Serving Large-scale Batch Computed Data with Project Voldemort

Prakhar Srivastav (ps2894)

Summary

Project Voldemort is a general purpose distributed storage and serving system inspired by Dynamo, to support bulk loading terabytes of read-only data. It constructs the index offline by leveraging the fault tolerance and parallelism of Hadoop.

Voldemort came out of a need to support the final phase of the feature data cycle at LinkedIn. After the data collection phase (log consumption) and the processing phase (running algorithms on the output) are complete a final phase is required to live serve this batch output even during data refreshes. By leveraging Hadoop's elastic batch computing infrastructure to build its index and data files, Voldemort can support high throughput for batch refreshes.

Architecture

The overall architecture of Voldemort was inspired from various DHT storage systems. Unlike the previous DHT systems, such as Chord [21], which provide O(log N) lookup, Voldemort's lookups are O(1) because the complete cluster topology is stored on every node.

A Voldemort cluster can contain multiple nodes, each with a unique identifier. All nodes in the cluster have the same number of stores, which correspond to database tables. Every store has a list of modifiable parameters through which it can change the cluster topology.

- Replication factor (N): Number of nodes which each key-value tuple is replicated.
- Required reads (R): Number of nodes Voldemort reads from, in parallel, during a get before declaring a success.
- Required writes (W): Number of node responses Voldemort blocks for, before declaring success during a put.
- Key/Value serialization and compression: The serialization format and compression settings to use.
- Storage engine type: One of the supported custom read-only storage engine for bulk-loaded data.

Other salient features of the architecture are

- *Modularity*: The pluggable architecture is consisted of modules each of which has exactly one functionality. This allows for each mock ups during testing phase as well.
- *API*: Voldermort's API is directly inspired by Dynamo DB's and exposes the same `get` and `put` methods.
- *Partitioning*: Voldemort splits the hash ring into equal size partitions, assigns them unique ids, and then maps them to nodes. This ring is then shared with all the stores.

- *Routing*: Voldemort supports two routing modes: server-side and client-side routing. Client-side routing requires an initial "bootstrap" step, wherein it retrieves the metadata required for routing by load balancing to a random node.

Data Storage and Read-only Extension

The lack of off-the-shelf solutions, along with the inefficiencies of the previous experiments, motivated the building of a new storage engine and deployment pipeline with the following properties.

- *Minimal performance impact on live requests*: The incoming get requests to the live store must not be impacted during the bulk load.
- *Fault tolerance and scalability*: Every step of the data load pipeline should handle failures and also scale horizontally to support future expansion without downtime.
- *Rollback capability*: To minimize the time in error, the engine must support very fast rollback to a previous good state.
- *Ability to handle large data sets*: Support terabytes of data and perform well under a large data to memory ratio.

To satisfy the requirements laid down above the workflow that was architected was as below

1. Leverage Hadoop as the computation layer for building the index as its MapReduce component handles failures while HDFS replication provides availability.
2. After the algorithm's computation completes, a driver program coordinates a refresh of the data.
3. It then triggers a build of the output data in our custom storage format and stores it on HDFS.
4. This data is kept in versioned format after being fetched by Voldemort nodes in parallel.
5. Once fetched and swapped in, the data on the Voldemort nodes is ready for serving

Conclusion

In conclusion, we can see that in this paper how the design team leveraged offline index generation coupled with a bunch of other optimizations to met the requirements they set out to solve. The key contributions of this work are -

- A scalable offline index construction, based on MapReduce, which produces partitioned data for online consumption
- Complete data cycle to refresh terabytes of data with minimum effect on existing serving latency
- Custom storage format for static data, which leverages the operating system's page cache for cache management