Project Horcrux: MapReduce on the Voldemort Distributed KVS

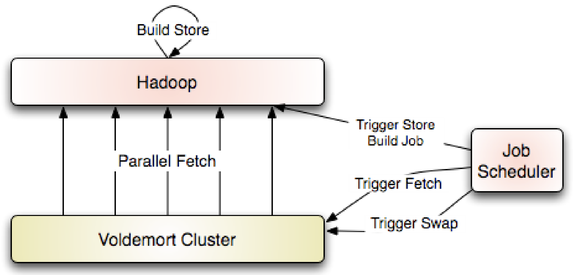
Jason Feriante, Kai Da Zhao, Sreeja Thummala

UNIVERSITY OF WISCONSIN–MADISON

Motivation

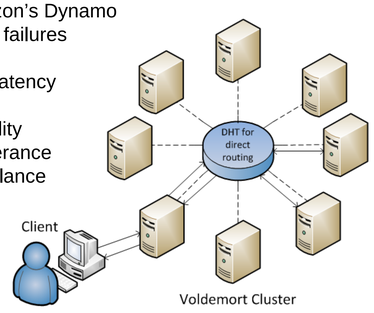
- Voldemort is a popular open-source Dynamo inspired distributed KVS and provides an excellent fault tolerant, high performance KVS
- MapReduce provides a low level abstraction for running parallel batch processing jobs
- No MapReduce implementation currently exists for Voldemort.

Currently, organizations push data to a separate Hadoop cluster in order to run MapReduce tasks



Why build MapReduce for Voldemort?

- An open-source clone of Amazon's Dynamo
- Sloppy quorums for temporary failures and high availability
- Zero-hop DHT routing for low latency
- Merkle trees for anti-entropy
- Vector clocks for write availability
- Hinted handoff for partition tolerance
- Consistent hashing for load balance
- Built-in caching



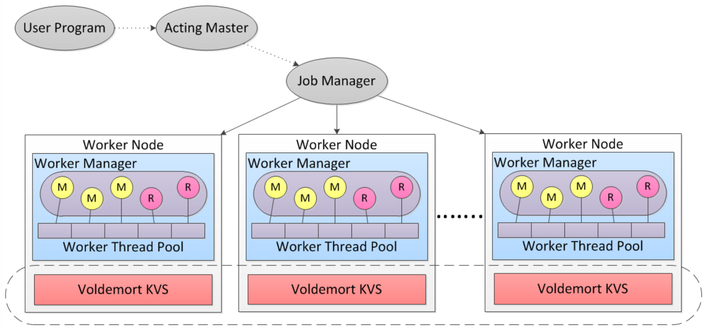
System Architecture

Acting Master:
- Responds to job requests by creating a job manager and passing the job request to the created job manager
- Keeps track of availability of node

Job Manager:
- Runs on the same node as acting master
- Uses the scheduler and distributes workload (job slices) among the worker nodes
- Acts as status server for the worker nodes
- Collects the result details from the worker nodes
- Reschedules job slices from stragglers and dead nodes
- Aggregates intermediate KVs and schedules reducer nodes on completion of map phase

Worker Manager:
- Runs on worker nodes
- Distributes workload to the worker thread pool
- Monitors the worker thread pool

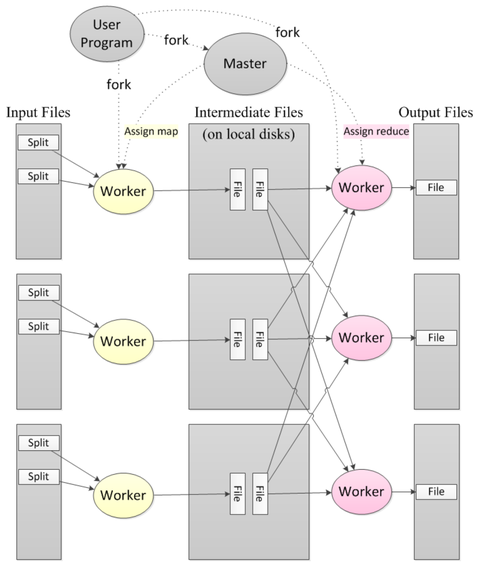
Worker Thread Pool:
- Runs scheduled map / reduce tasks in parallel



Contrasting Horcrux and Google Versions of MapReduce

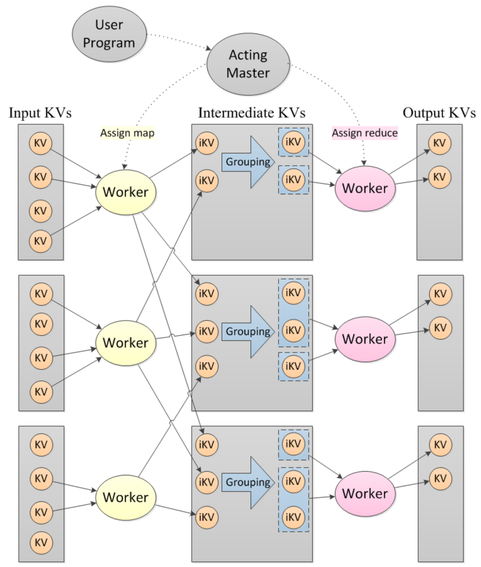
- Both systems have identical goals, but very different underlying architectures: traditional MapReduce assumes GFS / HDFS while Horcrux assumes a distributed KVS
- Although the workflow is the same on the surface, the underlying algorithms are very different
- Horcrux does not make any file system assumptions and algorithms are key-value based
- In contrast, Google's MapReduce algorithm depends on a global view of the file system and the algorithms are file-based

Google GFS MapReduce



- Scheduling based on global view of GFS file system
- Large file assumption, 100MB or greater which simplifies scheduling
- Master is very efficient for scheduling but is also a single point of failure (SPOF)
- Large files leads to large sequential reads
- Has no cache
- Adding nodes requires configuration and can be laborious

Horcrux KVS MapReduce



- Scheduling based on zero-hop DHT key-value node locations
- Most values are likely less than 16k which adds complexity to scheduling
- Automatic master failover with:
  - Akka leader algorithm
  - Resync with Kafka Multi-Broker
- Small values and consistent hashing result in random access disk reads
- Voldemort has a built-in cache
- Easy / simple to add new nodes

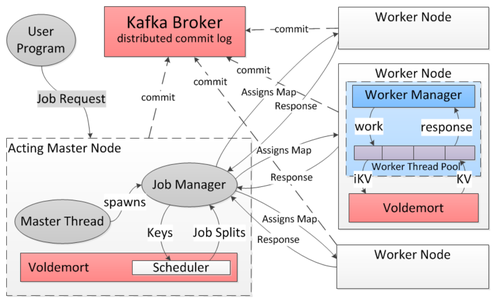
Design Goals

- Build our MapReduce system on the high performance foundation Voldemort provides
- System must scale with few required configuration changes (easy to add or remove nodes)
- System must be fault tolerant and any node can be acting master
- Provide a distributed commit system for log based recovery
- Use Voldemort's distributed hash table for scheduling tasks to exploit locality

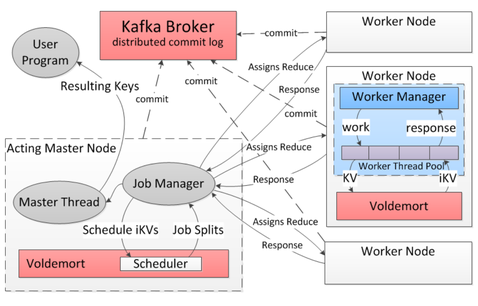
Implementation

- Horcrux is our own original MapReduce implementation that runs on Voldemort
- The Horcrux scheduling algorithm runs inside Voldemort to exploit locality through the zero-hop DHT
- The rest of Horcrux runs in a completely separate thread pool
- We used the Akka cluster singleton model is used to implement parallel distributed computations
  - Akka provides automatic leader failover
  - Akka reroutes job requests to the current acting master
- Apache Kafka multiple-broker cluster is used to store the distributed commit log
  - Kafka is used to recover the master state of the previous leader or a failed worker
- A worker node can easily be added to the cluster
- Client can connect to the cluster by submitting request to any node on the seed list
- This method provides robust fault tolerance because any node can act as master

Map Phase

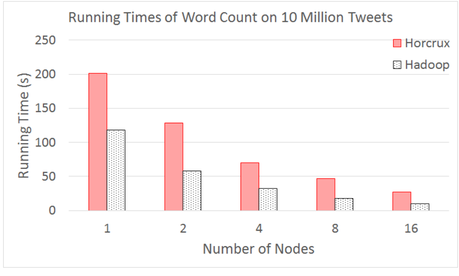


Reduce Phase



Evaluation

- Ran benchmarks to compare Horcrux and Hadoop performance on a word count
- Collected a sample of live Twitter Tweet stream.
- Ran word count on 10 million Tweets partitioned equally among the number of nodes
- Horcrux has worse performance due to seek times from many small random reads



Related Works

- Google published the original version of MapReduce in 2004 [1], which runs on GFS [4]
- Hadoop [2] is the most popular open-source implementation of MapReduce
- Dyrad [5] offers a low level, efficient graph based version of MapReduce
- Spark [6] is another MapReduce implementation with resilient distributed data sets and accumulators
- SSS [7] is MapReduce on the Tokyo Cabinet distributed KVS

References

[1] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large *Clusters". 6th Symposium on Operating Systems Design and Implementation, pages 137-149, 2004*

[2] Neethu Mohandas, Sabu M. Thampi. "Improving Hadoop Performance in Handling Small Files". *Advances in Computing and Communications in Computer and Information Science Volume 193, pp 187-194, 2011*

[3] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, Sam Shah. "Serving Large-scale Batch Computed Data with Project Voldemort". *FAST'12 Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18-18, 2012

[4] S. Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". *Symposium of Operating Systems Principles*, October 19-22, 2003

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dyrad: Distributed Data-Parallel Programs from Sequential Building Blocks". *EuroSys*, March 21-23, 2007

[6] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets". *Hotcloud*, 2010

[7] H. Ogawa, H. Nakada, R. Takano, and T. Kudoh. "SSS: An Implementation of Key-value Store based MapReduce Framework". *2nd IEEE International Conference on Cloud Computing Technology and Science*, 2010

Project Horcrux: Adding MapReduce to Voldemort Distributed KVS

Sreeja Thummala, Jason Feriante, and Kai Da Zhao
{sreejat, feriante, kzhao32}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin - Madison
1210 W. Dayton St.
Madison, Wisconsin 53706-1685 USA

Abstract

Voldemort, a clone of Amazon's Dynamo, is a high performance, open-sourced, distributed, scalable, reliable, fault-tolerant, and highly available key value store (KVS). Currently, MapReduce can't run directly on a Voldemort cluster. Data must be piped from Voldemort to an offline Hadoop cluster, which delays MapReduce analysis and consumes massive bandwidth. Aside from reporting delays, maintaining another entirely separate Hadoop cluster for offline reporting can be cost-prohibitive. Horcrux solves these problems by adding MapReduce functionality to a Voldemort cluster. This allows real-time reporting and this could supplement or potentially even replace the need for a separate offline Hadoop cluster. Horcrux has a built-in level of fault tolerance, scalability, and performance due to the well-engineered foundation that Voldemort provides. Our goal was to remain aligned as much as possible with the original goals and design choices that Voldemort offers, but to also allow new MapReduce functionality without losing scalability, fault tolerance, or availability.

1 Motivation

MapReduce, first introduced in 2004 by Jeff Dean and Sanjay Ghemawat[1], is a programming model and associated engineering implementation for running batch computations on clusters of commodity machines in parallel. The MapReduce algorithm applies a divide and conquer strategy by partitioning a query with key/value pairs into much smaller buckets, which are then mapped to other nodes in the cluster. After the nodes complete their map tasks, the system then completes a reduce phase where data is aggregated. Running small jobs throughout a cluster in parallel greatly speeds up batch computing tasks. MapReduce has proven to be an effective means of running data intensive parallel distributed jobs on a server cluster and Hadoop has become the most popular open-source implementation.

Although MapReduce only offers a low level interface which must be extended to run any given task, it still continues to grow in popularity because it provides a simple interface which simplifies the programming of otherwise complex parallel computations. Many popular key value store (KVS) systems have built-in MapReduce support including: HBase, Riak, Cassandra and others. Unfortunately, the Voldemort KVS does not have any known existing MapReduce support; open-source or otherwise. Most likely, this is because adding MapReduce functionality to Voldemort is a non-trivial task.

As shown in Figure 1, Voldemort KVS is a clone of Amazon's Dynamo; a highly available, fault tolerant, and scalable distributed store. Voldemort gives up ACID properties of a traditional database, as well as some immediate consistency in order to provide a level of availability which would otherwise not be possible. Key features of both Dynamo and Voldemort are zero-hop DHT routing for

low latency, merkle trees for anti-entropy, vector clocks for conflict resolution, hinted handoff for partition tolerance, and consistent hashing for load balancing. Although Voldemort is based on Dynamo, it is not identical. One key difference between the two systems is Voldemort's built-in caching system.

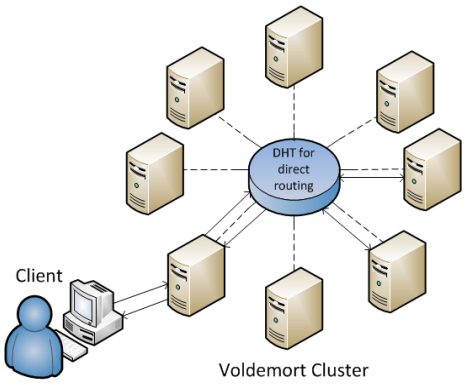


Figure 1: The architecture of a Voldemort KVS cluster

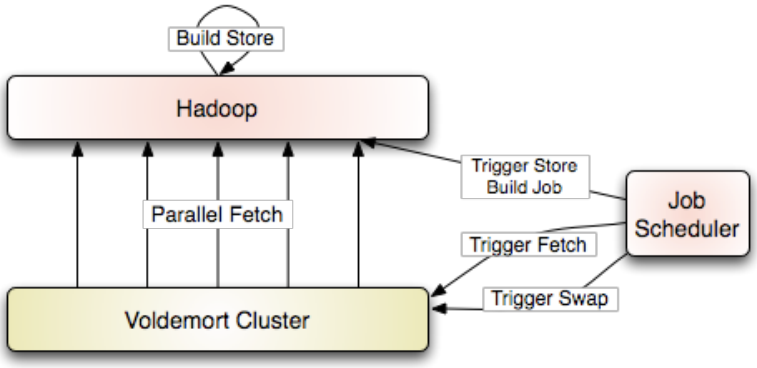


Figure 2: Pushing KVS data to a separate Hadoop cluster to run MapReduce tasks

Currently, as shown in Figure 2, LinkedIn uses Azkaban[2], an open source scheduler to operate a multi-terabyte data pipeline between Voldemort and their offline Hadoop cluster. MapReduce computations are only run on the offline cluster, and any reporting must be delayed until the data transfer is completed. The problem with this design is that there will always be some delay between the time data is added to the Voldemort KVS and when it becomes available in the offline Hadoop cluster.

The motivation for Horcrux is to reduce hardware, software, and bandwidth requirements by adding MapReduce functionality directly to the existing Voldemort cluster. By adding direct MapReduce support, an organization can avoid the need for building and maintaining an entire separate offline Hadoop cluster (and associated software complexity, and bandwidth consumption) just for offline MapReduce computations. Also, the added ability to obtain real-time results from MapReduce operations is an additional, and possibly more significant benefit, especially for financial markets or other industries with extremely time sensitive needs.

2 Design Goals

Our goal is to provide a MapReduce system on the high performance foundation that Voldemort provides. The system must scale with few required configuration changes; it will remain easy to add or remove nodes (as it is already with Voldemort). The system must be fault tolerant and allow any node to be acting master. The system must also be able to recover from failures; for example by using a distributed commit system for log based recovery. The system should also use Voldemort's distributed hash map for scheduling tasks to exploit locality.

3 Architectural Challenges

In order to add MapReduce functionality to the Voldemort KVS, we had to overcome a number of significant design challenges, most of which arise because Voldemort is built on a set of assumptions which are completely different from that of Hadoop and the original MapReduce paper written in 2004[1]. Numerous complications arise due to these differences in the core foundational systems. The three most significant difficulties we faced with adding MapReduce to Voldemort were:

1. No single master: we have no global view of the file system, which complicates scheduling.
2. Large file assumption[3]: we can't assume contiguous or large files like GFS or HDFS.
3. Algorithm file assumption: the MapReduce algorithm assumes files will be used at each step.

First, the traditional MapReduce algorithm is built on the assumption of an underlying file system such as HDFS or GFS[4]. Both of these file systems have a global master server which knows the locations of all files on all nodes in the cluster. This assumption is built into the core of the MapReduce algorithm, and it simplifies the scheduling during the map task, and it makes it trivial to exploit locality by sending jobs to the nodes that have the right files. In contrast, with Voldemort, we have no master, so although we have additional fault-tolerance with no single point of failure, we do not have a global view of the file system. Instead, we have a distributed hash map that only tells us which key belongs to which node. This makes it more difficult to exploit locality while scheduling a MapReduce job, and it means we can't use any of the traditional MapReduce scheduling algorithms.

Second, file systems like GFS and HDFS assumes file sizes of 100 MB or greater, which minimizes the bandwidth and computation required to schedule and communicate MapReduce jobs to the various nodes in the cluster. In contrast, with Voldemort we have no sense of which keys are in which files, so we can't assume large, contiguous files. This unfortunately means that even if we exploit locality, we will still need to communicate a large number of keys over the network. Once each server receives a set of keys for a Map or Reduce job, we are potentially forced to make many random disk accesses (unless the various values are cached in memory). In order to produce results with low latency, either the working sets must fit in aggregate memory or Horcrux must be deployed on servers that exclusively use flash storage.

A typical HDD is only capable of delivering around 1 MB/s of throughput for random access reads, while a commodity SSD can deliver over 30 MB/s per second. We believe that using persistent flash storage will generally prevent disk throughput from becoming the bottleneck on Horcrux. However, this is not required if Voldemort's cache is already warmed up with the data we need. In fact, this can provide a substantial advantage over a traditional MapReduce system that must go to disk multiple times during the course of a job. Regardless, our system is built on the assumption that persistent flash storage will be used. We believe that this is a reasonable assumption as the price of SSDs continues to drop.

We decided to add MapReduce functionality inside of Voldemort cluster because that would expose the internal API of the system and allow us to run functions natively, with maximum efficiency while also providing a scalable, fault tolerant, foundation with no inherent single point of failure. We considered running the Horcrux system as a separate process on top of Voldemort, but we found this prevented us from directly accessing internal system calls to structures such as the distributed hash map (which contains the locations of all keys in the cluster). We could have built a new interface which would allow us to do this outside Voldemort, but this meant we would have to build an entirely separate system which would replicate a lot of the work Voldemort was already performing (such as caching key locations, etc.). This would create unnecessary complexity because we would have to add new Voldemort API interfaces to provide system information to our MapReduce functions.

Traditional MapReduce assumes a global view of the file system. However, Voldemort only knows which node contains which keys, but not the file locations within the nodes. We explored options that could address this by allowing Horcrux to have a global view of all files, such as an overlay file system. However, maintaining a global index of all file locations would incur a significant performance penalty since Voldemort does not run on HDFS. Most values in Voldemort are likely to be very small; less than 128KB on average, and in contrast, HDFS often has file sizes of 100 MB or greater. Using a small number of large files for scheduling is very efficient for Hadoop, but this is not practical in Voldemort since a large job might require millions of small files.

Even if we could tolerate the performance penalty that would indexing millions (of small files would incur, there is an additional problem of staleness. Most likely, maintaining a global index for all the millions of small files in the Voldemort system would result in a perpetual level of staleness for some percentage of files in the overlay system. More significantly, an overlay file system does not address an important issue, which is random disk access. In general, the keys on our server tend to be spread out randomly due to the consistent hashing, which Voldemort uses for load balancing. Due to the inherent randomness of the system, reading large contiguous file chunks would generally not be feasible for our MapReduce jobs, and hence, most reads would likely still be random access (in spite of an overlay system).

We decided that adding an overlay file indexing system would add efficiency to the system without providing any real benefit since files are small where staleness and random access are inevitable. By not maintaining a global view of the file system, we keep our design simpler, and more scalable, and this also keeps our design in alignment with Voldemort (which intentionally has no global file index). This also prevents us from having to make any assumptions about the file system in order to achieve efficient and scalable MapReduce performance.

4 Horcrux Architecture

The Horcrux MapReduce system has four main components: the Acting Master, the Job Manager, Worker Managers, and Workers. The architecture consists of a heavyweight thread spawning many smaller threads to manage or execute tasks within an event driven framework.

1) The *Acting Master* accepts job requests from the User Program and spawns a Job Manager for each job request. The Acting Master also distributes Worker Managers to Job Managers. The Acting Master also sends the MapReduce results back to the client (as a set of keys).

2) The *Job Manager* communicates with the Acting Master to reserve available Worker Managers. The Job Manager also uses the Load Balancer in Voldemort to obtain a balanced schedule for each reserved Worker Manager. Job slices are then sent to the various Worker Managers to be executed. After the map phase, results are aggregated into sets of intermediate keys and finally reduced.

3) *Worker Managers* run on all the nodes of the cluster (one per node). The Worker Manager distributes the workload throughout the worker thread pool in round robin fashion. The Worker Manager also keeps track of the workers and collects the resulting keys. Finally, the Worker Manager sends the resulting keys to the Job Manager when a job is complete.

4) *Workers* run the scheduled jobs. Many workers run in a worker thread pool, and worker threads report to the Worker Manager on their respective nodes.

Like Voldemort, the Horcrux system is masterless with no single point of failure. As mentioned previously, Voldemort has no global view of the file system, which makes our system incompatible with current versions of the MapReduce algorithm. However, like Amazon Dynamo, all Voldemort nodes know where all keys are located throughout the cluster. The master node ‘maps’ the MapReduce request by equitably partitioning the requested Keys into buckets which favor key locality but also consider overall fairness (this way, nodes that have no locality for any keys may still participate in a MapReduce task). Instead of partitioning and scheduling jobs based on file locations (as in traditional MapReduce), we perform similar calculations but based on the node locations of the individual keys. These locations are provided by the distributed hash table (DHT) which Voldemort uses to store the node locations of all keys. In general, the job scheduling tends to be relatively equitable within a degree of error.

Once the scheduling is complete, the acting master sends jobs to the various nodes in the cluster. Requests are compressed before being sent and decompressed upon arrival to prevent unnecessary bandwidth consumption (effectively trading CPU cycles for bandwidth). Unlike the

original MapReduce algorithm, we implement a form of Resilient Distributed Data sets (RDD) based on ideas from Spark[6]. The key difference is that Voldemort does all the caching for us, and we cache key-value pairs instead of files. In Horcrux, each node caches the intermediate representation it builds, as well as any relevant keys in order to speed up iterative tasks. The results of each map job are also stored in each respective node so that only keys must be sent over the network at each step of the process.

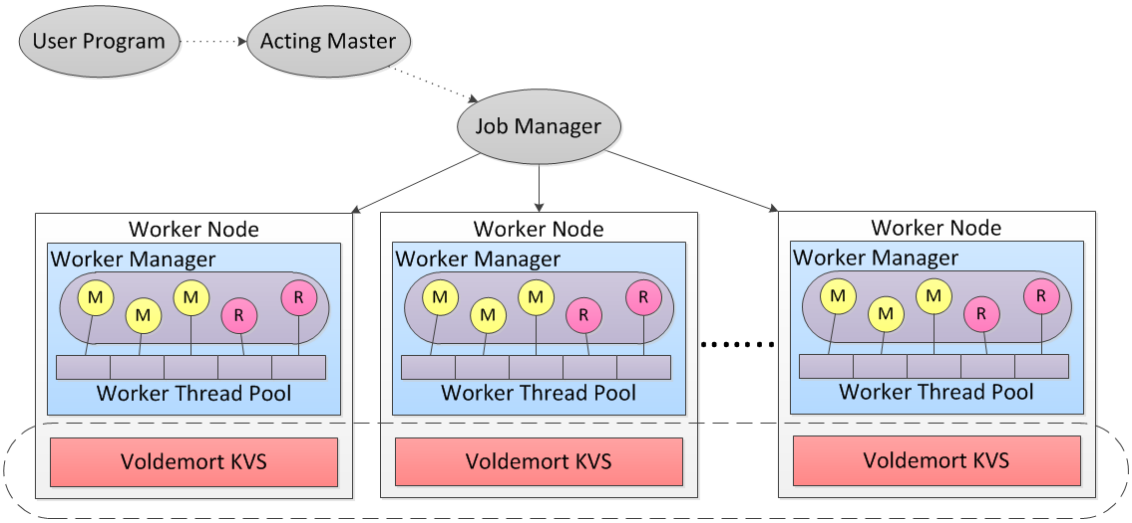


Figure 3: The Horcrux system architecture. A user program sends a job which is routed to the acting master, which spawns a new job manager for each respective job. The job manager then runs the jobs in parallel across the cluster.

5 Horcrux Implementation

New methods were added to the Voldemort shell client as well as the core client to pass MapReduce request messages to and from the internal Horcrux system. Horcrux uses a centralized load balancing algorithm which to achieve a near optimal workload distribution of keys to mappers. Although Voldemort's consistent hashing already provides a good overall load distribution, users might want to MapReduce only a subset of keys that could possibly be mapped to the same node.

The load balancer uses consistent hashing to create a table with all keys are initially assigned to its primary partition along with the number of keys assigned to each mapper. Then, the load balancer examines the workload of all the respective nodes. If there are any sizeable imbalances, the load balancer will scan for keys that are currently assigned to the heavily loaded node and shift them to a more lightly loaded replica. Load balancing is very beneficial when there is a small number of mappers because finding a key to exchange is faster if both mappers have a higher portion of keyset. The threshold for what constitutes as a balanced workload is set as a proportion in a configuration file and can be adjusted if needed by the user. We achieved only a near optimal load balance because it does not consider cascaded load balancing, where a more optimal load balance could be accomplished if Node A can pass keys to Node B, which in turn, can pass keys to Node C.

Horcrux consists of two distinct clusters running on separate nodes:

1. *MapReduce cluster:* (Akka) which runs on top of the Voldemort cluster
2. *Multi-broker cluster:* for Kafka which runs on separate nodes

The MapReduce cluster is modeled after Akka cluster singleton class which is P2P cluster with a single master. The Akka cluster singleton provides message routing to the acting leader, leader election on leader failure, gossip protocol for membership changes and dead letter queues for retrying

messages. The MapReduce cluster is bootstrapped using pre-configured cluster seeds. As part of bootstrapping, the cluster elects a leader. The acting master node in the MapReduce / Akka cluster is not predefined. It is elected during bootstrapping or when present leader fails. The acting master is the supervisor for all the Job Managers and Worker Managers. In turn, the Worker Managers act as supervisor to the worker thread pools (which are in fact Akka actor thread pools). When a node is elected as leader, the Worker Manager running on the node is removed from the pool of available Worker Managers.

The second cluster mentioned above is based on Kafka, which is used to provide a distributed commit log which provides a basis for component recovery on system failure. To improve reliability and performance, instead of using single-broker, multi-broker is used. Kafka multi-broker runs on a separate cluster. If any components fail, the supervisors contact Kafka brokers to get relevant log and schedule their functionalities on other equivalent components. The entries in the multi-broker commit log are then used to restore the state of the broken component.

The client process for sending a MapReduce job is fault tolerant to match the fault tolerance of the cluster which could have an arbitrary number of dead nodes. The client maintains a list of live nodes with a heartbeat system. Any live node will route the client's MapReduce request to the acting master. The job request is comprised of map and reduce instructions, as well as a set of input keys.

When a job request is received, the acting master creates a Job Manager to run the MapReduce task. The Job Manager then contacts the master for a set of available Worker Managers for scheduling. If there are not enough Worker Managers available, the job is queued. Once the Job Manager reserves enough Worker Managers to complete the request, it contacts the Voldemort scheduler with this information to get a partition map which tells which keys should be assigned to which Worker Manager for processing. The Job Manager creates job slice requests with this information and sends them to the relevant Worker Managers. When each respective Worker Manager receives the job slice requests, the input keys are divided uniformly throughout the worker thread pool (there is one Worker Manager and one Threadpool per node). When a job is completed, the Worker Manager stores the results in Voldemort and only the keys are sent to the next phase. Worker Manager waits till all the worker nodes have finished their assigned tasks. Then they sends results to the Job Manager.

The Job Manager waits until all the Worker Managers have completed their assigned job slices. Once they are done, it aggregates all the intermediate keys. Using these intermediate keys as input keys Job Manager schedules the reduce phase.

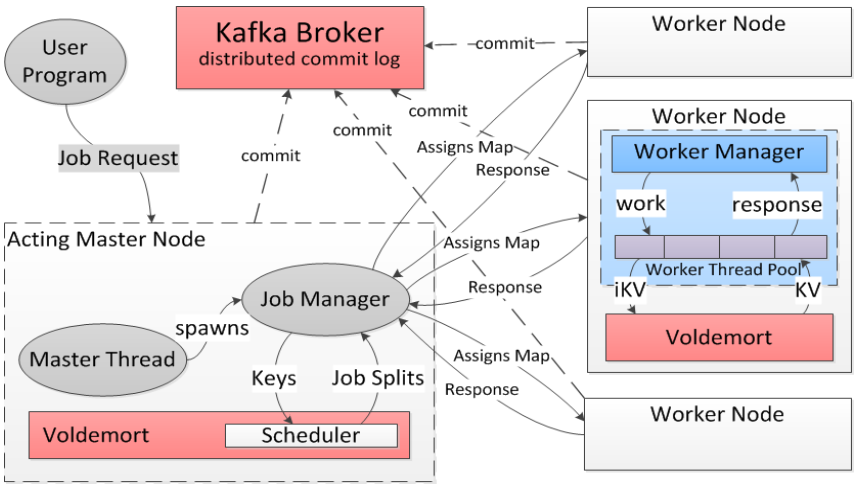


Figure 4: The implementation of the map phase. The User Program sends a job request to the Job Manager. In turn, the Job Manager uses the scheduler implemented in Voldemort to assigns mapping tasks for the Worker Manager. Then, Workers read KV, map, and store the intermediate KVs back into Voldemort.

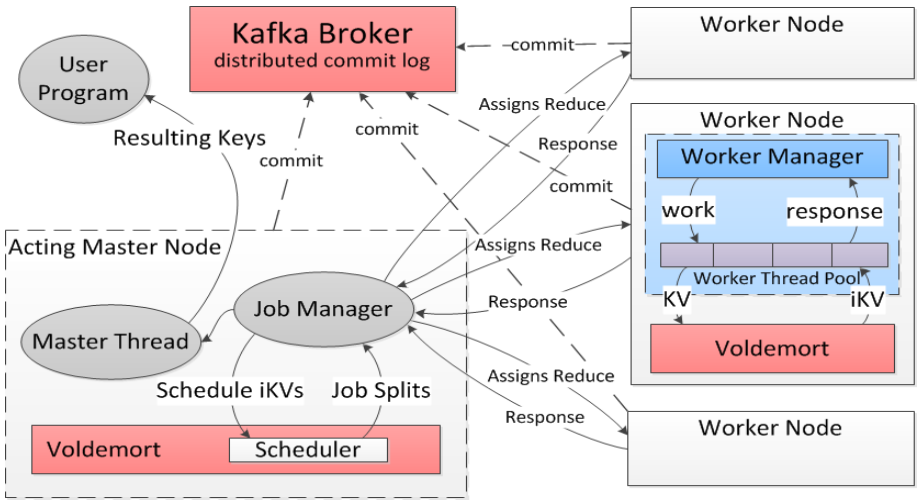


Figure 5: The implementation of the reduce phase. The Job Manager uses the scheduler implemented in Voldemort to assigns reduce tasks for the Worker Manager. Then, workers read iKV, reduce, store the resulting KV back in Voldemort, and return the resulting keys to the Job Manager. In turn, the Job Manager returns the resulting keys to the User Program.

6 Contrasting Horcrux vs Traditional MapReduce

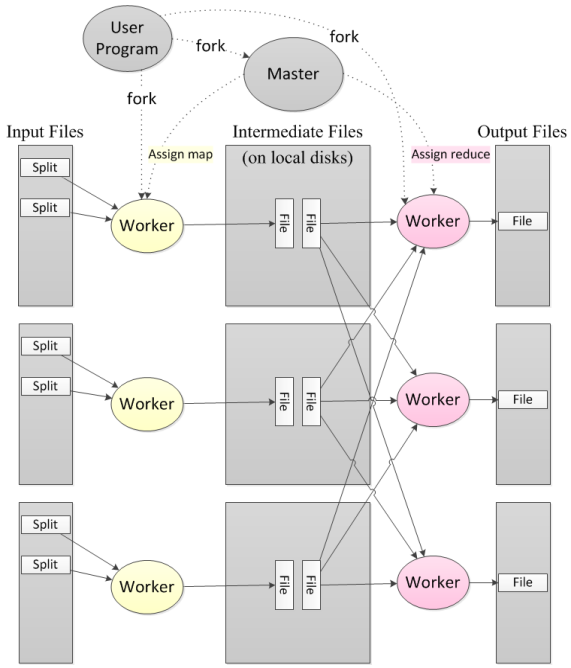


Figure 6: MapReduce on GFS

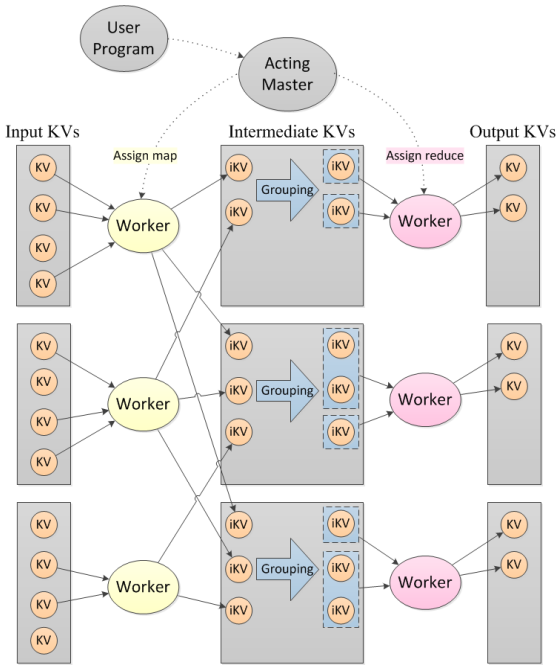


Figure 7: Horcrux: MapReduce on a KVS

There are five fundamental differences between MapReduce on GFS and Horcrux. First, MapReduce on GFS relies on a global view of GFS file system and can exploit the large file assumption for scheduling tasks. On the other hand, Horcrux does not make any file size assumptions and algorithms has to work efficiently on small key values. Second, MapReduce on GFS operates on large files for fast sequential reads. However, Horcrux operates on small keys, which tends to result in random disk access. To counteract this slow random access, Horcrux assumes persistent flash storage (although

Voldemort's built-in caching can help mitigate this as well). Third, MapReduce on GFS has a master that schedules tasks very efficiently with a few large files, but the master is also a single point of failure. Horcrux as uses a master with automatic failover for fault tolerance on the master node Furthermore, with many small keys on Horcrux, Horcrux needs to balance between balancing load and running map and reduce tasks. Fourth, as shown in the Figure 6, files must be transferred between every node for shuffle and sort. In contrast, as shown in Figure 7, Horcrux's KVS has a built-in shuffle and sort when storing the intermediate key values back into the KVS. Fifth, adding new nodes to GFS MapReduce is laborious and requires reconfiguration. However, adding new nodes to Horcrux / Voldemort is simple because Horcrux exploits Akka without a name server, and minimal configuration is required.

7 Horcrux Performance Results

The performance measurements were conducted on computers in University of Wisconsin-Madison Computer Sciences Department's instructional labs. We use galapagos-20 through galapagos-35, each of which has i5-4570 quad-core processors @ 3.20GHz with 6 MB cache and 16 GB of main memory. Tweets were collected from a live Twitter data stream to a local file in the /tmp directory. Then, we pushed collected Tweets data into HDFS and into the Voldemort cluster. When pushing Tweets into HDFS, we split the 10 million Tweets into 16 files, so each file contains 625,000 Tweets, each of which consumes about 313 MB of storage. When pushing Tweets into the Voldemort cluster, we use each key to store a set of 128 Tweets each, each of which consumes about 64 KB of storage.

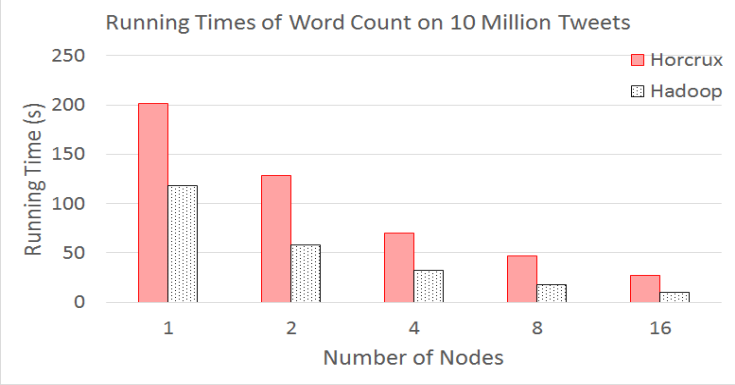


Figure 8: The performance results of running word count on 10 million tweets partitioned equally among the number of nodes.

The test suite benchmarks the performance of running word count on Hadoop with large 313 MB files and on Horcrux with small 64 KB keys. The 10 million Tweets were partitioned equally among the number of nodes. The results shown in Figure 8 is the minimum running time of word count out of 5 trials. The good news is that Hadoop outperforms Horcrux by only factor of two, probably because the Voldemort has built-in caching, which was warm after the first trial. We believe this a fair comparison because both, Hadoop and Horcrux, were ran on warm cache. However, the bad news is that Hadoop is still able to outperform Horcrux, probably because of Horcrux trades performance for high reliability, high availability, and fault tolerance on many small files.

8 Related Work

Apache Hadoop[10] is the most popular open-source implementation of MapReduce. Hadoop uses HDFS as the underlying file system, and the Hadoop ecosystem includes HBase, Pig, and Hive as well.

Our performance evaluation was done to contrast Horcrux with Hadoop since this is somewhat of a generally accepted baseline (for batch jobs).

The paper by Lakshman presents Apache Cassandra[9], a distributed KVS. Originally Cassandra was built by Facebook, although they later abandoned it. Cassandra is very similar to Voldemort (since it is also based on Amazon's Dynamo). Unlike Voldemort, Cassandra has no built-in cache, although MapReduce functionality is built-in to Cassandra. Generally, performance is very close between the two systems: Voldemort has lower latency and Cassandra has higher throughput.

The paper by Ogawa presents SSS[5], another implementation of MapReduce on KVS. SSS implements MapReduce on Tokyo Cabinet, which does not satisfy the CAP and ACID properties as well as Voldemort. For example, SSS only stores one copy of each key-value and cannot utilize a load balancer in case many keys maps to the same node.

The paper by Zaharia presents Spark[6], which partitions resilient distributed datasets (RDDs) and accumulators. Spark is ideal for iterative workloads, such as machine learning since the system both pre-loads and caches data (RDDs). This results in a factor of 10 to 100 in overall speed improvements for iterative workloads. On batch oriented jobs, traditional MapReduce frameworks like Hadoop tend to perform better.

The paper by Ekanayake presents Twister[7], which similar to Spark, also attempts faster iterative MapReduce by storing data in main memory. However, loading data to main memory would incur large amount of delay, which partially defeats the purpose of implementing MapReduce on the KVS cluster in the first place.

The paper by Isard presents Dryad[8], which offers a low level, efficient graph based version of MapReduce. Dryad offers finer task granularity, which may implement MapReduce on top of a KVS if we specify how to parse the input files. However, programming the parse phase with a KVS is not straightforward.

9 Conclusion

We have demonstrated that Horcrux can provide MapReduce functionality on top of Voldemort storage layer. The benchmark results show that building MapReduce on top of a KVS can have reasonably comparable performance when the entire working set can be cached in memory. The data processing benchmark likewise shows visible performance gains as the number of processing nodes increases.

Although Horcrux has achieved our project goals of building MapReduce with a Voldemort KVS backend, there is a fundamental requirement that keys must be stored in cache in order to obtain comparable performance (although we think persistent flash storage can also provide similar results as well). Storing data in cache is feasible for live data analysis because data recently stored in the KVS will be stored in cache as well. However, if data were removed from cache, and traditional slow hard disks are used, then Horcrux with small keys can be expected to run much slower than traditional MapReduce with large files. In the future we would like to benchmark Horcrux performance on solid state drives with a cold cache to confirm whether the system can compete with Hadoop.

We also believe our system can be optimized to further improve performance. One future optimization might involve never sending partitioned buckets to a node that already has the data cached in memory. A SHA1 fingerprint hash could be used to reliably identify the bucket (since the chance of a collision is less than 1 in a billion) to save bandwidth. Another optimization might involve running scheduling tasks on multiple nodes to better utilize bandwidth and CPU cycles available in the cluster to speed up the scheduling process.

References

[1] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. *6th Symposium on Operating Systems Design and Implementation*, pages 137-149, 2004.

[2] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, Sam Shah. "Serving Large-scale Batch Computed Data with Project Voldemort". *FAST'12 Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18-18

[3] Neethu Mohandas, Sabu M. Thampi. "Improving Hadoop Performance in Handling Small Files". Advances in Computing and Communications in Computer and Information Science Volume 193, 2011, pp 187-194.

[4] S. Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. *Symposium of Operating Systems Principles*, October 19-22, 2003, Bolton Landing, New York, USA.

[5] H. Ogawa, H. Nakada, R. Takano, and T. Kudoh. “SSS: An Implementation of Key-value Store based MapReduce Framework”. *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 754-761, 2010.

[6] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. “Spark: Cluster Computing with Working Sets”. *Hotcloud*, 2010.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. “Twister: A Runtime for Iterative MapReduce”. *The ACM International Symposium of High Performance Distributed Computing*, 2010.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. “Dyrad: Distributed Data-Parallel Programs from Sequential Building Blocks”. *EuroSys*, March 21-23, 2007, Lisboa, Portugal.

[9] A. Lakshman, P. Malik. “Cassandra: A Decentralized Structured Storage System”. SIGOPS Operating System Review, vol. 44, no. 2, 2010.

[10] T. White. “Hadoop: The definitive guide”. *O'Reilly Media, Inc.* 2012.