Architecture of a Big Data Platform for a Semiconductor Company

Daniel Müller, Stephan Trahasch
Institute for Machine Learning and Analytics
Offenburg University of Applied Sciences
Offenburg, Germany
e-mail: daniel.mueller@hs-offenburg.de, stephan.trahasch@hs-offenburg.de

***Abstract*—Apache Hadoop is a well-known open-source framework for storing and processing huge amounts of data. This paper shows the usage of the framework within a project of the university in cooperation with a semiconductor company. The goal of this project was to supplement the existing data landscape by the facilities of storing and analyzing the data on a new Apache Hadoop based platform.**

Keywords—Big Data Analytics; Big Data Storage; Apache Hadoop; Metrics

I. Introduction

Over the past few years, the world of data changed. More and more data-driven processes are coming up. They can be used to monitor or even improve existing processes. Especially the industry benefits from this new know-how that can be retrieved from analyzing Big Data where Big Data refers to the large volumes of structured, semi-structured, or unstructured data, acquired from a variety of heterogeneous sources. On the other hand, new devices and techniques need to be applied to enable such potential of Big Data. This often leads to high costs for the acquisition of new hard- and software, and even the knowledge how to implement a Big Data solution.

The semiconductor industry is one of the most complex manufacturing processes, and large amount of data retrieved during the manufacturing process has to be stored in huge databases [1]. The automatic analyses of that data may lead to reduction in the manufacturing cost. This is true for basic analyses, but especially for advanced analyses like anomaly detection and quality control. Insights can be gained about the production process if those data can be stored, retrieved and analyzed in an easy way.

The Institute for Machine Learning and Analytics (IMLA) [2] together with a semiconductor manufacturer from Germany examines how large data - collected during the manufacturing process - can be stored and analyzed in a Big Data system. The company has widened their machines with sensor technology, to be able to track their production process in large part. This led to a mass of new data, that must be stored and processed in an adequate duration of time, to be able to handle all this data and react as fast as possible on different events (especially in case of a problem). Another challenge comes with the previous analyses, which are based on different datasets – a lot of this data was collected and joined manually by some employees, which meant a huge overhead and delay on the analyses.

The goals of the project are to build a cost-effective and scalable database for storing and processing the sensor-generated data, to accelerate the search and analysis of data and to implement advanced analyses with machine learning. In this paper we focus on the first two goals: the architecture and implementation of a scalable data base and the integration in the IT environment to support the analysis process. The approach is based on the Apache Hadoop [3] and Apache Spark [4] framework, very popular platforms for subjects concerning Big Data handling. The next step of the project is the implementation of machine learning algorithms.

The structure of this paper is organized as follows: Section II provides background information about the data base, while Section III shows the basic information about the Apache Hadoop cluster that was used to implement the project. Section IV of this paper discusses various ways to store the data and shows ways to import and analyze this data. Section V provides first benchmarks on the imported data, and Section VI concludes the paper with a brief summary.

II. Background

This section describes the data and the database used so far. The company uses different database systems and Network-based File Systems (NFS) to store the data that accrues during the different production processes. The data consists of various types:

1) *Structured Data*
   - Lot history (tracking of the steps that were passed by each lot during production)
   - Machine data (events that occur on the different machines, lot independent)
   - Many more smaller datasets
2) *Semi-structured Data*
   - Results of quality tests (results of tests that run after production, e.g., power consumption, heat development, mechanical checks)

The structured data comes with a fix schema, like every entry has the same amount and datatypes of columns. Typical examples for this format are CSV files, which represent data in a table-like form.

In contrast to that comes the semi-structured data, which can have a very different schema from file to file. The quality test results are of this schema-less format, so every file (or even every entry) can have different number of columns and/or datatypes. Due to the various families of semiconductors there are also different test cases for each. This data diversity has also a second reason: In the course of time, the sensors for checking the products and the software changed, which also led to different test cases (which are reflected in the different schemas) within the same product families. Saving these differing data sets inside the same storage pool was also a big challenge on the project.

Altogether these information sets filled up (amongst others) an Oracle database with approximately 13 TB of data. Since this system run into capacity limits, one of the main goals of the project was to source older data out into another file storage, e.g., the Hadoop Distributed File System (HDFS), which is part of the Apache Hadoop ecosystem. More about the task of data moving in the “Solution” section.

As already mentioned in the introduction section, there were a lot of analyses that took a long time or even overloaded the system, that ran on capacity limits. In addition to this issue, many analyses needed some manually gathered and filtered data as input. These issues are pain points because a lot of time is wasted on getting and processing the desired data. Realtime results (or even getting any results at all) were not possible for these kinds of analyses. By using the parallelism of the Apache Hadoop platform, we wanted to be able to automate the information gathering and bring up new ways for faster analyses.

III. The Apache Hadoop Cluster

The IMLA runs an in-house Apache Hadoop cluster, which is based on the Hortonworks Data Platform [5]. This platform is a combination of different tools that can be used for storing and analyzing huge datasets [6]. Figure 1 shows the main structure of the cluster components:

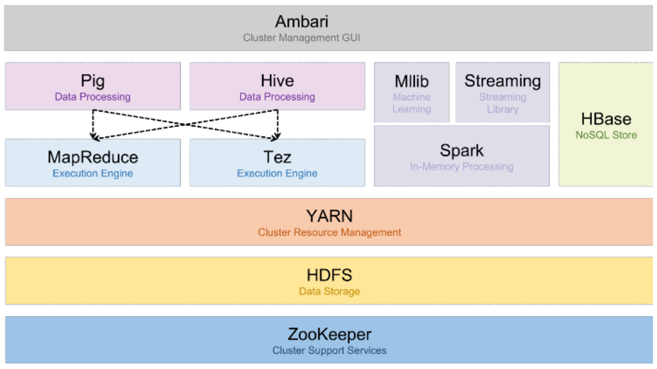


Figure 1. The coarse structure of the components in the Hadoop cluster

Since the Hadoop ecosystem is a collection of different tools for storing and analyzing datasets, it is applicable for most tasks all around working with Big Data. Some important tools that were used in the project are the following:

- **HDFS**: Distributed data storage inside a cluster.
- **Apache Hive**: SQL-like interface to structured data stored inside HDFS.
- **Apache HBase**: Distributed NoSQL database, using HDFS as background data storage.
- **Apache Spark**: In-memory processing engine with interfaces to various datastores, like HDFS, Hive, HBase and many more.

Stand May 2019 the Hadoop cluster of the university has the following setup: We use eight nodes, two of which are set up as name nodes (high availability) and the other six as data nodes. The name nodes are responsible for the administration of the metadata of the HDFS and the requests of the different services running inside the cluster and coordinate the incoming tasks submitted by the users. The data nodes hold the datasets in themselves and execute the processes, which in turn work with this data.

These are the actual components of the cluster:

- 2 x NameNode
  - o CPU: 2 x Intel Xeon E5-2630v4 @ 2.2 GHz (10 cores, 20 threads)
  - o RAM: 256 GB (DDR4, ECC-reg.)
  - o SSD: 2 x 480 GB (RAID-1)
  - o OS: CentOS 7
- 6 x DataNode
  - o CPU: 2 x Intel Xeon E5-2630v2 @ 2.6 GHz (6 cores, 12 threads)
  - o RAM: 64 GB (DDR3, ECC-reg.)
  - o HDD (System): 1 x 1 TB
  - o HDD (HDFS): 4 x 3 TB
  - o OS: CentOS 7

We use the Hortonworks Data Platform 2.6.5, which is a free Hadoop distribution from Hortonworks that is based on the Hadoop 2.7 stack. As operating system we use CentOS 7. In the future, the cluster will be upgraded to Hadoop 3 and equipped with graphics cards to enable also GPU computing.

IV. Solution

This section covers the realization of the project, which consists of the four thematic areas data import, storage of structured data, storage of semistructured data, and the data processing. For a better understanding of the other topics, first the storage of the data is treated, before continuing with the import of the data into the cluster.

A. *Storing the structured data*

First, the Hadoop framework contains a distributed file system that can be used to store all types of data. The user has access to appropriate interfaces for writing and reading this storage. Even other tools used in a Hadoop platform usually store their data on the filesystem called HDFS.

The basic Hadoop framework can be extended with a data warehousing software called Apache Hive that is based on HDFS [7]. Hive is an interface for working with structured data (that is stored in HDFS) using an SQL-like syntax called HiveQL. It brings also new interfaces (e.g., command line tool, JDBC driver and REST-based webservice) and supports the common data types like numeric, date/time, string, misc (boolean, binary) and complex (array, maps, structs).

There are also multiple file formats that can be read and written by Hive. One of the best-known formats in storing structured data in Hadoop is the Apache Optimized Row Columnar (ORC) [8] format, which is also supported by Hive and used in this project for storing the structured data. ORC is a memory-optimized and column-based data format with helpful features like ACID support, built-in indexes and support of complex types. It is optimized for Big Data workloads, especially for parallel readings from HDFS. It allows filtering close to the data, by passing the filter criteria to the data store, thus selecting and returning only the desired data at a lower level. This feature is called “predicate pushdown” and accelerates queries many times over, because it significantly reduces the network load and therefore the size of data, that must be processed in further steps. ORC also supports zlib and Snappy compression to reduce data size in addition to the default column-based compression.

There are different ways to bring any Hive-readable format into the ORC format and vice versa. This is very helpful, especially to bring data from external systems into this optimized format (e.g., if the external system can not work with ORC files but can export data as CSV). For example, to put CSV-based data into an ORC-based table, a user could go one of the following ways:

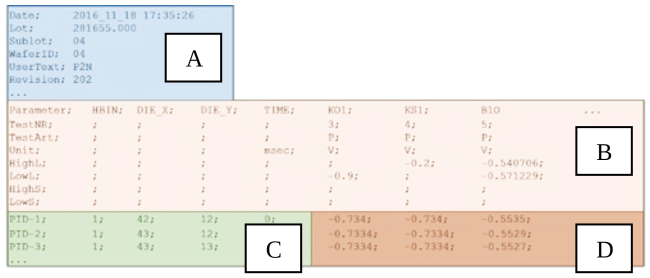
- Create an external hive table to reference the newly imported data (e.g., in CSV format) in HDFS. Then add an internal hive table that contains the same column definitions but uses the ORC format for storing [9]. After that, the ORC table can be filled using a simple "INSERT INTO ... SELECT ... FROM ..." command. This generates the corresponding ORC files on HDFS in background [10].
- Using a Spark job to read in the original files (e.g., in CSV format), optionally transform the data and write it to Hive (or HDFS) in ORC-based format.

B. *Storing the semistructured data*

As explained in the introduction section, the data base of the project partner also consists of semistructured data created during the quality tests after production (different schema from file to file, depending on test type, and software version). Since this data makes up a large part of the data base, the outsourcing of these files was also examined.

1) *Composition of the semistructured data*

The quality test results don not have this well-known CSV structure, as they contain some metadata at the beginning of each file, and every file can have another bunch of columns, that contain the test results. Figure 2 is an illustration of the coarse structure of such a test result file:



```
Date;      2016_11_18 17:35:26
Lot;       281655.000
Sublot;    04
WaferID;   04
UserText;  P2N
Revision;  202
...
Parameter;  HBIN;  DIE_X;  DIE_Y;  TIME;   K01;      KS1;      B1O        ...
TestNR;     ;      ;       ;       ;       3;        4;        5;
TestArt;    ;      ;       ;       ;       P;        P;        P;
Unit;       ;      ;       ;       msec;   V;        V;        V;
HighL;      ;      ;       ;       ;       ;         -0.2;     -0.540706;
LowL;       ;      ;       ;       ;       -0.9;     ;         -0.571229;
HighS;      ;      ;       ;       ;       ;         ;         ;
LowS;       ;      ;       ;       ;       ;         ;         ;
PID-1;      1;     42;     12;     0;      -0.734;   -0.734;   -0.5535;
PID-2;      1;     43;     12;             -0.7334;  -0.7334;  -0.5529;
PID-3;      1;     43;     13;             -0.7334;  -0.7334;  -0.5527;
...
```

Figure 2. Schema of a test result file

The header rows (A) have always the same structure and can help to identify a single test file within all the files. Also, the first columns (C) are always the same in every file, they identify the rows inside a test file. Area (B) contains meta information about the following rows, like column names, min / max values, and units. Its width depends on the amount of test columns. The test columns (D) contain the results of the test cases and can be different in each file (but are the same within one file, containing null values if necessary).

2) *Storing the semistructured data in HBase*

Since these files have different schemas, the default bulk-load mechanisms can not be used for importing this data. That is why we have decided to process the data with Apache Spark, because it has an extensive API and can handle almost any kind of data. Spark also has interfaces to almost all datastores that exist for Hadoop, e.g., HDFS, Hive and HBase. We examined two different approaches for storing these test data files, that follow in the next sections.

a) *Spark-Job that writes an HBase table*

The first approach of storing the data was a combination between Spark and Apache HBase. We decided to use Spark to read in and transform the data into key-value pairs, that could be written into the NoSQL database HBase afterwards. Since HBase stores data in form of key-value pairs this is an interesting opportunity for storing un- or semi-structured data. To do so, we had to split the regarding rows that contained the test results into a combination of row key and key-value pairs. The row key is used to identify a row globally within the entire data base (i.e., across all files). To get a unique key, we had to combine the information of the header rows (file identification) and the base columns, which identify the rows inside a single file. So, the key consists of the following parts:

id_{File} = <Lot>_<Sublot>_<WaferID>_<Date>_<Revision>_<UserText>
key_{Row} = <id_{File}>_<PID>_<HBIN>_<DIE_X>_<DIE_Y>

Figure 3. Composition of the row key. DIE_X and DIE_Y are the x and y coordinates of the die on the wafer.

In combination with the row key, a list of key-value pairs (that represent the test name and its value) can be added to a HBase Put object to be sent to the database by using a Spark application. After reading the different CSV files (test results) from HDFS, Spark creates these Put objects by processing the files in parallel and afterwards calling a bulk-load function on the HBase API. Here is the relevant code snippet:

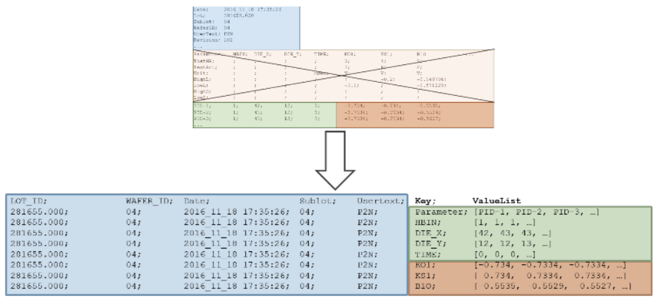
```
// This is inside the Spark Job
JavaRDD<HBaseRowEntry> rowsRDD = ...
hbaseContext = new JavaHBaseContext(sc, baseConf);
hbaseContext.bulkPut(rowsRDD, name, new PutFunction());

public class PutFunction
 implements Function<HBaseRowEntry, Put> {
  @Override
  public Put call(HBaseRowEntry entry) throws Exception {
    Put put = new Put(entry.getKey().getBytes());
    for(MyKeyValue kv: entry.getKeyValues()) {
      put.addColumn(kv.getCF, kv.getCQ(), kv.getValue());
    }
    return put;
}}
```

Figure 4. Writing to HBase using the Java API

We were able to try this approach in a test phase and we successfully imported more than 100 GB of test result files into HBase by running this Spark Job. The problem is that additional skills would be needed to launch and administrate the HBase infrastructure.

b) Spark-Job that writes an Hive table

After evaluating the first approach, we took a closer look at the data and found out that it was possible to bring this semistructured data into a structured form. That was possible, because we learned from the company's employees that they will always read at least one entire column of the test result files for analysis purposes. With this new knowledge, we were able to plan a new data structure, which then enabled to have a fix schema over all files. Figure 5 shows the transformation that brings the data into a globally identical schema:



```
LOT_ID;       WAFER_ID;  Date;                  Sublot;  Usertext;  Key;        ValueList
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       Parameter;  [PID-1, PID-2, PID-3, …]
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       HBIN;       [1, 1, 1, …]
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       DIE_X;      [42, 43, 43, …]
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       DIE_Y;      [12, 12, 13, …]
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       TIME;       [0, 0, 0, …]
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       KO1;        [-0.734, -0.7334, -0.7334, …]
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       KS1;        [ 0.734,  0.7334,  0.7334, …]
281655.000;   04;        2016_11_18 17:35:26;   04;      P2N;       B1O;        [ 0.5535,  0.5529,  0.5527, …]
```

Figure 5. Transforming the test result file

As shown in the figure, the headers that identify a file have been moved into the test rows below. They are still used to be able to find a specific file. The main difference is the transformation of the test columns, as they have been converted to row-based key-value pairs. The new "Key" column contains the test names, while the test results can be found in an array of values under the new column "ValueList". This means, that the array of the ValueList column has as much elements as the test result file had test rows before. Also, there will be generated as much rows for a test result file as the original file had columns. This schema can be used to store the contents of all test result files in a common, structured table together. As already mentioned, this only works performant under the condition that the smallest unit read out is an entire (former) test column, which means the reading of a complete ValueList array in the new format. Reading smaller units could cause performance issues, as the array has to be iterated to find the correct item (then the key-value approach with HBase would be a better solution). But since the company wants to read complete test columns this is a better solution, as we could use Apache Hive to save this data. As Hive is already chosen in the company for storing the structured data, they don nott need to administrate and learn a new tool. Spark also has native connectors to the Hive warehouse and can write this data to it in parallel, after bringing the test results into the new format.

C. *Realizing the data import process*

Now that it has been determined, how and where the data will be stored, the import process could be planned. An SAP system acts as the data supplier, while the HDFS and Apache Hive serve as data sink. We also decided to use Apache Spark for data import, as it is flexible and can operate natively with these Hadoop components, and this also avoids the introduction of another tool. The idea now was to upload the relevant data files into HDFS and then read them via Spark, transform (if necessary) and finally write them into the corresponding Hive tables. A basic scheme of the import process is shown in Figure 6.

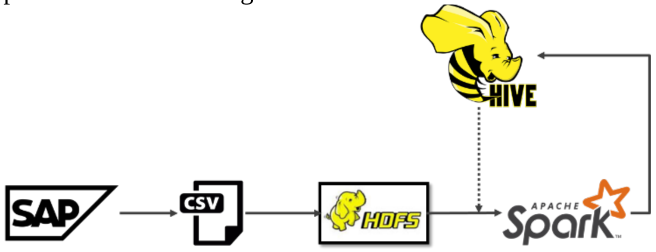
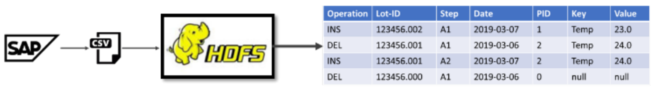


Figure 6. Import Process steps

The following steps are performed in the import process:

- SAP: Exporting the data as CSV files and uploading them into HDFS.
- Spark: Read the files from HDFS and read in the corresponding Hive table into datasets (in-memory).
- Update the dataset by combining the CSV file(s) with the Hive table content.
- Writing the final dataset content back to Hive table (overwrite).

The CSV file that is exported from SAP system must have a separate column that contains information, whether the corresponding row shall be added or removed – updates are realized by a deletion, followed by an insert. Figure 7 is an example of such an import file:



Operation	Lot-ID	Step	Date	PID	Key	Value
INS	123456.002	A1	2019-03-07	1	Temp	23.0
DEL	123456.001	A1	2019-03-06	2	Temp	24.0
INS	123456.001	A2	2019-03-07	2	Temp	24.0
DEL	123456.000	A1	2019-03-06	0	null	null

Figure 7. Example of an import file

As explained before, Spark first reads in the complete Hive table, where the updates should be run against. The table content is held in-memory during Spark job execution. In the next step, the update file is read from HDFS and split into delete and insert rows. Then the delete rows are used to remove the corresponding rows from the table content (null values are treated as wildcards). Afterwards the rows containing the inserts are appended to the table content. Finally, the dataset containing the updated content is written back into the Hive table. The complete Hive table is overwritten in this step.

To trigger the explained Spark import job, we use Apache Livy. Livy is a REST based interface that enables to submit Spark jobs from everywhere, also from outside the cluster. This is helpful since the company needs to start the import job from their SAP system, after writing the update files into HDFS. In the final status (May 2019), the data is gathered inside the SAP system at night and written to HDFS, before the Spark import job is triggered by a Livy call.

D. *Data processing*

The data processing is also done by using Apache Spark jobs. Since Spark has connectors to data stores like Hive and HBase, it is possible to read and write them from a native Spark application. There is also another tool that is particularly suitable for prototyping new Spark jobs. Apache Zeppelin is a web-based notebook, with interfaces to Hive (via JDBC), HBase (via Phoenix) and Spark. The respective tools are connected to Zeppelin via so-called interpreters. A user gets access to a Spark session, that is created automatically on starting the corresponding interpreter. In these notebooks, for example, Spark program code can be tried out in a direct and uncomplicated way, without the effort of creating Spark sessions, application packaging and publishing in the phase of prototyping.

The partner company also decided to use Zeppelin notebooks to introduce and try out new application logic. After a successful test phase, the logic is moved into a separate, stand-alone application, that is created, compiled and packaged in an appropriate IDE. Since the Spark interpreter for Zeppelin works with Scala (alternatively also with Python), we use a Scala IDE to export the final application logic as JAR files. These files are moved into the HDFS and can be executed by using the spark-submit script or by making a corresponding Livy call from outside the cluster. Second is the standard procedure, since a large part of the applications are started from the external SAP system or from user's client computers. Since the results of these data processing applications can be very big, storing these datasets in HDFS or Hive tables is a better approach than sending results back to the client, which could lead to local memory problems. After finishing a job, the user can preview (or download) the results by exploring the data in HDFS or querying the corresponding Hive tables.

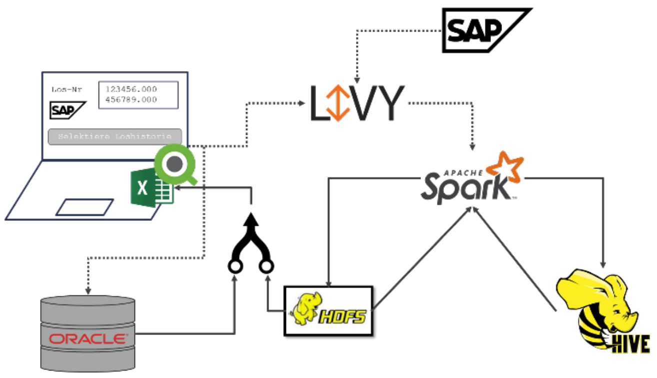


Figure 8. Data processing overview

Figure 8 shows the tools that are used for data processing with Spark (SAP and Oracle DB are external components that are used in the company).

V. BENCHMARKS

In order to show and compare the performance of the new Hadoop system, first benchmarks were carried out. For a better comparability, the queries were executed on the old and afterwards on the new system.

A. *Comparison of the data size*

The graph below shows the various amounts of data required to store the "HistStep" table, which contains the operations performed on the lots during production. The data has the following characteristics:

- approx. 275 mio. rows
- 17 columns (String, VarChar[1-20])

In Oracle database, this table required about 34.9 GB of disk space, plus optional (for performance reasons) index information of around 29.9 GB. So, the table thus required a total of 64.8 GB. In contrast, the ORC-based Hive table only requires about 5.2 GB in HDFS (partitioned, but without bloom filters and without replication). Using bloom filters would increase the storage space by a small amount, but these are not needed for current performance. So, the Hive tables reduced the disk space requirements by factor 6.7 (without Oracle table index) or even by factor 12.5 (with Oracle table index) (see Figure 9).

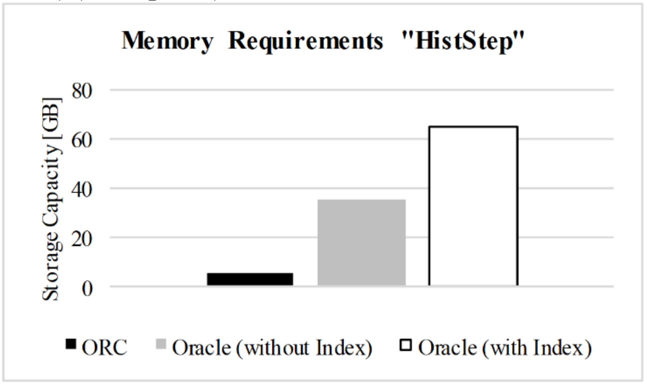


Figure 9. Storage requirements for table "HistStep"

The comparison of space needed to save the results of the product quality tests shows similar results. Saving this data takes around 13 TB in the Oracle database while Hive table only needs about 1.6 TB of HDFS storage without replication (see Figure 10).

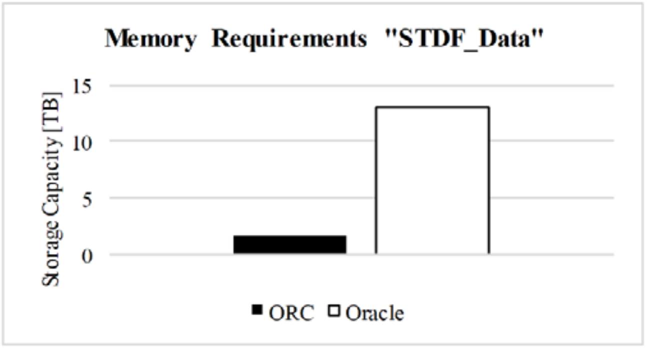


Figure 10. Storage requirements for table "STDF_Data"

We have also received first performance benchmarks from our project partner. The results show the runtimes of lot-based queries, executed on the "HistStep" table, showed in the first benchmark. On small queries, where only a few lots are selected, Oracle is much faster than Hive. But even with queries for a few hundred lots, Hive takes less time to select, process and return the data. Since lot-based analyses must include thousands of lots, the result on the far right of the diagram is the most interesting for the company. The difference in performance can be clearly seen in Figure 11.

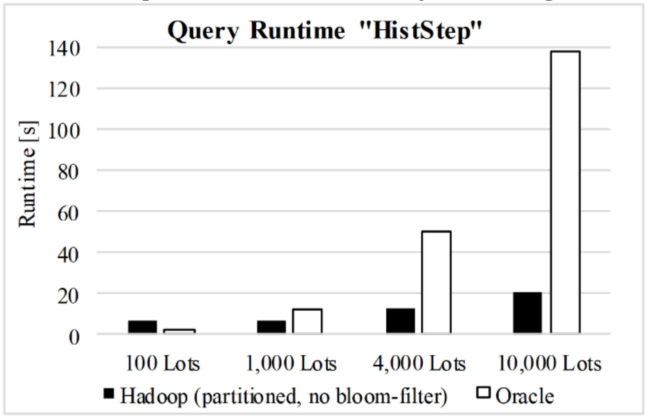


Figure 11. Query Runtime for table "HistStep"

While Oracle scales rather linearly, Hive's runtime increases only minimally. In this scenario, Hive also offers a major performance advantage over the previous system.

VI. CONCLUSION

This project examined the applicability of a Hadoop-based platform for the storage and processing of company-relevant data. Alternative ways to import, store and process different types of data were demonstrated on practical examples. Depending on the problem, the Apache Hadoop framework offers various components to implement the different tasks. In this project, a Hadoop-based cluster was successfully introduced to a company's existing data platform to store and analyze data over a longer time. Helpful tools are especially the basic Hadoop components like the HDFS, the SQL interface Apache Hive, the NoSQL database HBase, as well as the processing engine Apache Spark. This project confirms by means of an industrial project that Hadoop can be used to build such a data-driven platform. Hadoop comes with special storage formats and engines that can be used for efficient storage and high-performance analyses.

REFERENCES

[1] Y. Zhu and J. Xiong, „Modern Big Data Analytics for 'Old-fashioned' Semiconductor Industry Applications," Piscataway, NJ, USA, IEEE Press, 2015, p. 776–780.

[2] Institute for Machine Learning and Analytics - IMLA. [Online]. Available: https://imla.hs-offenburg.de. [retrieved: May, 2019].

[3] „The Apache Hadoop project," [Online]. Available: https://hadoop.apache.org. [retrieved: May, 2019].

[4] „The Apache Spark project," [Online]. Available: https://spark.apache.org/. [retrieved: May, 2019].

[5] „Hortonworks Data Platform," [Online]. Available: https://de.hortonworks.com/products/data-platforms/hdp/. [retrieved: May, 2019].

[6] A. Oussous, F.-Z. Benjelloun, A. Ait Lahcen, and S. Belfkih, „Big Data technologies: A survey," *Journal of King Saud University - Computer and Information Sciences*, Bd. 30, Nr. 4, p. 431–448, 2018.

[7] A. Ismail, H.-L. Truong, and W. Kastner, „Manufacturing process data analysis pipelines: a requirements analysis and survey," *Journal of Big Data*, Bd. 6, Nr. 1, p. 1, 2019.

[8] „Apache ORC," [Online]. Available: https://orc.apache.org/. [retrieved: May, 2019].

[9] „Apache ORC - Hive DDL," [Online]. Available: https://orc.apache.org/docs/hive-ddl.html. [retrieved: May, 2019].

[10] „Convert an HDFS file to ORC," [Online]. Available: https://docs.hortonworks.com/HDPDocuments/HDP3/HDP-3.1.0/migrating-data/content/hive_convert_an_hdfs_file_to_orc.html. [retrieved: May, 2019].