# A Distributed Key-Value Store using Ceph

Eleanor Cawthon
Summer 2012

## Introduction

The rise of distributed computing has given new importance to the question of how to effectively divide large sets of data. One popular approach is the B-tree. The B-tree and variations on it have been used in a variety of storage systems, distributed and not, since Bayer and McCreight first introduced the structure in 1972[i]. The basic structure of a B-tree involves nodes with a number of values between $k$ and $2k$, where $k$ is a constant. This outline allows for searches, insertions, and deletions in logarithmic time, with worse case height much smaller than that of a binary tree. Other approaches, such as Google's BigTable[ii], use fixed height structures instead of true trees. The library presented here uses some elements of a B-tree, including the $k$-value based node sizes, but uses a fixed height of two levels instead of a true tree structure.

The features of a single machine that are used to provide safe access for multiple writing threads – namely shared memory and locks – cannot be assumed in a distributed system. One intuitive solution to this problem is to replicate these features using a lock service or a single controller. These solutions introduce significant performance bottlenecks. In addition, where a single master is used instead of a Paxos cluster, the master constitutes a single point of failure for the distributed system.

Ceph is a massively scalable distributed object store with no single point of failure that avoids these bottlenecks by making full use of the resources of the machines in a cluster. Ceph uses a Paxos cluster to detect the deaths of object storage devices (OSDs) and synchronize maps of which objects are on which OSDs, but these monitors do not provide a locking service, thus avoiding a bottleneck over lock contention. Ceph provides safe access to multiple writers with locks at the OSD level, or with various lock-free algorithms[iii].

In addition to a name and a data field, Ceph objects contain a map of xattrs and a map of filesystem-independent, xattr-like key-value pairs called an object map, or omap. When writing to the data field, Ceph objects experience performance degradation when used to write small data. The omaps are far less expensive to access, so it is more efficient to use them than the data field to store small entries. However, a bottleneck arises when attempting to scale this. Unlike Ceph objects, omap entries do not split and distribute themselves across the OSDs to ensure maximum efficiency, and there is an upper limit to how much data can be stored in the omap of a single Ceph object.

This distributed key-value store uses omaps to store entries, taking advantage of the assumption that users' keys and values are small. The B-tree-based design ensures that the set of key-value pairs is distributed among many Ceph objects, which enables it to take advantage of Ceph's efficient handling of objects.

## Tools Ceph Provides

The Ceph object store provides several features that, from the perspective of someone designing an extension built on librados, mimic functions available on a single processor. Librados is a wrapper for RADOS, the underlying object storage device[iv]. The most important librados features this library uses are as follows:

- Transactional reads and writes on a single object – an arbitrary number of read or write operations can be combined into a single, atomic write or read operation such that if any of the sub-operations fail, the entire combined set fails. Reads and writes, however, cannot be combined with each other into a single atomic operation, and these operations are only transactional when performed on a single object. Librados groups all steps in a transaction into an ObjectWriteOperation or ObjectReadOperation.

- Several test-and-set operations, meaning that, as part of a write operation, the transaction can check if a condition is true and only proceed with the write if that condition is true. For example, librados provides methods to assert omap and xattr entries as part of a write.
- In addition to the built-in test-and-set methods, developers can write their own logic to execute on the OSD as part of a transaction. For example, the method for inserting a key only inserts the key if inserting the key would not bring the object's size above 2*k*.
- RADOS guarantees that all objects will be distributed and replicated efficiently. Code using librados does not make any assumptions about what is stored on any particular OSD.

# The Design

The data structure is essentially a two-level B-tree that uses Ceph objects as nodes. There is an index object that maps key ranges to other objects where keys and values are located. The index object's omap contains the highest key stored in an object as keys, and a data structure that includes the name of the Ceph object where that range of entries can be found as values:

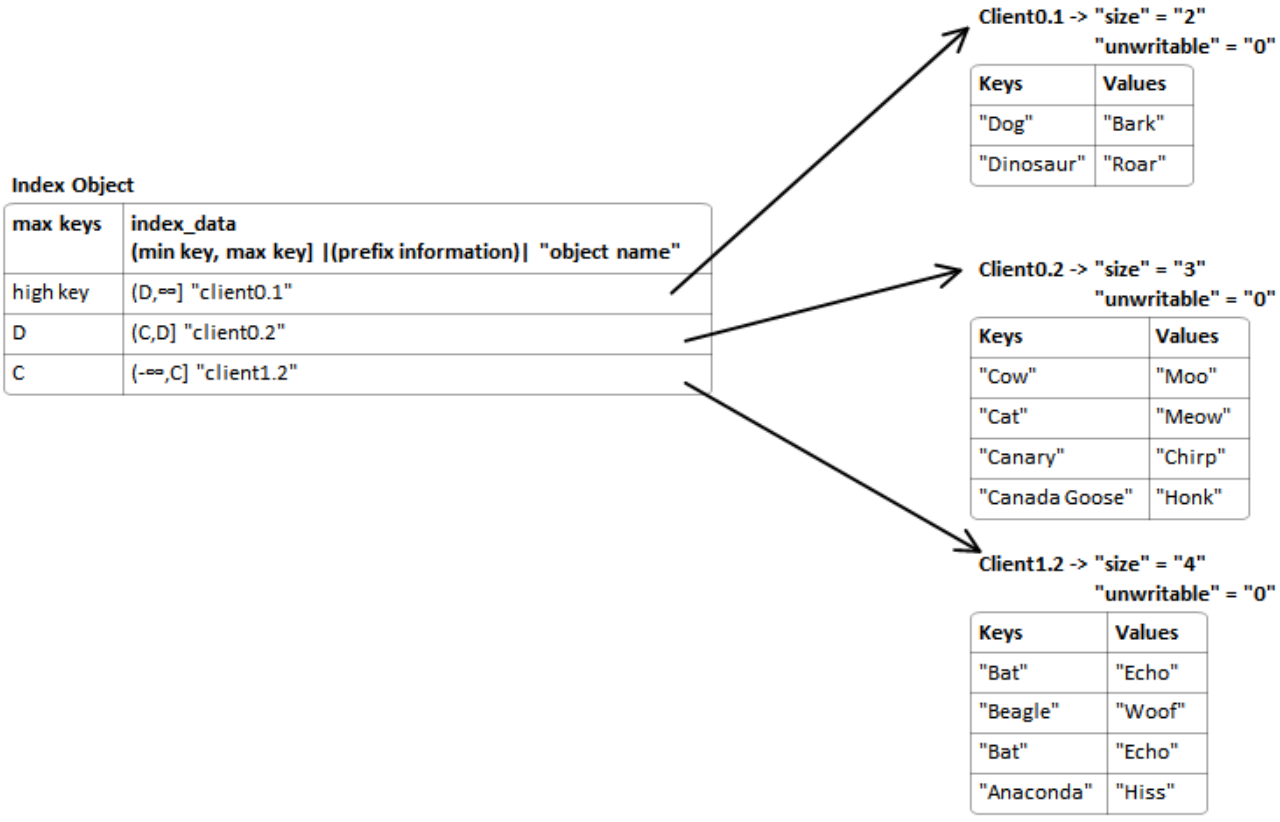



Figure 1. an index and three nodes, where $k = 2$

## Read

The algorithm for reads is as follows:

1. Read the index to find the entry where the key would be
2. Read the object pointed to in the index to find the value associated with that key
3. Return either an error code or a bufferlist with the value.

## Write

In normal cases, where a node does not need to be split or merged, the basic algorithm for writes (inserts, updates, and deletes) is as follows:

1. Read the index to find the entry where the key belongs
2. Attempt to write to the object pointed to by the index. Fail if:
   - The "unwritable" flag, stored in an xattr, is set on the object,
   - The object does not exist (i.e., has been deleted by another client),
   - The object already contains the key (in the case of an exclusive insert),
   - The object does not contain the key (in the case of a remove).
   - The object has exactly $2k$ entries (in the case of an insert) or exactly $k$ entries (in the case of a remove),
3. In the first and second failure cases, the method re-reads the index and tries again. In the third and fourth cases, the method returns an error code. In the final case, the method calls the appropriate balancing method (split or rebalance) and then retries.

In an average case write, where no splitting or rebalancing is needed, the operation is atomic because only one ObjectWriteOperation is performed, and each ObjectWriteOperation is guaranteed to be atomic.

## Split

Splits and rebalances are the most complicated operations. Splitting is the simpler of the two, because it only modifies one existing object. When an insert attempts to insert an entry into a node that already has 2k entries, it calls split. Split takes the index entry for the object as an argument, and performs the following steps:

1. Read the object to get its xattr and omap information
   - if the "unwritable" xattr is set, or if the size of the object has changed and it no longer needs to be split, abort.
2. Create two new maps, each with half of the contents of the omap of the object being split.
3. Mark the entry of the index that corresponds to this object with information about the split being performed. The prefix that gets added to the value in the index is enough for another client to roll back or roll forward the operation.
   - If there is already a prefix for another split or merge operation, do not change the index and abort the split. This is checked atomically – the index is only marked if no other client has already added a prefix.
4. Set the "unwritable" xattr on the object to true, asserting that the version number has not changed.
   - If the version number has changed, the information read from the object is no longer valid, so the split aborts.
5. Create the first new object, containing the lower half of the entries from the original object.
   - The object's name is globally unique – it is the name of the client followed by a number that is incremented every time the client attempts to create an object.
6. Create the second new object, containing the higher half of the entries from the original object.
7. Delete the old object
8. Update the index object, adding two new entries for the two new objects and removing the entry for the old object. This is done atomically with a single ObjectWriteOperation.

## *Explanation*

The table below illustrates how interference between multiple writers is resolved:

| Step number | What happens if an insert interrupts after this step | What happens if a remove interrupts after this step | What the clean-up method does if the client dies after this step |
|---|---|---|---|
| 1 (Reading the old object) | Both clients will be attempting to split, and one will discover the other has already added a prefix, causing it to abort | This client will add a prefix, but marking the object will fail because the version number has changed. After this fails, the splitting client will reset the index and abort. The insert that called split will start over and retry until it successfully inserts the key into the new appropriate object. | Nothing – no changes have been made. |
| 2 (Creating new maps) | This is the same as 1, since creating new maps does not affect the cluster. | This is the same as 1, since creating new maps does not affect the cluster. | " |
| 3 (Marking the index) | The other insert will discover that the object is full and attempt to split. The split will fail because there is already a prefix. | The remove will succeed. The splitting client will roll back the change to the index upon discovering the version number change. | Resets the index to its original state |
| 4 (Marking the old object unwritable) | The other insert will fail and retry because “unwritable” is true | The remove will fail and retry because “unwritable” is true | Marks the old object writable and then restores the index |
| 5 (Create the first new object) | " | " | 1. Discovers that the second new object does not exist and does not need to be deleted,<br>2. Marks the first new object unwritable,<br>3. Marks the original object writable,<br>4. Deletes the first new object,<br>5. Restores the index. |

| | | | |
|---|---|---|---|
| 6 (Create the second new object) | " | " | 1. Marks the second new object unwritable,<br>2. Marks the first new object unwritable<br>3. Marks the original object writable<br>4. Removes the second new object<br>5. Removes the first new object<br>6. Restores the index |
| 7 (Delete the old object) | The insert will fail with -ENOENT, causing it to check the time-stamp on the prefix and retry until it succeeds or until the splitting client times out. | The remove will respond the same way that insert does, at left. | Either the client that called cleanup passed in -ENOENT, which will cause it to immediately roll forward, or cleanup gets -ENOENT on marking the original object writable (step 3, above).<br>If cleanup had made changes, it rolls them back:<br>1. Marks the first object writable<br>2. Marks the second object writable<br>In either case, cleanup finishes by inserting the new entries into the index. |

## Rebalance

The rebalance method takes two nodes and distributes their omaps so that the object on which it is called has at least $k$+1 entries. Ensuring $k$+1 entries instead of exactly $k$ allows the remove operation to remove a key without creating the need for another rebalance. It does this either by moving entries from the larger object to the smaller object, or by merging the two objects into one object. When removes read the index to check which object contains the key, it reads two index entries: the one containing the key and the one after it. When remove attempts to remove a key from an object with only $k$ entries, the remove fails and calls rebalance. Rebalance executes the following steps:

1. Checks to see if the index data for the original object is the highest index data. If it is, read the entry before that from the index to use as the second object. Otherwise, use the second index entry provided as the index data for the second object.
2. Read the two objects for xattrs and omaps. If either is marked unwritable, abort.
3. Determine whether to perform a redistribution of entries or a merge
    1. If the total number of entries between the two objects is $2k$, merge them.
    2. If the total number of entries exceeds $2k$ and the object rebalance was called on has size $k$, create two new omaps such that the object in which key belongs has at least $k$+1 keys.
4. Add a prefix to the two entries for the objects being rebalanced. The prefix contains all information necessary to roll this forward or backward.
5. Set the “unwritable” xattr on the first old object to true, failing and rolling back if the version number has changed
6. Set the “unwritable” xattr on the second old object to true, failing and rolling back if the version number has changed
7. Create the new object (in case of a merge) or the first of the new objects (in case of a rebalance)
8. If rebalancing and not merging, create the second of the new objects
9. Delete the first of the old objects
10. Delete the second of the old objects
11. Update the index to contain entries for the new objects and remove entries for the old objects

## *Explanation*

The explanation of interference handling is the same as for splits, with the following additions:

| Step number | What happens if an insert interrupts after this step | What happens if a remove interrupts after this step | What the clean-up method does if the client dies after this step |
|---|---|---|---|
| 5 (Marking the first old object unwritable) | Inserting into the first object will fail and retry because "unwritable" is true. Inserting into the second object will succeed, unless it is full. If it succeeds, the second object will be modified, which will cause the rebalancer to roll back all changes upon discovering that the second object's version number has changed. If it fails because the second object is full, it will fail to split because there is already a prefix, and then retry until the rebalance completes. | Removing from the first object will fail and retry because "unwritable" is true. Removing from the second object will succeed if it has more than *k* entries, and otherwise it will fail to rebalance because there is already a prefix. In the first case, the second object will be modified, which will cause the rebalancer to roll back all changes upon discovering that the second object's version number has changed. In the second case, the remove will proceed after this rebalance completes. | Marks the object writable and then restores the index |
| 6 (Mark the second object unwritable | All attempts to modify either object will fail because they have been marked unwritable. | All attempts to modify either object will fail because they have been marked unwritable. | 1. Marks the first old object writable<br>2. Marks the second old object writable<br>3. Restores the index |
| 7 (Create the first new object) | " | " | 1. Discovers that the second new object does not exist and so does not need to be deleted,<br>2. Marks the first new object unwritable<br>3. Marks the first old object writable<br>4. Marks the second old object writable<br>5. Deletes the first new object<br>6. Restores the index |

| | | | |
|---|---|---|---|
| 8 (Create the second new object) | " | " | 1. Marks the second new object unwritable, if it exists<br>2. Marks the first new object unwritable<br>3. Marks the first old object writable<br>4. Marks the second old object writable<br>5. Deletes the second new object, if it exists<br>6. Deletes the first new object<br>7. Restores the index |
| 9 (Delete the first old object) | Attempts to modify the second object will fail and retry because it is unwritable. Attempts to modify the first object will fail and retry because the object does not exist. | Attempts to modify the second object will fail and retry because it is unwritable. Attempts to modify the first object will fail and retry because the object does not exist. | If cleanup was not called with -ENOENT, it starts with this:<br>1. Marks the second new object unwritable, if it exists<br>2. Marks the first new object unwritable<br>3. Gets -ENOENT on trying to restore the first old object to writable<br>4. Unmarks the first new object<br>5. Unmarks the second new object<br>Either way, it does this next:<br>6. Updates the index to contain new entries |
| 10 (Delete the second old object) | Attempts to modify either result in -ENOENT | Attempts to modify either result in -ENOENT | " |

## Cleanup

The cleanup method is called when changes made by an in-progress split or merge need to be rolled back, either by the client performing the split or merge if it discovers that it has lost a race, or by another client that discovers the client performing the split or merge has died. Cleanup takes two arguments: the `index_data` struct for one index entry, and an error code. Cleanup performs the most efficient rolling back or forward based on the error code. If there is a prefix, the `index_data` struct contains the following information:

- The time when the prefix was added
- The low key, high key and object name for every object to be created
- The low key, high key, object name, and version number for every object to be deleted. The version number is used for debugging.

When a client calls cleanup during a split or merge to roll back its own changes, for example, when it discovers that the version number has changed when it attempts to mark an object unwritable, cleanup simply performs the reverse operations that the client already performed in reverse order. Since this only results in states accounted for in the tables above, the above tables are sufficient to demonstrate the correctness of these versions of cleanup.

The remaining interesting cases are the cleanup methods that run when a client discovers a prefix that has timed out. In these cases, if either of the original objects is missing, the original split or rebalance got far enough that all data had migrated to valid new objects and there was no possibility of failure, so cleanup updates the index to reflect the new state. In any other state, cleanup rolls back the operation. The rollback operation is as follows:

1. Mark the newly created objects unwritable, in reverse order (i.e., starting with the object that was created last), asserting that they are not already unwritable,
2. Remove the unwritable flag from the original objects in the order that they were added
   - If any of these encounter -ENOENT, roll back 2 and 1 and then invoke roll forward
3. Delete the marked objects in the order that they were marked (which is the reverse of the order in which they were created), asserting that they still exist and are still flagged as unwritable
4. Restore the index to its original state, as it was before the split or rebalance started.

Unlike in split and rebalance, most of these steps do not check for errors. With the exception of step 2, which performs alternate steps if it encounters -ENOENT, cleanup proceeds through the steps regardless of whether any individual operation succeeds or fails. This makes it adequate for a variety of incomplete states that are possible when a client dies. For example, if a split dies after it has marked the original object unwritable but before it creates the new objects, the cleanup method will restore the original object even though it fails to delete the new objects that were never created.

### *Explanation*

There are two cases for potential interference between clients regarding the cleanup method: the case where two clients are racing to clean up after the same dead client, and the case where the cleanup method attempts to clean up after a client that has not actually died. In both explanations, a rebalance that removes two and creates two objects will be considered, since splits and merges are subsets of the operations involved in a rebalance.

In the first case, a table is unnecessary. Each cleanup follows a deterministic set of steps, and the expected state of the system is the same after each step for each client. At each step, regardless of which client wins the race, the outcome will be identical. This also resolves any potential problems associated with a client dying during cleanup – another client can simply restart the cleanup operation, and the result will be identical.

In the case of cleaning up after an in-progress rebalance that has timed out but not died:

| Last step completed by cleanup: | How an in-progress split or rebalance responds |
|---|---|
| 1 (Marking newly created objects unwritable in reverse order) | If it had not already created new objects, nothing happens. Cleanup does not delete the new objects later because they will not be marked unwritable.<br>If new objects had been created, the next step applies. Unmarking the old objects before attempting to delete the new objects ensures that no data loss is possible. |
| 2 (Unmarking old objects) | Either cleanup will succeed in unmarking the old objects before the split or rebalance deletes them, causing the split or rebalance to roll back, or the split or rebalance will delete the old objects first, causing cleanup to unmark the new objects and roll the rebalance forward. |
| 3 (Deleting the new objects) | If there is still a split or rebalance in progress at this point, it has marked the index but not marked the old objects unwritable. None of the cleanup operations made any changes, and the rebalance can go forward. |
| 4 (Restoring the index) | The split or rebalance will fail to update the index, since the index values it added will have changed. The split or rebalance will then roll back all changes, and the original set or remove will retry. |

# Design Trade-offs

## Height vs. Breadth of the Tree

A true B-tree would strictly enforce the $k$ and $2k$ limits to node sizes on all nodes, including the root. Most examples of B-trees follow this model and allow trees of arbitrary height, but in some cases a tree with fixed height is more efficient. A notable example of this use case is Google's BigTable[v]. A B-tree with arbitrary height reduces the worst case time needed to complete operations on the root node, but introduces additional latency proportional to log(N), where N is the number of nodes in the tree. In cases where keys or values are large, this is a worthwhile trade-off, but for purposes of this structure, limiting the tree to two levels was better for several reasons:

- With small keys and values, the time it takes to make the network round trip required to reach an object is often significant compared to the time it takes to read the object from the OSD. Since reading keys and values from an object is comparatively inexpensive, it makes sense to optimize for fewer network round trips.

- The complexity of ensuring exclusive access during splits and merges increases linearly with the average height of the tree. Although writes would only have to be blocked on the leaf node, each ancestor node would need to be marked with information about the operation in case of client death. This means that for each additional level of the tree, three round trips would need to be added: one to read the parent object to determine where the next object is, one to mark the parent, and one to unmark the parent when the operation completes.
- Since only keys are stored in the index, relaxing the constraint on its size only becomes a bottleneck if the number and size of keys is very large compared to $k$ – it is independent of the size of the values.

## Balance vs. Frequency of Splitting and Merging

A perfectly balanced tree would have the same number of entries in each node. This system does not attempt to enforce any balance rules more strict than the $k <= \text{size} <= 2k$ requirement. This is because splits and rebalances are by far the most expensive operations, while the difference in efficiency of performing a typical write on an object with 1.5 $k$ objects and on an object with 2k objects is insignificant. In this system, it is more effective to optimize for minimizing the likelihood of splits and rebalances happening. If reading very large or very small nodes were expensive and splits and rebalances were inexpensive, it might be reasonable to enforce more strict balancing rules.

## Optimism vs. Pessimism in Locking

When an object is in the process of being split or merged, it must block writes for some period of time. In particular, two correct ways to achieve this were considered here: Marking the old objects unwritable before creating the new objects, and marking them after creating the new objects. The advantage of creating objects first is that writes are able to succeed during the time it takes to write the new objects. However, this increased probability of writes succeeding while a split or rebalance is in progress increases the likelihood that the split or rebalance's assertion of the version number will fail. When this assert fails, the client must roll back all changes, which involves making three network round trips, and deleting one or two potentially large object(s). The advantage to marking the objects first is that the only window in which a write can succeed while a split or rebalance is in progress is between marking the index and marking the object. This means that if the client's assertion of the version number fails, it only needs to make one network round trip, and the only operation to perform on the OSD is changing one or two entries in the omap of the index object. The final version of the code marks the old objects before creating the new objects because splits and merges happen rarely enough that it is not important to optimize clients' abilities to continue writing to objects with in progress splits or rebalances, and because the time needed for the two extra network round trips is significant.

# Performance

This design has a very large number of configurable options to test, which made it difficult to create a set of tests that would adequately explore the wide range of variables. For each variable tested, assumptions were made about the best conditions for the other variables. After tests of a single variable revealed a best setting for that variable, that setting was used in subsequent tests of other variables.

## Hardware used

The machines used for testing used quad core hyper-threaded Intel Xeon E5630 @ 2.49 GHz CPUs with 7200 RPM Hard drives and 8 GB of RAM. Network connections were one Gb.

## Assumptions Made

### *Size of Initial Data Set*

In real-world use cases, reads and writes will likely come from disk (instead of from the machines' caches). To ensure that the tests were not simply reading from cache, before performing timed tests, all tests wrote four times the total cache size of all machines, divided by the number of replicas. All tests maintained the following relation:

$$\frac{4 \times \frac{kB\ RAM}{Machine} \times Number\ of\ machines}{Number\ of\ replicas \times Number\ of\ clients \times kB\ per\ entry} = Number\ of\ entries\ written\ per\ client$$

### *Number of Operations in Flight*

To accurately measure throughput, the OSDs must receive more requests than they can process concurrently. Based on prior testing of Ceph, it was assumed that 100 operations in flight per OSD would be enough to overwhelm the OSDs. Five clients wrote with the appropriate maximum number of operations in flight per client:

$$\frac{Number\ of\ OSDs \times 100\ concurrent\ operations\ per\ OSD}{Number\ of\ clients}$$

### *Caching*

Every operation begins with a read of the index object. To reduce contention over the index object, and to reduce the number of network round trips required for each operation, each client maintains a cache of recently read index entries. The cache has a maximum size, and when new entries are added, the oldest entries in the cache are removed. When a client discovers that a cached index entry is no longer accurate, it replaces that entry with the current information read from the index object. A client discovers that its cache information is outdated by attempting to perform a write and getting an error. After receiving an error, the client re-reads the index object and retries the write operation. This means that a successful cache lookup reduces the number of required round trips by one compared to no caching, while a bad cache lookup increases the number of required round trips by two.

There are two configurable aspects of the cache's behavior: The maximum cache size, and the number of entries the client reads each time it reads the index object. Informal monitoring of early results suggested that a max cache size of 1000, with 20% of the max cache size read each time the client reads the index object, resulted in a very high cache success to cache failure ratio. That is, most operations were able to find the object where the

key belonged based on cached information, and additional reads as a result of outdated cache information were extremely rare. Since the key size remained constant in all tests, the cost of that cache configuration was not expected to change significantly, and so the cache size of 1000 and 20% refresh interval were used in all tests.

### *Initial k value*

The first variable tested was entry size. It was assumed that having a number of objects greater than or equal to the number of operations in flight would maximize throughput, since there would be very little object contention. An additional consideration is the size on disk of the objects – LevelDB is best at handling objects between 4 and 100 MB. Finally, the initial populating of the data set is done with one client creating its share of the total objects at a time, so ensuring a minimum number of objects that would not limit throughput during this phase was useful for limiting the run time of tests. *K* values were selected so that the minimum object size was 26 MB, the maximum was 52 MB, and the expected number of objects was 2000 (5 clients times 400 operations in flight).

### *Number of operations*

Each test run created the initial data set and then performed 10,000 timed operations.

### *Workload Distribution*

For all tests except for the different distributions test, the operations are randomly chosen among reads, inserts, removes, and updates (25% probability each), and the keys are randomly generated (for inserts) or randomly chosen from the key set (for reads, removes, and updates).

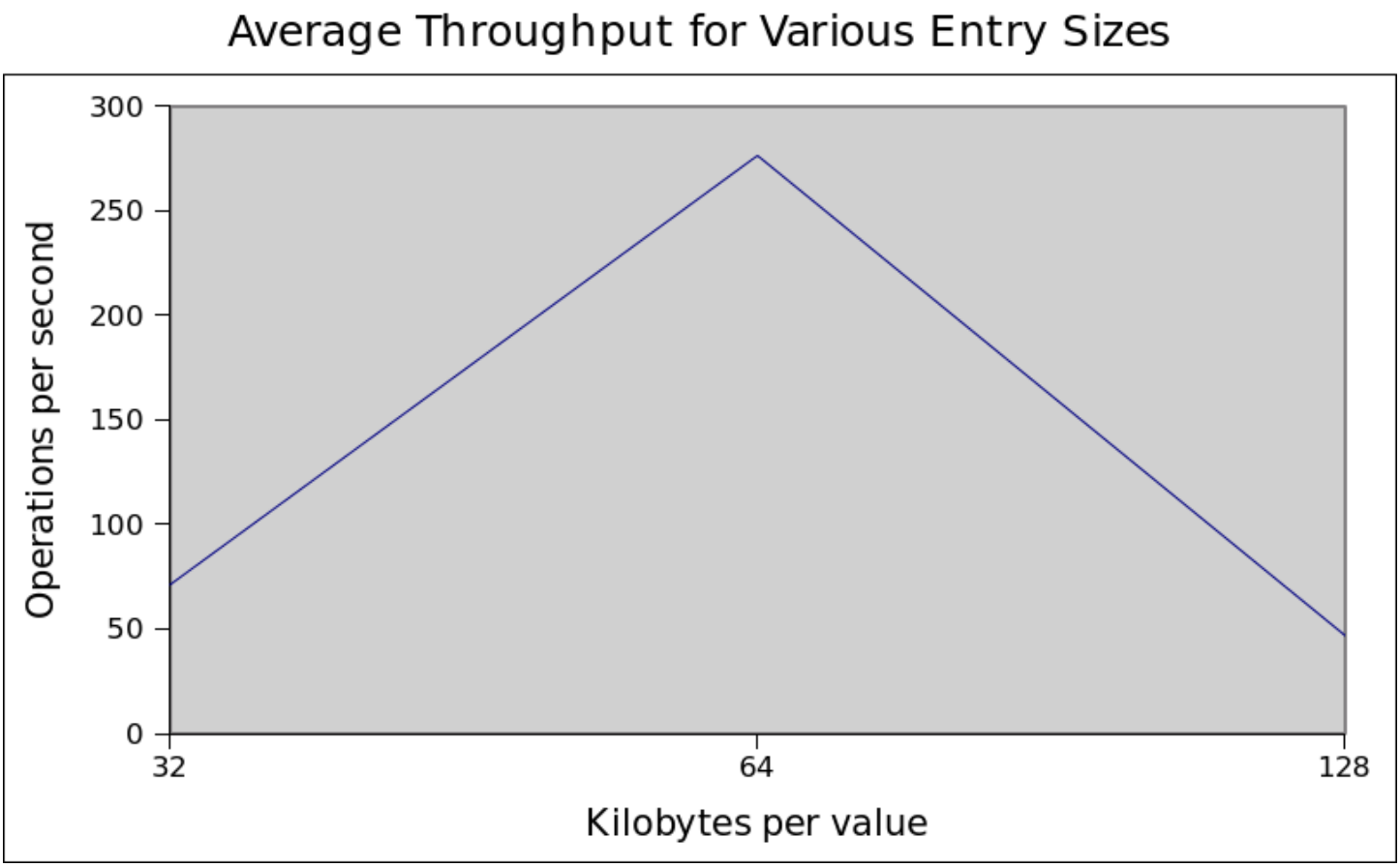
### *Initial number of OSDs*

For all tests except for the number of OSDs test, 4 OSDs were used.

### *Replication*

The Ceph configuration used for testing maintains two replicas of each object. This means that each write must be written to two different machines, and the total number of I/O operations per second is twice the value recorded in the graphs that follow.

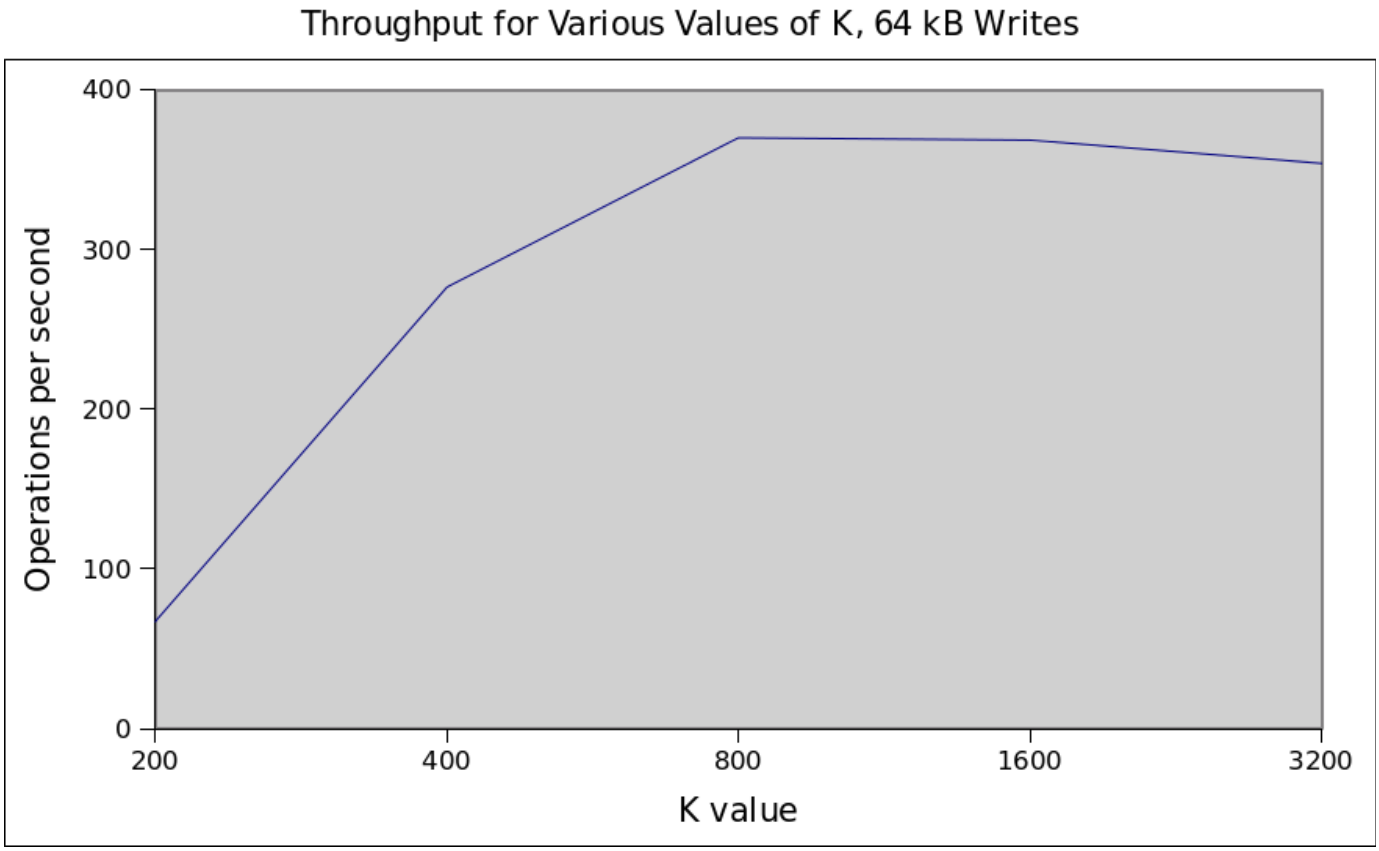
## Testing Entry Size




For all three runs of this test, there were approximately 2500 objects, each with an average size of 26MB. Runs with smaller values had more entries per object.

Smaller values reduce the cost of reading and writing the value, but increase the relative cost of network round trips, as well as the overhead of map operations that must iterate through maps with more entries than a map of the same size comprised of larger entries. This test found that 64 kB writes provide the optimal balance with the configuration used to test.

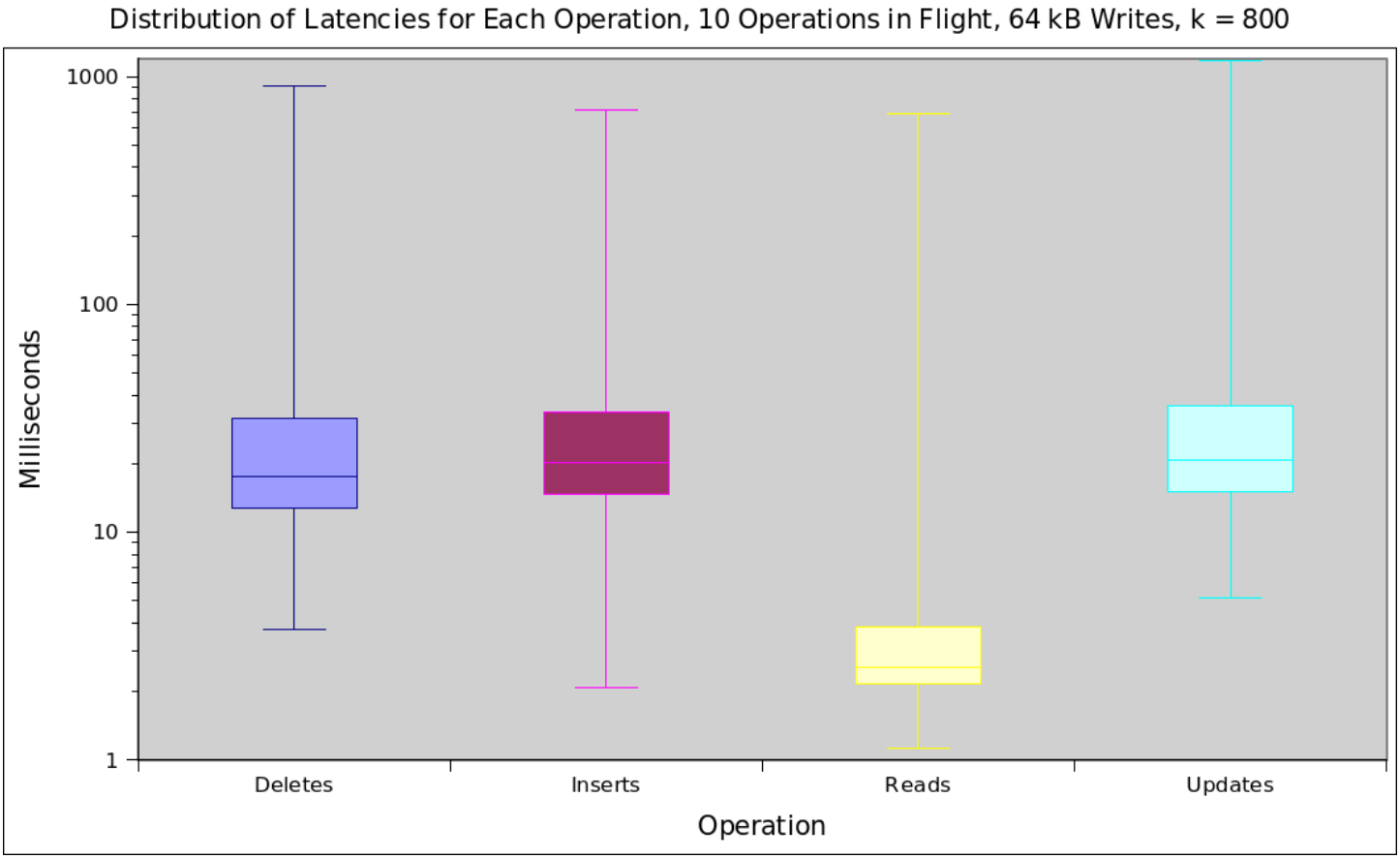
## Testing *K* Values




This test kept constant the size of writes and the total number of entries, but varied *k* values and therefore object size and object number. For the same number of entries, higher *k* values reduce the likelihood of splits and merges because there is a larger range of acceptable sizes, but having fewer objects increases object contention, which can reduce throughput. More objects also increases the likelihood that the index value for any given key will change if a single object is split or merged. This has the potential to increase the likelihood of cache failure.

At a *k* value of 800, objects have an expected size of approximately 80 MB, and there are approximately 900 objects. This *k* value is higher than the expected optimal *k* value used for the previous test (400). Throughput increases dramatically as *k* increases to 800, after which it drops off gradually as *k* increases further. This suggests that $k = 800$ is, in this case, the point at which the throughput-limiting factor becomes object contention rather than frequency of splits and merges.

# Testing Latency



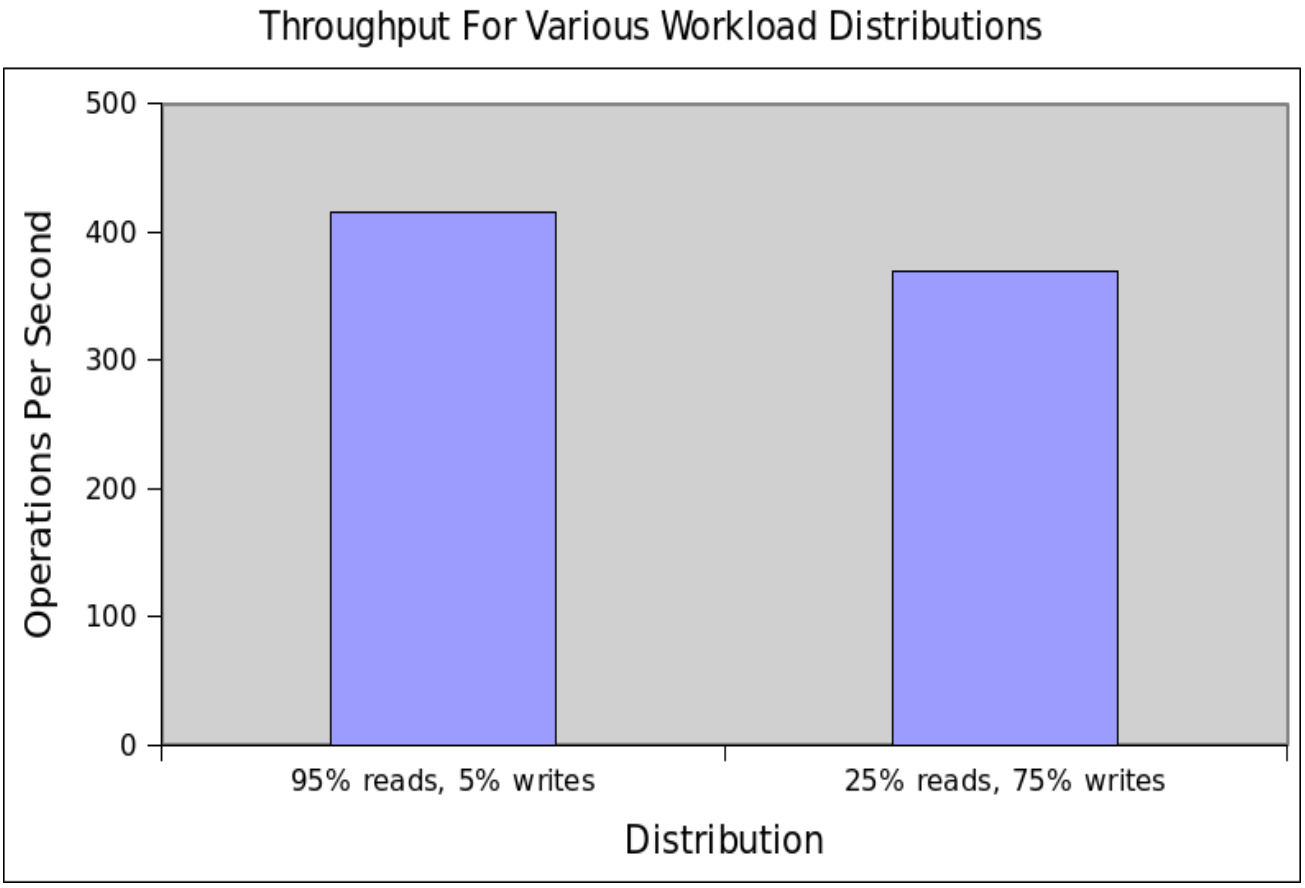

| | Deletes | Inserts | Reads | Updates |
|---|---|---|---|---|
| Median Latency (ms) | 17.681 | 20.111 | 2.547 | 20.866 |
| Average Latency (ms) | 30.78281 | 32.354464 | 9.5889992 | 34.340407 |

For the previous tests, since there were enough operations in flight to cause operations to wait for other operations to complete on the OSD side, it was not interesting to measure latency. For this latency test, two operations in flight per client were used. This low number should ensure that the latencies measured do not reflect queuing on the OSD. This test performed 64 kB writes with k = 800.

The time required to do a disk seek on a 7200 RPM hard drive is approximately 8 ms. Average reads are only slightly more time consuming than that, which suggests that the hard drive speed is the limiting factor. Median latency for reads is faster than the time required to do a disk seek because reading cached values from RAM is much faster than reading from disk. Because each write is replicated, each delete, insert, or update requires two disk seeks and two network round trips. Reads, however, only perform one disk seek, since they are read from only one OSD. This accounts for the difference in latency of reads and writes.

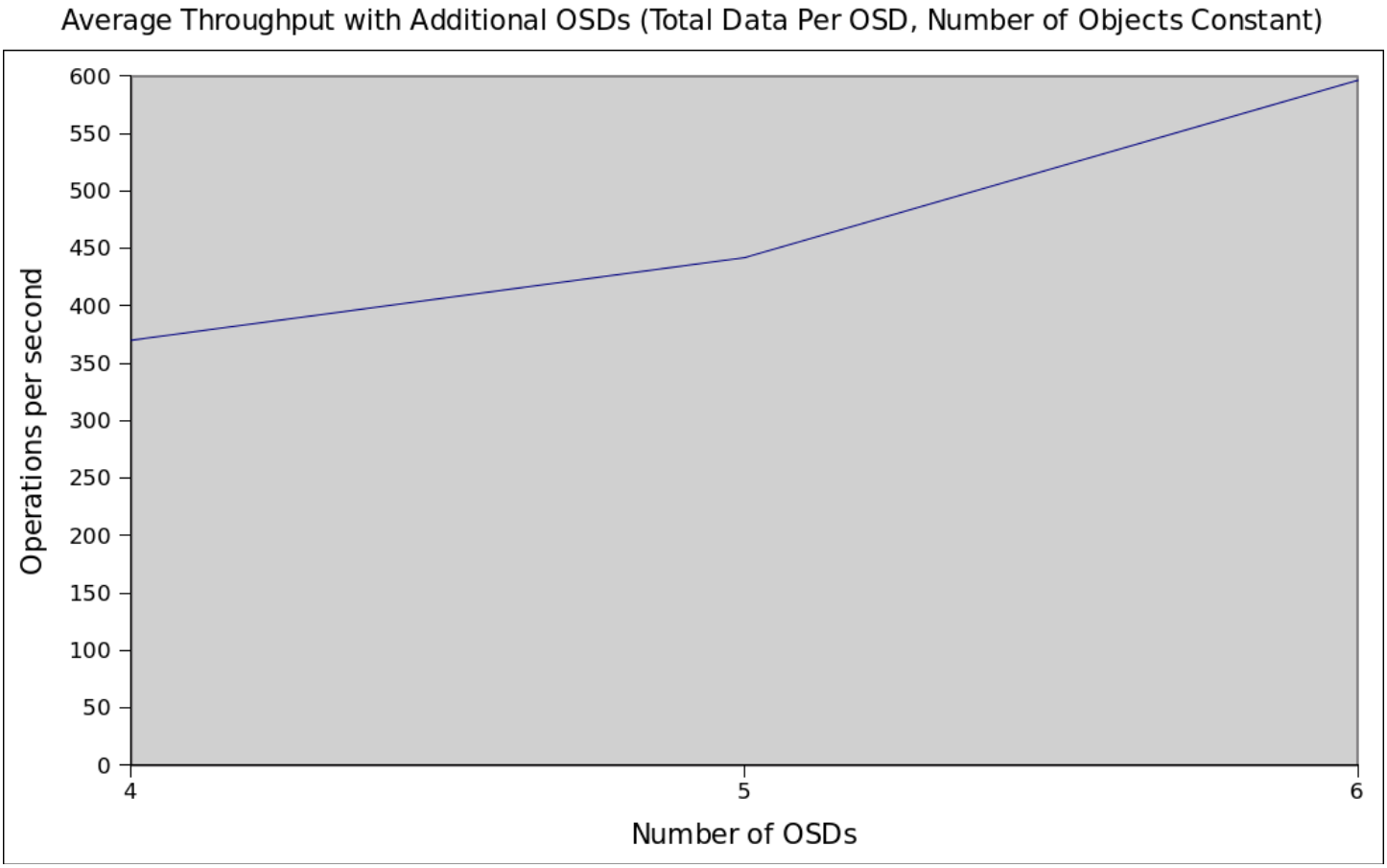
For writes, the discrepancy between average and median latency is explained by the expense and infrequency of splits and rebalances. For reads, the discrepancy reflects the difference between reading values from cache and reading them from disk – as with splits and rebalances, disk reads are infrequent but expensive.

## Testing Different Operation Distributions




The distribution of sets, reads, removes, and updates used for previous tests was write-heavy (75% writes, 25% reads). This test compared throughput from a read-heavy workload to that of a write-heavy workload. This test used the throughput-optimized setting of 80 operations in flight per client. All other settings are the same as in the previous test.

## Testing Scalability




One of Ceph's most important features is its scalability. In this test, the total data written was scaled based on the added machine, and *k* was increased so that the total number of objects remained constant. All other settings were the same as in the previous test. As expected, adding additional machines hosting OSDs drastically improves throughput.

# Conclusions and Further Questions

The best configuration of this key value store design achieved 370 replicated operations per second on four OSDs (740 operations per second total), and 596 replicated operations per second (1192 total) on six OSDs. This demonstrates that the design has reasonable throughput and scalability up to at least six machines in typical cases. Median latencies for writes were close to 20 ms with 64 kB writes on a 7200 RPM hard drive, with 17 ms for deletes. The median latency for reads was 2.5 ms. Assuming 8 ms for a disk seek and 0.5 ms to write the 64 kB per disk, replicated across two disks, plus 1 ms per network round trip, these numbers are consistent with observed OSD and network latencies at this time. Average latencies, however, were significantly higher than median latencies, which implies that splits and rebalances can be further optimized, either by using different settings so that they occur less frequently, or improving the algorithms, or both.

Inherent in the large scope of this design is an inability to fully test all possible configurations. A logical next step would be to derive an equation to find the optimal k-value, cache settings, and number of operations in flight given the number of entries and the number of OSDs.

An additional configuration to change would be how k-values are enforced. In the current design, splitting an object creates two objects with exactly *k* nodes, which makes them candidates for rebalancing. Other B-tree-based structures relax the 2k max size to 3k, which reduces the likelihood of a popular subset of the key space continuously being split and merged after a small number of insertions or deletions.[vi]

It would also be valuable to explore the potential advantages of a variable-height tree in more depth. The algorithms provided would not be difficult to scale to apply to a variable height tree. The discrepancy between the average and median latency numbers suggests that splits and rebalances, while infrequent, do have a significant effect on the overall efficiency. Not requiring every split and rebalance to modify the same object would reduce contention over the index and cause cache values to become invalid less frequently.

# Works Cited


i R. Bayer, E. McCreight, 1st Initial. , "Organization and Maintenance of Large Ordered Indices," *Acta Inf.*, Vol. 1, no. 3, 173-189, 1972.

ii F. Chang, J. Dean, S. Ghemawat et. al., "Bigtable: A Distributed Storage System for Structured Data," *Operating Systems Design and Implementation*, 2006.

iii S. Weil, "Ceph: Reliable, Scalable, and High-Performance Distributed Storage," Ph. D. *Thesis*, December 2007.

iv S. Weil, A. Leung, S. Brandt, "RADOS: A Fast, Scalable, and Reliable Storage Service for Petabyte-scale Storage Clusters," *Petascale Data Storage Workshop*, November 2007.

v F. Chang, J. Dean, S. Ghemawat et. al., "Bigtable: A Distributed Storage System for Structured Data," *Operating Systems Design and Implementation*, 2006.

vi O. Rodeh, "B-trees, Shadowing, and Clones," *ACM Transactions on Computational Logic*, Vol. 3, no. 4, 1-27, 2008.