Overview of petabyte-scale metadata storage methods and frameworks

Tim Oelkers
Fachbereich 4 - Angewandte Informatik
Hochschule für Technik und Wirtschaft
Berlin, Deutschland
Tim.Oelkers@Student.HTW-Berlin.de

***Abstract*—This work addresses methods and frameworks for storing metadata in the petabyte range. SQL and NoSQL databases are compared and their advantages and disadvantages are described. Relational databases reach a limit in the Big Data domain, NoSQL databases offer better scaling behavior. This is illustrated by the requirements of the H.E.S.S. experiment. Distributed file systems, various file formats, federated databases, and data lakes and data warehouses are presented. Specifically, the frameworks Lustre, Apache Parquet, ROOT, Delta Lake, Apache Hive, and Greenplum are evaluated. Finally, some suggestions for improving the use of frameworks in the H.E.S.S. experiment are given.**

***Index Terms*—Big Data, Petabyte scale, NoSQL, Data Warehouse, Data Lake, Distributed file systems, Open Data, File formats**

I. Introduction

Nowadays, the processing of data has reached a size that overwhelms normal systems. This work is therefore intended to demonstrate methods and frameworks for dealing efficiently with these data volumes. In particular, the handling of metadata is considered. Also, the aspect of scaling will be addressed, as this is the most important aspect for future projects. This project should show an approximate direction and the state-of-the art of the Big Data. Concrete frameworks can only be considered briefly. In particular, a performance analysis will not take place. It is also limited to open source frameworks and criteria such as commercial support are not considered.

There are countless application areas that consider the problem of huge amounts of data. In this thesis, the field of astronomy will be considered in particular, since they produce corresponding amounts of data. A project that has been running for several decades and is therefore suitable to determine typical requirements of a Big Data project, is called The High Energy Stereoscopic System (H.E.S.S.). At its core, it consists of a system of telescopes for the study of cosmic gamma rays. It is located in Namibia and was first commissioned in 2002. After an expansion in 2012, 5 telescopes are now operating in conjunction to increase the detection area and cover different energy ranges. Among them is the largest reflecting telescope in the world. In total, the project has about 250 employees [1].

Ramin Marx [2] provided concrete insights into the project's requirements. According to him, the telescope produces a data volume of 300 GB per hour. This involves operating the telescope every night, with an estimated average runtime of 6 hours per night. However, this can vary due to external influences such as weather or full moon. This amount of data, however, only concerns the raw data in the form of images with a resolution of about 4 megapixels. The total amount of raw data has thus reached a size of petabytes and is stored in a classical file system. This is sufficient for this project, as the storage is more for archival purposes. Queries of the raw data take place very rarely. The data is stored in the ROOT file format, which was developed by CERN for their data volumes (see section V-B2). These data are then further calibrated in several steps. For example, noise is removed or broken pixels are interpolated. This calibration ideally runs only once per data set and reduces the amount of data by a factor of about 20. A large amount of metadata is also extracted from the raw data. Examples include target coordinates, duration, date, telescope temperature, and weather data such as the thickness of the atmosphere. These data are stored in a MySQL database due to frequent analysis. This also allows parallel access by the different staff members. The database stores about 1000 different attributes and has about 100,000 entries. In addition, the database is kept redundant to prevent data loss. Critically, it must be mentioned that this amount of data is far away from the considered petabyte size.

It can therefore be seen that the number of write accesses far exceeds the number of read accesses. So there is no need to focus much on optimizing the access times here. Also, all data are independent of each other. So no classical relations between the data are necessary. At most, historical queries can concern different runs, for example, several runs of the same galaxy at different times. However, this occurs very rarely. Furthermore the number of columns with 1000 seems very high. Here the question arises whether a structured, SQL-based database is useful. It is still important how a change of the schema is handled. Here, currently the database is refilled every time. This is possible because all metadata can be derived from the raw data. This refilling occurs only about 2 times a year, but of course no access is possible during this time. Therefore, this work shall investigate, among other

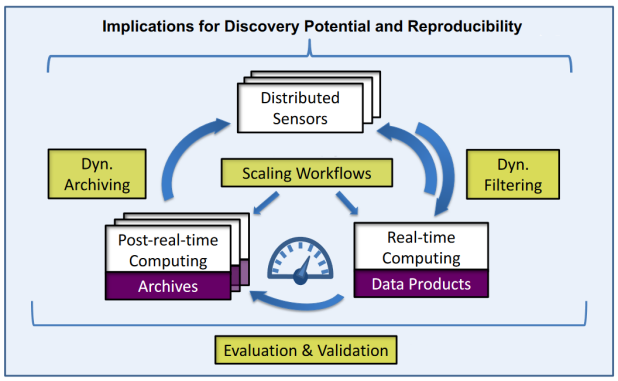

Fig. 1: Dynamic model for processing data

things, whether a NoSQL-based database would be better suited for the H.E.S.S. experiment.

II. Big Data

The problem of growing data sizes is summarized in computer science under the term Big Data. However, Big Data is also often used as a buzzword for various technologies and social problems. It is therefore important to find a uniform definition for it. The most common definition describes the term using the 3 "V "s [3]. The *volume* describes a volume of data that cannot be processed by conventional systems. The *Variety* describes the characteristic of the data to appear in different forms. This includes, above all, unstructured data that must nevertheless be processed. The *Velocity* describes the speed with which new data is generated. It is so high that the data cannot be cached and must be processed in real time. However, this leads to the problem of **Data Irreversibility**. Data can only ever be processed once, so if an error occurs or certain information is skipped during processing, this results in a permanent loss of information.

The PUNCH4NFDI consortium has addressed these requirements and developed a model for processing data, which is shown in Figure 1. It is not determined in advance which data will be processed, but this is decided dynamically during the process. Data is processed in real time, for example by machine learning applications, and the result influences which data is collected by filtering. The data is then evaluated and validated, and then archived. Since the amount of data cannot be stored completely, dynamic archiving must again take place. This again influences the collection of the data. All the processes described here must be adapted to the high data rates, which is made possible by good scalability [4][5].

The handling of large-scale metadata in view of “Data Irreversibility“ is intensively explored in the Task Area 5 of the PUNCH4NFDI consortium [5]. Here, use cases and their challenges from astronomy and high-energy physics are discussed. Task Area 3 is preparing a document on handling metadata in simulations. In astroparticle physics, metadata use cases are considered by the projects KCDC (KASCADE Cosmic-ray Data Center) and GRADLCI (German-Russian Astroparticle Data Life Cycle Initiative) [6]. A design of future metadata systems in high-energy physics is discussed in Ref. [7]. In Lattice QCD, a flexible description of metadata (based on XML formats) was introduced already two decades ago [8] and has received a status update recently [9].

III. Databases

Databases are the most obvious form for storing huge amounts of data. Here, however, there are different forms that have to be compared. If properties of databases are to be evaluated, then usually the **ACID** properties of database are looked at. The term ACID stands for *Atomicity, Consistency, Isolation and Durability* and was introduced in 1983 by Theo Härder and Andreas Reuter [10]. It basically describes how robust the database is against common problems. Thus, the database should remain consistent even in the presence of errors or concurrent queries.

A. ACID

Atomicity describes the desired behavior, that changes of the data could be executed either completely or not at all. It should never happen, for example, that when writing a set of data, only half of the data is written. For this, transactions are usually used that execute the changes one after the other, but only make the data visible to other processes after completion. For other processes the data change thus happens atomically. If errors occur in the meantime, everything that has been written so far can be deleted again by a rollback.

Consistency describes the property that the consistency of the data is maintained by the database. On the one hand, this concerns the integrity conditions defined in the schema of the database. These include, for example, adherence to value ranges (e.g. only positive numbers) or adherence to referential integrity. In other words, the database must ensure that data referenced by foreign key also exists in a reachable manner. The other type of consistency concerns distributed systems. Here, data is usually kept redundantly in replicas to increase access times at different locations in the cluster. The system must now ensure that the data is identical in all replicas. Accesses at different points in the cluster must therefore always provide the same responses.

Isolation describes the property that transactions must not get in each other’s way. Thus, if one user makes a read request and another writes to that area at the same time, then the read user must not be affected by the write operation. Also, write transactions running in parallel may get in each other’s way. These problems are usually solved by locks that achieve mutual exclusion (mutex) for the critical area. Since this limits the degree of survivability and in the worst case deadlocks can occur, some database systems neglect this point or apply it only to certain partial problems.

Durability describes the behavior that data is permanently stored in in the database. This sounds obvious at first glance, but this behavior must also be guaranteed if the database

fails due to a crash. Many systems do not write the data immediately to the hard disk, but keep them in a much faster cache in the main memory. Here, many database systems keep a transaction log that documents all changes to the database. If a crash occurs, the changes that have not yet been made can be implemented by re-executing the saved queries.

B. CAP-Theorem

In theory, all ACID properties can be fulfilled, but in practice it usually looks different. Above all, since many databases represent a distributed system. This is where the **CAP theorem** comes into play. It was established in 2000 by Eric Brewer and states that in a distributed system the properties Consitency, Availability and Partition Tolerance can never all be achieved at the same time. Consitency describes the property that data accesses across all nodes produce the same result. Availability describes not only the general availability, but also the speed at which accesses are made. Partition tolerance describes the ability of the database to remain consistent if individual nodes can no longer communicate with each other. Also the general scalability over nodes is covered by this point. The individual points influence each other, as shown in the Venn diagram in Figure 2. So, for example, if a database focuses on fast access times and scalability, then consistency suffers. Another example is the focus on consistency and fast access times. This is the case with classic, relational database systems. However, these then have significant disadvantages in scaling, which is problematic for Big Data requirements.

C. SQL

Basically, there are two subdivisions of databases: SQL- and NoSQL-based databases. They differ in whether they use the SQL query language for changing and reading data. However, the distinction by language is not simply an arbitrary choice, because certain architectural conditions must be met for the language to be used. SQL is based on relational algebra, which transforms each query into a set of operations for manipulating relations. These operations are usually: projection, selection, cross product, union, difference and renaming. It is important to note that data for this must be in relational and structured form. On the one hand, the type of data must be defined beforehand, which is called database-**schema**. On the other hand, data must be normalized beforehand and relations between data must be defined.

This normalization and creation of relations is difficult in practice due to widely varying data. Also, schema migrations are often very difficult to implement when requirements change. Another point of criticism is the weak scalability of relational databases. In most cases, they can only be installed on a physical server. Also, there are limits on data sizes for most databases, which can be problematic in the Big Data area. Well known databases like MySQL have table limits of 64TB and row limits of only 64KB. Other databases like Oracle even have a limit on the entire database of 2PB [11]. There are also performance issues when indexing huge amounts of data. In practice, these can take a very long time and lead to exponential growth in memory consumption. Read and write accesses also scale only conditionally with an increase in data volume and become increasingly cost-intensive.

D. NoSQL

In contrast to SQL-based databases, NoSQL databases go without a relational approach. They often neglect strong consistency and the ACID criteria. This does not mean, of course, that the data within the database is unusable, since weak consistency is nevertheless realized. This can be realized on the one hand by the principle of *eventual consistency*. Thereby it is guaranteed that the data becomes consistent at an undefined point in time. For example, one cannot rely on the fact that directly after a write access the read access can already find the data. This principle is also called **BASE** (Basically Available, Soft state, Eventual Consistency). On the other hand, NoSQL databases often do not offer any transactions or restrict them, for example, only to certain commands.

NoSQL databases can be divided into several categories. On the one hand, there are **key-value-stores**. These can be compared to hash tables, as they assign exactly one value to each key. Moreover, they are able to read these values again in constant time. A variation of these are **document-oriented** databases. These allow the storage of more complex data models for a key. The documents are not structured and can be whole files. Semi-structured data such as JSON or XML can also be stored and efficiently read. There are also **column-oriented** databases. These store columns one after the other and separate attributes of an entity. This has on the one hand a positive influence on the access times, since less data must be scanned, which takes up with databases the majority of the time. Also columns, which are not needed, can be skipped more easily. The disadvantage is the higher effort when writing a data set, because here the scanning effort is significantly higher. In addition, a column-oriented format allows better compression, since similar data is stored next to each other. Many compression methods such as LZW or RLE build precisely on this spatial similarity and thus achieve a higher compression. Finally, there are **graph-based** databases. These model relations between data as graph, with which hierarchical or interlaced structures can be represented well. While relational databases determine these relationships at runtime via joins, these are stored in graph-based databases and thus achieve fast access times. Popular areas of application are, for example, social networks where users can follow other users.

The different NoSQL database types have different strengths and weaknesses. However, all of them have the ability to do without a predefined schema. This enables them to store very diverse data. It also makes them capable of responding to changes in the data model. Furthermore, the mitigation of consistency allows them to be distributed across different servers, which leads to good horizontal scaling. This leads, for example, to constant access times as the amount of data

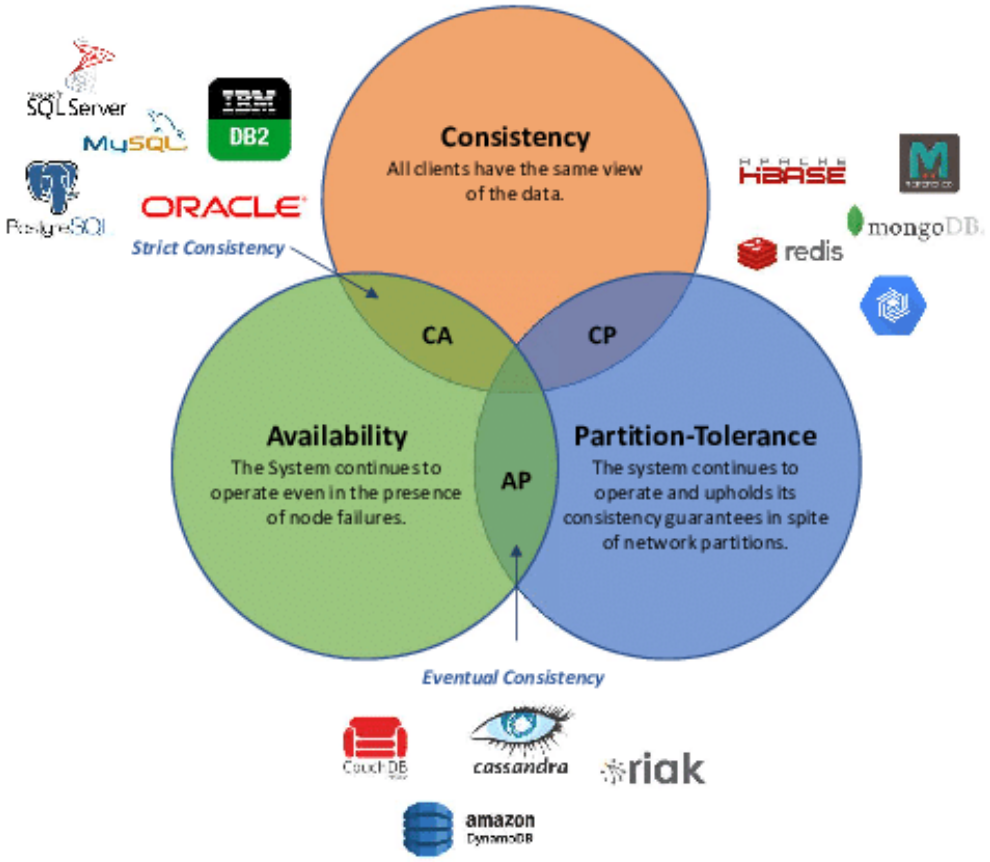

Fig. 2: CAP theorem on the basis of different databases

grows, which is not the case with a relational database. A disadvantage is that joins between data are very difficult to model. Graph-based databases are an exception. All NoSQL databases instead do without normalization of the data. In this case, the relation is stored as a new entry. Thanks to the good scalability, the higher amount of data is negligible in practice. Another disadvantage is the lack of a standardized query language. Here, each database usually offers its own language, which is specialized for the respective use case. There are alternatives like GraphQL, which are based on SQL and try to be established as a standard. Weak consistency and missing transactions are also an exclusion criterion for some use cases. One example is banking, where the order and timing of write accesses are of course critical.

IV. Aggregated Data Sources

Besides the direct use of databases, there is another view of data storage. It combines several databases, which mostly contain data on a specific area, into a new, more comprehensive data source. This then greatly helps to process huge amounts of data, as they are all accessible under the same interface. There are two ways to enable this view here.

One is that it can be just a pure abstraction layer. These systems are often referred to as **federated database systems**. In this case, the data is not copied and remains physically in their respective data stores. Access to the individual databases is then handled transparently for the user by the application through a so-called mediator, so that the user has the impression of accessing a single data source. The mediator must split the query into sub-queries and send them to the individual databases, whereby different query languages can also be used. It is also possible to combine both SQL and NoSQL based databases.

It is important that a global schema can be found, which the individual data sources follow at least in parts. This integration of many local schemas into a global schema is a challenge in itself. Here, the global schema can either be derived from the local schemas (**Global-as-View** (GaV)) or the local schemas are subsets of the global schema (**Local-as-View** (LaV)). A well-known algorithm called bucket algorithm is able to build a global schema from existing schemas, but with a complexity of NP-complete. It is accordingly slow for huge amounts of data. However, since the whole thing only has to be executed when the schema changes, it is not an exclusion criterion.

It is also important that the individual data sources remain autonomous and do not have to be specially adapted for an integration. It is the task of the mediator to ensure that the data from the local sources can be merged. A good definition by Andreas Bauer and Holger Günzel is: "A federated database system is a multi-database system with a (global) conceptual schema that encompasses all component systems. All component systems must thereby preserve their autonomy and local conceptual schema, i.e., they remain independent with respect to design, execution, and communication." [12]. The big advantage of this architecture is that it does not increase the amount of memory. The major disadvantage is the high complexity of finding a common schema.

On the other hand, there is the option of copying interesting data to a new, central data store. This has the advantage that data can be transformed simultaneously and thus unified. Fast access times can also be expected, since no aggregation of the data is necessary at runtime. The obvious disadvantage is increased storage requirements, which can be very critical for many Big Data projects.

A. Data Warehouse

The paradigm of aggregated data sources was first mentioned in the 1980s and is known as **Data Warehouse** Architectures. Its origin lies in business informatics and was intended to help companies make corporate decisions. Although no fixed definition could be developed, there are some criteria that apply to most data warehouses. First, data is loaded and unified from heterogeneous and distributed data sets. The unification is an important point to enable a common view. Then it is important that the data is processed according to fixed, previously defined methods. Well-known methods are, for example, data mining, visualizations or Big Data analyses, especially with machine learning. Data mining uses statistical methods to find hidden relationships and structures in data. In machine learning, a model was developed in advance from similar data that can make certain statements with new data. Bauer and Günzel define data warehouses as "a physical database that presents an integrated view of (any) data to enable analysis" [12]. Here it becomes clear that in contrast to federated database systems a copy of the data takes place.

One of the main problems with this architecture is that the data must be available in structured form. This is necessary because the use cases for the data warehouse are already defined. The way in which the data is processed is therefore strictly predetermined. Therefore, a schema must be defined in advance that applies to all data. This procedure is also described as **schema-on-write**. A process called Extract, Transform, Load (ETL) is used. In this process, data is loaded from various sources, transformed into the schema of the warehouse, and then loaded into the warehouse. For example, extraction can be periodic, event-driven, or query-driven. There are both syntactic and semantic criteria to consider during the transformation. All in all, data warehouses are very specialized data sources that are aggregated only for analysis.

B. Data Lake

In contrast to the fixed structure of a data warehouse, a **Data Lake** can also contain unstructured data. There is no common schema. Instead, the type of analysis defines the schema, which is also referred to as **schema-on-read**. This is especially important when the way the data will be used cannot be determined in advance. This is especially the case with projects that rely on voluntary, open collaboration. These projects are often referred to as Open Data projects. Data in a data lake can have a wide variety of data formats. For example, images, audio, JSON or CSV. There is no ETL process, i.e. no transformation of the data. This leads to a number of disadvantages. On the one hand, the memory consumption requirements are significantly higher and on the other hand, a lot of data does not offer any real value. It can happen that analyses become increasingly difficult and machine learning models are overwhelmed with irrelevant data. This condition is also referred to as a **Data Swamp**.

Strongly federated data lakes are also a possibility. Here, data is exchanged across institutions. A well-known data lake of this kind is operated by CERN and is called ESCAPE. It is designed to make exabytes of data in the field of astronomy and particle physics available between different research institutions. It is interesting to note that different institutions use different memories such as EOS, DPM, dCache, StoRM or XRootD. To synchronize this, three software solutions are used: GFAL, FTS and Rucio. Grid File Access Library (GFAL) is used as an abstraction layer to make the different storages known to each other using different protocols like FTP and HTTP. File Transfer Service (FTS) is used to copy data between the storages. It uses a technique called Third Party Copy (TPC), which allows data to be directly linked between compatible stores. Rucio is used as a data orchestrator. It is the layer that users interact with and which in turn uses FTS and GFAL [13].

V. Possible frameworks

A. Filesystems

File systems in themselves are a method of storing data in the Big Data domain. Of course, every database eventually also stores on the file system, but a direct use can be useful. Especially if no complex queries are needed, the overhead by a database is not necessary. Especially if data is to be archived over a longer period of time, this can be a given. It also provides very direct access to the data. Various other frameworks have their own access options, which do not always have to be compatible. For example, different query languages for databases. Files are more universal and do not have this problem. Distributed file systems can also achieve very good scaling. Above all, good performance (RAID 0) and redundancy (RAID 1) can easily be achieved by using RAID without having to implement this through software.

Well-known implementations of distributed file systems are, for example, Hadoop Distributed File System (HDFS), Lustre

or BeeGFS. HDFS is a file system from the Hadoop environment and therefore well integrated into classic Big Data frameworks such as Apache Spark. BeeGFS is a file system from the Fraunhofer Center for High Performance Computing, which, in addition to good performance and scalability, also offers ease of use in particular. This makes it an ideal file system in the academic domain. Lustre focuses on very high performance and is therefore heavily used in today's supercomputers. 60 of the 100 fastest supercomputers in the world, including the fastest, use this file system. It supports several thousand clients, several petabytes of storage, and more than a terabyte per second of throughput. All distributed file systems have a similar structure, so they are described in more detail below using Lustre as an example.

Distributed file systems are usually not directly responsible for storing files, but form an abstraction layer over classic file systems such as EXT4. Instead, they are responsible for coordination and the management of metadata. The architecture is shown in Figure 3. Lustre has one or more metadata servers with one or more metadata targets. Here the separation becomes clear, the servers are logical interfaces which serve physical targets in the form of local file systems. The server administers metadatas such as file names, access authorizations and folder structures in a Linux inode. Storage locations are also stored in an FID location database. The data is then stored by object storage servers on one or more object storage targets. For clients this separation is transparent and they interact with a classic POSIX compatible file system. Clients cannot edit the underlying file systems to maintain consistency in the system. Files are usually split into smaller chunks and distributed in distributed file systems. This *file striping* provides chunks in the size of 1 MB, which are stored on different object storage servers with a round-robin procedure. This leads to a better performance, because data can be read in parallel. Ideally, RAID hard disks are used here. In addition to classic Ethernet, communication between the servers can also take place via remote direct memory access (RDMA) implementations such as InfiniBand or Omni-Path, which can further increase performance. To achieve POXIS compatibility, concurrent accesses to the same data must be excluded. This is ensured by a *distributed lock manager* (LDLM), which uses byte range locks to achieve this. This allows multiple clients to have read access to overlapping sections. Also thanks to striping, multiple clients can write to different chunks at the same time, leading to a reduction in the locking bottlenecks. Cache coherency is also ensured by the metadata server. Other features of Lustre include the ability to store data and metadata together. This data-on-metadata can reduce the overhead associated with very large numbers of small files. This would otherwise lead to a bottleneck with a single metadata server. Lustre also has the ability to build a layered storage structure. It is often useful to store data on different media depending on how often it is requested. For example, old data can be archived on magnetic tapes, while newer data is stored in caches in RAM. Lustre manages this archiving based on defined rules and takes care of the coordination between the systems. Finally, distributed file systems always have mechanisms against a failure of nodes. Here, heartbeat messages are usually exchanged and failover servers are defined. These are usually other servers in the cluster to cause no overhead in normal operation [14].

B. File formats

1) Apache Parquet: Apache Parquet is an open source, column-oriented file format. As described in section III-D this offers a performance advantage over row-oriented formats when reading the data. And disk accesses are the biggest bottleneck in many systems, so this performance advantage is noticeable. Also, a significantly higher compression can be achieved, which is very helpful for archiving. Specifically, *Dictionary Encoding*, *Run Length Encoding (RLE)*, *Bit Packing* and *Delta Encoding* are used for encoding. The different encodings are used automatically so that the best result is achieved. Thanks to the column-oriented storage, this can also be different for different columns. For the compression then different, very well known methods like Snappy, GZip or LZO can be used [15].

Apache Parquet was also designed to separate metadata from the actual data. This allows related data to be split across multiple Parquet files. Furthermore, Parquet uses Apache Thrift to describe the metadata. Apache Thrift is a description language for the exchange of data between different applications. These can also be programmed in different programming languages. Thrift generates, on the basis of an API definition, different client and server implementations and data classes in languages like C++, Java or Python. The reading and writing of Parquet files is thereby strongly simplified [16].

Besides Apache Parquet there are other column-oriented file formats. Here, for example, RCFile or ORC should be mentioned, which cover the same use cases. One advantage of Apache Parquet, however, is that it has greater support for various Big Data frameworks and is optimized for Apache Spark. Even pandas, a popular Python framework for data analysis uses Parquet as a file format. It is also explicitly designed for self-referencing datasets and uses a novel *record shredding and assembly* algorithm for more efficient storage. Furthermore, Parquet has the ability to combine different schemas and add new columns [15].

2) ROOT: ROOT is not only a file format, but also an open source software for reading this data. It was developed at CERN to be able to process the data volumes of the LHC. In the meantime it has developed to a general file format for different use cases. It holds data worldwide in an order of serveral magnitude of Exabyte. It was written in C++ and has its own interpreter called Cling, which can also be used as a command line tool. Bindings for Python also exist, so that many machine learning libraries can be connected. Meanwhile, the interpreter is C++11 standard compliant. Nevertheless, it can also ignore minor syntax errors like missing semicolons to allow easier usability. Data is stored in ROOT in a binary

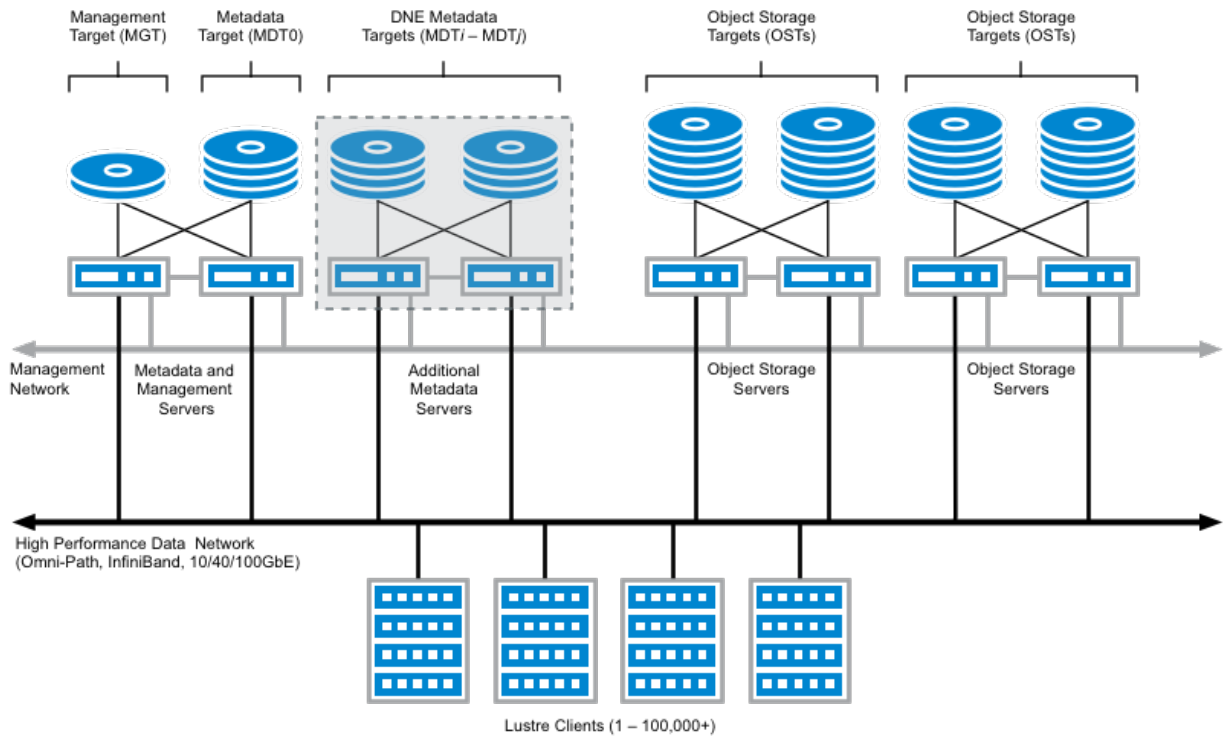

Fig. 3: Architecture of the Lustre filesystem

format and forms a tree structure, which provides for faster reading times. This tree structure can also be spanned across multiple ROOT files. This allows data to be stored in different locations and still be read as a single object. Furthermore, ROOT stores metadata with the actual data. The files are thus self-describing.

Furthermore, the ROOT software is able to perform statistical analyses. It supports a number of mathematical functions, some of which can be executed in parallel. The generation of data is also possible, so that simulations can be easily generated. It is also possible to create 2D and 3D visualizations. Various graphs, plots or histograms are available here. These can be easily published thanks to various export formats, so this is very suitable for the research field [17][18].

C. Delta Lake

Delta Lake is an open source framework for petabyte-scale data storage. It uses its own architecture called Lakehouse. This combines the two architectures data warehouse and data lake and was presented in a whitepaper by the developers. They promise that "Lakehouses combine the key benefits of data lakes and data warehouses: low-cost storage in an open format accessible by a variety of systems from the former, and powerful management and optimization features from the latter" [19]. According to the authors, the evolution of Big Data architectures is illustrated in Figure 4. It shows that the focus was originally on structured data in data warehouses. Due to growing data volumes, the focus then switched to unstructured data lakes, which, however, continued to offload higher quality data to data warehouses in an ETL process. The authors argue that this unnecessarily increases complexity and that the data does not reach the user quickly enough because the processes are time-consuming. They also criticize the fact that no direct access to conventional data lakes is possible and that this always has to be done via integrated interfaces. Delta Lake therefore uses open standards such as Apache Parquet and still enables important features such as ACID compatibility, versioning and indexing. It is also still important how the ETL process is executed. Delta Lake uses a Medallion architecture for this. In this process, data is refined further and further in a pipeline and schemas are made more and more restrictive. Raw data from various sources ends up in the bronze layer. No selection is made. In the silver layer, data is validated, duplicates are removed and transformed. The gold layer then contains data in the form in which users need it. In particular, aggregations, joins and filtering are performed to minimize access times [20].

Other design decisions included direct support for machine learning and data science applications. Here, Delta Lake uses declarative DataFrames, an abstraction layer of the Python library pandas. A metadata layer on top of the Data Lake is also being introduced. It enables ACID compatibility and other management features such as rollbacks and change logs. Performance is not constrained and there is no copying of data between layers. Instead, this layer can actually improve performance through caching by prioritizing frequently loaded

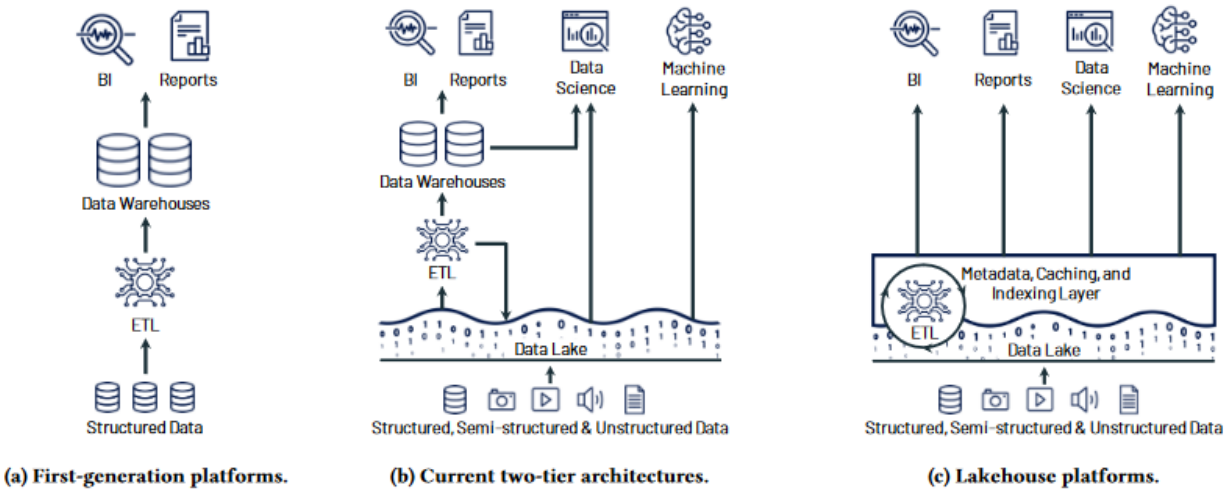

Fig. 4: Evolution of Big Data Systems towards Lakehouse Architecture

data. Performance measurements by the authors show that Delta Lake is more performant and significantly cheaper than well-known cloud providers such as AWS, Azure and Google Cloud [19].

Another feature is the ability to integrate various other compute engines. These can then access Delta Lake both read and write. Here, the documentation mentions Apache Spark, Apache Flink or Apache Hive, among others. There is also an application called Delta Standalone, which provides interfaces for more direct access. However, these are only available for Java applications. There is also a project for importing data from SQL-based databases [21].

D. Apache Hive

Apache Hive describes itself as a distributed, fault-tolerant data warehouse that can process petabytes of data via SQL [22]. It was originally developed by Facebook, but is now being further developed under an open source license. It builds on the well-known MapReduce framework Hadoop and offers possibilities to integrate different database systems such as Apache HBase and HDFS-like file systems under a central interface. Other well-known HDFS-like file systems are, for example, Amazon's S3 or Google Cloud Storage, so that the integration of cloud-based storage also seems to be given. Various YARN-based execution engines can also be integrated, which ultimately execute the queries. The queries are made in a SQL-like language called HiveQL. This is transformed at runtime into queries that the underlying execution engines understand. Here, for example, MapReduce, Spark Jobs or Apache Tez Queries are to be mentioned. A fixed schema is not necessary, schema-on-read is applied. Data integrity is checked at runtime.

A central component in Apache Hive is the Hive Metastore Server (HMS). It provides the various integrable clients with metadata such as tables and partitions in a central database. This abstracts the nature of the data from the actual storage and provides a way to respond to updates to the data. This feature is also referred to as **Data Discovery**. This enables the integration of frameworks such as Apache Spark, Impala or Presto to process the data. The access to the metadata is thereby controlled via a separate API. The database here is a classic, relational database. By default, an integrated Apache Derby database is used, but the integration of other databases is also possible.

Like any RDBMS, the ACID criteria are very important. Hive supports the full ACID criteria, but not for all SQL commands. According to its own statement, Hive is also not designed for classic multi-user queries with transactions. Instead, it maximizes horizontal scalability [23].

E. Greenplum

Greenplum is a Big Data application that applies a *massively parallel processing* (MPP) architecture to the very well-known relational database PostgreSQL. According to its own statement, it is capable of processing petabytes of data without performance loss. It achieves this by the possibility of distributing PostgreSQL databases on arbitrarily many instances and to scale so horizontally. It thus represents a federated database system. It is an open source project under the Apache 2 license, which is strongly sponsored by VMWare and is also very actively developed. Other features include the ability to integrate various, external data sources. Among others, Hadoop, Google Cloud Storage, ORC, AVRO and Parquet are mentioned. It also natively supports various ways to run Machine Learning and Big Data analytics distributed on the cluster. This is achieved through the integration of Apache MADlib. Its query planner called GPORCA is also singled out by the developers. It was designed specifically for Big Data applications and transparently distributes write and read queries across different nodes [24].

Technically, Greenplum consists of a cluster with a coordinator server, a backup coordinator and various segment nodes. The

data resides on the segment servers, with the coordinator server managing the system's metadata. Queries are also processed by the coordinator server and automatically distributed. Each segment server in turn runs different segments, which represent different database instances. In addition, segments are kept redundant, with data residing on different physical nodes for resilience. How the data is distributed must be specified by the user when defining the schema via data definition language (DDL). Thus, it requires a common schema across all nodes [25].

Greenplum supports the SQL:2003 standard and is ACID compliant according to its own statement. However, the performance should also be better for non-transactional queries. Performance measurements show very good parallelizability for the latest Greenplum version 6. Overall, it is considered a very good database system in the Big Data area for the majority of all use cases [26].

VI. Conclusion

The problem of the growing metadata volume can be well illustrated by the H.E.S.S. experiment. It is true that the methods currently used there are sufficient. However, the dynamic character of their metadata is characteristic for future requirements, such as at the Square Kilometre Array Observatory (SKAO): for the reconstruction of measurement results, in particular, a large number of time-varying boundary conditions must be stored.

The limitations of SQL-based relational databases have been demonstrated. As a solution, there are NoSQL databases without a fixed schema and with better scalability. Different features and their advantages and disadvantages have been explained. Furthermore, aggregated databases as well as data warehouses and data lakes have been described. Different frameworks for storing and processing data were presented. These are of course differently suitable for different use cases. In the following, the frameworks of the H.E.S.S. experiment are evaluated from the author's point of view.

For the H.E.S.S. experiment, the storage of raw data in a file system is very well suited. It is more for archiving purposes and therefore no complex queries take place. In addition, storage as a file offers advantages such as easier export to external researchers. Here, the use of a distributed file system like Lustre can be considered to minimize access times. Since distributed file systems distribute data transparently for the user and have the same POSIX specifications as existing file systems, this should be easy to integrate. Especially in the sense of Open Data and the FAIR principles, an easy and performant access for researchers worldwide should be realized. The ROOT data format currently in use is suitable for research thanks to integrated analyses, but ROOT software must also be used to read it out. This is not suitable for other use cases. A more open file format such as Apache Parquet would not only fulfill the FAIR criteria better, but also offer advantages in read times and compression due to the column-oriented format.

The storage of metadata in a MySQL database can be improved. Even though performance is not an issue here due to the relatively small amount of data and number of users, the use of a NoSQL database should be considered. In the future, the amount of data may increase further due to better telescopes. The number of users can easily multiply as the data is published. Current refills with annoying downtimes can also be prevented by databases without a fixed schema. Since no relations between data are necessary, a document-based database such as CouchDB or MongoDB is a good choice thanks to better scalability.

For projects with larger data volume requirements, federated databases can be useful. They offer very good scalability thanks to the distributed system. Here Greenplum is a very good alternative if relational databases are necessary. It offers, besides the good performance, especially a good integration with other Big Data frameworks. Delta Lake is also a good framework that combines the advantages of data lakes and data warehouses. However, it seems to focus more on companies and integrated analyses. In particular, the weak API seems to be a problem here. Apache Hive also seems to be very well suited when petabytes of data have to be stored. It convinces especially with a good integration into the Hadoop Ecosystem.

The frameworks considered are far from complete. There is a need for further research, especially in the aspect of performance. Frameworks such as Rucio or XTENS [27] also seem to be very well suited for the scientific area and should be considered. RUCIO in particular has a very convincing deployment with CERN's ATLAS experiment. There it administers more than one billion files, with altogether 450 Petabyte of data which are world-wide distributed [28].

VII. Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 460248186 (PUNCH4NFDI). Special thanks to all involved PUNCH4NFDI members, especially to Andreas Redelbach for reviewing this paper and to Hermann Heßling for his continous support and topic suggestions.

References

[1] H.E.S.S. collaboration. *H.E.S.S - The High Energy Stereoscopic System*. URL: https://www.mpi-hd.mpg.de/hfm/HESS/pages/about/.

[2] Ramin Marx. Center for Astronomy at Heidelberg University, private conversation.

[3] Isitor Emmanuel and Clare Stanier. "Defining Big Data". In: BDAW '16. Association for Computing Machinery, 2016. DOI: 10.1145/3010089.3010090. URL: https://doi.org/10.1145/3010089.3010090.

[4] The PUNCH4NFDI Consortium. *PUNCH4NFDI Consortium Proposal*. 2020. DOI: 10.5281/zenodo.5722895.

[5] The PUNCH4NFDI Consortium. *Curation and metadata - concepts for data irreversibility*. in preparation. 2023.

[6] Victoria Tokareva. *Metadata curation use cases in astroparticle physics*. Oct. 2022. DOI: 10.5281/zenodo.7157292. URL: https://doi.org/10.5281/zenodo.7157292.

[7] T. J. Khoo et al. "Constraints on Future Analysis Metadata Systems in High Energy Physics". In: *Computing and Software for Big Science* (2022). DOI: 10.1007/s41781-022-00086-2. URL: https://doi.org/10.1007/s41781-022-00086-2.

[8] Eric H. Neilsen and James Simone. "Lattice QCD Data and Metadata Archives at Fermilab and the International Lattice Data Grid". In: (2004). DOI: 10.48550/ARXIV.CS/0410026. URL: https://arxiv.org/abs/cs/0410026.

[9] Frithjof Karsch, Hubert Simma, and Tomoteru Yoshie. "The International Lattice Data Grid – towards FAIR data". In: *PoS* LATTICE2022 (2023), p. 244. DOI: 10.22323/1.430.0244.

[10] Theo Haerder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Comput. Surv.* 4 (1983), pp. 287–317. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[11] *Limits relationaler Datenbanken*. 2023. URL: https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems#Limits.

[12] Andreas Bauer and Holger Günzel. *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung*. dpunkt, 2013. ISBN: 3-89864-785-4.

[13] Di Maria, Riccardo, Dona, Rizart, and on behalf of the ESCAPE project. "ESCAPE Data Lake - Next-generation management of cross-discipline Exabyte-scale scientific data". In: *EPJ Web Conf.* 251 (2021). DOI: 10.1051/epjconf/202125102056. URL: https://doi.org/10.1051/epjconf/202125102056.

[14] *Introduction to Lustre Architecture*. 2017. URL: https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf.

[15] *Apache Parquet GitHub Repository*. URL: https://github.com/apache/parquet-format.

[16] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable Cross-Language Services Implementation". In: 2007. URL: http://thrift.apache.org/static/files/thrift-20070401.pdf.

[17] I. Antcheva et al. "ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization". In: *Computer Physics Communications* 180.12 (2009), pp. 2499–2512. DOI: https://doi.org/10.1016/j.cpc.2009.08.005.

[18] *ROOT Documentation*. URL: https://root.cern.

[19] Matei Zaharia et al. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics". In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf.

[20] Databricks. *Medallion Architecture Documentation*. URL: https://docs.databricks.com/lakehouse/medallion.html.

[21] *Delta Lake GitHub Repository*. URL: https://github.com/delta-io/delta.

[22] *Apache Hive Homepage*. URL: https://hive.apache.org/.

[23] *Apache Hive GitHub Repository*. URL: https://github.com/apache/hive.

[24] *Greenplum Homepage*. URL: https://greenplum.org/.

[25] *Greenplum GitHub Repository*. URL: https://github.com/greenplum-db/gpdb.

[26] Steven Nuñez. *Greenplum 6 review: Jack of all trades, master of some*. 2019. URL: https://www.infoworld.com/article/3452517/greenplum-6-review-jack-of-all-trades-master-of-some.html.

[27] *XTENS GitHub Repository*. URL: https://github.com/xtens-suite/xtens-app.

[28] *Rucio Homepage*. URL: https://rucio.cern.ch/.

[29] Eric Brewer. "Towards Robust Distributed Systems". In: *Symposium on Principles of Distributed Computing* (2000). URL: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.