SNA Projects Blog

{ 2009 06 15 }

Building a terabyte-scale data cycle at LinkedIn with Hadoop and Project Voldemort

Many of LinkedIn's products are critically dependent on computationally intensive data mining algorithms. Examples of these include some modules like People You May Know, Viewers of This Profile Also Viewed, and much of the Job matching functionality that we give to people who post jobs on the site. To support these data-intensive products we have begun to move many of the largest offline processing jobs to Hadoop. These jobs form a fairly typical data cycle. Data is moved out of twenty or so online data storage systems (Oracle, MySQL, Voldemort, etc) as well as from our centralized logging service, where they go to offline systems like Hadoop, AsterData, and our Oracle Data Warehouse. Moving all the data into centralized offline processing systems like these dramatically simplifies the implementation of complex algorithms which may use data from dozens of sources. Once data has been extracted a sequence of offline processing jobs are run on it. Finally the results are automatically loaded back into the live system to feed parts of the website. All offline data we produce is read-only once it goes live to avoid the complexity of merging the offline computations with online updates during the next run of this data processing cycle.

The difficulty in these systems comes with the fact that large amounts of data need to moved around every day. Thus although hundreds of gigabytes or terrabytes of data are not to difficult when sitting still in a storage system, the problem because much, much harder when it must be transformed to support quick lookups and moved between systems on a daily basis.

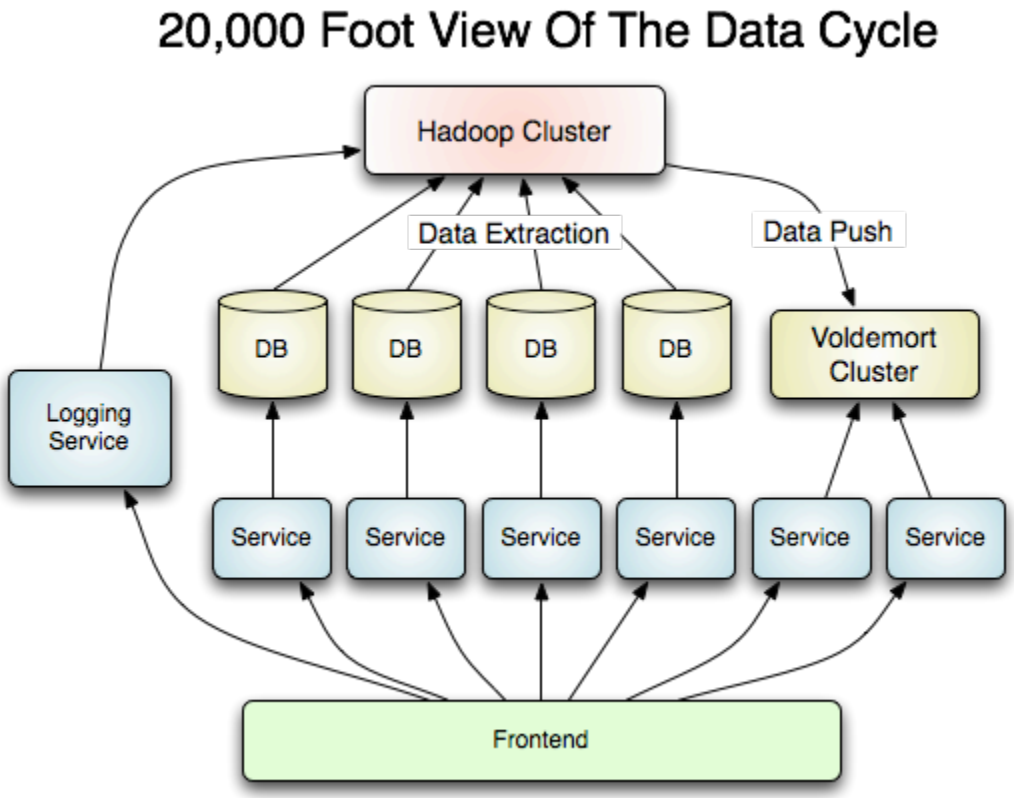
This post describes the system we built to deploy data to the live site using our key-value storage system, Project Voldemort.

Why do we end up with so much data? The size of the output is usually determined by the quantity of something on the site: we might compute something for each member profile, each question that gets asked, each news article that is posted, etc. These jobs may process a lot of data, especially if they involve any of the very large logging data streams, but the results, though large, are manageable. We have a second kind of job that is at least as common and produces results for each *pair* of users, or each *pair* of companies, or, say, the relationships *between* users and questions, or between the many other types of content on our site. As you might imagine the number of interesting pairs of items is much larger than the number of actual items (it isn't as large as the square of the number of items, since most pairs aren't interesting, but it is still huge). This seems to be a natural use case for social networks where the relationships are of central importance. Previously we did not need to confront this problem both because our data size was smaller, and also because our ability to produce large offline datasets was limited by computation constraints. Hadoop has been quite helpful in removing scalability problems in the offline portion of the system; but in doing so it creates a huge bottleneck in our ability to actually deliver data to the site. As is often the case, removing a bottleneck in one area

creates a new bottleneck somewhere else.

To solve this problem we spent some time thinking about how to build support for large daily data cycles. Voldemort was designed to support fast, scalable read/write loads, and is already used in a number of systems at LinkedIn. It was not designed specifically with batch computation in mind, but it supports a pluggable architecture which allows the support of multiple storage engines in the same framework. This allows us to integrate our fast, failure-resistent online storage system, with the heavy offline data crunching running on Hadoop.

Here is a picture of what our world looks like:



Some existing approaches

There are plenty of other ways to approach this problem, but no one we talked to had a good solution. We saw many variety of things being done, including pushing static text files by hand, FTPing giant XML files or doing JDBC batch inserts in an (Oracle) DB. None of these are really good approaches to the problem, since they typically have one of two common problems. The first is that the data transfer is centralized, creating an un-scalable bottleneck in the delivery of the data. The second is that the process of building the lookup index (generally a btree) is happening on the same live server that is serving lookups. This is a big problem since building a large index is a huge and computationally intense operation that may take hours, and by doing this on the live server we are effectively mixing this huge throughput-oriented operation with short-latency sensitive lookups, generally with poor results for your users.

So what alternatives are there?

The best online system for data lookups right now is memcached. Memcached is stable and has excellent performance for common caching needs. The obvious problem with memcached are the "mem" and the "cache" parts. Memcached is all in memory so you need to squeeze all your data into memory to be able to serve it (which can be an expensive proposition if the generated data set is large). In addition memcached is a cache, so if you need to restart your servers then your data will disappear and need to be re-pushed! Another problem is the apparent lack of batch set operations. Without this the majority of time will inevitably be spent on unnecessary network round-trips no matter how efficiently we implement them. We could easily build a map reduce job to do this in parallel, but that only works around the underlying weakness in the per-node transfer rate.

The next best online system is MySQL. MySQL can avoid the one-round-trip per insert problem by doing batch inserts, but even that seems to give rather low throughput. MySQL's InnoDB table format has too high space overhead to make it a real competitor. However MySQL has a very slim and simple MyISAM format. MyISAM isn't used as much for normal read/write usage since it uses a global table write lockand lacks many transactional features, but this isn't a problem for read-only usage since it is write-free. MySQL also supports an optimized "load data infile local" statement that provides bulk load capability. This is an extremely important feature for a disk-based storage format in this use case–building a 100GB index can not be done effectively as a sequence of b-tree updates that incrementally re-arrange data as the they go because the total IO casued by all the little updates is extremely high. To avoid this the tree needs to do a batch build that builds as much of the tree at once as possible, and this is exactly what the "load data" statement does. All-in-all MySQL is slim, quick, and generally the best off-the-shelf solution to this problem we have seen. Still to make this build effective you need a ton of memory, and it will lock the table for the duration of the build. This means that if you are running this on your live servers they will be extremely heavily worked for the duration of the load (which can easily take hours). Not to mention that MySQL provides little in the way of ability to parallelize this, making constructing a system on top of this a difficult proposition.

Clearly building an index like this is an offline operation and should not be done on a server that is serving live traffic as it will likely choke the CPU and IO resources from serving the live requests. In principle this is possible as MySQL (rather frighteningly) seems to allow you to just copy the files for a database into the database directory of a running server which will immediately make the table appear available without restarting. But this would mean maintaining a whole separate cluster of MySQL servers just for the purpose of index building as well as devising some way of parallelising this process. Finally a practical point is that you will likely have to write the data to disk multiple times–once to copy it to the server as a text file, then again as it is built as a database, and finally a third time if it copied to a live server (not to mention that fact that MySQL unfortunately seems to make its own internal copy of the data as well when you build the index to support its transactional requirements). Since the load data statement doesn't seem to support compression, storing your data as CSV is a rather large blow-up. These things don't seem like they should be a big problem, but when your 400GB data set turns into a 1,200GB dataset because all the numbers are in ASCII, and this file is then copied multiple times, that creates a serious problem.

Requirements for a better solution

These alternatives weren't attractive, so we thought through what would be needed to do a good job with this problem. We came up with the following things:

1. **Protect the live servers.** Uploading a new data set can't impact the services relying on the data. We want the upload of new results to go as fast as possible but no faster. This means moving as much computation out of the online system as possible, and guaranteeing the live servers are not negetively impacted.
2. **Horizontal scalability at each step**. Hadoop gives us a scalable approach to the build, and Voldemort gives a scalable system for the lookups. The trick is just ensuring that there is no centralized bottleneck in the process.
3. **Ability to rollback**. Like any code, the processes that generates the data may have some kind of error or bug that leads to generating corrupt data, but unlike most code problems, fixing things may not be so quick. Since the processes may take many hours to run, and the data automatically goes live without human perusal, this kind of failure can leave us in a bad position. It may take hours to rerun (or for some very computationally intense processes, days), and we will could be stuck with the bad data until we manage to fix the bug, rerun everything, and re-push the fixed data. This is clearly unacceptable. As a result we would like to retain multiple copies of the data set, one for each of the last N pushes (where in the common case N = 1) so that we could revert to this known good state. This allows us to have a constant time rollback to a previous data set.
4. **Failure tolerance.** This is where the Voldemort consistent hashing comes in to play–a server failure in the live system will redirect 1/K of that server's traffic to each of the remaining servers without impacting the client. We also get similar failure tolerance in the build from using Hadoop.
5. **Support large ratios of data to RAM.** The original problem we are trying to solve is that the data size is very large so we need to design accordingly. Improving performance in the case where the data is all in memory is not terribly valuable, the focus is on supporting a data size significantly larger than memory on each node.

Our approach

Bhupesh, Elias, and I toyed with solutions to these requirements, and here is the design we came up with.

One thing was clear, the Hadoop cluster is the natural place for the index build to occur. Hadoop is where the data is when the processing is done, and the goal of these machines is to run at full utilization so however computationally intense the build process is, it will not be a problem.

For the live system we wanted to adapt our key-value system, Voldemort. To do this we wanted to add an on-disk structure optimized for access to very large read only data sets we could deploy in batch. In particular we wanted some kind of simple file-based format we could stream to the servers to avoid doing many network round trips during the deployment. Ideally we should be able to deploy data at the rate possible by the network or disk system of the Voldemort and Hadoop clusters.

In early versions of the storage engine we toyed with different lookup and caching structures. But some simple benchmarking revealed this to be a rather academic exercise. The fundamental fact of

filesystem access is that you may or may not be accessing the underlying disk depending on whether your request can be served by the OS's pagecache or not. A pagecache hit on an mmap'd file takes less than 250 nanoseconds but a page miss is around 5 milliseconds (a mere twenty thousand times slower). Any fancy data structure we build is likely to reside in-memory. Hence it would only help the lookups for things that would be in page cache anyway (since the process of loading them into memory would put them there) and so lookups on these would be fast no matter what. And worse this in-process lookup structure will likely steal memory from the pagecache to store its data, and since this will duplicate things in the pagecache it is extremely inefficient. Thus even if we manage to improve the lookup time for the things in our process memory, it is already quite low; and by doing so we use up memory that moves more requests out of the ns column and into the ms column. In short, Amdahl wins again.

To take advantage of this we have a very simple storage strategy that exploits the fact that our data doesn't change–all we do is just mmap the entire data set into the process address space and access it there. This provides the lowest overhead caching possible, and makes use of the very efficient lookup structures in the operating system. Since our data is immutable, we don't need to leave any space for growth and can tightly pack the data and index. Since the OS maintains the memory it can be very aggressive about this cache, and indeed it will attempt to fill all free RAM at all times with recently used pages. In comparison Java is a very inefficient user of memory since it must leave lots of extra space for garbage collection, etc. Plus anyone who has gotten intimate with Java GC tuning will not object to moving things out of the Java heap space.

How data is stored

Sometimes it's nice to know what is going on under the covers. The data for a store named my_store would consist of the following files:

```
my_store/
  version-0/
    0.index
    0.data
    ...
    n.index
    n.data
  version-1/
    0.index
    0.data
    ...
```

As you can see a store is just a directory of simple files. The *.data* files contain variable length values and the *.index* files contain the lookup structure necessary to map keys to values. In principle only one *.index* and *.data* file would be needed, but since writing a file is inherently single-threaded we break it into chunks numbered 0 through *n* to allow greater parallelism in the build. These chunks are then grouped into version directories containing a complete version of the data, with *version-0* containing the current live data set.

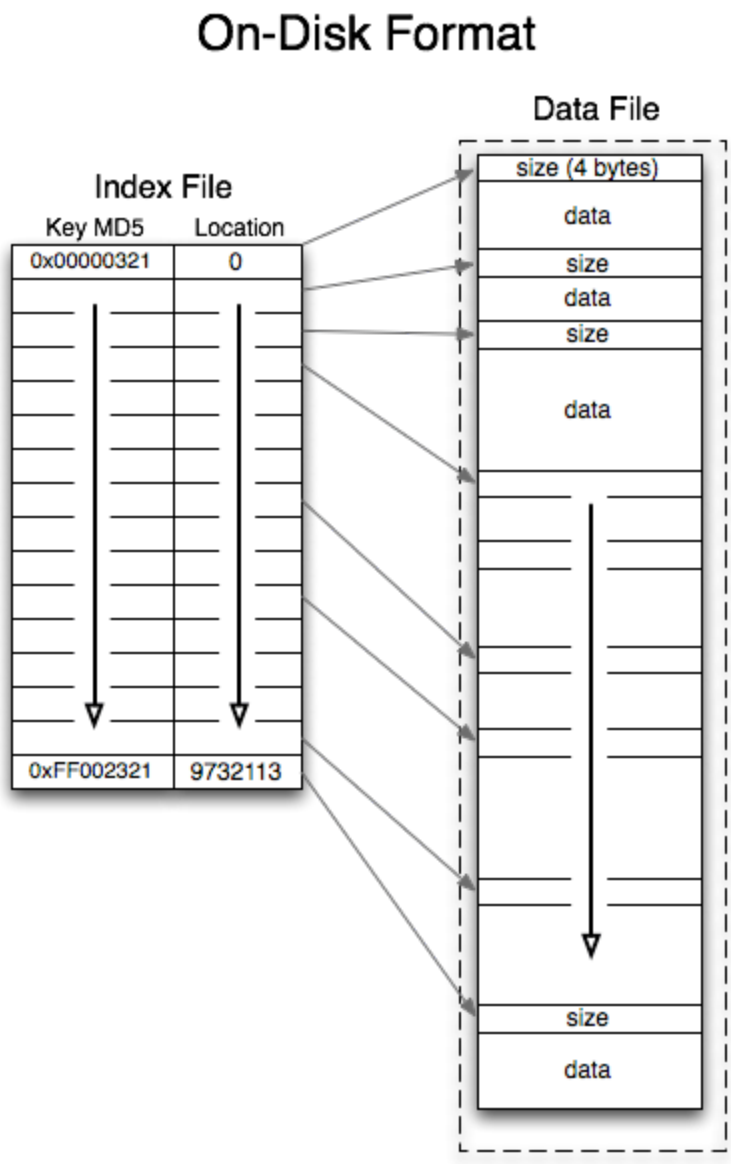
Deploying a new version of the data consists of adding a new directory and renaming the existing ones. Storing multiple copies of the data is clearly a huge waste of space, but this is not too important

as inactive files use no pagecache space just disk space. Small low latency reads on a huge data set will be largely seek bound, so we are going to need a lot of disk spindles no matter how we store things. Hard drive space is fairly cheap, so getting slightly larger disks to store additional copies is not a big problem.

To reduce the size of the file pointers and to work around limitations in Java's mmap implementation, we limit chunk files to a maximum size of 2GB, so a reasonably sized store will consist of tens or hundreds of chunks per Voldemort node.

The order in which the values in the .data files are are stored is not important. Each value is prefixed by a 4 byte length indicating how many bytes to read. Each value is uniquely identified by the offset in the file at which its 4 byte length begins. The index contains 16 byte MD5 hashes of the keys along with the associated 4 byte position offset of the value in the data file. Because we hash keys, each key/value pair we store has a fixed overhead of exactly 24 bytes in addition to the length of the value itself. Furthermore this can be stored very efficiently as we can calculate the location of the *i*th index value as 20 * *i*. All lookups are positional; no internal pointers are needed within the index to locate entries.

The question is how to structure the keys in the index for quick lookups? A page- or block-organized tree is a good data structures if the data does not fit in memory. But this complicates both the lookups (which would need to be block aware), and the build process. In particular we want to perform our build in hadoop, which means that we will be limited to the amount of memory available to the mapper and reducer tasks which may leave only a few hundred megabytes for the build–if this does not fit our index data then we will have to perform some kind of external tree build. However on consideration we realized that since the index contains only 20 bytes per key even a very moderate amount of memory can hold several hundred million entries. Given this low overhead, very likely the whole index can (and should) be in memory (pagecache, not java heap), and so organizing the data by block or page is not really very important. As a result we greatly simplified our design–we just store the index entries in sorted order by md5 hash of the key.



A lookup in the store proceeds as follows:

1. Calculate the MD5 of the key
2. The first 4 bytes of this md5, modulo the number of chunks, is the chunk number to search in
3. Do a binary search for the key md5 in that chunk's .index file to get the position of the value in the data file
4. Finally read the appropriate number of bytes for the value from the data file starting at the given position

The code for this storage engine is quite simple, only a few hundred lines, with the distribution and fault tolerance–the hard problems–being provided by the rest of Voldemort.

Binary search is not a very efficient algorithm for finding the location of the data. Most of the time this is not important since the index is in memory and so data access time dominates, but there are two

cases that could be improved. The first is the case where all data and index fit entirely in memory. With very small keys, a chunk might have an index with, say, 100 million entries, which means a binary search does 27 key reads and comparisons and a single data read. In this case the cost of the search will dominate. Another suboptimal case is when we have an entirely uncached index. We explicitly transfer index files last in the data deployment to avoid this case, however in the case of rolling back to a previous index version it is unavoidable. To page the 100 million entry index for a chunk into memory will require 500k page faults no matter what the structure is. However it would be desirable to minimize the maximum number of page faults incurred on a given request to minimize the variance of the request time. In this case a page-organized tree, where each parent had 204 20 byte children, could do only log_204(100 million) = 4.5 page faults in the worst case and would be superior.

To resolve these cases we are working on an improved search algorithm which takes into account the uniformity of the key distribution, whch results from the fact that MD5 is (somewhat) cryptographically secure and so its keys are uniformly distributed. Rather than always beginning with a comparison to the middle entry such an algorithm would use the uniformity of the key distribution to compute the expected quantile of the key being looked up attempting to jump immediately to the correct location. If we can get a reliable implementation this promises to greatly improve the number of both page faults and comparisons needed in these corner cases.

Index Building

To build these store files we created two programs: a single-process command-line java program and also a distributed Hadoop-based store builder. The single process program uses a simple external sort to build the index files. Since this is a centralized process it is only useful for small data sets, testing, or one-time builds.

The Hadoop-based store builder is actually substantially simpler than the single-process builder as it leans heavily on Hadoop's native capabilities to do its work. The store building processes proceeds as follows. An user-extensible Mapper extracts keys from the source data. This mapper can be parametrized to work with different InputFormats, and provides hooks to allow custom ways to construct the key and value from the data. A custom Hadoop Partitioner then applies the Voldemort consistent hashing function to the keys, and assigns all keys mapped to a given node and chunk to a single reduce task. The shuffle phase of the map/reduce copies all values with the same destination node and chunk to the same reduce task. Thus each of the reduce tasks will create one .index and .data file for a given chunk on a particular node; and as a result the number of chunks specified in the configuration acts as a parameter to control the parallelism of the build. These values are then sorted by Hadoop in order to group them by key for reduce. Each reduce task copies the key/value pairs it is given into a pair of .index and .data files in sorted order to build its store chunk.

Data deployment

It is important that we be able to swap in a complete data set all at once without any downtime or impact to the live cluster. As described above, multiple data versions are kept in the *version-* subdirectories, but only *version-0* is used for serving data. Versions 1 through *n* are effectively

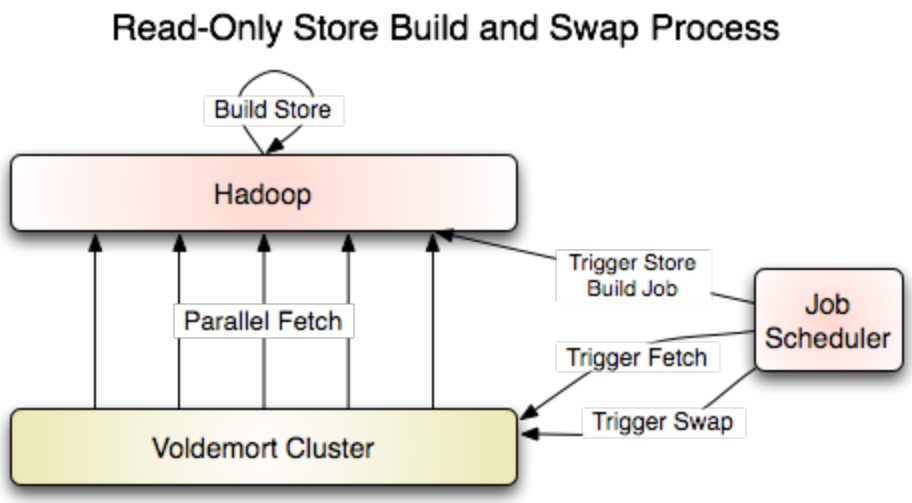
backups. When a new data version is deployed, the version number of each existing data set is incremented, and the new set becomes the new *version-0*. To perform this swap a simple reader/writer lock is used to halt readers, each directory *version-i* is moved to *version-(i+1)*, and the new data is moved to *version-0* and the store is reopened and unlocked using this new dataset. Since only file renames are used, this is an O(1) operation, and in practice the whole procedure seems to complete in a few milliseconds irrespective of file size. The deleting of the *N*+1st version is prolonged until after the lock is released as delete may not be an O(1) operation, and may take over a minute on a filesystem that lack extents such as ext3.

The actual method for transferring data is pluggable. The original prototype used rsync in hope of efficiently supporting the transferring of diffs. However, this has two practical problems. The first was that the rsync diff calculation appears to be quite expensive, and half of the expensive calculation is done on the live server. Clearly if we want to do diffs, that too should be done on the batch system (Hadoop) not the live system (Voldemort). In fact due to this heavy calculation rsync was actually slower than just copying the whole file, even when the diff was rather small (though presumably much more network efficient). The more fundamental problem was that using rsync required copying the data out of HDFS to some local unix filesystem–which had better have enough space!–to be able to run rsync. This copying took as long as the data transfer to Voldemort, and meant we were copying the data twice.

To avoid these problems we switched from a push model to a pull model. It was important that we could schedule the transfer from the batch system to run automatically when the build completed successfully, so this took the form of a RESTful fetch command which triggers the Voldemort servers to fetch the data directly from HDFS. This mechanism is pluggable and a third party can provide an alternate implementation of the fetch command to support non-HDFS based mechanisms.

HDFS provides great throughput and seems to be able to max out the write capabilities of the Voldemort node. This is a blessing and a curse. Anyone who has lived with JDBC-based data transfer and seen it bottleneck on a measly few hundred KBs/sec will be overwhelmed with joy. But once again, removing performance problems in one area creates performance problems elsewhere: the high rate of data transfer to the live servers, even without any index building, can potentially starve live requests. However, in this model, where the server controls the pull, the Voldemort nodes can be configured to throttle itself to a fixed MB/sec limit so as not to overwhelm the I/O capabilities of the local Voldemort node. We have implemented a Voldemort configuration property, fetcher.max.bytes.per.sec, that controls this rate.

We have provided a driver program which initiates this fetch and swap procedure in parallel across a whole Voldemort cluster. In our tests this process can reach the I/O limit of either the Hadoop cluster or the Voldemort cluster.



Some benchmarks

There are two things to benchmark: the build time for a store in Hadoop and the request rate a node can sustain once live. We completed our benchmarks on EC2, since this is an easy way to get big clusters up and running for a quick test. Hopefully this will aid in making the results reproducible by others interested in testing different scenarios. We used extra large instances for both Hadoop and Voldemort as these most closely match our own hardware, and we used the Cloudera Hadoop AMI to get the test cluster up and running quickly.

Benchmarking anything that involves disk access is notoriously difficult because of sensitivity to three factors:

1. The ratio of data to memory
2. The performance of the disk subsystem, and
3. The entropy of the request stream

The ratio of data to memory and the entropy of the request stream determine how many cache misses will be sustained, so these are critical. A random request stream is more or less un-cachable, but fortunately almost no real request streams are random. They tend to have strong temporal locality which is what page cache eviction algorithms exploit. So for our testing we can assume a large ratio of memory to disk, and test against a simulated request stream to get performance information.

The performance is still very sensitive to the quality of the disk subsystem used for the Voldemort nodes. A live system like this will do lots of quick seeks with relatively small reads and will likely be bound by the seek time of the hard drive and the number of drives. The drives on the EC2 machines are fairly weak and not configured with RAID so they are not optimal if you are purchasing hardware, and in our tests all the processes we benchmark are IO bound. To help make sense of all these variables we provide a comparison to MySQL's performance on the same tasks on the same hardware.

We are not aware of an existing system that does full build and data deployment in parallel, so there no direct comparison possible. But any build process will consist of three stages: (1) partitioning the

data into seperate sets for each destination nodes, (2) gathering all data for a given node, and (3) building the lookup structure for that node. We can only compare results for the actual build (i.e. part 3) with MySQL as there is no off-the-shelf method for (1) and (2).

For our tests the keys are integers in ascii form. The values are meaningless 1024 byte strings.

Build Time

We tested the Hadoop build for a variety of store sizes. This time is the complete build time including mapping the data out to the appropriate node-chunk, shuffling the data to the nodes that will do the build, and finally creating the store files. In general, the time was roughly evenly split between map, shuffle and reduce phases. The number of map and reduce tasks are a very important parameter, as experiments on a smaller data set showed that varying the number of tasks could change the build time by more than 25%, but due to time constraints no attempt was made to optimize these, we just used whatever defaults Hadoop produced. Here are the times taken:

- 100GB: 28mins (400 mappers, 90 reducers)
- 512GB: 2hrs, 16mins (2313 mappers, 350 reducers)
- 1TB: 5hrs, 39mins (4608 mappers, 700 reducers)

To compare the build time we created a RAID 10 array on a single extra large instance, and did a build using one node's worth of data (100m keys). This process to 6 hours and 3 minutes to build the 100GB table for single node. Assuming similar performance for partitioning and copying data around this would indicate a complete build time of almost 8 hours per destination node. But this comparison ignores the time necessary to extract the data from the source system and convert it to CSV format for loading. And, of course, this neglects the additional benefits of Hadoop for handling failures, dealing with slower nodes, etc.

In addition, this process is scalable: it can be run on a number of machines equal to the number of chunks (700 in our 1TB case) not the number of destination nodes (only 10).

Data transfer between the clusters happens at a steady rate bound by the disk or network. For our Amazon instances this is around 40MB/second.

Online Performance

Lookup time for a single Voldemort node compares well to a single MySQL instance as well. To test this we ran local tests against the 100GB per-node data from the 1 TB test. This test as well was run on an Amazon Extra Large instance with 15GB of RAM and the 4 ephemeral disks in a RAID 10 configuration. To run the tests we simulated we simulated 1 million requests from a real request stream recorded on our production system against each of storage systems. We see the following performance for 1 million requests against a single node:

	MySQL	Voldemort
Reqs per sec.	727	1291
Median req. time	0.23 ms	0.05 ms

Avg. req. time	13.7 ms	7.7 ms
99th percentile req. time	127.2 ms	100.7 ms

These numbers are both for local requests with no network involved as the only intention is to benchmark the storage layer of these systems.

How to actually use it

The code is all checked in to the main project repository on github. The commands for building a store, and executing a swap can be found under the bin/ directory. Elias has written a blog entry on how to use these, and how he has put this system into action at Lookery.

Future work

Nothing is ever finished, and below are a few of the ideas we didn't quite get to. There are a lot of huge performance wins that exploit the immutable nature of the data that we have not yet taken advantage of. If any one is interested in playing with one of these problems here are a few ideas. LinkedIn is also looking for engineers to work on Project Voldemort full time, so if that sounds interesting send us a resume.

Incremental data updates

Despite the problems with rsync, incremental data pushes would be quite a big improvement for the case where the data changes by only 5%. This is a common case for a job that runs daily to recompute a large set of values. Getting efficient incremental performance is a harder problem than it sounds. We have never gotten a production system that will do this well in our past attempts: rsync didn't seem to work well, MySQL's load data performance is destroyed by pre-existing unique indexes, and Oracle insert/update is slower than a complete transfer and rebuild for anything but the most minor of changes.

There are two ways we can think of to support this. The first is the easiest to implement, and just consists of creating a diff file in much the same way that Unix diff or rsync , and using it in combination with the existing data on the live server to create the new set (rather than deploying everything each time). Since computing the diff turns out to be a very computationally intensive task for a large file, this work must be done in the offline Hadoop system. There is little point in trying to do incremental updates to the .index files as these are comparably small, and the changes are liable to be randomly interspersed throughout–so the diff won't be much smaller than the original file. The .data files, however, do not have any inherent order so all the new data could be placed at the end of the file allowing for extremely efficient diffs. The "patch" could be applied in the process of doing the fetch, so that existing data segments would be read from the current *version-0* dataset and new segments would be read from the HDFS diff file.

The above strategy reduces the network transfer necessary and could be run in steady state each day. However it does still require writing the complete data set to disk for each deployment, rather than writing the smaller diff only. One strategy that could avoid this would be to create a seperate set of .index and .data files for each day and store each days data seperately on the live system. A naive lookup would have to check each of these directories from latest to oldest to retrieve a value. However a more sophisticated approach could keep a Bloom filter tracking which keys are in each day's patch. This would give a quick way to determine which files need not be searched without actually performing the search (with high probability).

Improved key hashing

The consistent hashing algorithm Voldemort uses has the nice property that N copies of a key are more or less randomly distributed over the cluster. As a result a failed node redistributes load evenly to the remaining nodes. This is an essential property to be able to tolerate node failure. However this algorithm has the unfortunate property that the data on each machine is different, and as a result the data built for each machine is unique. This means that a replication factor of two, doubles the size of the data that must be built. A hashing algorithm tuned to this use case could avoid this problem by replicating at the chunk level. This would provide less fine-grained load distribution since each chunk of data would be fully replicated on N machines, but would avoid the blow-up due to replication factor. Voldemort supports pluggable hashing algorithms so this should not be too difficult to implement, but didn't make the cut for our first attempt.

Compression

Because the data is known upfront and does not change, it is a good target for compression. LZO compression or another compression algorithm tuned towards fast decompression speed could improve performance by reducing IO.

Better indexing

Fancier index structures are another area for improvement. We think the probabilistic binary search will prove to be a very effective approach, but since we haven't implemented it yet it is worth considering a few other approaches.

The idea of a 204-way page-aligned tree instead of a binary tree was mentioned above. Each set of 204 20-byte index entries would take 4080 bytes which with 16 bytes of padding would then be exactly page aligned for a 4k page. This would mean the first 4k page of the index file would contain the hottest entries, the next 204 entries would contain the next hottest, and so forth. Thus even though the number of comparison necessary to locate an entry would not asymptotically improve the maximum number of page faults necessary to do an index lookup would decrease substantially in practical terms (to 4 or 5 for a large index).

This is not the optimal tree structure for an immutable tree such as ours, though. A much better approach to this problem was brought up by Elias who was familiar with the literature on cache-oblivious algorithms, and was aware of a cache-oblivious structure called a van Emde Boas tree.

Cache oblivious algorithms uses a data structure that recurses in on itself in a way that requires no assumptions or special treatment for page sizes or CPU caches to get optimal cache performance (under some reasonable assumptions). There is a well developed set of cache-oblivious data structures for dealing with disk-based tree lookups. These algorithms manage to nicely utilize CPU cache as well, all without explicit assumptions about the memory hierarchy.

Still another alternative would be an on-disk hash-based lookup structure. Such a structure can reduce the number of required comparisons in a lookup, though as with a tree, its creation could be difficult. The trade-off between extra space used in the hashing and the collision rate is well known. At the far end of this spectrum is the minimal, perfect hash which is a function that hashes a fixed set of *N* keys to exactly *N* hash table slots. This structure would seem to be optimal for the lookups since it guarantees that we will require no more than one lookup to find the location of the data (indeed we could entirely avoid storing the hash value itself reducing the index entry size to only 4 bytes for the position). The hash function itself requires only about 3 bits per entry to be stored once it has been found. Computing these hashes can be difficult, though, so only a real implementation would show if the superior lookup time was justified by the possible increase in build time. There is an off-the-shelf MPH implementation from the author of mg4j but we have not yet investigated the feasibility of this in much detail.

Posted by jay on Monday, June 15, 2009, at 11:28 am. Filed under Uncategorized. Tagged hadoop, linkedin, read-only.

Follow any responses to this post with its comments RSS feed. Comments are closed, but you can trackback from your blog.

{ 2 } Trackbacks

1. Hadoop and Project Voldemort « François Schiettecatte's Blog | June 18, 2009 at 12:48 pm | Permalink

   [...] Software Development by François Schiettecatte on June 18, 2009 Great article in two parts (part 1, part 2) co-authored by Elias Torres on using Hadoop and Project Voldemort for managing very large [...]

2. SNA Projects Blog : Beating Binary Search | June 16, 2010 at 11:55 pm | Permalink

   [...] as Voldemort stores. This allows us to support a big batch datacycle run out of Hadoop as described here. The data structure for these uses a large sorted index file to do lookups, what is stored in this [...]