**Topic: Apache Kudu**

**Name: Vijet Muley**

**E-mail Address:** [**vmuley@syr.edu**](mailto:vmuley@syr.edu)

**SUID: 299366058**

**What is Apache Kudu?**

Apache Kudu is a data storage system from the Hadoop ecosystem that works on structured data (n.d., 2019). Kudu is a sweet juxtaposition of properties from both, HBase and HDFS in order to carry out analytics over rapidly changing data. Despite being structured database based upon HDFS, Kudu enables both rapid access of data rows and execution of analytical procedures on the data to examine patterns. Kudu is a column-oriented data storage system. Kudu’s first version was released on 19th September 2016 as an open source project to public.

**Why was Kudu made?**

Hadoop has been around as a result of the increase in production and consumption of data. Hadoop became the go-to system for big data due to its provision of massive scalability. Hadoop has many loosely coupled elements such as Solr, HBase, HDFS etc. Thus, a Hadoop ecosystem can be using one or more of these elements as per the need. Data in Hadoop is stored in forms of files in a file system. The files are usually Paraquet, Avro etc. binary formats to store static data. These files are good for storage, rapid scans and trying to be read in bulk for analysis. On the other hand, rapidly mutable data that needs to be changed frequently row wise and read randomly uses HBase or Cassandra. Often, users had to develop pipelines or backend programs when they wanted to both store data in a static manner as well as analyze the incoming data for patterns. The reason is, HDFS files fill up in form of a block file. As in, the data would fill up a block file and once the block file fills up to its size, it spills into HDFS. On the other hand, HBase involves ingestion of data as it comes. These systems developed by the users had a few downfalls. The users need to make sure that there is consistency on both ends. They also need to make sure of security policies being consistent, reducing the lag between the 2 elements of Hadoop to as low as possible. Also, updating data on HBase end is easier than the same on HDFS end. That adds up to another issue that was faced while using this solution.

As a result, Apache came out with a new system, Kudu, to achieve both the goals of storing the data with an intent of rapid reads and writes and to analyze the stored data like OLTP data. Introduction of Kudu, however, does not reduce the capabilities of the individual systems that it tries to provide an alternative for. Kudu simply tries to find a balance between the 2 cases. (Lipcon et al., 2015)

**CAP theorem:**

A database is classified using CAP theorem where, C stands for consistency of data, A stands for availability of data and P stands for partition tolerance for the system. Apache Kudu is a CP system wherein the data will be partitioned over multiple nodes in the Hadoop environment and will be consistent at the expense of instant availability.

**Kudu Data Model:**

Apache Kudu is a column-oriented storage system and it is designed to store structured data, meaning it has a defined data model and user-defined schema to be implemented. The data is stored in form of tables. Every table has finite number of columns. Columns can have following data types:

a). int8

b). int16

c). int32

d). int64

e). unixtime\_micros

f). single-precision

g). double-precision

h). decimal

i). UTF-8 encoded string

j). binary

Kudu also allows column compression in order to save storage space. However, the trade-off to compressing data is increase in raw scan time. Kudu uses LZ4, Snappy or zlib compression codecs. By default, columns that are Bitshuffle-encoded are compressed using LZ4 compression method. If the columns are not Bitshuffle-encoded, then they are left uncompressed.

Kudu allows using one column or a subset of columns as primary key in order to uniquely identify rows. Every table must have a primary key and the primary key, because it is used to identify individual rows, follows uniqueness constraint. Primary keys cannot be null, they must contain a value. After creating a table, the set of primary keys cannot be altered to add or remove columns from the subset. Unfortunately, Kudu does not have an auto-increment feature for primary keys unlike relational databases. The tables in Kudu are clustered indexed on primary keys by default. (Cloudera, n.d.)

To perform write operations on Kudu tables, the primary key needs to be specified completely, and not in parts. Although Kudu will batch multiple mutations in order to have efficient performance, every individual mutation will be treated and executed as a separate transaction. For read operations, users may use predicates that either define a range for primary keys or compare values in a certain column with a constant value of the same data type. Users can also choose to select a subset of columns from table upon reading. (Zabavskyy, 2017)

As per Kudu’s website, certain guidelines must be followed in order to attain perfect schema for a Kudu table. The reads and writes must not be unbalanced on a few tablet servers, hence, must be evenly spread across all the tablet servers. Similarly, addition of new data must not skew one or a few of the servers. These 2 concerns can be balanced by implementing a proper portioning strategy. The design and strategy implemented must also allow for the least number of possible reads before extracting data as a result of a query. This will involve partitioning and proper design of primary key.

**Apache Kudu Cluster Architecture**

Tables in Kudu are partitioned horizontally, and every partition is called a tablet. For a cluster architecture, Kudu follows in the footsteps of BigTable by using a single Master server and arbitrary number of tablet servers dependent on the Master server. The Master server stores the metadata while the tablet servers store the actual data. Usually, Master server can be replicated so that in case of a node failure, the responsibilities of directing incoming queries of data to tablet servers can be transferred to backup Master server nodes. (Cloudera, n.d.)

Kudu’s Master server serves 3 roles: (Lipcon et al., 2015)

a). Catalog Manager

Catalog manager itself acts like a table with one tablet, which cannot be accessed by the user. Catalogs the tables and tablets that exist, the schemas that are implemented for these tables, replication levels for different tables and other metadata. The catalog manager is responsible to facilitate the alterations made to tables in Kudu.

b). Cluster Coordinator

The Master server, as the tablet servers are dependent on it, needs to monitor which tablet servers are active and which are not active. In case of node failure, the Master server needs to redistribute and redirect queries and data.

c). Tablet Dictionary

Master server needs to keep track of which tablet server(s) host the replicas for different tablets.

**Partitioning Strategy**

Kudu works on a distributed system, that is Hadoop. In order to facilitate the scalability associated with Hadoop, Kudu supports partitioning. In Kudu, the tables are partitioned horizontally into subsets called tablets. Tablets are distributed over multiple tablet servers and are usually replicated. Tablets are made by assigning ranges of rows to a certain tablet and its respective replicas. The ranges are defined by contiguous range partition keys. It is to be noted that a single row belongs to a certain tablet and cannot occur in a separate tablet. In other words, the ranges of partition for tablets can never overlap.

Efficient and proper usage of Kudu requires balanced number of write operations over the number of tablet servers in usage and that will require proper partitioning. Similarly, in case of having operations that involve small read operations, the time required to contact a remote and busy tablet server will outweigh the time required to do the actual read. Proper partitioning strategy will enable resolve this bottleneck. There are 2 simple partitioning strategies provided by Kudu:

a). Range Partitioning

b). Hash Partitioning

Kudu also allows multilevel partitioning as a result of combination of range and hash partitioning or multiple instances for hash partitioning. In this case, however, the hashing and ranges should occur on different columns. (Cloudera, n.d.)

Range partitioning occurs on totally ordered partition keys. User select a subset of primary key columns to make the range partition key. The split values then form partitions and based on the value of this split and the range partition key, the rows are distributed to different tablets. To initialize partitioning, user specifies set of bounds at the time of table creating. For every bound value, a corresponding partition is created. The next bounds then split the partitions. For range partitioning to work, the partitions must always be non-overlapping for, logically, one row will fall under only one partition. Range partitioning allows dynamic addition and removal of partitions. Removing an existing partition abstains from addition of new data into that partition. Also, it will delete the data that existed in that partition. That data can be re-added into another existing partition or a new partition. While creating a new partition the user needs to make sure that the new partition does not overlap any existing ones. There are 2 types of range partitions, bounded and unbounded.

Eg.: For data with date values column, user can split the data by year. Say the data ranges over the years 2010-2019. The user can select unbounded range partitioning with splits at 2014 and 2016. Thus, the data partitions will be:

Partition 1: Data before 2014

Partition 2: Data from 2014 to 2016

Partition 3: Data after 2016

If the user were to make bounded partitions, the data would be like:

Partition 1: Data from 2010-2014

Partition 2: Data from 2014-2016

Partition 3: Data from 2016-2019

When reading over range partitioned data for execution of a query, the predicates used should always be inequalities in order to use the ranges. (For eg. year<2014)

Hash partitioning will involve distribution of rows into different buckets based on hash values. The hash values are derived, usually, from the primary key columns. Either all the primary key columns or subset of primary key columns can be used. Range partitioning distributes data serially. Hash partitioning won’t do it necessarily. In order to have randomized distributions of data to reduce overloading of a select few tablet servers, hash partitioning will work better than range partitioning. The number of buckets to be created is decided at the time of table creation That constrains hash partitioning as it will not allow dynamic addition of removal of buckets.

Kudu allows combination of these partitioning strategies in order to implement multilevel partitioning. One or more than one hash partitioning can be combines with a range partitioning. One thing to remember while implementing multiple hash partitioning is that different levels of hash partitioning must hash different columns. The benefit is that it can use benefits of either or both the methods.

Kudu’s partition pruning is the method of skipping entire partitions and saving time while reading by comparing the comparable value from predicates with hashes or range values. Usually, for hash partitioning, equality can result in partition pruning. While for range partitioning, equality or inequality signs would suffice.

Kudu offers a choice from 2 consistency modes for read operations:

a). READ\_AT\_SNAPSHOT

The data read is corresponding to a certain timestamp provided

b). READ\_LATEST

The system will return the data that is committed at the time of query’s execution. Kudu uses this mode by default.

In case of write operations, writes are internally consistent for individual tablets. However, this consistency is not guaranteed outside that tablet. In order to work around this issue of uncertainty in case of write inconsistency, Kudu offers to send the timestamp along with the data. (Zabavskyy, 2017)

**Competing Products**

Apache Kudu was not made specifically to replace either HDFS, Impala or HBase. However, it was made to attain a balance between the advantages from these 2 polar systems. HDFS and Impala are used to enforce schema over static data stored in Parquet or Avro files and then analyze the static data. HBase is used to monitor the streaming data rather than static data. Apache Kudu provides low latency operations over moving data, like HBase. It also provides ability to run analytical procedures over the fast-paced data flowing through it, synonymous to Impala. Hence, the competition to Apache Kudu are both HBase and Impala. However, this competition is case specific. Apache Kudu can be used to have an efficient usage over Hadoop if the user’s requirements comprise of having structured schema over fast moving data with necessity of rapid read and write operations and implementation of analysis methods. In such cases, Apache Kudu outperforms the system made from amalgamation of HBase and HDFS using a pipeline. However, if the user’s requirements lie in only either of the areas (either analyzing static data or rapid reads and writes over fast moving data) then Kudu cannot outperform the specific systems used.

**Limitations and Issues**

Apache Kudu has been reported to have large number of changes over different versions. It allows a few numbers of data types for columns thus making it less flexible for variety principle of big data. Multi-row transactions are prohibited. As mentioned earlier, each mutation is treated as an individual transaction. (Zabavskyy, 2017)

Kudu allows only up to 300 columns per table and the documentation mentions that the number should be maintained below this limit for efficient performance. Primary key columns cannot be updated after the addition of a record. Also, the subset of columns determined to be primary key cannot be altered in terms of data types or cannot have columns added or removed from it. Kudu won’t allow alteration of column data type nevertheless; either it be a part of the primary key or not. Partitions are immutable with the only exception of range partitions. Once partitions are made, they cannot be split further or combined. (Cloudera, n.d.)

Kudu also does not a SQL like interface. It is just a storage system and it requires usage of Impala or other query languages to work on it.

**References**

Lipcon T., Alves D., Burkert D., Cryans J., Dembo A., Percy M., Rus S., Wang D., Bertozzi M., McCabe C. P., Wang A. (2015). Kudu: Storage for Fast Analytics on Fast Data. <https://kudu.apache.org/kudu.pdf> .

Cloudera. (n.d.). Apache Kudu Schema Design. Retrieved from <https://kudu.apache.org/docs/schema_design.html>

Zabavskyy, A. (2017, March 10). A Closer Look at Apache Kudu. Retrieved from <https://www.slideshare.net/andrijz/a-closer-look-at-apache-kudu?from_action=save>

n.d. (2019, December 4). Apache Kudu. Retrieved from <https://en.wikipedia.org/wiki/Apache_Kudu>