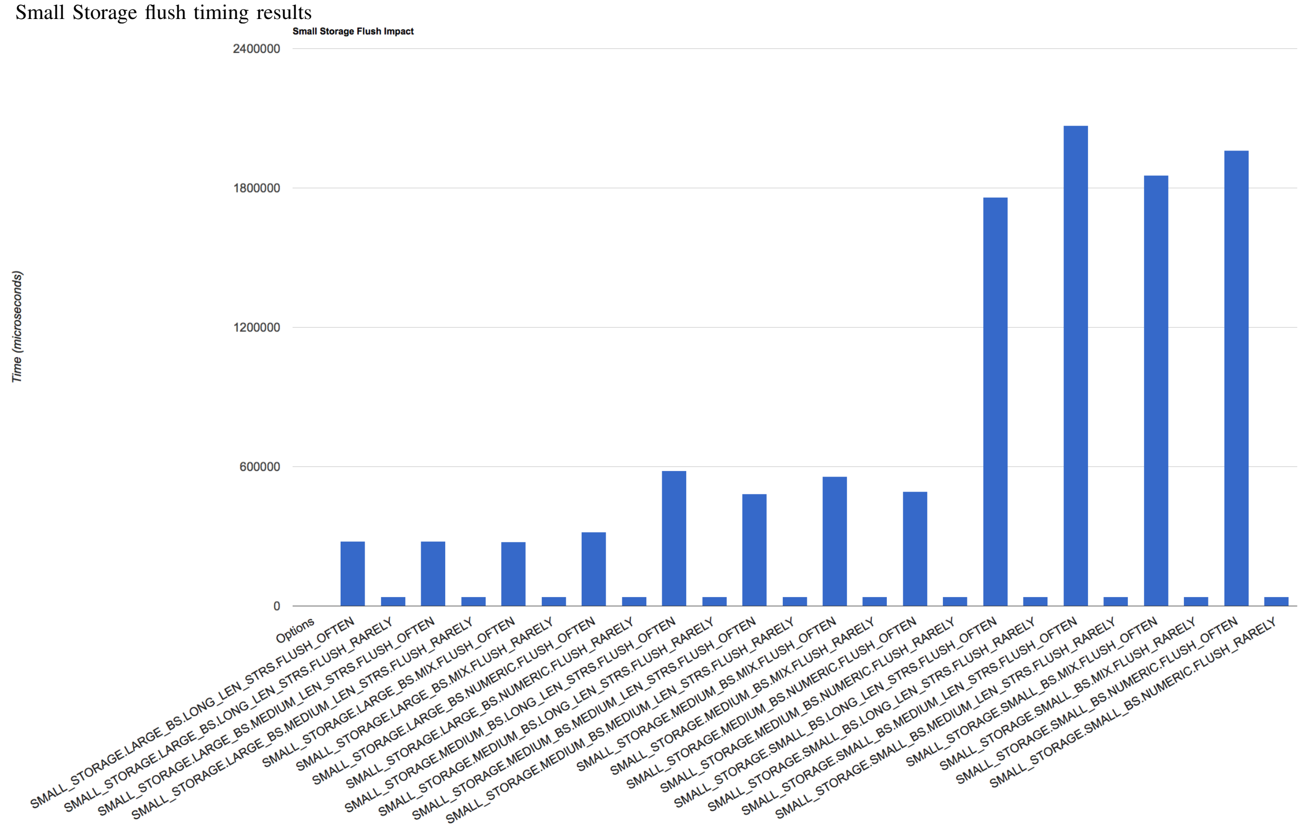


Fig. 9. Large Storage timing results

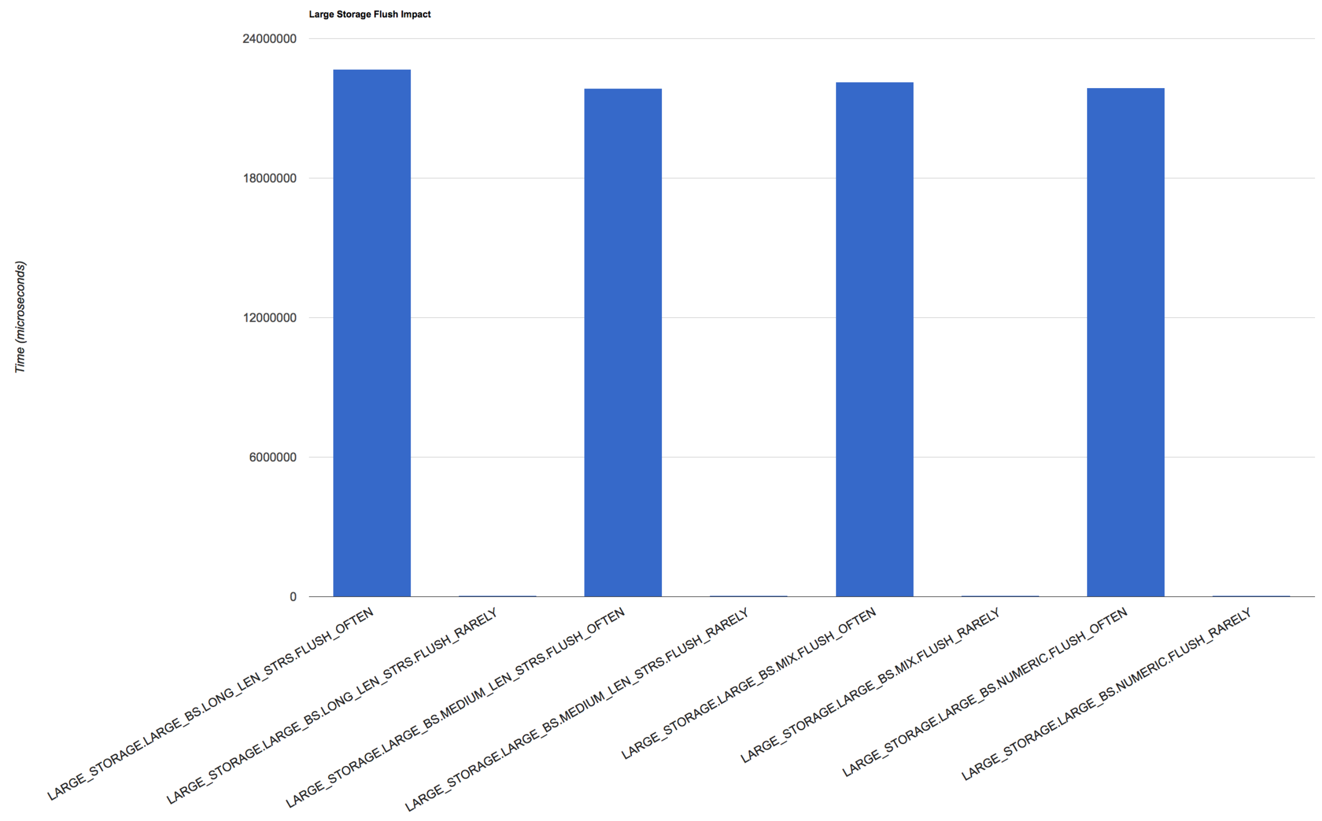


Fig. 10. Overall Timing Comparison

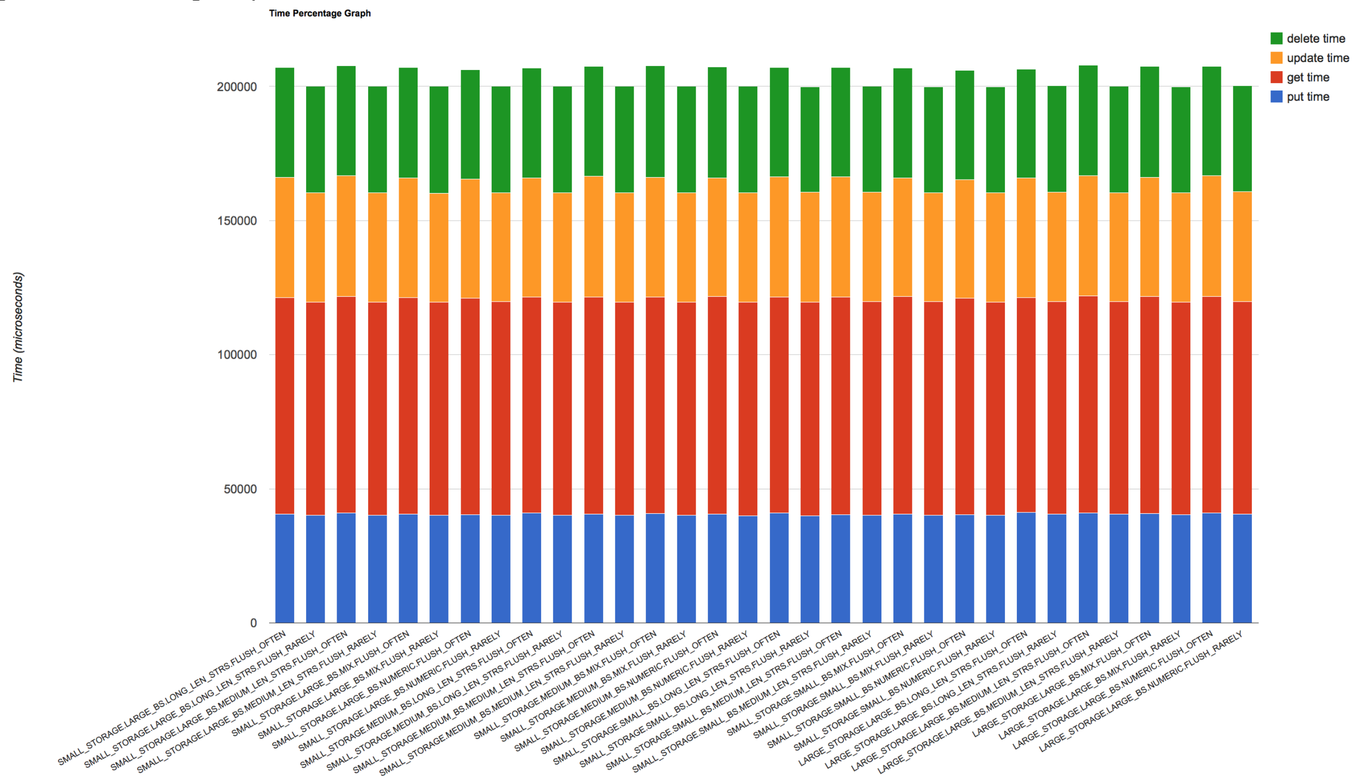
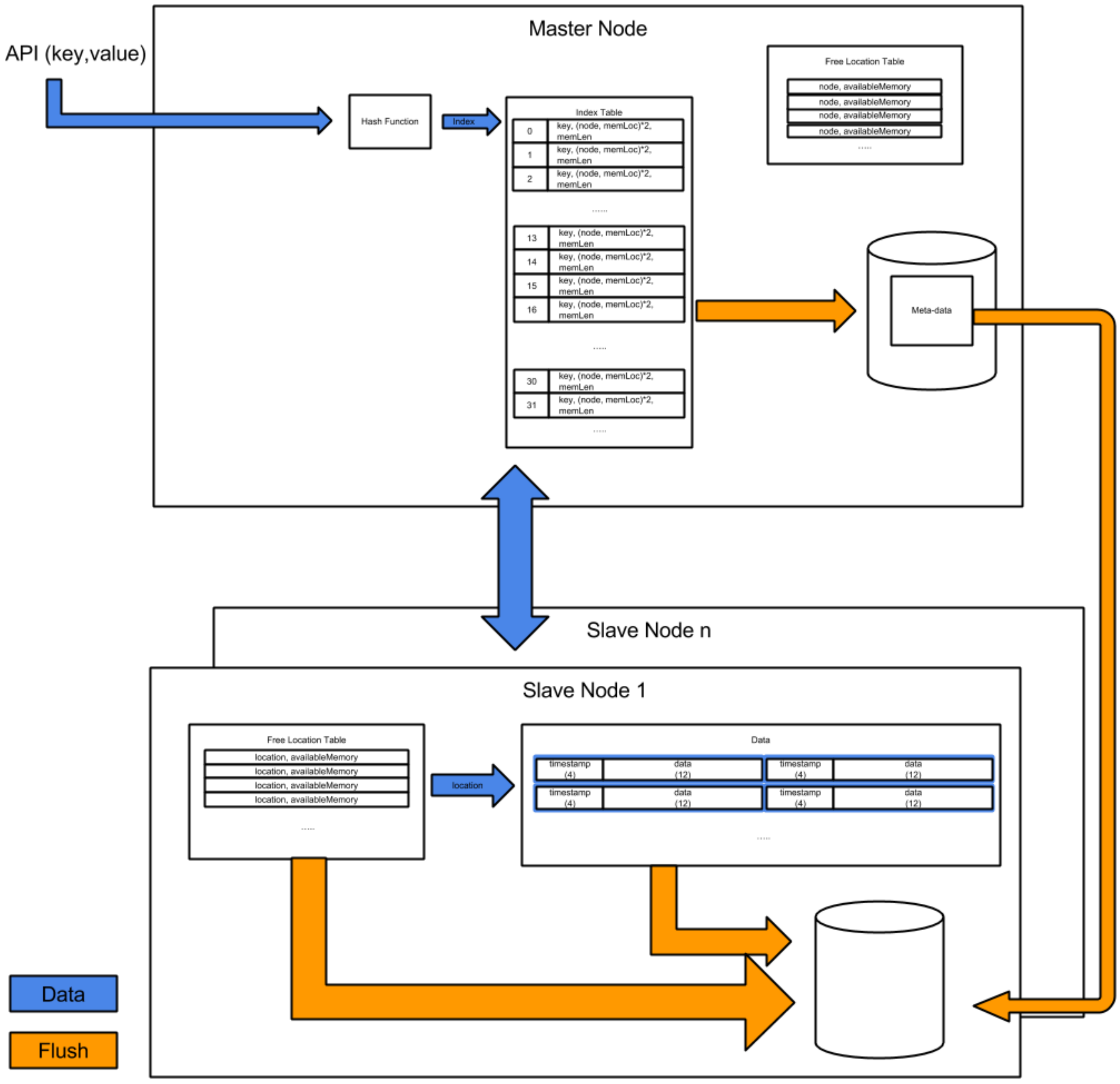


Fig. 11. SquirrelDB architecture


Master Node
API (key,value)
Hash Function
Index
Index Table
0 key, (node, memLoc)*2, memLen
1 key, (node, memLoc)*2, memLen
2 key, (node, memLoc)*2, memLen
13 key, (node, memLoc)*2, memLen
14 key, (node, memLoc)*2, memLen
15 key, (node, memLoc)*2, memLen
16 key, (node, memLoc)*2, memLen
30 key, (node, memLoc)*2, memLen
31 key, (node, memLoc)*2, memLen
Free Location Table
node, availableMemory
Meta-data
Slave Node n
Slave Node 1
Free Location Table
location, availableMemory
location
Data
timestamp (4)
data (12)
Data
Flush