Project Voldemort : Batch computed read-only data serving

Blah1
LinkedIn Corp
blah@blah.com

Blah2
LinkedIn Corp
blah2@blah.com

Blah3
LinkedIn Corp
blah3@blah.com

Abstract

Many internet based companies have started building analytics products backed by computationally intensive data-mining algorithms. One of the important system requirements for these products is the ability to present these results to the end user under strict latency constraints. A good systems practice to achieve this is by decoupling the computation layer from the serving layer. MapReduce paradigm, through the open-source solution Hadoop, has seen a wide spread adoption as the perfect solution for the computation layer. In this paper we present a distributed serving and storage layer called Project Voldemort. We have built and open-sourced this fault-tolerant layer with the ability to serve both read-write and read-only batch computed data. In this paper we will focus primarily on the batch data serving cycle and talk about how it has helped LinkedIn serve multiple TBs of data regularly. With its very simple key-value based interface Voldemort today powers most of our social recommendation products.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management; H.3.3 [*Information Search and Retrieval*]: Information Storage; H.3.3 [*Information Search and Retrieval*]: Information Search and Retrieval

General Terms Design, Performance

Keywords database, low latency, distributed, offline

1. Introduction

The product data cycle of a typical consumer web company consists of a continuous cycle of three phases - data collection, processing and finally serving. The data collection phase deals with gathering raw events like user activity, shares, etc. from various storage solutions as well as logging systems. The ability to horizontally scale with the increase in number of data sources is a tough problem and has been solved in the form of various distributed log aggregation systems. Some examples of these include LinkedIn's Kafka[**?**], Facebook's Scribe[**?**] and Cloudera's Flume[**?**]. Besides providing access to numerous data-sources these systems also provide pluggable sink interfaces, in particular the ability to push the data into batch processing systems like the Hadoop ecosystem (in particular HDFS)[**?**]. The Hadoop ecosystem then in-turn provides a diverse offering of query languages thereby providing the ability to run complex product driven algorithms as batch jobs. The size of the output generated by these jobs can end up being massive. For example in the context of social networks most algorithms tend to derive relationships between items (where items could be users or any other important facet). This sparse relationship data can get really huge when dealing with millions of items. The end goal of most of this processing is to surface insights which can then be provided back to the user.

In this paper we talk about our solution for the last phase of this cycle, called Project Voldemort, and how it fits into our product ecosystem. We've built this serving platform with the aim to quickly update the online index with new offline data and still have minimum latency impact for the live lookups. The tricky part of this system is the efficient movement of large amounts of the data, especially since we have constraints to refresh these 'insights' on a daily basis. At LinkedIn our largest cluster deals with multiple TBs of data and regularly pushes around 3 TBs to the site. Another important feature that we wanted to support was the ability to still deliver results under regular failures. We also built the system with horizontally scalability in mind so as to support addition of new products immediately. Voldemort has been running in production at LinkedIn for the past 2 years and has successfully been serving various user-facing features like 'People you may know', 'LinkedIn Skills' and 'Who viewed my profile'. One of the reasons behind its success and quick adoption within the company has been its simple key-value like API which makes it easy to test. This has helped our engineers to quickly iterate and now we have our largest cluster serving around 100 features (stores), most of which are serving our 120 million user base.

The rest of the paper has been structured into 5 sections. In Section 2 we talk about some of the existing solutions and how it compares with Voldemort's design. The next section,

Section 3, talks about our system architecture and its evolution. Voldemort is heavily inspired from Amazon's Dynamo[**?**] and was hence initially designed to only support the fast online read/write load. But its pluggable architecture, in particular the ability to add multiple storage engines, allowed us to build our own custom read-only storage engine and integrate with the offline data cycle. Section 4 goes into details about the read-only storage engine and the various design decisions we made while building it. In Section 6 we give details about our experiences while using Voldemort at LinkedIn. We will also talk more about performance results. We finally discuss future work and conclude in Section 7.

2. Related Work

The most commonly deployed serving system in various companies is MySQL. The two most used storage engines of MySQL - MyISAM and InnoDB - provide bulk loading capabilities into live system with 'LOAD DATA INFILE' statement. MyISAM is the better amongst the two because of its compact on-disk space usage and its ability to delay the re-creation of the index to a time after the load[**?**]. Unfortunately we still have to deal with the problem of needing lots of memory during the re-creation which in turn may affect our live requests performance. MyISAM also provides the ability to build compressed read-only tables [**?**] which can increase the speed of loading. So even though this decreases the load time, we still have no way to parallelize loads while also locking the complete table for the duration of the load. Obviously this doesn't work for us since applications at LinkedIn cannot have down-time.

A lot of work has also been done to add bulk loading ability to some new shared-nothing cluster[**?**] databases similar to Voldemort. In [**?**] tackles the problem of bulk insertion into range partitioned tables in PNUTS [**?**] by adding another *planning phase* to gather more statistics for the movement. Another paper from the same team [**?**] shows how they used Hadoop to batch insert data faster into PNUTS.

All of the above approaches try to optimize the performance on the current serving cluster during loads and update the indexes (B-tree in most scenarios) on the live system. Since we cater to data-sets that completely need to be changed at once the approach that we have adopted builds the full index offline. Using MapReduce for this purpose has been a well researched idea with various systems. For example various search systems, like Katta[**?**] and [**?**], have hooks to build their indexes in Hadoop and then pull the indices from HDFS to serve search requests. This idea has also been extended to various databases. Both [**?**] and [**?**] build HFiles (equivalent to Google's SSTable file - Persistent and immutable map file) offline in Hadoop and then ship them over to be consumed immediately by HBase. They bypass the conventional client API which would do something similar to MySQL i.e. first persist to log, buffer in memory and then flush periodically. The same approach has been adopted by Cassandra in its new *sstableloader*[**?**] functionality. These approaches definitely alleviate the cluster from sudden memory usage pattern changes due to index changes.

But all of the above solutions don't solve one very important use case of ours. The general development pattern that we've seen for data products is that engineers come up with new algorithm tweaks and then push out the corresponding data changes in steps, while continuously monitoring their metrics. The problem kicks in when a new push of data is resulting in a major drop in metrics. In such a scenario having to run a batch Hadoop job to push the data back again would be very time-consuming. Our novel storage layout tries to solve this problem by providing some form of rollback mechanism.

From an architecture perspective, Voldemort falls into the league of many previous P2P storage systems. There have been various structured DHTs, like Chord and Pastry, which provide O(log N) lookup. We are more inspired by Dynamo and Beehive which tries to decrease the variance in latency due to multi-hop routing by saving more state and providing lookups in O(1). Similarly our consistency mechanisms has resemblance to previous work done by Bayou[**?**] wherein we provide application level resolution in case of inconsistencies.

3. System Architecture

Before we start let us define some of the concepts and terms we will use in this paper. A Voldemort *cluster* can contain multiple *nodes*. All the nodes in the cluster store the same number of *stores* which correspond to database tables. General usage patterns have shown that a site-facing feature may map to either one store or multiple stores. For example, a feature dealing with grou ...

Voldemort has been inspired from Amazon's Dynamo paper and has a similar pluggable architecture. Figure 1 shows the architecture of Voldemort. Each box represents a module, all of which share the same simple interface. Every module has exactly one functionality, which we'll explain in the following sub-sections. This type of layering makes it easy to interchange modules and place them in any order. For example we can have the routing module on either the client side or the server side. Functionality separation at module level also allows us to easily mock these modules for testing purposes. One of the most common uses of this can be found in our unit tests where we mock the storage layer to use an in-memory hash map based engine.

3.1 Client API

- Server side transforms Starting from the top our client has a simple $get()$ and $put()$ API with no support for complex operations like joins, foreign key constraints, etc. We also support optimistic locking, which can be very useful for applications that require a 'read, modify, write if no change' loop (Eg. Counters).

3.2 Conflict Resolution

3.3 Routing and replication layer

Zone aware and Consistent routing

3.4 Consistency Mechanism

Read repair and hinted handoff

3.5 Storage engines

- Pluggable - BDB, in-memory, MySQL, Krati, read-only
  Talk about rebalancing of partitions

Now that we know the individual components, let us look at how Voldemort is used inside LinkedIn. As shown in Figure 2 we support both server side as well as client side routing. Client side routing gives us the benefit of having one less hop as the clients now have information about exactly where the replicas for a key are. This is beneficial from a latency as well as throughput aspect since we have less number of services acting as bottlenecks. The disadvantage of client side routing is it makes the client side code-base large because of the extra routing and replication layer. It also makes rebalancing difficult because we now need to update the cluster configuration on all the clients.

4. Read-only storage engine

The ability to have a pluggable storage engine allowed us to experiment with various storage engines for batch loading. Tried two approaches (Explain why they sucked)

- MySQL
- BDB

Came up with our own storage engine with the following requirements

- Ability to roll back
- Index building should not degrade performance of user-facing requests
- Scalable building and deployment
- Able to work on large data to RAM ratio

4.1 Versioning of data

4.2 Rollbacks

4.3 Storage format

- Memory footprint
- HFile, RCIndex, HIndex

4.4 Search

As suggested in [**?**], Interpolation search is a good fit for uniform distribution. Since md5(key) is uniform distribution we didn't explore other methods like Fast and Pegasus.

4.5 Rebalancing

5. Experience

- Random dataset - Run on member_id to uniform key of 1024 bytes - The time to completion increases linearly with the size of the dataset and is comparable between different types of data

  a) Single node - Increase in size of dataset - 1GB, 2GB, 4GB, 8GB, 16GB, 32GB, 64GB, 128GB [Metric 1] Time to build [Metric 2] Time to query (Median, 99th) - YCSB - 1 million keys - Uniform, Zipfian

  b) 16 node - Increase in size of dataset (128 GB, 256 GB, 512 GB, 1TB) [Metric 1] Time to build [Metric 2] Time to query (Median, 99th) - YCSB - 1 million keys - Uniform, Zipfian

- PYMK dataset a) 16 node - Time to build - Time to query (Median, 99th) - YCSB - 1 million keys - Uniform, Zipfian

6. Conclusion

Acknowledgments