Exploring the Implications of Building a Column Store using a Key Value Store System

Sathyanarayanan Gunasekar, Kamiya Motwani, Karthik Narayan

Abstract

Key Value stores have been a rage these days. They attract the attention of the application developers since they guarantee very high availability and can scale to high numbers. However, they do not provide a rich API to the application programmers in most cases. The APIs are in the form of get and put operations. This is not suitable for OLAP queries which work on huge amount of data. On the other hand, column stores are very good for OLAP queries but do not perform well on scaling. Hence we try to analyze the implications of building a column store with a key value store as a building block. We first build a naive column store using Voldemort, which is a key value store, and then improve the performance of this column store by introducing some indexing mechanisms for aggregate queries over a range. We also use machine learning to build an intelligent column store that performs better than the naive column store.

1 Introduction

Key Value stores are widely used in all web applications. They are used in large systems like Dynamo [3], running across nodes geographically separated and also small desktop applications which run on a single machine. The reason being that, key value stores are fast if you know which data to access exactly. They provide an access interface through keys. Companies like Facebook and Google have their own versions of key value stores like Cassandra [4] and BigTable [1] respectively. These systems are in contrast to the traditional RDBMS where queries are relational in nature. Users generally specify predicates and conditions, and the RDBMS fetches the data satisfying them. Most of the RDBMS store the data in a row oriented fashion, but some adopt the approach of storing them column wise [5]. This helps them serve OLAP queries, which are generally read-only and do not work on all attributes of a single tuple. These queries have strict latency requirements and need high availability but at the same time need efficient scaling since they work on large amount of data. This is guaranteed by key value stores, but at the same time, these stores do not accept queries in the OLAP format. Building a column oriented database which also has the features of a key value store would be ideal for OLAP based applications.

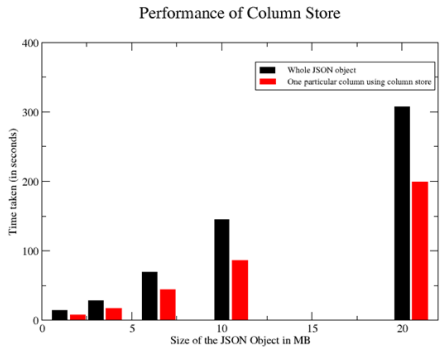

Figure 1: Performance of Voldemort on different sizes of JSON object

We used the Voldemort key value store to implement a column store over it. It is basically just a big, distributed, persistent, fault-tolerant hash table. It is not a relational database and does not attempt to store arbitrary relations while satisfying ACID properties. Voldemort combines in-memory caching with the storage system so that a separate caching tier is not required. We provided an API over the existing get/put methods that vertically partitions the tuple into columns and stores it onto different stores. It fetches from each of the stores when a whole tuple is requested.

In Section 2, we describe a naive approach to build such a column store. In Section 3, we present some indexing mechanisms that we used to make the key value store service aggregate queries over a range. In Section 4, we explore machine learning techniques to improve the naive column store approach. We finally conclude in Section 5 with our results and some future directions.

2 Building a Naive Column Store

We build a naive column store by first observing the performance of a key value store system for JSON objects of different sizes, as in Figure-1. Based on these observations, we conclude that as the JSON object size grows, the latency to fetch it also increases. Hence, in cases where only a part of the JSON object is relevant to the user query, a lot of time is wasted piggybacking the other irrelevant attributes. For example, a JSON object representing a user profile can store various attributes like userid, hash of the password, details like name, age, sex, tracking info etc. During a login, only the user-id and the hash need to be fetched to authenticate the user. However, in the traditional key value system, the whole JSON object representing the user would be fetched. This can be improved by sharding the user information into separate key value stores, one for each attribute and then fetching only the relevant attributes from the corresponding attribute store.

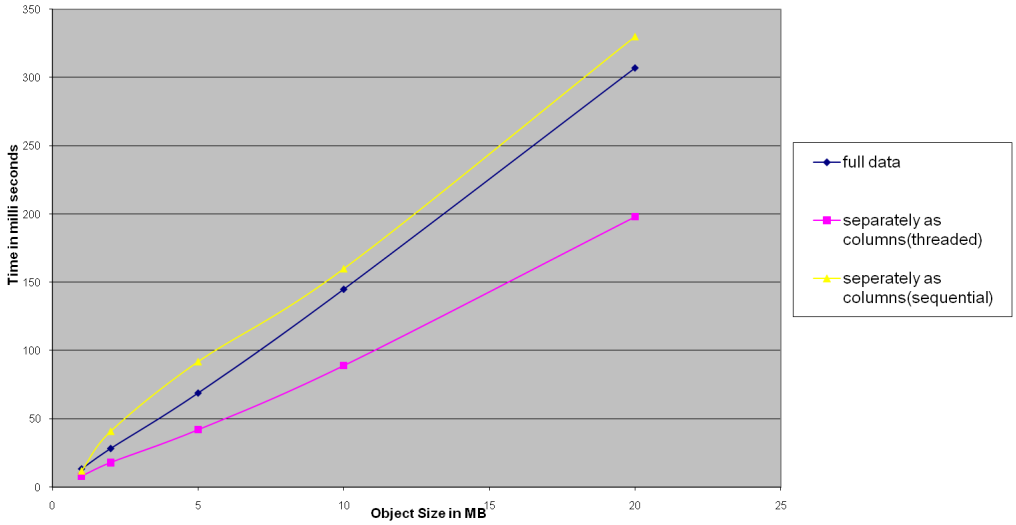

Figure 2: Performance of naive column store

To build such a system, on a put request for a tuple t that is identified by a key k , N put requests to N different stores are generated, where N is the number of attributes in the tuple (in this case a JSON Object). Similarly on a get request for the entire tuple, N different get requests are generated. It is notable that one need not require all the fields of a tuple while issuing a get request and hence the response time can be improved. Hence we provide a getf API, where the user can specify the attributes which he specifically wants for a key k. In such a request, the underlying get requests to different stores can be issued asynchronously so that the time to get all the fields of a record is equivalent to a get request for a single attribute.

We measured the response time for querying a specific attribute value in the traditional key value store approach wherein all the attribute values are stored at one store and also in a column store environment where each attribute is stored in a different store. We notice(as expected) that the column store approach guarantees better response time as the unnecessary attributes are never fetched from the persistent store. We issue get request for both single attribute value and whole JSON object in Voldemort running on a single node cluster. The summary of the results are presented in Figure-2.

It is noteworthy that we have just provided an abstraction for column store over key value store till now. We do not store the column values together to enable fast OLAP queries that involve aggregations and selections. Hence we need an efficient indexing mechanism to be equipped to fetch the attribute values in a given range within a deterministic low latency.

3 Indexing a Key Value Store

Implementing a naive column store gives better performance on get requests since we only fetch relevant parts of the data and not the whole object. However, the user still has to be specific on what he needs and cannot issue OLAP type queries. For example, a user can fetch all details of a product using the product id as a key. But he still cant get all products which are manufactured by a particular manufacturer or all products which lie in a particular price range. These are typical OLAP type queries and to serve them via a key value store, we need to maintain some summary or meta data associated with the original data. To solve these, we incrementally describe indexing approaches to service *select-by-attribute-value* queries and *aggregate* queries over a range. Throughout this section, a store refers to a key value store. We target two specific query formats which are briefly described below.

1. **Selection by attribute value** : These queries are of the form *select * where* $attribute = value$. They look for specific items satisfying the predicate.
2. **Aggregate queries over a range** : These queries are of the form *select COUNT where* $X < attribute < Y$. They perform aggregation operations like count, sum, average over a range of tuples which satisfy the predicate.

3.1 Inverse Index

To service the queries of first type, we maintain an inverse index on the corresponding attribute. Hence each original store has a index store built on a particular attribute. An inverse index on a *(key,value)* is a pair *(attribute-value,key-list)*, where the value of the attribute we index on is used as key in the index key value store. The key-list that we maintain is a list of the keys corresponding to the original store. Note that for a given attribute value, there can be multiple keys of the original store and hence we use a key list. This way, searching for objects matching a particular value on a attribute can be done in constant time. These indices are very easy to build and can be updated as and when put operations are performed on the original store. As expected an inverse index performs in constant time compared to brute force approach which iterates through the entire original store. We plot the time taken against the number of objects in the store in Figure-3.

The inverse indexing however performs equally bad on queries of the second type. This is because the query range could be huge and sparsely populated. In such cases, each and every value in the range has to be searched for in the index store and the corresponding aggregate action would have to be applied on the selection. This performs badly even if we maintain a inverse index. To tackle this, we use another indexing strategy traditionally adopted in RDBMS systems like [2] to maintain statistics.

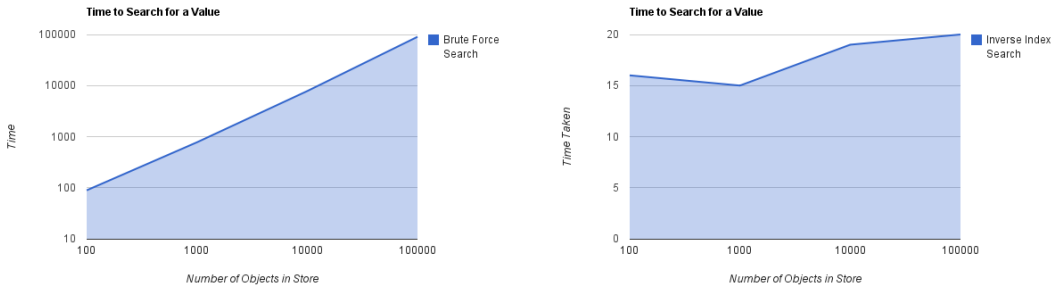

Figure 3: Performance on queries of Type-1

3.2 Bucket Index

The main idea behind bucket index is to maintain the keys of the original store in buckets which represent ranges. The buckets could be of *equi-depth* or *eqvi-width*. Equi-width buckets store the keys partitioned over static ranges of equal width. When more keys are added, buckets can become crowded and due to skew in distribution, some buckets will be more populated than others. Hence we use the equi-depth buckets where the range representing an individual bucket can vary with time, but the maximum size of each bucket is fixed. When a bucket reaches its maximum limit and overflows, the range corresponding to the bucket is altered to fit the bucket and a new bucket is introduced to hold the overflown items. All the buckets are associated with a bucket key in the bucket store. The information about ranges and the corresponding bucket keys is stored in a special key called the *Master Bucket Record key(MBR)*. The MBR also stores the summary information for the ranges to improve the performance. We can represent both the MBR data and the bucket using a single JSON object representation.

Any aggregate query, would first fetch the MBR and try to figure the buckets which fall in the query range. For those buckets which completely lie in the range, it uses the pre-computed summary information and for the buckets which overlap partially with the query range, the bucket is fetched using the corresponding bucket key and relevant aggregate operations are performed. Thus, any range query needs to issue at the maximum $2T + 3$ number of get requests, where T is the size of a bucket. By varying T, the performance can be improved according to the granularity of query ranges. Since bucket sizes are smaller compared to the entire range of values corresponding to an attribute, we achieve better performance compared to the inverse indexing.

However, the main challenge is to handle the cases of overflow while insertion. Our implementation again guarantees that there be will be puts proportional to the number of the buckets maintained in the worst case. As we can see in Figure-4, the bucket indexing

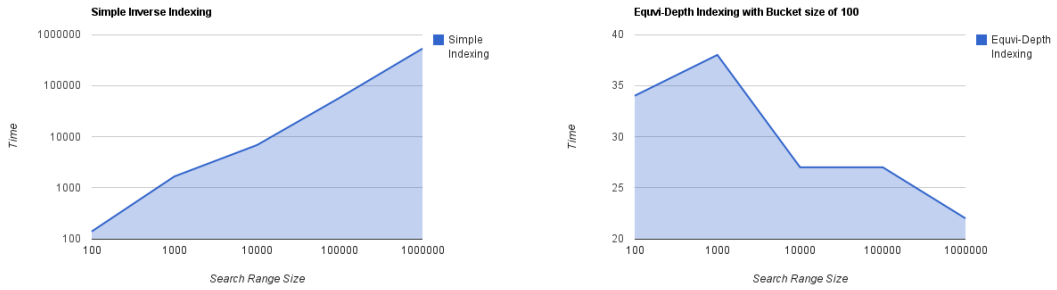

Figure 4: Performance on queries of Type-2

performs in constant time compared to the inverse indexing on aggregate queries over a range. We vary the size of the range and plot the time taken for different ranges.

The equi-depth has a negative slope since for bigger query ranges, the buckets that partially overlap tend to hold very few items on an average compared to the same buckets for smaller query ranges

4 Intelligent Sharding using Machine Learning

In this section, we explore opportunities to improve the performance of a column store by using some machine learning. Using the naive approach to completely shard the data can result in transforming the data from a fat table to a thin one. This could lead to serious performance degradation when the user always queries a set of attributes together. Determining the optimum method for sharding the data can be crucial to the performance of a database. The sharding strategy changes from one application to another. A good sharding mechanism should decide how to shard the data considering factors like transaction rates, table volumes, key distribution, and other characteristics of the application. We use machine learning to decide how to partially shard the data. Learning it based on the training set coming from the real world queries might prove to be better than naively decomposing the table. This could later be scaled to have a feedback mechanism which improves the selection based on the user feedback.

We used greedy forward selection mechanism to intelligently partition the data. We divide the query set into training and test for our experiment purposes. Greedy selection is an alternative to exploring exhaustive enumeration of the ways in which the data can be sharded. We restrict our search space by specifying two key parameters which are: the maximum number of partitions and the maximum size of a single partition. Even though it is sub-optimal, it is a popular technique as it is faster.

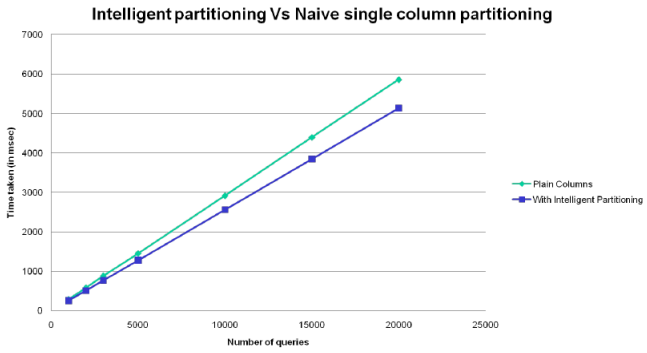

Figure 5: Performance of Intelligent Sharding

4.1 Greedy Forward Attribute Selection

The algorithm we implemented is as follows :

1. Let S = ϕ be the current set of selected column and Q be the full set of columns.
2. While size of S is smaller than the max. number of partitions specified by the user or we exhaust all columns :
   (a) For each v $\in$ Q
      i. set S' = {v} $\cup$ S
      ii. Train the database with S' and keep the performance
   (b) Set S = { v* } $\cup$ S where v* corresponds to the best performance obtained in above step
   (c) set Q = Q \ v*
3. Return the best set P

For our experiment, we used the frequency at which subsets of columns are accessed together as a performance criteria. However, one could also use a performance criteria based on various other factors like improving the average time, improving the worst case latency or improving the best case latency. After getting the set of partitions to shard the data based on the above algorithm, we used those partitions to serve the remaining queries. The graph in Figure-5 shows the performance of doing intelligent sharding using greedy approach versus sharding the data on all possible attributes. Clearly, with the increase in the number of queries, intelligent sharding performs better than naive sharding for the random query set we generated.

5 Conclusion

We have successfully built a column store using a key value store. To support OLAP queries, we have extended the basic interface provided by Voldemort as an abstraction to the user which use inverse indexing and bucket indexing. These indices can be built on the fly unlike traditional column stores where write operations are only performed at once and any updates after that is costly. To improve the performance of the column store, we have used machine learning to intelligently shard the data into columns. We have observed through our implementation that one can build linked data structures through key value system, using keys as pointers. Hence it is possible to build more complicated indexing mechanisms like interval trees or B-trees that make use of this and can serve OLAP queries efficiently. Another interesting aspect that could be explored is how to use the user feedback and improve the sharding of the column store system we built.

References

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (Berkeley, CA, USA), USENIX Association, 2006, pp. 15–15.

[2] Surajit Chaudhuri, *An overview of query optimization in relational systems*, Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (New York, NY, USA), PODS '98, ACM, 1998, pp. 34–43.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, *Dynamo: Amazon's Highly Available Key-Value Store*, Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (New York, NY, USA), SOSP '07, ACM, 2007, pp. 205–220.

[4] Avinash Lakshman and Prashant Malik, *Cassandra: Structured Storage System on a P2P Network*, Proceedings of the 28th ACM symposium on Principles of distributed computing (New York, NY, USA), PODC '09, ACM, 2009, pp. 5–5.

[5] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik, *C-Store: A Column-oriented DBMS.*, VLDB (Klemens Bohm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Ake Larson, and Beng Chin Ooi, eds.), ACM, 2005, pp. 553–564.