Technological Organization in Big Data Analysis with Apache Kudu Analytical Tool

Assoc. Prof. Dr. Pavel Petrov
University of Economics - Varna, Varna, Bulgaria
petrov@ue-varna.bg

Prof. Dr. Julian Vasilev
University of Economics - Varna, Varna, Bulgaria
vasilev@ue-varna.bg

Dr. Stefka Petrova
University of Economics - Varna, Varna, Bulgaria
s.petrova@ue-varna.bg

Dr. Liliya Mileva
University of Economics - Varna, Varna, Bulgaria
l.mileva@ue-varna.bg

Abstract

The main purpose of this article is to present the Hadoop Analytical Tool Apache Kudu in connection with the creation of methods and models for big data analysis in the field of construction. The first part of the document provides guidelines for creating a data model with the Apache Kudu analytics tool, as well as guidelines for creating methods for analyzing different types of data with the Apache Kudu analytics tool. The second part presents appropriate methods and models with analytical tool Apache Kudu in the business field of construction and ICT prototype solution with analytical tool Apache Kudu in economic field of construction.

Keywords: big data analysis, Hadoop, Apache Kudu, analytical tools

JEL Code: C88, O32

Introduction

Apache Hadoop's Hadoop Distributed File System (HDFS) is suitable for efficient storage of large data. A key feature of HDFS data storage is that once saved, the data cannot be changed, only read and delete operations are possible. Also, the architectural features of HDFS do not allow for effective real-time big data analysis. For real-time analysis, it is appropriate to use the Apache HBase system, which uses HDFS, but the time to crawl the recorded data volumes is large. Apache Kudu can be used to overcome these problems (Fig. 1).

Kudu was established in 2015 by Cloudera as an internal project to create a new columnar database management system of the NoSQL type and supporting SQL for operational analysis of rapidly changing big data. In 2016, the source code was distributed under an open license by the Apache Software Foundation.

The main purpose of Apache Kudu is to work with fast-changing data. There are options for both quick add operations and for updating and deleting rows in real time. Relatively fast crawls of columns for real-time analysis of one layer for data storage are performed (Fig. 2).

Figure 2 shows the elements of the architecture of the Apache Kudu tool, where the elements are depicted, in light (yellow) are shown the leading Master and Tablet, and in dark (gray) - the followers (equivalents), equivalent to Master and Slaves in Hadoop (for more information: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html).

The considered features of the analytical tool Apache Kudu aim to facilitate the work of specialists dealing with big data analysis. It should be noted that despite the known limitations and

peculiarities of working with Apache Kudu and the strict sequence of operations, the tool allows flexibility in operation and integration with other data analysis tools in Hadoop (Impala, MapReduce, Spark, Flume).

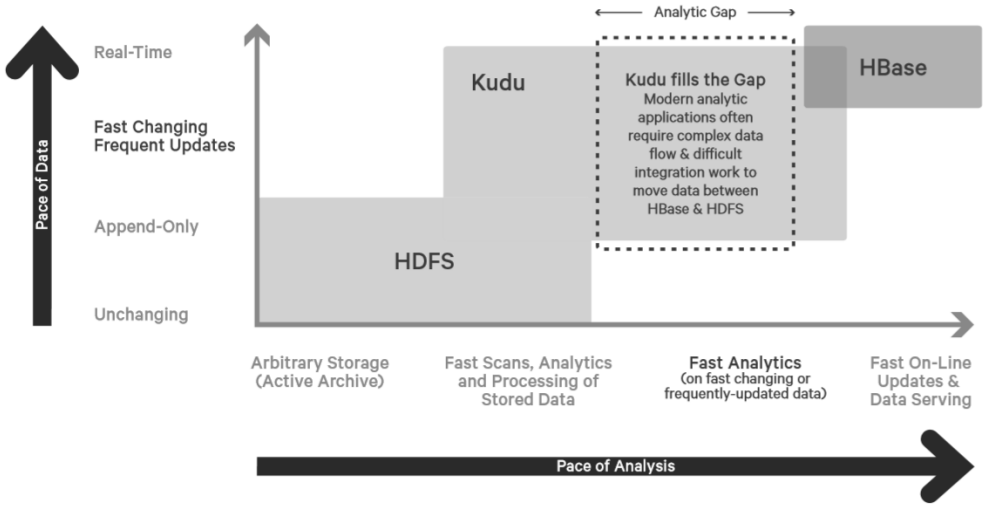

Figure 1. Apache Kudu's place in relation to HDFS and HBase in big data analysis via Hadoop.

Source: Apache Kudu Datasheet (2017). https://www.cloudera.com/content/dam/www/marketing/resources/datasheets/cloudera-kudu-datasheet.pdf.landing.html.

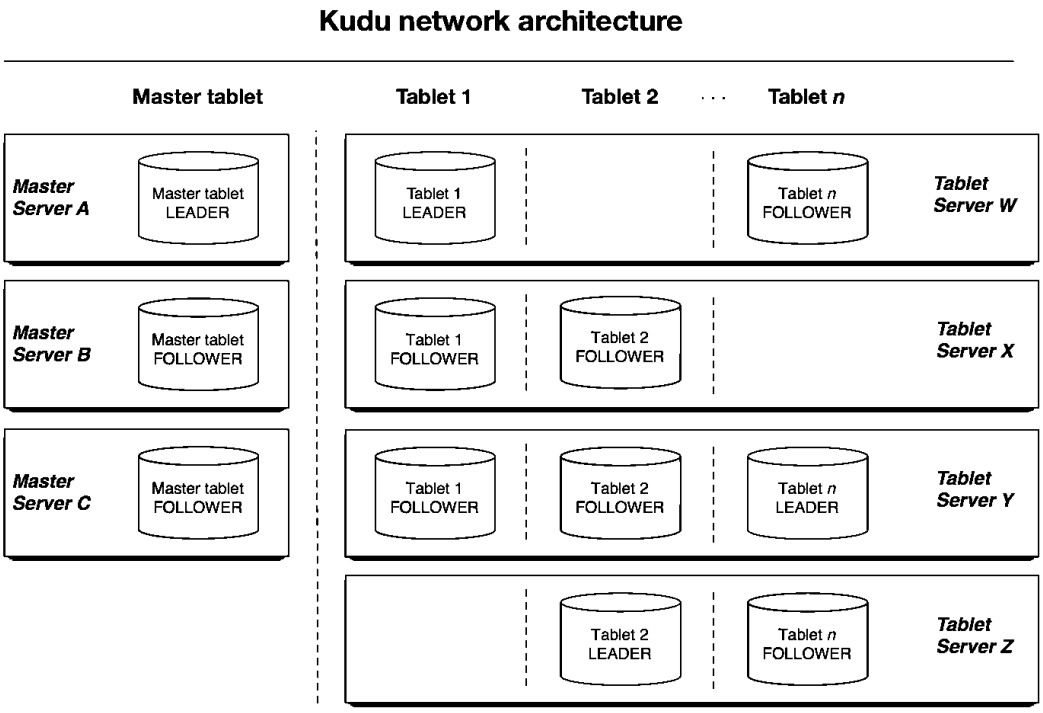

Figure 2. Architecture of the Apache Kudu.

Source: Introducing Apache Kudu. Architectural Overview. https://kudu.apache.org/docs/

Apache Kudu is a top-level project in the Apache Software Foundation that can be used as a column storage manager developed for the Hadoop platform. Kudu shares the general technical features of Hadoop's ecosystem applications, running on stock hardware, being horizontally

scalable, and supporting highly accessible operations.

1. Advantages and limitations

The main **advantages** of using Apache Kudu are:

- Fast load processing, full integration with Apache Impala, as well as high performance, so it is one of the most preferred analysis tools in Hadoop, as a good, changeable alternative to using HDFS with Apache Parquet.

- Fast processing of OLAP workloads - OnLine Analytical Processing - OLAP technology for data discovery, including possibilities for unlimited review of reports, complex analytical calculations and forecast planning of the type "what will happen" (budget, forecast); https://olap.com/olap-definition/).

- Developed to integrate with MapReduce, Spark, Flume and other components of the Hadoop ecosystem.

- Strictly defined, yet flexible sequencing model, which allows on-demand consistency requirements to be selected, including a strict sequence option.

- High productivity at simultaneous execution of consecutive and arbitrary loadings;

- Easy to administer and manage via Cloudera Manager;

- High availability. Tablet servers and Masters use the Raft consensus algorithm, which provides availability as long as more replicas are available than not. Readings can be served by sequential read-only tablets, even in the event of a leading tablet failure.

The main **limitations** that professionals should consider when working with Apache Kudu are the following:

- The primary key cannot be changed after creating a table. A table must be run and recreated to select a new primary key.

- The columns that make up the primary key must be listed first in a diagram.

- Automatically generated primary keys are not supported.

- For cells and columns: no single cell can be larger than 64 KB before encoding or compression. The cells that make up a composite key are limited to a total of 16KB after Kudu's internal composite key coding. Inserting rows that do not meet these limits will return errors to the client.

- Tables in Kudu can have a maximum of 300 columns, so schemes that use fewer columns for best performance are recommended (for more information: https://docs.cloudera.com/documentation/enterprise/latest/topics / kudu_limitations.html).

- Does not support relational functions, such as foreign switches;

- Tables must first be divided into tablets using simple or complex primary keys. Automatic splitting is not yet possible.

By combining all of these features, Kudu aims to support applications that are difficult or impossible to implement in Hadoop's currently available storage technologies. Applications for which Kudu is an appropriate solution include:

- Reporting applications where the new data must be immediately available to end users.

- Time series applications that must support requests for large amounts of historical data while returning detailed requests for an individual site.

- Applications that use predictive models for real-time decision making, with periodic updates of the predictive model based on all historical data.

These applications (Radev & Aleksandrova, 2013) are typical in the field of logistics, where it is necessary to track the movement of goods and goods from one point to another, and throughout the process they are monitored continuously in real time.

In view of the above, we can summarize that the main points in terms of concept and architecture are the following:

- Kudu is a column database, a data warehouse storing data in strictly typed columns.

- Kudu is characterized by reading efficiency, as for analytical queries one or part of a

column can be read, ignoring other columns. Or, the query can be executed while reading a minimum number of blocks on the disk. This gives Kudu a significant advantage over row-based repositories, as this type of repository must read the entire row, even if it returns values from only a few columns.

- Kudu works with Tables, Tablets, Teblet servers and Masters.

In Kudu, data is stored in tables, the table has a schematic and a fully ordered primary key. The table is divided into segments, called tablets, by primary key. A tablet is an adjacent segment of a table, similar to a partition in other storage mechanisms or relational databases. A tablet replicates on multiple tablet servers, and at any given time, one of those replicas is considered a master tablet. Each replica can read services. Enrollment requires consensus among the set of tablet servers serving the tablet. Tablet servers, in turn, store and serve customers' tablets. For a tablet, one tablet server acts as a leader, and the others serve followers of that tablet's followers. Only leaders leading a particular service write queries, while leaders or followers of each service can read queries. Leaders are elected by Raft consensus. One tablet server can serve multiple tablets, and one tablet can be served by multiple tablet servers The Master tracks all tablets, tablet servers, catalog tables, and other metadata associated with the cluster. There can be only one acting Master at a time. If the current Master disappears, a new leader is selected using an algorithm to determine the next Master (Raft consensus - for more information: https://raft.github.io).

2. Creating a data model with the Apache Kudu analytics tool

The data model when working with the analytical tool Apache Kudu is realized through tables.

Tables in Kudu are similar to tables in traditional RDBMS, and their design is important for high performance. It is necessary to approach each case individually, as there is no common design approach that is suitable for each table. Important points in designing the structure of the tables are setting the appropriate data type of the columns, selecting the primary key and allocating the columns in different tables.

Tables in Kudu are divided into parts called tablets, which are distributed among many tablet servers. An entire row of a table is always located on one tablet. The way the rows are distributed on the tablets is set when creating the table.

In order to choose an appropriate partitioning strategy, it is necessary to know the data model and the expected operations that will be performed with the table - mostly reading or mostly writing. In treatments which mainly involve recording, it is appropriate to distribute the recordings between the tablets so as not to load a single tablet. For predominant reading, it is appropriate that all data be located on the same tablet.

The two types of data splitting by tablets are: splitting ranges and splitting by hash value.

When dividing by range, the rows are distributed on the tablets on the basis of a special common key for multiple rows (Fig. 3). Ranges should not overlap and can be added and removed dynamically.

When dividing by hash value, the rows are divided into tablets based on the hash value by which the rows are placed in one of several groups (Fig. 4). This splitting strategy is appropriate when the order of the records is irrelevant. In this case, the recording operations are randomly distributed between the tablets, which reduces the likelihood of obtaining "bottlenecks" in data processing.

It is also possible to apply a multi-level separation strategy that combines the benefits of the two types of separation, reducing their disadvantages (Fig. 5 and Fig. 6).

Our research in connection with the state and trends of the literature sources published in scientific databases on methods and models for big data analysis show that one of the most valuable resources today is the information or that part of the data, which after their processing acquires value and brings value to those who use it. The specialist who has more information always benefits

from a greater advantage in terms of the activities he performs. In the process, big data are generated as structured, semi-structured, unstructured and multistructured, but only 5% (Durcevic, 2020) can be categorized as structured data. Big data usually comes from different geographical or other points in the form of unstructured or semi-structured data.

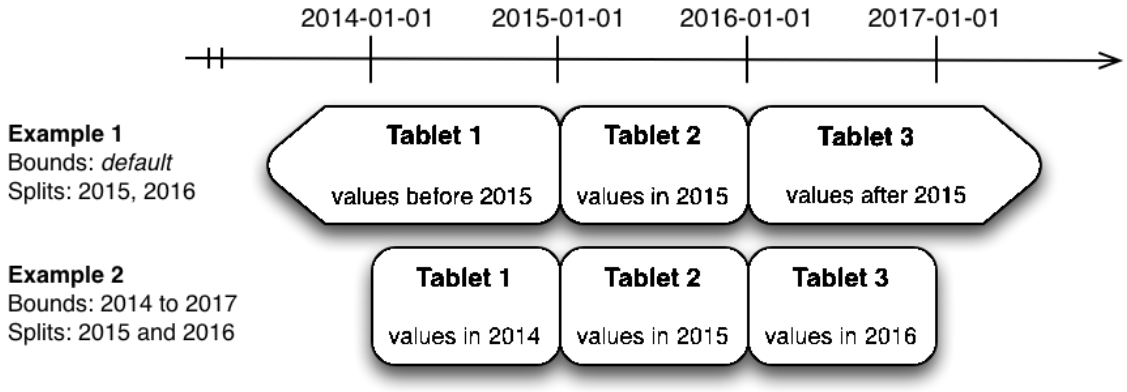

Figure 3. Examples of dividing rows by ranges in different tablets.

Source: Apache Kudu Schema Design (2019). https://kudu.apache.org/docs/schema_design.html

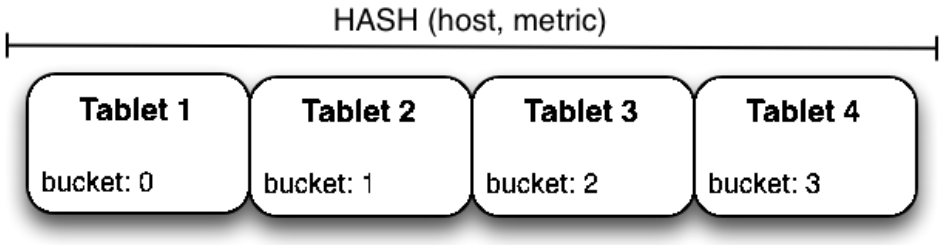

Figure 4. Example of dividing rows by hash values into different tablets.

Source: Apache Kudu Schema Design (2019). https://kudu.apache.org/docs/schema_design.html

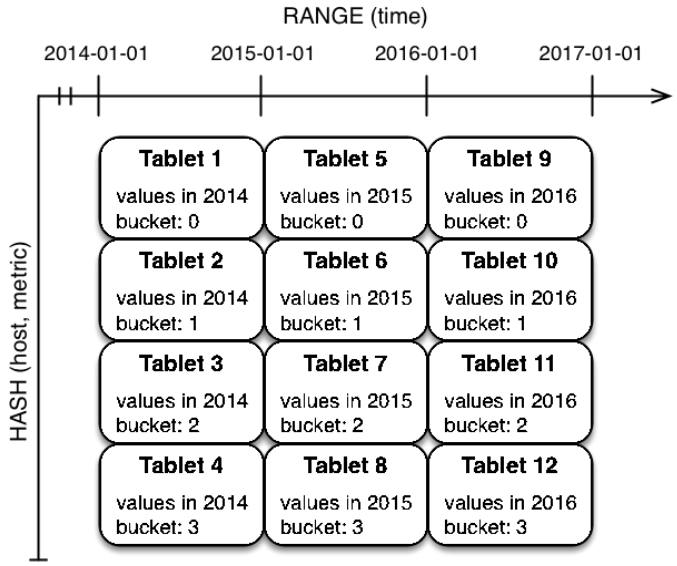

Figure 5. Example of dividing into several levels - by ranges and by hash values.

Source: Apache Kudu Schema Design (2019). https://kudu.apache.org/docs/schema_design.html

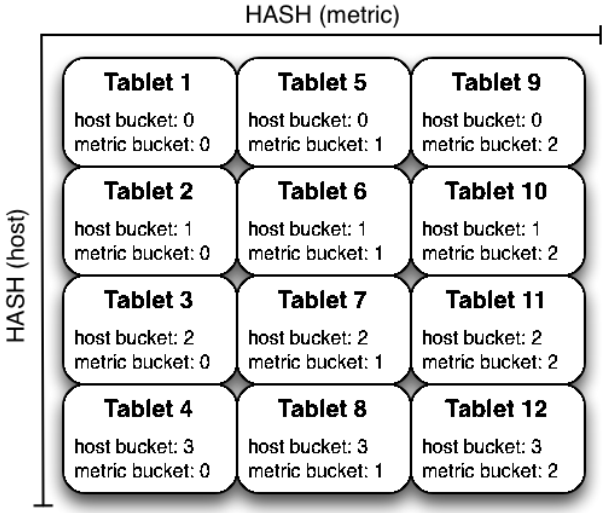

Figure 6. Example of dividing into several levels - by hash values and again by hash values.

Source: Apache Kudu Schema Design (2019). https://kudu.apache.org/docs/schema_design.html

In order to extract valuable information from the big data set, the tools and techniques used for big data processing and analysis must be analyzed.

The overall process of acquiring valuable information from large data sets can be divided into stages (which have already been considered at an earlier stage of the study), which include retrieval and cleaning and presentation, modeling, analysis and interpretation of data . These stages are divided into two parts, which are called the data management phase and the analysis phase. Data management includes techniques, tools and technologies that support data management for data collection and storage and for making changes, or this leads to the conclusion that data is thus prepared for the analysis phase. Analytical methods are used to extract valuable information from big data and interpret the conclusions about them. It is logical to add additional value for the data user through these two phases.

Tables in Kudu use a model similar to the relational model. In designing them, some of the limitations that Kudu currently has must be taken into account (Apache, 2019; Cloudera, 2020):

1. The maximum number of columns when creating tables is up to 300 columns. For best performance, it is recommended to use schemes that use fewer columns.

2. The maximum size of a cell is up to 64KB, before encoding or compression. The size of the cells involved in a composite key is limited to a total of 16KB after the internal coding performed by Kudu for the composite key. Inserting rows that do not meet these constraints generates error messages.

3. It is recommended that the data in a row be no larger than a few hundred KB, although theoretically at 300 columns of 64KB, the row can store up to about 18MB of data.

4. Table and column name identifiers must be valid UTF-8 strings and no longer than 256 bytes.

5. After creating the tables, Kudu does not allow changing the columns with a primary key.

6. The values for the primary key are constants and cannot be changed.

7. After creating a table, the column data types cannot be changed.

8. When deleting rows, the disk space remains occupied by the deleted rows until the table is compressed.

In traditional row data storage, in order to traverse the values in a field, it is necessary to read all the data in the table. The use of Kudu requires changes in data storage compared to the

traditional approach to using DBMS (Fig. 7 and Fig. 8).

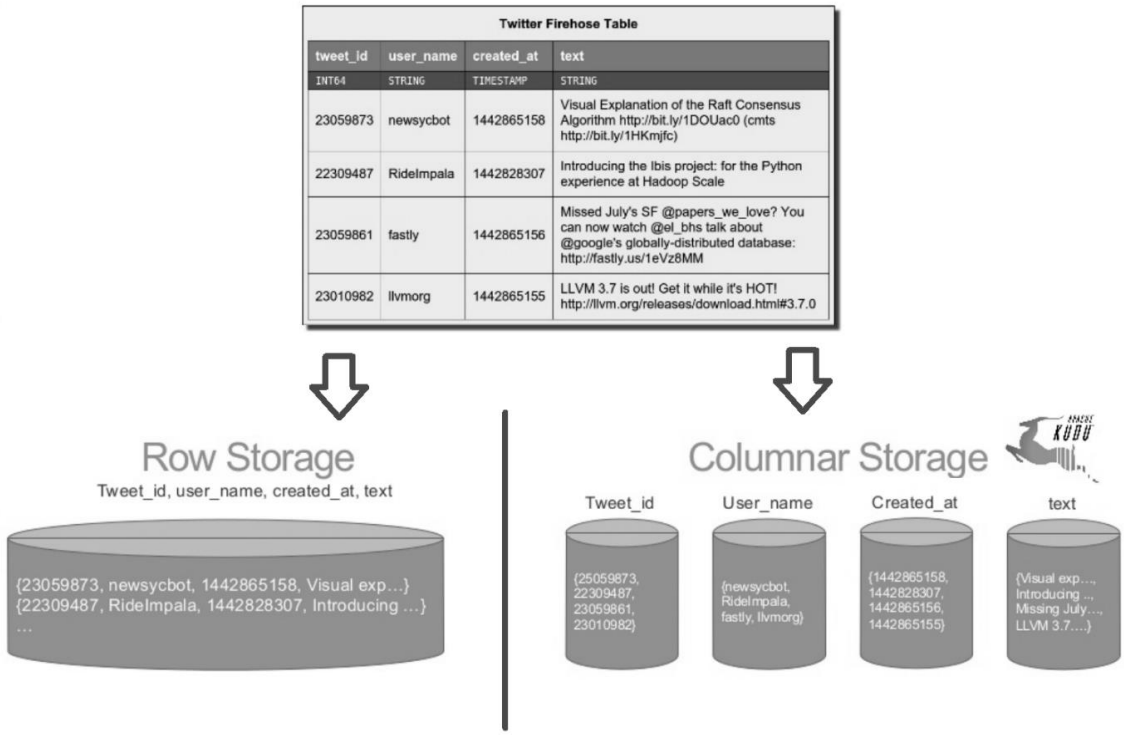

Figure 7. Comparison between the traditional approach of storing data in row tables and the use of column tables in Kudu.

Source: Combination by the author of graphics from Cloudera's documentation. (http://getkudu.io/overview.html)

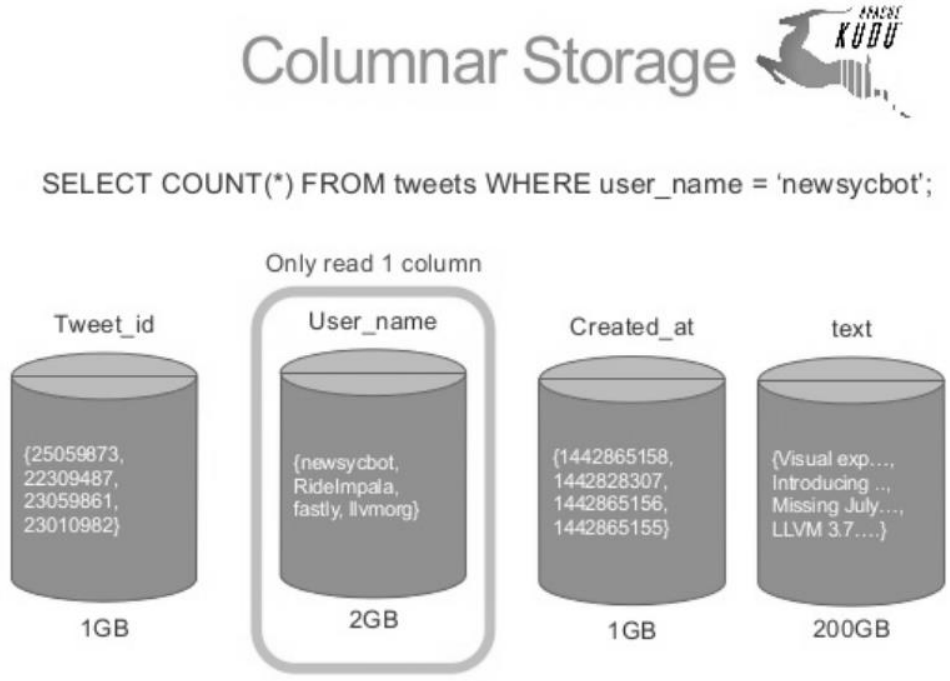

Figure 8. Extract data from 1 column when using column tables in Kudu.

Source: Cloudera documentation. (https://www.slideshare.net/cloudera/apache-kudu-incubating-new-hadoop-storage-for-fast-analytics-on-fast-data-by-todd-lipcon-software-engineer-cloudera-kudu-founder)

Column data when using Kudu can be efficiently compressed to achieve significant space savings and a corresponding increase in read speed. The greatest effect is obtained with data of type date and time, time series (ie constant increase of values) and repeating strings (ie low cardinality of values).

The possible operations for already created tables are: renaming of tables, columns with primary keys and ordinary columns; adding and removing ordinary (non-primary key) columns.

The main types of data that can be used for columns in Kudu are:

- int8, int16, int32, int64 - respectively 8, 16, 32 and 64 bit integers with a sign;
- float, double, decimal - respectively 32 and 64 bit floating point numbers, and for numbers with fixed accuracy;
- date, unixtime_micros - respectively 32 bit number for the number of days from the beginning of the UNIX era and 64 bit number for the number of microseconds from the beginning of the UNIX era;
- bool - for logical values;
- string, varchar, binary - for UTF-8 strings and binary values up to 64KB uncompressed.

Each Kudu table must have a primary key consisting of one or more columns. Like the primary key in RDBMS (Kuyumdzhiev & Nacheva, 2019), the primary key requires uniqueness. Adding a row with the same primary key values as an existing row results in a duplicate key type error. The columns of the primary key cannot have zero values or be of Boolean or floating point type. Unlike RDBMS, Kudu does not have the ability to automatically increment values in a column, so the value for the primary key must be provided by the application when adding new rows.

The set of columns used as the primary key cannot be changed later.

When deleting and updating rows, the full primary key of the row must always be used. Kudu does not support deleting or updating by specifying a range of values.

Primary key values cannot be changed after adding a row to the table. For these purposes, it is possible to delete the row and add it again with another value of the primary key.

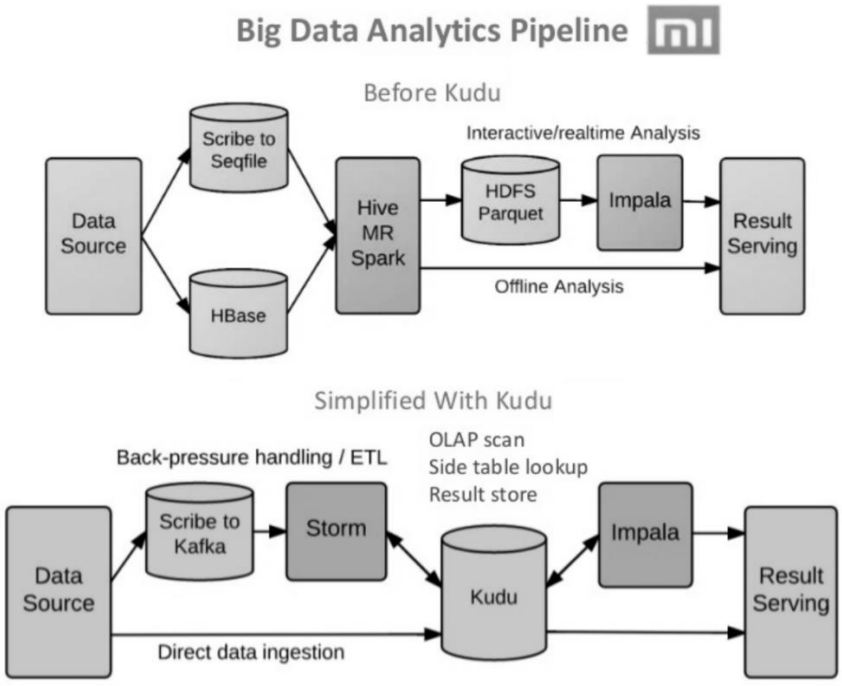

Figure 9. Functional architecture of big data analysis processes using Kudu on the example of Xiaomi.

Source: Presentation by Xiaomi (https://www.slideshare.net/jhols1/introduction-to-apache-kudu).

Publications describing the application of Kudu in practice prove the effectiveness of this approach in big data analysis. For example, based on the experience of Xiaomi, it was found that the latency (latency) in the analysis is reduced from 1 hour - 1 day to 0 - 10 sec. A log file with data from 1 day with 5 billion records was processed in 2-3 days. After using Kudu, the analysis is performed in almost real time. The functional architecture of the big data analysis processes using Kudu on the example of Xiaomi is shown in Fig.9.

3. Basic scenarios for using Apache Kudu in big data systems

The main scenarios for using Apache Kudu in large data systems are the following:

- Processing of a constant input data stream in real time. These are situations where new data arrives frequently and constantly. They must be available in near real time for reading, crawling and changing. Apache Kudu has the ability to quickly insert and update with efficient column crawling.

- Study of time series. In time series, data groups are organized according to the time in which they occurred. It is suitable for studying the effectiveness of various indicators over time or for predicting future behavior based on past data. Apache Kudu is suitable for working with time series.

- Creating forecast models. In machine learning using large data sets, the model may need to be updated and changed frequently. This can happen, for example, while training is taking place as a result of a change in the simulated situation. A researcher may want to change one or more factors in the model to see what happens over time. Updating a large set of data stored in files in HDFS requires a lot of resources, as each file must be completely overwritten. In Apache Kudu, data updates are performed in near real time, and the researcher can adjust the parameter, restart the query, and refresh, for example, a chart in seconds or minutes, not hours or days. Batch or incremental algorithms can also be executed on data in near real time.

- Combining data in Apache Kudu from existing ("legacy") systems. The business generates data from multiple sources and stores it in various systems and formats. It is possible by using other subsystems in Hadoop, such as Impala (Quinto, 2018), to use data from different sources and in different formats without the need to modify existing systems.

Apache Kudo is a data storage tool. Data entry and retrieval is carried out through program libraries that offer the so-called client application programming interface (Stoev, 2019). Java, C++ and Python programming languages (Developing Applications With Apache Kudu, <https://kudu.apache.org/docs/developing.html>) are supported. It is possible to use MapReduce via a Java client and Spark via Spark SQL.

Client API libraries provide classes from which objects must be created to define and modify the data stored in Kudo (Kudu C++ client API, <https://kudu.apache.org/cpp-client-api/>). The work proceeds in two main steps.

1. The first step is to create a Kudu client object, which sets a list of master addresses. Subsequently, the object can be used to check if the table exists and to perform operations such as creating, deleting, and modifying a table.

Example of creating a Kudu C++ client based on the instructions given in (Kudu C++ client example README, <https://github.com/apache/kudu/tree/master/examples/cpp>):

```
#include "kudu/client/client.h"
using kudu::client::KuduClient;
using kudu::client::KuduClientBuilder;

static Status CreateClient(const vector<string>& master_addrs,
```

```
                                shared_ptr<KuduClient>* client) {
  return KuduClientBuilder()
     .master_server_addrs(master_addrs)
     .default_admin_operation_timeout(MonoDelta::FromSeconds(20))
     .Build(client);
}
```

In the main program, the function is used as follows:

```
int main(int argc, char* argv[]) {
        if (argc < 2) KUDU_LOG(FATAL) << "usage: " << argv[0] << " <master host> ...";
        vector<string> master_addrs;
        for (int i = 1; i < argc; i++) master_addrs.push_back(argv[i]);
        shared_ptr<KuduClient> client;
        KUDU_CHECK_OK(CreateClient(master_addrs, &client));
        KUDU_LOG(INFO) << "Created a client connection";
//... Work with a table using the client directory...
        return 0;
}
```

The commands of the master hosts are set as command line arguments when starting the program.

2. In the next step, you can perform the insert, delete, change, and search rows operations with a table object.

Example of working with a table object in the main program (the source is from there again):

```
const string kTableName = "test_table";
shared_ptr<KuduTable> table;
KUDU_CHECK_OK(client->OpenTable(kTableName, &table));
KUDU_CHECK_OK(InsertRows(table, 1000));
KUDU_LOG(INFO) << "Inserted some rows into a table";
```

Program code for data insertion function:

```
static Status InsertRows(const shared_ptr<KuduTable>& table, int num_rows) {
  shared_ptr<KuduSession> session = table->client()->NewSession();
  KUDU_RETURN_NOT_OK(session->SetFlushMode(KuduSession::MANUAL_FLUSH));
  session->SetTimeoutMillis(5000);

  for (int i = 0; i < num_rows; i++) {
    KuduInsert* insert = table->NewInsert();
    KuduPartialRow* row = insert->mutable_row();
    KUDU_CHECK_OK(row->SetInt32("key", i));
    KUDU_CHECK_OK(row->SetInt32("integer_val", i * 2));
    KUDU_CHECK_OK(row->SetInt32("non_null_with_default", i * 5));
    KUDU_CHECK_OK(session->Apply(insert));
  }
  Status s = session->Flush();
```

```
if (s.ok()) {
  return s;
}

// Test asynchronous flush.
KuduStatusFunctionCallback<void*> status_cb(&StatusCB, NULL);
session->FlushAsync(&status_cb);

// Look at the session's errors.
vector<KuduError*> errors;
bool overflow;
session->GetPendingErrors(&errors, &overflow);
if (!errors.empty()) {
  s = overflow ? Status::IOError("Overflowed pending errors in session") :
      errors.front()->status();
  while (!errors.empty()) {
    delete errors.back();
    errors.pop_back();
  }
}
KUDU_RETURN_NOT_OK(s);

// Close the session.
return session->Close();
}
```

Conclusion

The main purpose of this article is to present the Hadoop Analytical Tool Apache Kudu in connection with the creation of methods and models for big data analysis in the field of construction. The first part of the document provides guidelines for creating a data model with the Apache Kudu analytics tool, as well as guidelines for creating methods for analyzing different types of data with the Apache Kudu analytics tool. The second part presents appropriate methods and models with analytical tool Apache Kudu in the business field of construction and ICT prototype solution with analytical tool Apache Kudu in economic field of construction.

References

1. Alla, S. & Muglurmath, K. (2019) Powering real-time analytics on Xfinity using Kudu (презентация) <https://slideplayer.com/slide/13949939/>
2. Apache (2019). Apache Kudu Schema Design, <https://kudu.apache.org/docs/schema_design.html>
3. Apache Kudu Datasheet (2017). https://www.cloudera.com/content/ dam/www/marketing/resources/ datasheets/cloudera-kudu-datasheet.pdf.landing.html
4. Apache Kudu Schema Design (2019). https://kudu.apache.org/docs/schema_design.html
5. Apache Software Foundation (2020). Using the Hive Metastore with Kudu, <https://kudu.apache.org/docs/hive_metastore.html>
6. Baesens, B. (2014). Analytics in a big data world: The essential guide to data science and its applications. John Wiley & Sons.

7. Baesens, B. (2020). Real-world techniques for analyzing big data (Part 1) [Online] Available at: https://www.sas.com/en_us/insights/articles/big-data/real-world-big-data-tips.html).
8. Cloudera (2020). CDH Component Guides Kudu, <https://docs.cloudera.com/documentation/enterprise/latest/topics/kudu_schema_design.html>
9. Cloudera Docs. Storage Apache Kudu overview, <https://docs.cloudera.com/runtime/7.2.1/kudu-overview/topics/kudu-intro.html>
10. Developing Applications With Apache Kudu (2019), <https://kudu.apache.org/docs/developing.html>
11. Durcevic, S., 2020, Your Modern Business Guide To Data Analysis Methods And Techniques [Online] Available at: https://www.datapine.com/blog/data-analysis-methods-and-techniques/;
12. Harrison, G. (2015). Column Databases. In Next Generation Databases, Apress Berkeley, pp.75-85.
13. Introducing Apache Kudu. Architectural Overview. https://kudu.apache.org/docs/
14. Kudu C++ client API (2018). <https://kudu.apache.org/cpp-client-api/>
15. Kudu C++ client example README (2020). <https://github.com/apache/kudu/tree/master/examples/cpp>
16. Kuyumdzhiev, I. & Nacheva, R. (2019). Correlation Between Storage Device and Backup and Restore Efficiency in MS SQL Server. Serdica Journal of Computing, 13(3-4). Institute of Mathematics and Informatics. BAS, pp.139-154.
17. Maheshwari, M. (2019). Impala Performance Tuning Guidelines. Tuning SQL queries, p.19. <https://docs.cloudera.com/best-practices/latest/impala-performance/bp-impala-performance-tuning.pdf>
18. Oleś D., Nowak Z. (2019) The Performance Analysis of Distributed Storage Systems Used in Scalable Web Systems. In: Borzemski L., Świątek J., Wilimowska Z. (eds) Information Systems Architecture and Technology: Proceedings of 39th International Conference on Information Systems Architecture and Technology – ISAT 2018. ISAT 2018. Advances in Intelligent Systems and Computing, vol 852. Springer, Cham. https://doi.org/10.1007/978-3-319-99981-4_27
19. Patel, J. (2019). An Effective and Scalable Data Modeling for Enterprise Big Data Platform. In 2019 IEEE International Conference on Big Data (Big Data), pp.2691-2697.
20. Radev, M. & Aleksandrova, Y. (2013) Combining Virtualization Technologies in SOA-Application. 3rd International Conference on Application of Information and Communication Technology and Statistics in Economy and Education (ICAICTSEE2013), Vol. 200, Sofia: UNWE, pp.56-61.
21. Stoev, S. (2019). Using of Additional Packages of Components for Accelerated Application Development. Izvestia Journal of the Union of Scientists - Varna. Economic Sciences Series, 8(2), 171-179. doi:10.36997/IJUSV-ESS/2019.8.2.171
22. Quinto B. (2018) Introduction to Kudu. In: Next-Generation Big Data. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-3147-0_2
23. Quinto, B. (2018). Big data warehousing. In Next-Generation Big Data, Apress Berkeley, pp.375-406.
24. Quinto, B. (2018). High Performance Data Analysis with Impala and Kudu. In Next-Generation Big Data. Apress, Berkeley, CA. pp.101-111