Design of Data Models and Distribution Systems based on Big Data ecosystems HPCC and Hadoop with HBase

Braunmiller Nina

Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
braunmiller.nina@tutanota.com

Abstract. Nowadays the data processing requirements are highly increased because of the immense amount of data that is collected by companies for different contexts. For this reason, the belonging thesis has a look at two famous Big Data processing and storage ecosystems High-Performance Computing Cluster (HPCC) systems and Hadoop which are both open-source projects. On top of that also the NoSQL database HBase will be discussed because it is an add-on to Hadoop. The question is how data querying, data models, and storage have to be designed such that all three systems can effectively work with Big Data. It will become clear that there is no single winning schema design. The performance will always depend on data, query types, and a combination of different design patterns.

Keywords: HPCC · Hadoop · HBase · NoSQL

1 Introduction and background: Big Data

Relational Data Base Systems were quite common in usage. However, nowadays companies collect more and more data in the volume of zettabytes. Furthermore, lots of services like search engines need to handle large consumer queries. Big Data can be described in the form of the three "V"s. The first V is the volume standing for the data amount. The second V is equivalent to variety referring to the different data formats, like videos, texts, and so on. Last but not least, velocity targets the high user demand. Surprisingly, especially data variety is in the interest of Big Data work [8]. An even more extending view to the definition of Big Data is, that it is about analyzing existing data to get a guess about the upcoming future data. [4] claims to drop the volume component.
Databases for Big Data need to fulfill the requirement of the "Three Highs": High performance by lots of queries, high storage for an enormous amount of data, and high scalability to work with continuously increasing data amounts [5]. For this purpose, "Not only SQL" (NoSQL) was developed working beyond fixed table structures where data is unstructured and has the potential of abrupt change. The NoSQL databases separate data storage and management trying to make use of high-performing and scalable data storage hardware. One main advantage compared to relational databases is that their structures can be adapted

to the circumstances instead of needing a rewrite of a table [1, 8].

The following thesis will have a look at two ecosystems famous for working with Big Data, namely "High-Performance Computing Clusters" (HPCC) developed by LexisNexis and Apache Hadoop. Both differ in their approaches of data handling. HPCC doesn't touch its data in the storage but modifies it to the query demands during processing. In contrast, Hadoop aims for strategic data storage. One part of it is the independent database HBase which was integrated into the ecosystem.

2 HPCC Systems of LexisNexis

The open source, cloud-computing HPCC system or also called "Data Analytics Supercomputer" (DAS) [15, 18] was created as a data lake platform meaning that different software technologies work together to process and analyze complex data with the help of hardware for storing [23]. It was developed with the idea to handle huge amounts of data no matter which data format would be used.
The data itself stays untouched on the storage level such that it always keeps its type and content as it was originally inputted into the storage. No implementation of data schema is needed. The key is that the data structures shall be experienced while actively querying. So, the storage, more precisely the data lake, doesn't matter at all. That's a contrasting point compared to Hadoop which lays its focus on storage strategies [2, 22].
Even more interesting, LexisNexis claims that its HPCC system is by weeks faster than the competitor Spark [23]. That's a fancy concept as it seems to contradict the aim of fast Big Data processing as data is not analyzed in advance but during processing.

2.1 Architecture

The storage is a modern version of the "Indexed Sequential Access Method" (ISAM) where data files are simply stored on disks with direct data access. However, it isn't by far the focus of the HPCC systems as it is mentioned only as a side note [23]. The data model is nested by the design that data sets contain fields or child data sets supported by ECL [16]. ECL, "Enterprise Control Language", was developed as a data-centric, user-friendly programming language tailored to the HPCC ecosystem. It also provides basic SQL functions, like JOIN [2, 22]. This gives an advantage compared to HBase which misses the JOIN option.

Fig. 1 gives a complete overview of data processing in the HPCC ecosystem. It shows the idea of creating data clusters that connect information between single nodes. Its idea is further explained in the upcoming text.
First of all, data gets transferred from its original location into a Thor cluster.

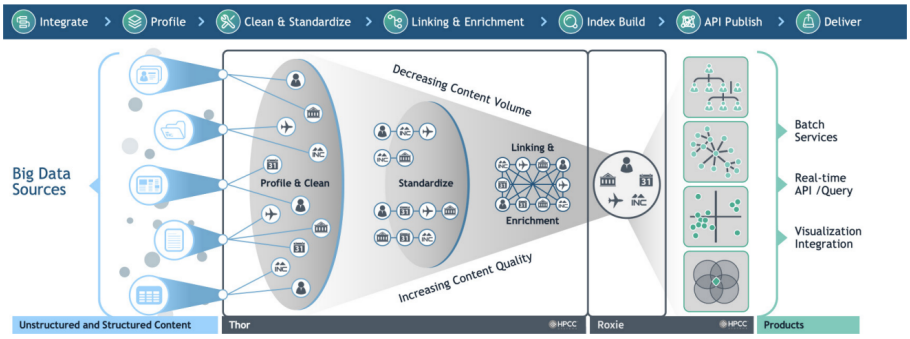

Fig. 1. Data modification in the course of processing [22].

Thor is an important component in processing. There data is cleaned, standardized, enriched and information will be extracted. Also, links between the single entities are created such that a cluster graph is generated.
Thor makes also use of parallel data processing and works on the master/slave concept where the master deals with data partitioning and manages the progress of processing and co-working of several slaves. Thor also indexes data for the upcoming querying process Roxie. Moreover, it has the strength of performing operations on individual node levels locally or across a whole cluster globally.

Then at least one Roxie cluster follows taking over indexes, providing data patterns that are interesting for querying and putting out the final result. In Roxie, the query becomes the focus of interest. For each query, it is first determined which server would fulfill the needed resources for execution. Then, the server will activate agents who actually process. The number of servers and agents isn't dependent on the number of Roxie clusters. Also, flexible data partitioning takes place only stored in the cluster by fitting to the current demands. The partitioning number represents the number of computation nodes or an interesting feature. This is shown by fig. 2. Each involved server can act as an individual agent taking care of information. Also, each server has a duplicate data version of another server for safety reasons [22].
Clusters also contain the metadata stored in RAM done by DALI which also manages work units for jobs in the HPCC ecosystem.
When there is the need of updating a file, the updated version can be added to the SuperFile which is the bin for logical files treated as one element [16, 23].
Generally speaking, data manipulation only takes place in memory, not on the original data stored on disks. It becomes clear that only after data modification as part of processing, working with data schemas becomes possible. However, the records of computation history are stored such that it is possible to build up the modified data by instructions [16, 22, 23].

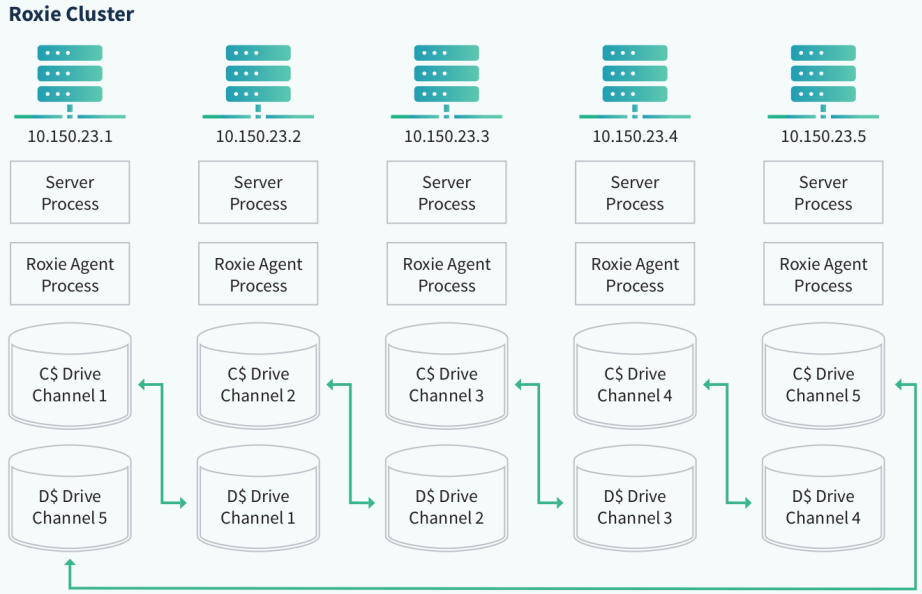

Fig. 2. Server-related distribution of a Roxie cluster. Each data pool has a saved copy as D$ Drive on a second server. Each server can be thought of as a computation node with individual agent [22]

2.2 Studies

As mentioned above Thor and Roxie build up graph-like clusters [22]. [15] grapples graph databases which are defined as graph-generating storage, representation, and querying systems. The study wants to find out how to design graph databases the best. The strengths of the models are their flexibility and the possibility of information expansion without changing data schema as there is none. Software and distributed hardware are both involved in such structures. Different concepts of efficient querying and management were introduced. One concept is to distribute the data over several servers for parallel computing. To ensure the availability of data, caching is recommended. Parallel querying turns out to be an important topic. Therefore, queries have to be thought in a different manner than relational systems. Therefore, the index choice is essential which shall support both, nodes and the in-between edges.

[9] focuses on the question of how to handle lots of clients simultaneously on HPCC. Two methods were presented. The first one is called the "Partitioned Data Model" (PDM) which cuts the database into areas of the same size and mixes up the areas to new computing nodes. The idea is that in the end the computing nodes shall be equally utilized such that data load through lots of clients at once can be compensated. Single areas can be split and merged again to enhance data transportation and fitting to new computing nodes. Also, the

requests should be able to find the new area location. Therefore, a routing process with the help of an allocation table was implemented. Through simulation, performance observation, and predictability it shall become possible to spread the areas strategically. A dependency graph consisting of query nodes can track the dependencies. The graph is reduced step by step. A danger is that a client can crash the process while modifying an area. Therefore, the area stays locked. For this reason, a timestamp is integrated and regularly checked.
The second presented method is called the "Replicated Data Model" (RDM). Nodes spread replications to other computing nodes. Through replication, a locking request is sent to all computing nodes. When they agree, a lock happens. When locking fails, a consensus protocol is used as a solution. Synchronization of duplicated areas shall take place in times of low client numbers.

[18] compares HPCC performance to Hadoop for which the programming language Pig is used. Pig converts requests to MapReduce jobs. The reason for the comparison is the rough similarity of the two ecosystems. Both work on distributed Big Data in a master-slave design and have the same hardware design according to [16]. On the basis of the high flexibility of HPCC PigMix data can be converted to HPCC's internal format. HPCC outperformed Pig-Hadoop on the basis of 10 million sample rows. The only exception was given in the case of the sorting task queries as Hadoop has an advantage in efficient parallelism in the Reducer as will be explained later. Also, for 270 million rows Hadoop topped HPCC in 15 out of 17 cases showing that Hadoop is more scalable. So, HPCC can be beaten when using enormous data amounts by Hadoop. However, so far the thesis hasn't cleared up what Hadoop actually is. What are its potentials and weak points? The upcoming chapter will declare the question.

3 Hadoop

3.1 Components and architecture

First of all, Hadoop is an ecosystem meaning that several technologies are brought together to overcome Big Data struggles. The draft horses of Hadoop are its "Hadoop Distributed File System" (HDFS) and the processing, retrieval, and data generation architecture "MapReduce" [8, 4, 18, 21].

HDFS is the underlying storage for the whole system. Fig. 3 describes the basic idea of how the Hadoop storage concept interacts with a client. The client wants to read and write over the data file which is stored across several 64-megabyte-sized storage blocks which are represented by the DataNodes (slaves) building up one cluster. DataNodes create, remove, and duplicate blocks. Furthermore, they give feedback to the NameNode like a "heartbeat" periodically.

Fig. 4 shows this concept quite simply. It makes clear that each DataNode consists of several disks. The same is the case for the NameNode. Within that

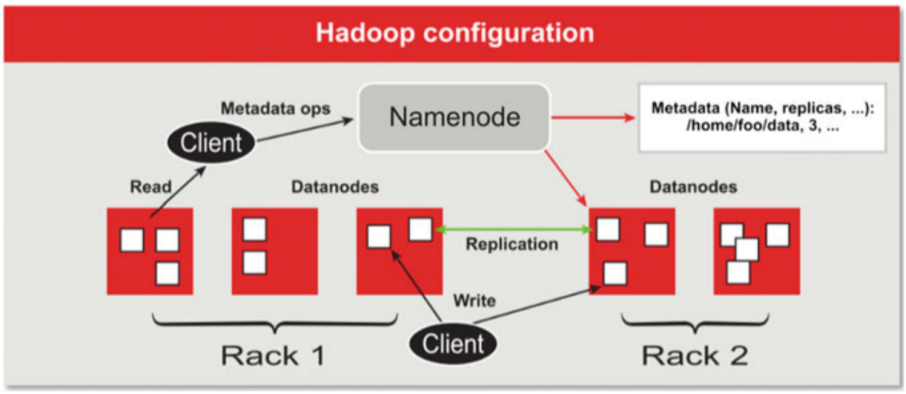

Fig. 3. Storage concept of Hadoop in interaction with a client [16].

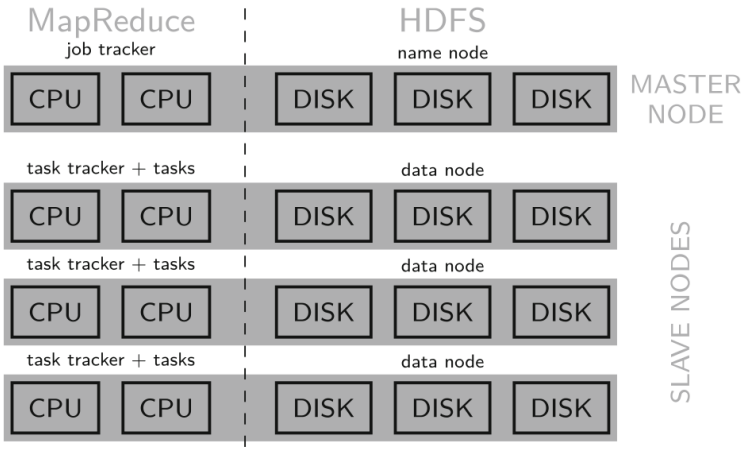

Fig. 4. Hadoop's storage architecture [25].

cluster, each block gets three replicates by default meaning that four blocks in total carry the same information. The cluster is organized by its master, the NameNode, with the task of storing file and block records that contain metadata and information about the block location. Also, the client HDFS interaction is stored by the NameNode. The representation of a NameNode is done by the file "namespace" [4, 8, 16, 19, 21]. Since Hadoop version 2.0 each NameNode gets duplicated. For the case of failure the duplicate, standby NameNode can replace the active NameNode [25].
Moreover, Hadoop clusters underlay the "shared nothing" distributed processing paradigm by connecting single systems with local memory, disk resources, and local processor [16].

Let's come to the processing strategy of Hadoop. The distributed MapReduce algorithm considers a whole cluster of semi-structured or structured data. Its first component parts are the Mappers which extract the relevant rows for a given key-value pair query where the values represent the data. The Map process splits the data which leads to parallel processing [3, 25]. Next, the Reducers aggregate data to count along all rows how often a query is fulfilled [4, 8, 14]. The fig.5 gives an overview of the MapReduce process. It explains with the help of a concrete example how it works.
Fig. 4 shows that the TaskTrackers are the slaves. They guard Map and Reduce processes. The JobTracker is the master which organizes and keeps track of the TaskTrackers. The whole MapReduce process takes place on CPUs. Each slave and the master can have several of them. Furthermore, the JobTracker is linked to a NameNode [4, 16, 19, 21, 25].

Both technologies, HDFS and MapReduce, keep the concept of distribution meaning that several processes can work in parallel to faster achieve results for high data amounts.

However, that was only the basis of the Hadoop ecosystem. There is the option to work with the NoSQL databases Cassandra and HBase. The high-level programming language Pig can be used for converting commands into MapReduce jobs in a NoSQL style. In addition, a second language, namely Hive, can be utilized for operating in a SQL-like manner. Both harmonize with HBase.

Even further, the underlying fig. 6 gives an overview of the different technologies in Hadoop. The following technologies are part of Hadoop [8]:
- Chukwa and Flume for logging
- Ambari, HCatalog, and Zoo Keeper for managing services
- Hama and Zoo Keeper for synchronization
- Avro for handling certain data formats
- Avro and Zoo Keeper for data serialization
- HCatalog and Thrift for schema creation
- Lucent for text content searching

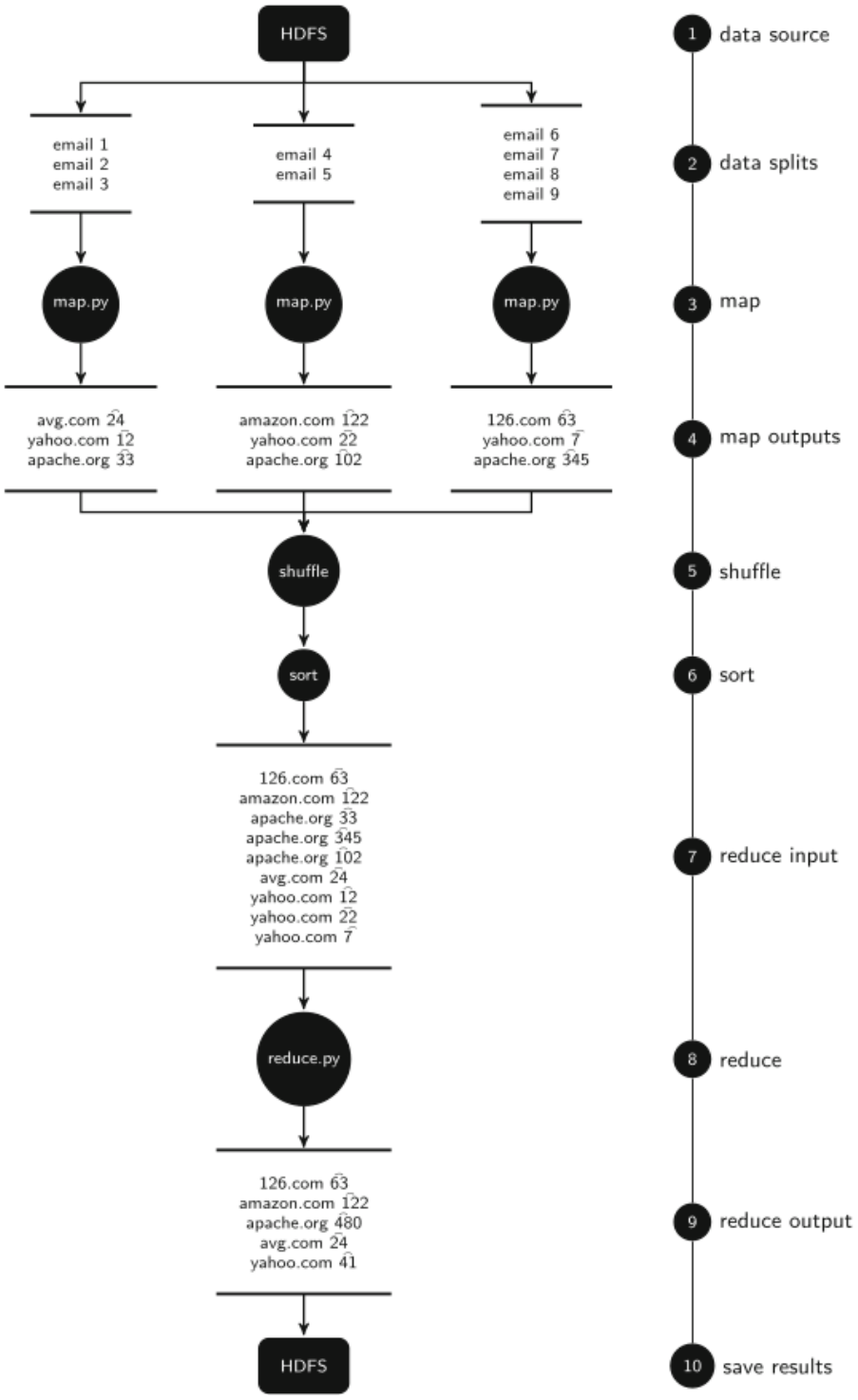

Fig. 5. An example for the MapReduce process. The Mapper outputs consist of the individual values, e.g. "apache.org", and the number of their occurrences when going through all keys, e. g. for "apache.org" 102 times. The Reducer aggregates the results such that the occurrence numbers for the same entity are added up [25].

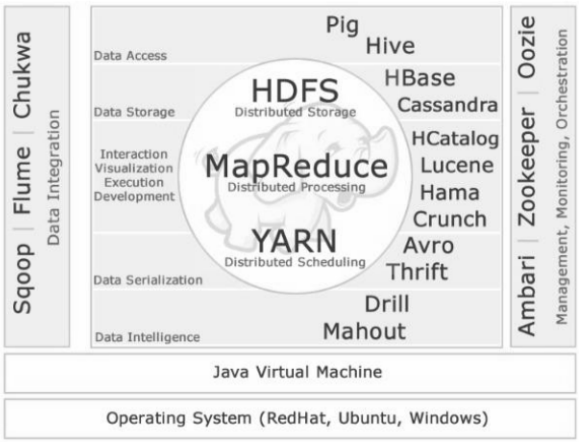

Fig. 6. Hadoop ecosystem components [8].

- Drill for data analysis
- Mahout for machine learning
- Sqoop for converting data work to relational data thinking
- Oozie for connecting partitioning projects

However, [16] also criticizes the approach of gluing different methods. It explains it with mismatching programming languages, traits, and operating traits.

3.2 Principles of read and write operations

When the client wants to read certain information, the NameNode first delivers the storage location of the searched blocks. In fig. 7 this step is marked as with "1". Afterwards, all needed blocks from different DataNodes are read such that the needed information can be put together. These steps are highlighted with "2", "3" and "4" in fig. 7.

The process to fulfill a write operation is described by 8. In step "1" the NameNode verifies the client's request by checking for conflicts. Then in step "2" the write operation takes place. As the third step, the replicas are updated. "4" verifies the success or failure of the single operation. Finally, the NameNode has the power to succeed in the writing process in the case enough write operations were successful (step "5").

3.3 Design choices

Hadoop was developed to handle Big Data in a range of terabytes. What happens when it has to work with lots of bytes distributed among a high amount of

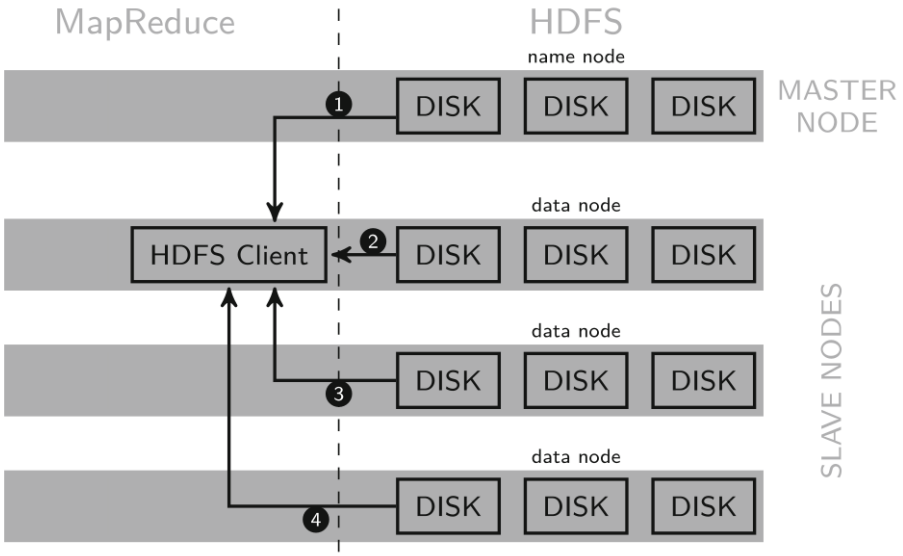

Fig. 7. Client reads information stored in HDFS. The step-by-step process [25].

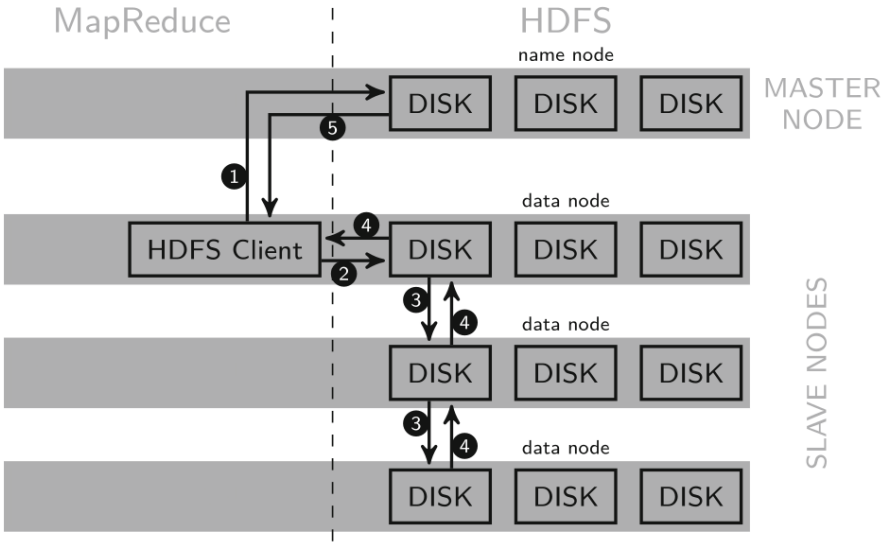

Fig. 8. Fulfillment of the write request in Hadoop. All steps have to be successful for executing the write [25].

small files? A small file is smaller than the size of a single storage block meaning that it is smaller than 64 megabytes [4]. The more small files there are, the more the NameNode is burdened. Also reading each block is more time-consuming because it has to be hopped from block to block (data sparsity). In addition, each file serves as the input of the Mapper which can lead to an overload [4].
To solve the mentioned problems [4] looks at different approaches. The first one is the "Hadoop Archive" (HAR) which aims to reduce NameNode and Map burden. The small files are put together into a tuple. However, there is a need to distinguish between single files. Therefore, an index is needed. It is contained in the master's metadata information such that the method is too slow. At least, scalability is kept high by reducing namespace usage.

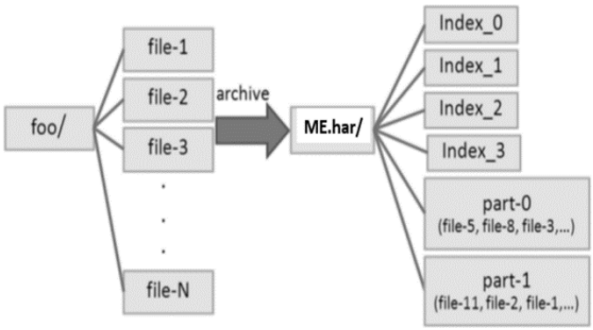

Fig. 9. File structure of NHAR. The individual "part" files consist of several concatenated small files. Several "index" files are generated to evaluate where which file is stored [4].

For this reason "New HAR" (NHAR) was developed. Fig. 9 describes the file structure of NHAR. The small files are merged into bigger files but this time files are binned by a hashing function which makes use of the individual file names. The bins provide the belonging index. The index tells us where the individual small file is stored. Several hash tables were created to store the indexes. According to fig. 10, the merged files are merged to one big "Part File". NHAR has high scalability because of its implementation to simply add files. Its competitors need a bit less memory.
Besides that, "Improved HDFS" was introduced, the only method provided in [4] working on DataNode instead of NameNode basis. Small files get integrated into a big file where a cache manager reads the files. All files in the same directory will end up in the same big file. Because of the cache managers high scalability is reached. Accessing performance of files increases.
"Extended HDFS" (EHDFS) makes use of pre-fetching and file merging. Firstly, files get merged into bigger files that have the size of a storage block. Those merged files contain the beginning of the indexes of the small files stored in this

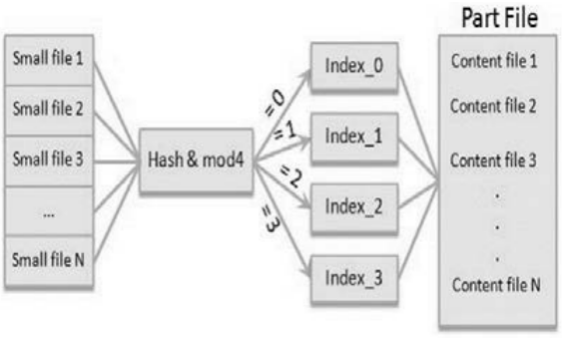

Fig. 10. Hashing for NHAR. Location information of a file through an "index" file is given by the hashing of the file name. The "Part File" merges the single "part" files [4].

merged file. The NameNode only collects metadata on the level of the merged files. This leads to decreased memory usage and improves file access performance. Then, file mapping takes place. It makes it possible to tag the single file names to the relating blocks by requesting the NameNode for the exact location of the single file and also finding the DataNode.

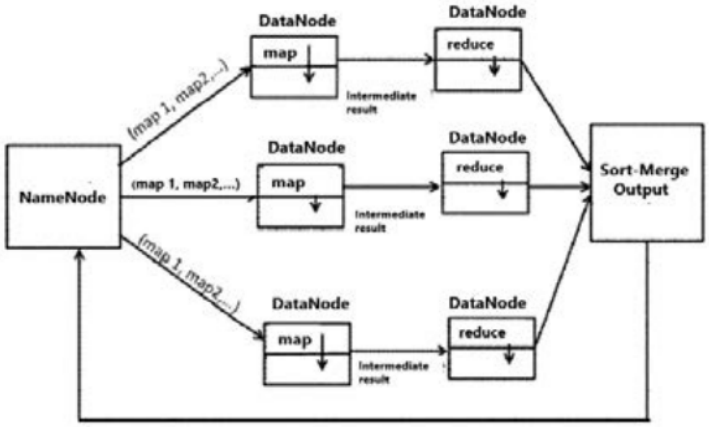

Fig. 11. The workflow of CombinedFileInputFormat [4].

The last method is named "CombinedFileInputFormat" and is inspired by the MapReduce method. Fig. 11 visualizes the workflow of the method. In the Map process files are merged together to be treated as a big file. The Reducers will then run on the single files in parallel such that processing works much

faster. However, it struggles with memory consumption in the NameNode. Finally, the methods were compared. For different disciplines, various methods were the best performers.
- Best memory savings: HAR
- Best reading performance: CombinedFileInputFormat
- Best scalability: CombinedFileInputFormat
- Smallest overhead: EHDFS, NHAR

4 HBase

4.1 Architecture

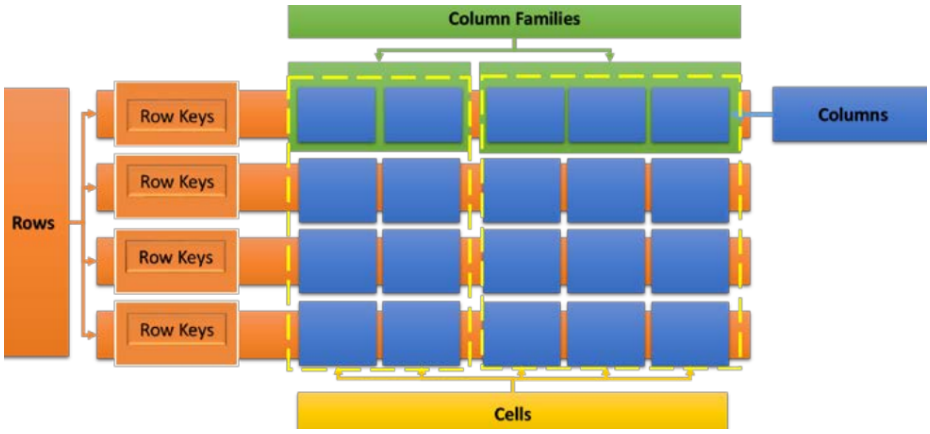

Fig. 12. The architecture of an HBase table [20].

First of all, let's have a look at the architecture of HBase to understand the design choices better. HBase is a column-oriented database being part of the NoSQL approach, meaning that it aims to process Big Data [20]. It stores information in table-like structures. Fig. 12 gives a rough overview of such a table. In the following, Column Families (CF) are described closer in the context of column usage.

Each CF hosts several columns, which are embedded entities [20]. The columns are also called qualifiers [12]. CF ties the single qualifiers in one physical storage file together. When a read query searches for information in an HBase table contained in a single CF, it will only have a look at exactly the file which belongs to the CF in question. All qualifiers of the CF contained in this CF will be searched through until the query is answered. With this strategy, lots of search time can be saved [21].

Row keys	Time_Stamp	Column family 1 (CF1)		Column family 2 (CF2)		
		CF1:Col 1	CF1:Col 2	CF2:Col 3	CF2:Col4	CF2:Col 5
Row1	Time stamp 1			Value 3	Value 4	Value 5
Row2	Time stamp 2	Value 6	Value 7	Value 8	Value 9	Value 10
Row2	Time stamp 3	Value 11	Value 12	Value 13		

Fig. 13. The architecture of an HBase table. Different timestamps can be assigned to the same row key [20].

The curiosity of HBase is that it is suited for working with time-related data. Timestamps can be assigned to data when it enters the table. Fig. 13 shows that the individual key, which is needed to find wanted information, can relate to several timestamps of an entity. So, when an entity changes over time, it is assigned to the same row but with an additional updated timestamp [14, 20]. So, it is possible to follow the time track [12].
Cell contents can be found by the following structure: row key, CF:ColumnName, and timestamp. Bring them together in one query to find a precise cell [21]. Row keys are sorted lexicographically and are represented as byte arrays telling the start and end borders of regions [12, 20]. Regions can be formed by splitting a table by rows. Several sequential rows can build up one region [7, 12]. How the region design can be used for efficient HBase usage will be discussed in the following subchapters. The single cells can contain any data type encoded in byte arrays, for instance, boolean, float, and characters [10, 12, 20].

4.2 HBase in the Hadoop ecosystem

As mentioned above HBase was implemented as part of the Hadoop ecosystem. The main idea is that it can work well with HDFS to enrich the ecosystem with the possibility to use fast queries. The collaboration of the two technologies works well as both are designed for lots of read operations but struggle with writing commands, especially since HDFS doesn't support writers in parallel. Moreover, MapReduce fits into those constraints [21]. The database works with the help of key-value pairs where the row IDs are the keys and the column's contents of the single rows are the values [10].

However, several papers observed that there may come up problems. Especially, for writing queries, HBase can struggle with its performance time. Therefore, it is essential to develop design methods for data storage, search key representation, and data model strategies. Looking at the CAP theorem, Consistency, Availability, and Partition tolerance, HBase can provide consistency and partition tolerance. It fails at the availability meaning that read and write queries sometimes can't be immediately fulfilled. Then the client would have to wait until the needed node number is free [5, 12]. HDFS has an HRegionServer which

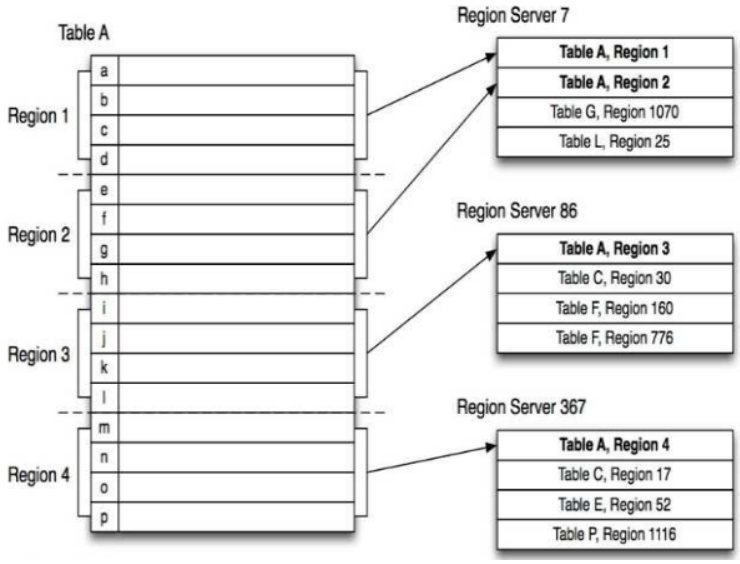

Fig. 14. The storage concept of regions in HBase [12].

stores several regions and serves as a slave [1, 12, 19]. Fig. 14 gives an example in which regions of several tables can be assigned to the same server. The HRegion-Server manages the logs, Memstores, each StoreFile per CF, query processing, splitting regions, and client interaction [7, 12]. In the upcoming text, it will be clear what regions are. When the HRegionServer tries to find a replication on the storage level, a new HRegionServer needs to take over to handle other client requests. This is organized by the master HMaster. From this it follows, that availability can't be fulfilled [1, 12, 19].

[21] describes a hybrid architecture according to which the cells of the HBase tables are only links to the image files stored in HDFS. A writing query can add new images to the current data set. Compared this HDFS-HBase structure with the HDFS-MySQL structure. For a read operation, it was observed that the HDFS-HBase has an advantage although the differences weren't that high. More precisely, for 1000 users a maximal difference of circa 50 requests per second was observed [21]. Also, [12] shows that HBase harmonizes with HDFS provided by Hadoop.
So, HBase is excellently suited for read queries but it struggles with write operations as every duplicate has to be overwritten. The default replication factor of three generates three copies of each file which are stored on different Data Nodes and therefore are located on several discs [19]. However, the strength of replication lies in the fact that it is possible to replace failing nodes with replications [12].
Also for write requests, HDFS-HBase had a slight advantage. However, the ad-

vantage increased when the task was to handle a read-write-load for a maximum of 100 clients. This shows that the HDFS-HBase structure harmonizes and therefore it makes sense to have a closer look at the upcoming HBase review.
The following discussed papers handle the topic of how to integrate HBase into Hadoop. Moreover, it investigates improving HBase's performance through various design choices. It will become clear that the studies follow to some extent similar ideas but also come up with completely new approaches. In the following sections will be discussed the architecture design by modifying Column Families, and the query design by developing search keys and combinations out of it.

4.3 Column Family design

[5] has shown that the HBase table design can be crucial for the performance of Big Data. Concretely, it had a look at the design of CFs and columns itself. The idea of [5] is to manipulate the column number per CF. First, it is pushed into one CF with one column. Here it is named the single-CF approach. The competitor multi-CF was designed in form of ten CFs. All of them contain one column. In total, this approach has the same information amount. Then the study accomplished read and write queries with different data loads.
It was shown that in all cases the single-CF clearly outperformed the multi-CF condition in average delay time. The reason for that observation was already explained above. For the single-CF only one file has to be searched through whereas for the multi-CF up to ten files had to be opened. For writing queries the multi-CF approach needs circa five times longer because several storage files have to be visited to overwrite information in there.

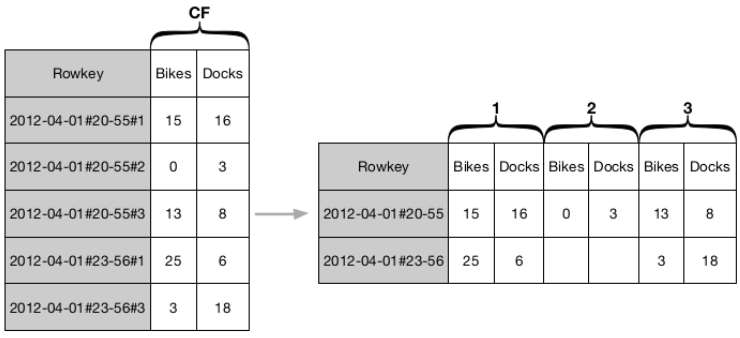

Fig. 15. An instance for CFIDM in which one CF is split up into several ones decided by the last row key's element [7].

[7] brought up the question of handling multidimensional queries and thus developed the data model "Column Family Indexed Data Model" (CFIDM). The

rough concept is to treat one query target column as several CFs. An instance is provided by fig. 15. In concrete, the idea of vertical partitioning comes into play. With this approach, it would be possible to not only access one storage file containing a long column of information but several storage files at once through parallel processing. The number of CFs to be formed is the maximal number of different entities for the same timestamp. To prevent building up a single region, the CFIDM is split up across several tables. It follows that the CFs are distributed over several tables. Therefore, the number of CFs stays the same for individual tables but the number of regions is increased. The query time will stay the same. Also writing performance stays similar. However, with too many CFs query time can be drastically increased due to compaction through lots of query requests. However, it would still be better than the one-table CFIDM. Also, the storage space gets wasted through too many tables and/or CFs. However, the CFIDM with only one table still performs the best when not too many requests appear even compared to different key formulation strategies. This shows, that it always depends on the data amount whether splitting one column into several CFs is beneficial. This study provides the opposite result of [5].

Furthermore, [27] deals with the CF schema by implementing a "Genetic Algorithm" (GA). It is orientated at machine learning for the approximation of the best CF building construction. The algorithm is defined by the following stages.
The Encoding states that one column can appear in different CF but not in the same CF twice.
The Fitness Function Design evaluates the single solutions. The cost function is the sum of the number of needed CF for the request, the variance of a number of columns between the different CFs, the duplicate rate,

$$\frac{1}{numberCFsCurrentSolution}$$

and the loading frequency of individual CFs. Each member within the sum gets its own weight factor.
The Recombination method helps to find a global solution by adding columns inside one CF and removing repeating columns within the CF. In addition, it exchanges columns between CFs where again duplicate columns within a CF are deleted. Also splitting and merging CFs is done.
Finally, take two solutions to merge them together. Again remove duplicate columns. Afterward, split data into train, test, and validation sets, to train the algorithm on the train set, choose the algorithm on the validation set, and evaluate the general performance on the test set. Results of the study show that the CF schema generated by GA reduces at least 20 percent of response time and performs significantly better than the baseline CF designs. However, as soon as the data changes by getting new values or simply removing and adding new column types the GA has to be trained again on the new data.

4.4 Key design for faster querying and better data distribution

In designing HBase schemas several options are possible. First of all, through the key-pair design of HBase, the key plays a crucial role in querying. When tying up several value attributes to one key together this key is called a composite key. Interestingly enough, the order of the row key can be manipulated for faster query results. For example,

key = rowKey + (maxTimestamp - currentTimestamp)

[20]. Also, the length of column titles, CF designation, and row key names can influence the query speed. The longer the name is, the worse the performance will be. For row key design the binary representation format saves space [20].

HBase works on the basis of enormous data amounts in the area of petabytes. Ironically, it becomes on HDFS inefficient by using a small data volume as it would be too small to be distributed automatically and therefore only processed as one unit [12]. [13] wants to fix this issue with the help of pre-partitioning and hashing. The work aims for avoiding the "hot spot" problem. This happens when after the region split new data will be added afterwards to the table. Then the data goes only into the latest region. Meanwhile, the older regions get ignored and therefore can't use their storage potential. To solve the issue, [13] strategically formulates regions based on time, more precisely 60 regions for 60 minutes is equivalent to one hour. As a result of that, a new key for querying is built as follows where '+' shall only be the symbol that represents the connection of byte arrays:

key = timedRegionNumber + random8bitString + currentTimestamp + uniqueDataRecordID

. Through that key design, it becomes possible that data is evenly distributed across all regions and thus the hot spot issue is solved.

Also [17] deals with balancing loads on HDFS or other distributed data storage but the used solution path is a different one. The study makes use of the petabyte-scale storage for "facility information access protocol" (FIAP). It shall fit into the concept of HBase. Therefore, it is built for time-series data. Time shall be part of the key to access values. Each time sequence entity, e. g. a sensor, has a unique PointID which can be the path to the data structure. Referring to HBase, FIAP shall help in developing keys to handle lots of sequences, delivering good performance, and providing uniform data distribution. FIAP is interposed between HBase's driver and the input which shall be embedded into an already built-up HBase table. This is also shown by fig. 16. In conclusion, also in this paper comes up the "hot spot" issue which shall be solved by the FIAP layer. As mentioned above an efficient key shall be designed. For this reason, the paper used the MD5 hash function to bin the time sequence entity's value and also its unique serial number. The bin corresponds to a certain region for equal distribution and avoidance of hot spots. This leads to the following key:

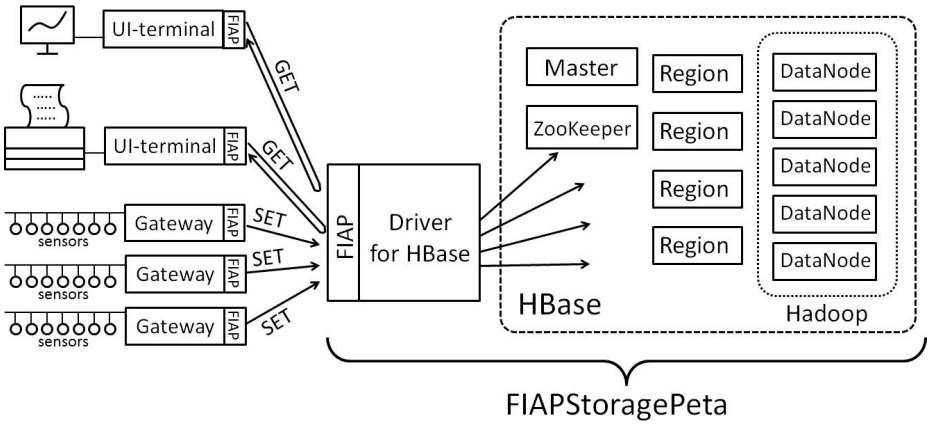

Fig. 16. FIAP as insertion layer between inputs/outputs and HBase [17].

key = HashBinRegion + SerialNumber + LatestTimestampOfSequence

Ḃeyond that, on HBase, three tables are built which make the querying process effective by strategically using the developed key. The results show that the study was a success. Data was as planned uniformly distributed. Also, the reading process worked in the space of $10 * *3$ faster when systematically using the time series key. Even for a data amount of $10 * *7$ the average processing time only needed ca. two seconds.

Secondary indexing goes even further. So, [24] actively used past user behavior to design an HTable. Pre-partitioning and secondary indexing were used for performance improvement and Join integration.

RowKey = HorizontalPrepartionRegion + userID + timestamp

, where pre-partitioning and timestamp shall result in more uniform data partitioning. The key design was explicitly chosen because it fulfills the needs of the most often query type [24]. Therefore, faster access is possible [20].
A secondary index was built to take care of the second most frequent query scenario. It is used additionally as a second key for finding the searched value. In doing so, an isolated CF is created for faster querying. The paper states

key = RegionStartKey + IndexName + BankBranchValueAndTransactionTime + rowKey

, where the bank branch value and transaction time are the query attributes we are looking for. The experiment found that indeed the queries fitting to the first-row key design were retrieved the fastest [24].

The previous papers showed us that using metadata and time information out of data to build up keys can increase performance dramatically.

4.5 Data partitioning beyond key design - the concept of regions

[5] outlines that partitioning strategies of data in the HBase storage model play a crucial role. The background is that the rows of HBase are split into different storage/server regions. The storage buffer MemStore collects newly to add data rows for the HBase table. As soon as a memory threshold with a default value of 128 MB is reached the region is split. The data will append the StoreFile which collects data also until a certain threshold. Then StoreFiles of the same CF is merged together. They are represented by HFiles which carry row key, qualifier, and timestamp. Finally, split the HFiles to transfer data to the disc storage on HDFS, namely a block within a DataNode. So, it is not possible that a region ends up on several DataNodes. It follows, that the RegionServer has to run on the basis of the individual DataNode [5, 7, 13, 19, 20]. The memories of HBase, MapReduce, and HDFS are separated and communicate with each other [19]. It becomes clear, every time working with data a data transfer is needed which costs computational power. Therefore, too extreme data partitioning can be obstructive.

[5] came up with the idea of implementing different region numbers on the same amount of data rows such that the more regions we have, the more DataNodes can be used. Indeed reading performance increased with the usage of several regions due to the fact that several nodes can be visited at once. In addition, the more regions were implemented, the faster the writing process could be executed.

In addition, the [19] picks up the idea of horizontal and vertical partitioning. The first main idea is that the more region we have, the higher the level of parallelism becomes as the regions can be fed into the Mappers in parallel. However, when there are too many of them, the MapReduce processing would need more time.
Vertical partitioning improves reading performance which was already discussed in [5]. It converts a column into several CFs. The more CFs are generated, the later the threshold for region-building is reached. [19] created 60 CFs out of one column which results in bad writing performance because in each of the 60 storage files values have to be added when a new row is inserted into the HTable.
The one-column approach has also advantages compared to the multi-CF approach. It can be queried fast with the help of row IDs instead of the needed column name as the key. Moreover, it scales better. When only reading one CF, it performs faster and only has to search for one file in one region [20].
It becomes clear that the partitioning of regions goes hand in hand with CF design. In the case of vertical partitioning IRA and FSS strategies from [19] needed the multi-CF approach as the one-column approach led to much higher execution time [19]. Those approaches will be explained in the upcoming subchapter. Here they are used to explain that it always depends on the context

which design pattern performs the best.

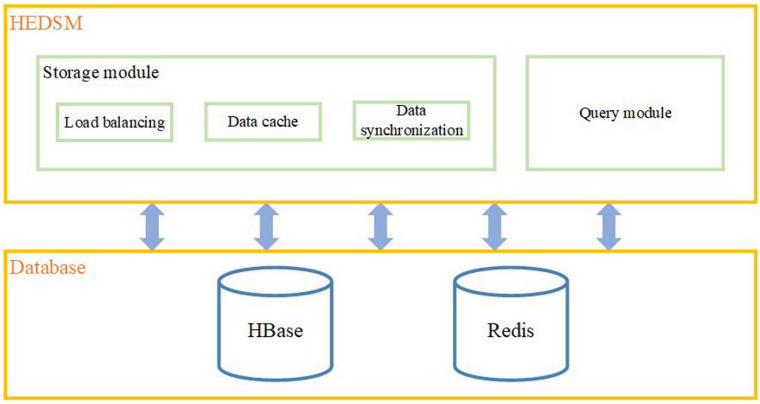

Fig. 17. HEDSM architecture [26].

Moreover, [26] had a look at the global load balancing. It presents the "High Efficient Distributed Storage Middleware" (HEDSM) which consists of a storage and a query module. Its rough architecture is shown by fig. 17. The storage module cares about load balancing, data cache, and data synchronization. The Redis database is used for storing frequently visited data. Therefore, it's the cache. Through Redis, there is less need to search several disks for certain queries using HBase. Therefore, the query process starts at Redis. HBase carries data and is synchronized with Redis. Again the hot spot problem is solved with load balancing but this time for regions and RegionServers. First, the region balancing is described as a local version. In pre-partitioning, region size gets defined by the formula

$$RegionSize = \frac{MemorySizeRegionServer \cdot ProportionRegionServerToMemStore}{SizeMemStore \cdot NumberCF}$$

. To build up the row key the FNV1-32-HASH hash algorithm is used to create the evenly distributed region starting keys. However, it still can happen that busy regions are stored together on one RegionServer whereas lazy regions could end up together on the same RegionServers. So, the distribution of operation burden between RegionServers is not given.
What makes this work special is the fact that thematizes global load balancing to solve the above-described issue. HEDSM observes the requests and tracks which RegionServers are exhausted and which are seldom used. Then the regions of over-loaded and under-loaded RegionServers are changed partwise such that

both RegionServers are loaded more uniformly.
Finally, the paper states that data is more uniformly distributed for HEDSM than for HBase but also the number of regions is by far more. When HBase would have a higher partitioning rate, also the HBase approach would reach more even distribution. Besides that, it's a big success in the findings, that the RegionServer loads are more equally distributed whereas in HBase there are high loads for few servers. Another effect is, that HEDSM needs a bit more than half of the time of writing operations compared to HBase. For read operations, the effect is very drastic as HEDSM needs only a very small partition of what HBase needs [26]. This paper shows the importance of local and global data distribution. Data storage seems to be the essential ingredient for the effective use of HBase. Also asking in a clever way for searched data can help to improve HBase.

4.6 Query design for faster processing and Join handling

That the approach of building secondary indexes is quite famous but has variety in its implementations becomes clear in the following paper with the aim of analyzing multidimensional data fast as possible with the help of Hadoop and HBase. The paper [19] discusses two algorithms being integrated into MapReduce jobs.
The first one, "Indexed Random Access" (IRA), makes use of secondary indexes. The secondary index can be used to generate artificial regions by building up borders. So, reading performance is sped up. Also, certain samples can be found faster by combining searched attributes as keys [20].

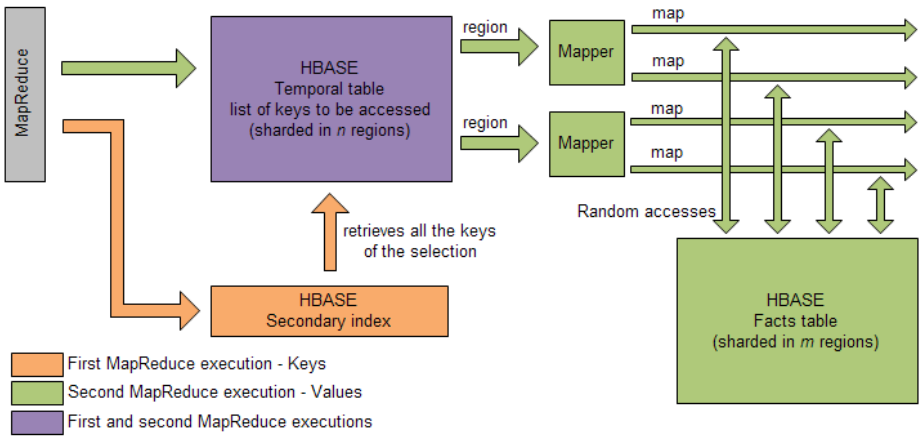

Fig. 18. IRA. Secondary index design for faster querying [19].

Fig. 18 visualizes the method's architecture. The query is split into its single attributes which will be the single keys by the Map process. This is the orange

part of fig. 18. Each of those keys is applied independently of the other keys to the HBase table to find and sort the right key-value pairs (mapper process). After that, a Merge-Sort layer follows which brings the found rows together, meaning that each found key appears once and the values of the previous phase are merged together when there were identical keys. Finally, the Reducer only chooses those keys containing the number of searched attributes in a number of their values [1, 19].
The second algorithm "Full Source Scan" (FSS) works simpler. It looks at the whole HBase table and filters out rows with fitting values according to the query in a parallel manner.
The results show that FSS is much faster in processing than the IRA up to a factor of ca. seven. The performances are on the same level when searching through a single column instead through a CF. Besides that, FSS was not impressed by the increasing number of attributes in individual queries in contrast to IRA.

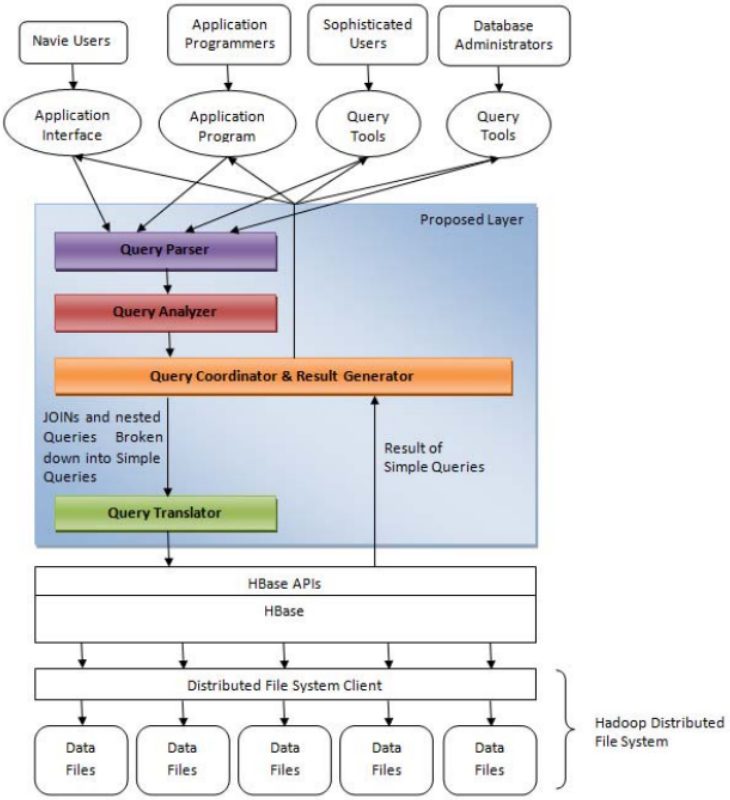

Fig. 19. Developed layer design for using Join queries in HBase [11].

Relational databases support Join operations. Through them, it is possible to create relations between two tables with the help of foreign keys. It is possible to map one entity to another. For example, the entity "customer" could have bought several products (second entity) which is a one-to-many mapping [11]. To come back to HBase, Joins aren't supported by it. To be able to execute all the different Join versions, like the inner Join and the left outer Join, three methods were designed. It was built a layer in which the join queries were split up into simple queries. Fig. 19 makes clear that an extra layer is designed. It is inserted between the query and HBase. It follows that the queries are modified but not the HBase table.
The "Hash Join Method" is the first implementation possibility. Queries are split into blocks which are called one by one. The query-relevant information of the table gets binned by a hash function. The second table can be then also blocked and compared to the current block of the first table. When ending up in the same bin we can merge the belonging rows. This leads to an inner Join. This is also the slowest option.
The second method is the "MapReduce Join Method" where the Reducer of the MapReduce processing brings all values of rows of two tables with the same key together.
The last option, "Tiled MapReduce Join Method", relaxes the memory requirement for parallel processing. It loops iteratively over the whole MapReduce process to handle data in the form of mini-batches. All in one, this paper makes use of the query and data design to enable joins to HBase [11].

4.7 Strategic logging

[6] points to the fact that through the distributed nature of HBase, transaction execution can become complicated. More precisely, it aims for transactional isolation meaning to tackle conflicts when several clients try at once to manipulate the data in one HBase region, speaking of multi-row transactions. To fulfill this aim the "Transaction Status Oracle" (TSO) was used to keep track of potential conflicts and to handle them with the help of timestamps. Information about the transaction status is collected. Secondly, a "Direct Serialization Graph" (DSG) table was created containing all read and write queries. The study could show that with the developed design HBase was able to fulfill more transactions per second for several clients.

HBase has a loose schema [1], designing one could lead to a strong impact. [14] implements a schema design for HBase sorted by event ID analyzing large process event logs. But how can it be defined? Following the paper's line of thought "eXtensible Event Stream" (XES) data structure is implemented. It creates traces of metadata logs that contain query attributes. A more precise structure is given by 20. It states that an attribute can be split into logs, traces, and events. An event defines the start and end of a happening. Within an event traces take place. Those traces are then stored by logs.

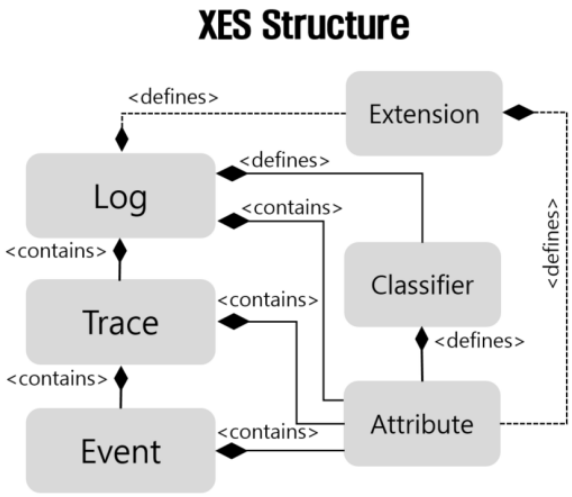

Fig. 20. log data structure of XES [14].

Beyond XES, the control flow alpha algorithm or sigma algorithm is used to get information about the process model. Also here a key for HBase is designed to handle the hot spot problem. So, we get

$$rowKey = eventID + SuccessorEventID + timestamp + traceID$$

. The column event attribute is located in one CF. It contains the original HBase data and the column event relation containing flow information of entities. Through control flow similar events are stored on the same region server.

4.8 HBase disclosure

From the HBase section, it becomes clear that there is heavy research concerning HBase improvement for different case usages. Papers often have similar approaches but implement them in a completely different way matching up the requirements of certain query types and data set characteristics. Successes in performance can be observed but those are always limited by different circumstances like the data set size, request type, and so on.

5 Conclusion

This thesis was only a small glimpse into the world of data management, analysis, and processing Even though, it could show the weak points and strengths of the

different technologies HPCC, Hadoop, and HBase. It always depends on the context to decide which system should be used. When searching for a system for a real project, it makes sense to also look at studies investigating how to overcome the weaknesses of the single approaches concerning the own project needs.

References

[1] Bakshi, K. (2012). Considerations for big data: Architecture and approach. *IEEE Aerospace Conference Proceedings*.
https://doi.org/10.1109/AERO.2012.6187357

[2] Bayliss, D. (2016). Aggregated Data Analysis in HPCC Systems. In B. Furht & F. Villanustre (Eds.), *Big Data Technologies and Applications* (pp. 225–235). Springer International Publishing.
https://doi.org/10.1007/978-3-319-44550-2_8

[3] Bayliss, D. (2016). Models for Big Data. In B. Furht & F. Villanustre (Eds.), *Big Data Technologies and Applications* (pp. 237–255). Springer International Publishing.
https://doi.org/10.1007/978-3-319-44550-2_9

[4] Bende, S., & Shedge, R. (2016). Dealing with Small Files Problem in Hadoop Distributed File System. *Procedia Computer Science 79*, 79, 1001–1012.
https://doi.org/10.1016/j.procs.2016.03.127

[5] Cai, L., Huang, S., Chen, L., & Zheng, Y. (2013). Performance analysis and testing of HBase based on its architecture. *2013 IEEE/ACIS 12th International Conference on Computer and Information Science, ICIS 2013 - Proceedings*, 353–358.
https://doi.org/10.1109/ICIS.2013.6607866

[6] Cai, P., & Ni, L. (2012). An approach of multi-row transaction management on HBase with serializable snapshot isolation. *Proceedings of 2nd International Conference on Computer Science and Network Technology, ICCSNT 2012*, 1741–1744.
https://doi.org/10.1109/ICCSNT.2012.6526257

[7] Cao, C., Wang, W., Zhang, Y., & Ma, X. (2017). Leveraging Column Family to Improve Multidimensional Query Performance in HBase. *IEEE International Conference on Cloud Computing, CLOUD, 2017-June*, 106–113.
https://doi.org/10.1109/CLOUD.2017.22

[8] Dhyani, B., & Barthwal, A. (2014). Big Data Analytics using Hadoop. *International Journal of Computer Applications*, *108*(12), 1–5.
https://doi.org/10.5120/18960-0288

[9] Ellis, C., Babenko, P., & Goldiez, B. (2014). Distributed Dynamic Terrain in a High Performance Computing Cluster Environment.
https://www.researchgate.net/publication/237412326_Distributed_Dynamic_Terrain_in_a_High_Performance_Computing_Cluster_Environment/link/542fcf100cf27e39fa99796e/download

[10] Frozza, A. A., Defreyn, E. D., & Mello, R. dos S. (2020). A Process for Inference of Columnar NoSQL Database Schemas. *Brazilian Symposium on Databases, 35*, 175–180.
https://doi.org/10.5753/sbbd.2020.13637

[11] Gadkari, A., Nikam, V. B., & Meshram, B. B. (2014). Implementing joins over HBase on cloud platform. *Proceedings - 2014 IEEE International Conference on Computer and Information Technology, CIT 2014*, 547–554.
https://doi.org/10.1109/CIT.2014.77

[12] Goyal, A. (n.d.). HBase - Hadoop Database.

[13] Guo, J., & Wu, X. (2015). RESEARCH ON OPTIMIZATION OF COMMUNITY MASS DATA STORAGE BASED ON HBASE.*CCT 2015*.
https://doi.org/10.1049/cp.2015.0863

[14] Ham, S., Ahn, H., & Pio Kim, K. (2019). HBase based Business Process Event Log Schema Design of Hadoop Framework. *Journal of Internet Computing and Services*, *20*(5), 49–55.
https://doi.org/10.7472/jksii.2019.20.5.49

[15] Kalmegh, P., & Navathe, S. B. (2012). Graph database design challenges using HPC platforms. *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, 1306–1309.
https://doi.org/10.1109/SC.Companion.2012.160

[16] Middleton, A. M. (2016). Data Intensive Supercomputing Solutions. In B. Furht & F. Villanustre (Eds.), *Big Data Technologies and Applications* (pp. 257–305). Springer International Publishing.
https://doi.org/10.1007/978-3-319-44550-2_10

[17] Ochiai, H., Ikegami, H., Teranishi, Y., & Esaki, H. (2014).Facility Information Management on HBase: Large-Scale Storage for Time-Series Data. *Proceedings - IEEE 38th Annual International Computers, Software and Applications Conference Workshops, COMPSACW 2014*, 306–311.
https://doi.org/10.1109/COMPSACW.2014.54

[18] Ouaknine, K., Carey, M., & Kirkpatrick, S. (2015). The Pig Mix Benchmark on Pig, MapReduce, and HPCC Systems. *Proceedings - 2015 IEEE International Congress on Big Data, BigData Congress 2015*, 643–648.
https://doi.org/10.1109/BigDataCongress.2015.99

[19] Romero, O., Herrero, V., Abelló, A., & Ferrarons, J. (2015). Tuning Small Analytics on Big Data: Data Partitioning and Secondary Indexes in the Hadoop Ecosystem. *Information Systems, 54*, 336–356.
https://doi.org/https://doi.org/10.1016/j.is.2014.09.005

[20] Shriparv, S. (2014). The Storage, Structure Layout, and Data Model of HBase. In A. Hussain, K. Colaco, P. Bisht, P. Kadam, J. Dharmaraj, & S. Mukherjee (Eds.), *Learning HBase: Learn the fundamentals of HBase administration and development with the help of real-time scenarios* (pp. 121–144). Packt Publishing.
https://www.programmer-books.com/wp-content/uploads/2019/12/Learning-HBase.pdf

[21] Vora, M. N. (2011). Hadoop-HBase for large-scale data. *Proceedings of 2011 International Conference on Computer Science and Network Technology, ICCSNT 2011, 1*, 601–605.
https://doi.org/10.1109/ICCSNT.2011.6182030

[22] Taming the Data Lake: The HPCC Systems® Open Source Big Data Platform. (2022).
https://cdn.hpccsystems.com/whitepapers/wp_introduction_HPCC.pdf

[23] Understanding HPCC Systems and Spark - A Comparative Analysis. (2022).
https://cdn.hpccsystems.com/whitepapers/wp_hpccsystems_spark_comparison.pdf

[24] Wang, X., Liu, Y., & Zhang, L. (2016). Research on the Application of Bank Transaction Data Stream Storage based on HBase. *Proceedings of the 2016 International Conference on Engineering Management (Iconf-EM 2016), 30*, 253–358.
https://doi.org/10.2991/iconfem-16.2016.46

[25] Wiktorski, T. (2019). Hadoop Architecture. In X. Wu & L. C. Jain (Eds.), *Data-intensive Systems Principles and Fundamentals using Hadoop and Spark: Advanced Information and KnowledgeProcessing* (pp. 51–61). Springer Nature Switzerland AG.
https://doi.org/https://doi.org/10.1007/978-3-030-04603-3

[26] Xu, L., Chen, Y., & Guo, K. (2021). A Distributed Storage Middleware Basedon HBase and Redis. In Y. Sun, D. Liu, H. Liao, H. Fan, & L. Gao (Eds.), *15th CCF Conference, ChineseCSCW 2020* (Vol. 1330, pp. 364–380). Springer Singapore.
https://doi.org/10.1007/978-981-16-2540-4_27

[27] Yang, F., Cao, J., & Milosevic, D. (2015). An evolutionary algorithm for column family schema optimization in HBase. *Proceedings - 2015 IEEE 1st International Conference on Big Data Computing Service and Applications, BigDataService 2015*, 439–445.
https://doi.org/10.1109/BigDataService.2015.20