SQL-on-Hadoop Engines Explained

A Technical Whitepaper

Rick F. van der Lans
Independent Business Intelligence Analyst
R20/Consultancy

May 2014

Sponsored by

Table of Contents

1 Management Summary

Big Data And Hadoop – *Hadoop* is being regarded as one of the best platforms for storing and managing *big data*. It owes its success to its high data storage and processing scalability, low price/performance ratio, high performance, high availability, high schema flexibility, and its capability to handle all types of data. Unfortunately, Hadoop APIs, such as *HDFS*, *MapReduce,* and *HBase*, are quite complex. They require expertise in Java programming (or similar languages) and require in-depth knowledge of how to parallelize query processing efficiently. The downsides of these interfaces are a small target audience, low productivity, and limited tool support.

The technical interfaces of Hadoop lead to a small target audience, low productivity, and limited tool support.

The Need For SQL-on-Hadoop Engines – What is needed is a programming interface that retains HDFS's performance and scalability, offers high productivity and maintainability, is known to non-technical users, and can be used by many reporting and analytical tools. The obvious choice is evidently *SQL*. SQL is a high-level, declarative, and standardized database language, it's familiar to countless BI specialists, it's supported by almost all reporting and analytical tools, and has proven its worth over and over again. To offer SQL on Hadoop, SQL query engines are needed that can query and manipulate data stored in HDFS or HBase. Such products are called *SQL-on-Hadoop engines*.

SQL-on-Hadoop engines can query and manipulate big data stored in Hadoop.

Lately, the popularity of SQL-on-Hadoop engine is growing rapidly. Here are just a few of the many SQL-on-Hadoop engines available: Apache Drill, Apache Hive, CitusDB, Cloudera Impala, Concurrent Lingual, Hadapt, HP Vertica, InfiniDB, JethroData, MemSQL, Pivotal HAWQ, Progress DataDirect, ScleraDB, Shark, and SpliceMachine.

On the outside most of the SQL-on-Hadoop engines look alike. They all support some SQL-dialect that can be invoked through ODBC or JDBC. Internally, they can be very different. The differences stem from the purpose for which they have been designed. Here are some potential use cases for which they may have been designed:

- batch-oriented query environment (data mining)
- interactive query environment (OLAP, self-service BI, data visualization)
- point-queries (retrieving and manipulating individual objects)
- investigative analytics (data science)
- operational intelligence (real-time analytics)
- transactional (production systems)

Undesired Big Data Silos – Most Hadoop-based systems have been designed and developed by organizations for one or two use cases. The workload characteristics of these use cases are usually massive data load and execution of non-interactive, complex forms of analytics. However, Hadoop implementations can support other use cases, including interactive reporting, data stream processing, transactional processing, and text search. The growing availability of SQL-on-Hadoop engines has just widen the range of use cases of Hadoop even more.

Unfortunately, when deployed for a different use case, a specific Hadoop implementation may be unsuitable with regard to functionality or performance. Development of another use case may force an

organization to develop a second solution in which data is stored again. In the long run, this results in many data management platforms: each one designed and optimized to support a limited number of use cases. Finally, this leads to undesirable *big data silos*.

Big data silos leads to data duplication, high data latency, complex data replication solutions, and data quality problems.

The disadvantages of having big data silos are: high costs because of data duplication, high data latency, complex data replication solutions, and data quality problems. Silos may work well temporarily, but history has shown that eventually the users of these silos will want to combine data from multiple data sources. When this happens, each application is extended to access multiple data sources. This leads to a dedicated integration solution for each one of them. The result is another undesired solution: an *integration labyrinth*. For an organization it's almost impossible to guarantee that all these integration solutions are correct, efficient, and lead to consistent results.

One Hadoop platform should support all the current and future use cases.

The Need For One Data Management Platform – The ROI on all big data stored in Hadoop is increased when it's made available for as wide a range of use cases as possible, including all the new use cases offered by the SQL-on-Hadoop engines. What is needed is one Hadoop data management platform that has been designed to support all the current and future use cases, so that the need for duplication of all that big data is minimized and that the development of big data silos and an integration labyrinth is avoided.

The Whitepaper – This whitepaper explains what SQL-on-Hadoop engines are, what the technological challenges are, and what potential use cases of SQL-on-Hadoop are. Besides a high-level comparison of several of these engines, it also contains a detailed description of *Apache Drill* that brings to light some of the pertinent issues in providing SQL capabilities on big data. In addition, the MapR Technologies data management platform M7 is also described as an example of a big data platform that can support many different use cases.

2 Requirements for Big Data Systems

IT Systems in the Old Days – There was a time when the IT industry measured database sizes in gigabytes. It was a time when IT systems were primarily designed to support business processes, such as invoicing, financial accounting, manufacturing, purchasing, and product planning. These *production systems* were developed to make these processes more efficient and cheaper.

Most business intelligence environments are still processing data copied from these production systems. Periodically, data is retrieved from them, and integrated, cleansed and stored in a data warehouse. A wide range of reporting and analytical tools is used to analyze what has happened. These ERP-type production and data warehouse systems are incredibly valuable to organizations, but because every organization has them, they don't allow organizations to stand out, they don't always offer a real distinguishing capability or competitive advantage.

Increasing Importance of Analytics – Now, fast forward. Today, IT systems *can* make organizations stand out and dramatically improve the internal operations of organizations as well as improve how they are perceived by and interact with their customers, suppliers, and agents. IT systems can improve customer

care and pre-active customer care, optimize business processes, personalize products and services, and so on.

One dominant technology that can help an organization to stand out is *analytics*. Algorithms for, for example, data mining, predictive modeling, and forecasting, have become incredibly powerful and help organizations to identify *business insights*.

Analytics can make organizations stand out.

Analytics itself has been around for many, many years. There are three main reasons why it is so much more popular now. First of all, relatively inexpensive and fast computing power and data storage technology have become available for advanced forms of analytics that were unthinkable of a few years ago. They allow larger data sets to be stored for acceptable costs and working with these larger data sets can improve the quality of analytical exercises. Next, the tools have become much easier to use. Users don't need PhDs in statistics anymore to use analytical tools. And finally, more external data sources, such as social media networks, weblogs, web interactions, socio-demographic data, and numerous open data sources, have become available and can be used to enrich analytics.

Big Data – All these improvements resulted in much bigger databases than organizations were used to. This gave way to the *big data* trend, which can be considered to be one of the most popular trends in the IT industry.

Big data applications store amounts of data magnitudes larger than those in more traditional applications. For example, click-stream applications, sensor-based applications, text-analysis, and image processing applications, all generate massive numbers of records per day. The amount of records stored surpasses more often than not hundreds of millions of records.

Big data applications store amounts of data magnitudes larger than those in more traditional applications.

The sheer amount of data has a direct impact on the database technology used. For this reason, organizations started to consider other types of data storage technology that were different from familiar and classic SQL database servers. This need convinced vendors and startups to research and develop new database technology, which resulted in the market of *Hadoop* and *NoSQL* technology. Products such as Hadoop, MongoDB, Cassandra, Riak, and many more were introduced.

Requirements for Data Storage Technology – Together, big data and the new forms of analytics, raise the bar for data storage technology. To be ready for big data, data storage technology must adhere to the following requirements:

- **High data storage scalability:** To deal with the volume of the data, the data storage technology should be designed and optimized to store, process, analyze, and manage massive amounts of data over a large set of nodes and disks.

- **High data processing scalability:** To handle the database size and workloads, it's important that data storage technology is able to distribute its data processing over very large sets of nodes and supports a highly parallel architecture. It must have a *scale-out* architecture with almost no limit to the level of parallelization.

- **High performance:** It's important that analytical and reporting requests are processed fast, even if they're highly complex and even if they access massive amounts of data.

- **Low price/performance ratio:** The price/performance ratio consists of two key aspects. First, because so much data has to be stored, data storage technology must be able to exploit inexpensive commodity hardware and disk technology. Second, license fees of data storage technology should not be related to the database size. Else, it would make a big data environment very costly and thus forcing analysts to work with less data, which limits analytical capabilities. A low price/performance ratio is required to make analytics on big data affordable.

Data storage technology must be able to exploit inexpensive commodity hardware and disk

- **All data types:** Some big data, such as sensor data, such as sensor data, is highly structured. Weblog files and text messages are clear examples of data that cannot easily be organized in relational columns. Sometimes this type of data is unjustly referred to as *unstructured data*. There is structure in this type of data, it's just not an obvious structure. Nevertheless, to deal with the variety of big data, the data storage technology should support functionality for all types of data: structured, not structured, and everything in between.

- **High schema flexibility:** Data may be stored without a schema. Because analysts may want to study that data differently at different times, data storage technology should be flexible enough to allow applications to read schema-less data with different schema's at different times. Schema flexibility means that there is no need for time- and resource consuming data reorganization work. High schema flexibility demands the support for *schema-on-read*. This concept is explained in Section 6.1.

- **Fast loading:** To support the velocity level that big data systems require, data storage technology must be able to load thousands and thousands of records per second or minute. In addition, this massive ingestion workload should not cause any form of delay or disruption on reporting and analytics,

- **Enterprise-grade:** More and more organizations rely on big data systems. They have become crucial to core business processes. Therefore, data storage technology must offer robustness and stability, 24x7 availability, high-end concurrency (large numbers of concurrent users), data consistency (users should never see two inconsistent versions of the data at the same time), data security and authorization, and stable performance (one application should not be able to consume all the resources and slow down processing for all others).

3 Analytics with Classic SQL Database Servers

Production systems commonly use classic SQL database servers to store data. Therefore, it makes sense that organizations first evaluate these database servers for storing big data. Practice has shown, however, that some of the requirements listed in the previous section are hard to meet by them. With respect to the requirements for high performance, data storage and processing scalability, and enterprise-grade, they have proven themselves for years, even in the heaviest data- and query-intensive environments, and even when the databases grew to many terabytes large.

For the other requirements, it's a different story:

- **High data storage and processing scalability:** The *scale-out* level of the architectures of most classic SQL database servers is limited. They have been designed for *scale-up*. They do a reasonable job with scale-out, but because of their architectures leaning towards centralization, there is a limit to the scalability level. They can't really distribute their processing over hundreds of nodes, which may be needed for big data systems.

- **All data types:** Originally, classic SQL database servers have been designed to store and manipulate structured data. They have been extended to handle text, XML-structured data, and some other non-standard structured data types. The operations applied to this type is usually complex. An example of such a complex operation is sentiment analysis on text. Because it's a resource intensive operation, it's important that it's executed in parallel on as many nodes as possible. Distributing complex operations over many nodes is difficult for most classic SQL database servers.

- **Low price/performance ratio:** As indicated, the license fee for several products is related to the database size, which can lead to expensive big data environments. In addition, most products can only offer high performance when installed on high-end SMP machines using expensive SAN or NAS data storage hardware.

- **High schema flexibility:** Handling unstructured and semi-structured data is not always easy, because in SQL systems data is organized in tables and columns. Afterwards, columns can be added or dropped, but storing and especially accessing data without a structure is difficult—no real support for schema-on-read. Usually, SQL database servers support *schema-on-write* and not schema-on-read.

To summarize, currently implementing big data systems on classic SQL database servers with non-decentralized architectures may be an expensive choice and can lead to scalability problems when unstructured data has to be processed. For ultimate scale-out, data storage technology with a highly non-centralized architecture is required.

For ultimate scale-out, data storage technology with a highly non-centralized architecture is required.

4 Hadoop as Platform for Big Data Analytics

The Coming of Hadoop – Because classic SQL database servers do not always meet the requirements for big data systems, the hunt was on for other data storage products. In fact, long before traditional organizations experienced the need for big data systems, companies such as Google, Yahoo!, Facebook, and Twitter, were already struggling with scalability and performance demands. They were the first to recognize that classic SQL products wouldn't suffice. Therefore, some decided to develop their own data storage solutions. Most of these home-made projects inspired several open source products. For example, Google's Big Table was the inspiration for the development of many others, including Apache HBase, Hypertable, and Cassandra.

One of the most popular and successful new data storage products is without any doubt *Apache Hadoop*. Hadoop has been designed to store, process, and analyze large amounts of data from terabytes to petabytes and beyond. It was also designed to process all this data in parallel on a hardware platform consisting of inexpensive commodity computers (no need for expensive high-end machines and storage technology). Also, because moving big data across networks is hard, Hadoop was designed to move the processing to the data and not vice versa.

For example, on a traditional platform, a full scan of a data file containing one hundred millions records takes a long time. With Hadoop this file can be distributed over hundreds of nodes and disks resulting in a parallelized execution of the scan greatly improving its performance, even if the file contains billions of rows.

The Modules of Hadoop – Hadoop can be regarded as a stack of software modules from which the developers can pick and choose. Figure 1 illustrates the Hadoop modules on which this whitepaper focuses.

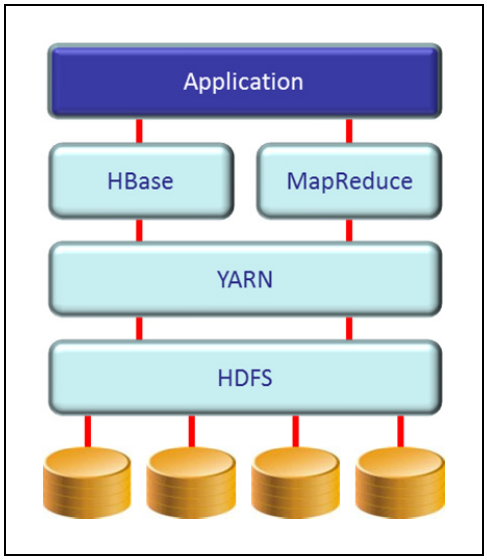

Figure 1 *Hadoop consists of a number of modules including HDFS, YARN, MapReduce, and HBase (the light blue boxes).*

We briefly introduce the core modules here. For more extensive descriptions, we refer to Tom White's book[1] on Hadoop.

- **HDFS:** The foundation of Hadoop is formed by the *Hadoop Distributed File System* (HDFS). This module is responsible for storing and retrieving data. It's designed and optimized to deal with large amounts of incoming data per second and for managing enormous amounts of data up to petabytes. The key aspect of HDFS is that it can distribute data over a large number of disks and has been designed to exploit an MPP (Massively Parallel Processing) architecture. HDFS supports a well-designed programming interface that can be used by any application. Note that Apache HDFS is an *append-file only system*, data stored in Apache HDFS files cannot be changed—a record in a file cannot be replaced.

[1] White, Tom, *Hadoop, The Definitive Guide*, O'Reilly Media, 2012, third edition.

- **YARN:** *YARN* (Yet Another Resource Negotiator) is one of the newer modules introduced in Hadoop version 2. As the name indicates, it's a *resource manager*. It's responsible for processing all requests to HDFS correctly and for distributing resource usage correctly. And like all other resource managers, it should assure that performance is stable and predictable. The main reason why YARN has been introduced is to let multiple computing frameworks run on the same Hadoop cluster using the same underlying storage. So, for example, a company could process data using MapReduce, a graph engine, or a data streaming engine (see Section 11), all without having to run multiple clusters.

- **Hadoop MapReduce:** *MapReduce* offers a programming interface with which developers can write applications to query the data in HDFS. MapReduce can efficiently distribute query processing over hundreds of nodes. It pushes any form of processing to the data itself, and thus parallelizes the execution and minimizes data transport within the system. MapReduce has a batch-oriented style of query processing. MapReduce is well suited for schema-on-read environments. MapReduce benefits from the concept of data locality supported by HDFS. So, to improve scalability, clusters containing no relevant data, are not used when a MapReduce job executes. However, MapReduce doesn't know where individual records are stored. So, selecting a particular customer record from a file, can still lead to scans. Important to note is that the MapReduce programming interface is very technical and requires a deep understanding of the internal workings. Version 1 of MapReduce operates directly on HDFS and contains its own crude resource manager. Version 2 runs straight on YARN.

- **Hadoop HBase:** The *HBase* module is designed for applications that need random, real-time, read/write access to data. The new version operates on top of YARN to exploit its resource managing capabilities. Where HDFS offers a typical file API (open file, append to file, and close file), HBase has an API consisting of operations such as insert record, get record, and update record. HBase's API is not set-oriented. HBase is usually categorized as a *NoSQL system*.

APIs and Implementations – The Apache Software Foundation has developed implementations of all these modules, and they defined and documented their interfaces. Other vendors have developed alternative implementations of the same modules but with the same interface. Examples of alternative HDFS implementations are Amazon S3, the MapR Data Platform, and CassandraFS; see Figure 2. Vendors have also developed alternative implementations of MapReduce, HBase, and YARN; see also Section 10.

Many alternative implementations exist of the Hadoop modules all supporting the same APIs.

Many reasons exist why vendors develop alternative implementations of these modules. It may be that they have been designed to make them easier to manage and use, to use computing resources differently, or to run specific workloads faster.

Alternative implementations must support the same technical interface guaranteeing that an application developed for one, can run unchanged on another. They must be binary compatible.

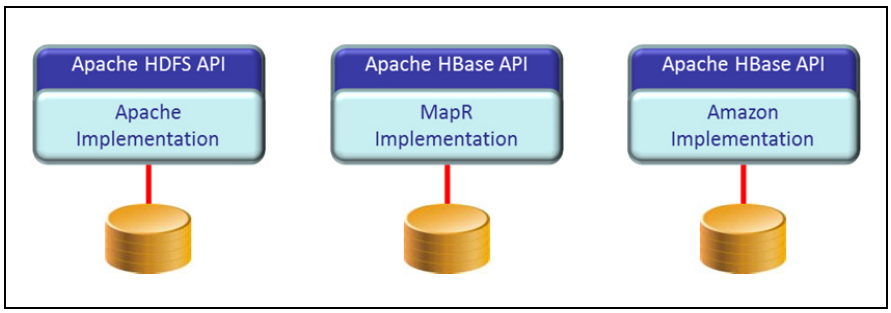

Figure 2 *Alternative implementations of the Apache Hadoop modules exist. To be able to port applications, they all support the same API.*

Does Hadoop Meet the Requirements for Big Data Systems? – Section 3 describes whether SQL database servers meet the requirements for big data systems. Here, we describe how Hadoop satisfies these requirements:

- **High data storage scalability:** HDFS has been designed and optimized to handle extremely large files. For example, there is not a real maximum file size and each file is broken into big blocks (or chunks) of 64MB. In real life projects, Hadoop has repeatedly proven that it's able to store, process, analyze, and manage big data.

- **High data processing scalability:** Hadoop has been designed specifically to operate in highly distributed environments in which it can exploit large numbers of nodes and drives. There is almost no centralized component that could become a bottleneck and lead to performance degradation. For example, one hundred drives working at the same time can read one terabyte of data in two minutes. In addition, MapReduce processing is moved to the nodes where the data is located.

- **High performance:** Together with HDFS, MapReduce offers high performance reporting and analytics. One of the features of HDFS is data replication which makes concurrent access to the same data (on different nodes) possible.

- **Low price/performance ratio:** Hadoop has been designed to exploit low-cost commodity hardware and the license fees of commercial Hadoop vendors are not based on the amount of data stored.

- **All data types:** HDFS is a file system, so it has no knowledge of what is being stored in the files, nor does it require that knowledge. Whether weblogs, emails, or records with sensor data are stored, for HDFS it's all bytes. In addition, functions can be developed in MapReduce that have the same complexity found in SQL statements and beyond. MapReduce is not limited to playing with structured data. MapReduce allows applications to access any form of data. For example complex functions can be developed to analyze text or complex weblog records. If programmed correctly, MapReduce is able to process these complex functions completely in parallel, thus distributing this complex and I/O and resource intensive processing over a large set of nodes.

- **High schema flexibility:** Using HDFS, data can be stored in its original, raw form and MapReduce can be used to assign any structure to the data that the applications wants. Schema-on-read is fully supported.

- **Fast loading:** HDFS has been designed to load massive amounts of data. For example, benchmarks with HDFS implementation of MapR show that it's capable of loading one gigabyte of data per second.

- **Enterprise-grade:** HDFS supports built-in data replication capabilities for fault tolerance and high levels of data availability. However, there are various features, which are common in classic SQL database servers, that are missing in HDFS. HDFS is robust and stable, and does offer 24x7 data availability due to replication across clusters. Apache HDFS itself does not offer read consistency, other HDFS-compatible platforms may. With respect to high-end concurrency, that's the responsibility of the application running on HDFS. YARN offers rudimentary resource management capabilities to guarantee a stable performance.

Summary Hadoop – Hadoop's high data storage and processing scalability, high availability, high performance, and support for flexible data structures, make it an attractive platform for supporting big data systems. In addition, because it runs on relatively inexpensive hardware, it offers an attractive price/performance ratio.

5 The Need for SQL-on-Hadoop

The Hadoop APIs are complex and require expertise in Java programming.

The Drawbacks of Low-level Interfaces – HDFS, MapReduce, and HBase offer APIs for accessing data. Both allow applications to be developed for the simplest queries to the most complex forms of analytics. Unfortunately, all three APIs are quite complex. They require expertise in Java programming (or a similar language) and in-depth knowledge of how to parallelize query processing efficiently. Here are the drawbacks of these interfaces:

- **Small target audience:** These interfaces make Hadoop unsuitable for most of the business analysts, data scientists, and other non-technical users, who typically don't have such skills. The consequence is that the potential analytical power of MapReduce is limited to a happy few.

- **Low productivity:** Due to their technical interfaces, the productivity of developing in Hadoop APIs is not high (compared to, for example, developing in SQL) and analysis can be time-consuming as well.

- **Limited tool support:** Many tools for reporting and analytics don't support the MapReduce and HBase interfaces and can therefore not be used for developing reports on big data. Most of them only support SQL. If such tools must be used, the only option is to copy all the data to a SQL database server. This is a costly and time-consuming exercise.

The Need for SQL-on-Hadoop – What is needed is a programming interface that retains HDFS's performance and scalability, offers high productivity and maintainability, is known to non-technical users, and can be used by many reporting and analytical tools. The obvious choice is evidently SQL.

The first *SQL-on-Hadoop engine* introduced a few years ago is called *Apache Hive*. Hive offers a SQL-like interface for querying data, manipulating data, and for creating tables. It supports a dialect of SQL called *HiveQL*. HiveQL offers the traditional features of SQL, including the windowing functions. To parallelize

query processing, Hive translates SQL statements to MapReduce jobs; see Figure 3. If the data is stored using HBase, the SQL statements are translated to the HBase API. Note that the SQL-on-Hadoop engine Hive does not replace MapReduce and/or HBase, but augments them. Applications can still select their preferred API.

Apache Hive is the first SQL-on-Hadoop engine allowing big data to be queries using SQL.

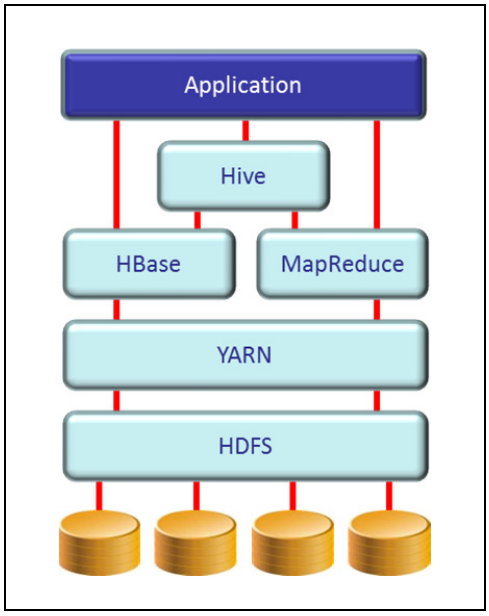

Figure 3 *Hive offers a SQL interface to data stored in HDFS and works on top of MapReduce and HBase.*

In 2013, the market started to understand the importance and business value of a SQL interface. The result was that many other SQL-on-Hadoop engines were announced and introduced, such as Drill, Facebook Presto, HAWQ, Impala, and Shark. Furthermore, the Stinger project was started to improve Hive and lead to, among other things, the development of YARN. Even vendors that were not always associated with open source software released solutions, such as IBM BigSQL, Quest Toad for Cloud Databases, and Salesforce.com Phoenix.

To summarize, the big advantages of having a SQL interface on Hadoop are as follows:

- **Large target audience:** Many business analysts and data scientists are familiar with SQL and can exploit all the big data. This must increase the ROI of a big data investment.

- **High productivity:** Writing queries in SQL requires a lot less coding than when the same logic is written in MapReduce. This improves the time-to-market for reports and improves maintainability.

- **Openness to many tools:** Almost every reporting and analytical tool is capable of accessing data when it's accessible via SQL. Due to these SQL-on-Hadoop engines, these hundreds of tools can now be used to access big data stored in Hadoop. The same applies for all the data integration tools, such as ETL tools and data virtualization products, that can use the SQL interface to extract data from and pump data in HDFS.

Not All SQL-on-Hadoop Engines Are Created Equal – On the outside, most of the SQL-on-Hadoop engines look alike. They all support some SQL-dialect that can be invoked through ODBC or JDBC. Internally they are different. The differences stem from the purpose for which they have been designed. Here are some potential use cases for which they may have been designed:

- batch-oriented query environment (data mining)
- interactive query environment (OLAP, self-service BI, data visualization)
- point-queries (retrieving individual objects)
- investigative analytics (data science)
- operational intelligence (real-time analytics)
- transactional (production systems)

Organizations must know which types of usage they need, because most of the SQL-on-Hadoop engines are not generic SQL implementations. This is not very different from the market of SQL database servers itself. For example, there are products optimized for a transactional workload while others exceed in complex forms of analytics.

Multiple SQL-on-Hadoop Engines Accessing the Same Data – Many applications won't be able to tell the difference between a SQL-on-Hadoop engine and a SQL database server. Internally, there is one big difference: with SQL-on-Hadoop engines there is *independence* between, on one hand, the SQL-on-Hadoop query engine and, on the other hand, the file system and its files; see Figure 4.

Data can be inserted with one SQL-on-Hadoop engine and accessed using another.

In SQL-on-Hadoop engines query engine and the file system with its files are interchangeable. The consequence is that data can be inserted in a particular HDFS file using, for example, Impala, and afterwards accessed with Drill; see Figure 5. Or, a SQL query engine can seamlessly switch from the Apache HDFS file system to the MapR Data Platform or to the Amazon S3 file system depending on the file system requirements of the applications.

One big advantage of this independence is that different query engines, each with its own strengths and weaknesses, can be deployed on the same data. For example, for one group of users a SQL engine can be selected that is designed to support high-end, complex analytical queries, and for the other group an engine that's optimized for more simple interactive reporting. This is comparable to having one flat screen that can be used as TV, computer screen, and as projector screen, instead of having three separate screens. The second advantage is that there is no need to copy and duplicate the data, which, in a big data environment can be very costly.

This independence does not exist for classic SQL database servers. They come with their own file systems and file formats. This means that a database file developed with, for example, Oracle, can't be accessed afterwards using DB2. To be able to do that, the data must be exported from the Oracle database and imported in the DB2 database. In these systems, the SQL query engine plus the file system and its files form one indivisible unit. In a way, it's a proprietary stack of software for data storage, data manipulation, and data access.

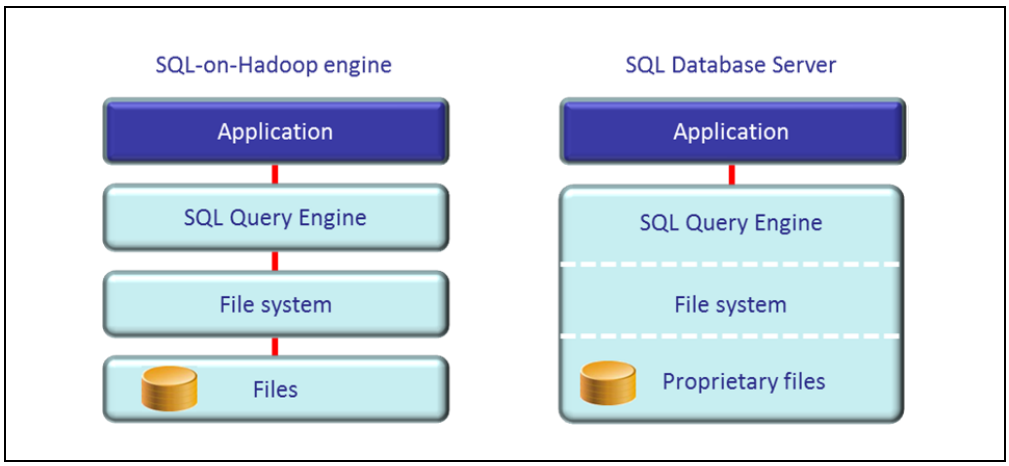

Figure 4 *In a SQL-on-Hadoop engine the three layers are independent. This is not the case for classic SQL database servers, here the layers are indivisible.*

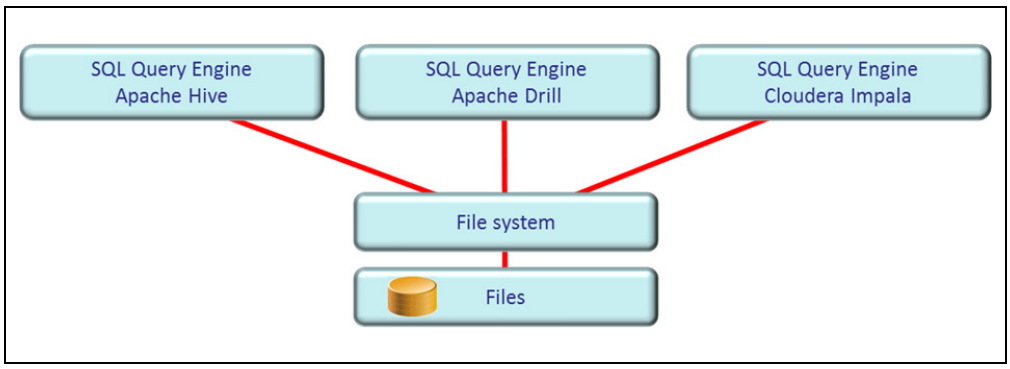

Figure 5 *Different SQL-on-Hadoop engines can access the same file system and thus the same big data.*

SQL-on-Hadoop and the Metadata Store – Another difference between SQL-on-Hadoop engines and SQL database servers relates to the metadata. Most SQL-on-Hadoop engines document the structures of the tables in a system called the *Metadata store*, which is also a key module of the Hadoop stack. By accessing this Metadata store, a SQL-on-Hadoop engine knows how to read the data from file. The metadata store can be queried using an API called *HCatalog*. In a way, this metadata store is comparable to the catalog of classic SQL database servers. The big difference is that many SQL-on-Hadoop engines can use and share one and the same Metadata store, whereas all the classic SQL database servers use their own metadata store (the catalog).

Implementing SQL-on-Hadoop Engines – Developers of SQL-on-Hadoop engines have to select an interface to store and access data in HDFS files. There are several options: on top of HDFS, MapReduce, HBase, or some other interface. When running on MapReduce, the SQL-on-Hadoop engine translates SQL statements to MapReduce jobs; when running on HBase, it translates SQL to the typical HBase commands, such as get record; and, when using HDFS, it uses the low-level, file-oriented API of HDFS. Which one is used does have an impact on the performance of the SQL-on-Hadoop engine. For example, HBase works well for so-called point queries with which just one or a few records are selected, and MapReduce works well if complex analytical functions are processed requiring the access of many records.

6 Technological Challenges of a SQL-on-Hadoop Engine

If only structured data is stored and manipulated, if the set of columns in each table is static, and if the SQL queries are straightforward, developing a SQL-on-Hadoop engine is not that difficult for a vendor. Unfortunately, it's not that simple. First of all, not all big data is structured and static, and secondly, analytical queries can be far from straightforward. These are the main key reasons why development of a SQL-on-Hadoop engine raises a number of technological challenges. These technological challenges can be classified in two groups: *Non-SQL-to-SQL transformational challenges* and *architectural challenges*. They're described in the following two subsections.

6.1 Non-SQL-to-SQL Transformational Challenges

How is Data Stored? – When data is stored in HDFS there are no limitations on how data is stored. In fact, this applies to most file systems—no structure is imposed on the storage format of data. The applications themselves assign a format to the data. This format-free concept is one of the attractive aspects of Hadoop.

For HDFS multiple predefined file formats exist, such as Avro, ORCFile, Parquet, RCFile, Text, and SequenceFile. When these are used, many applications understand how to manipulate the data. Each of these file formats is designed for specific form of usage. For example, Parquet is a column-oriented binary file format intended to be highly efficient for large-scale queries, whereas the Text file format is ideal for archiving data.

SQL-on-Hadoop and File Formats – When a SQL-on-Hadoop engine operates directly on HDFS, it determines how data is organized in the files. It can use one of the aforementioned file formats or use a new one. A `CREATE TABLE` statement usually contains an indication of the format to be used when storing the data in HDFS. When `INSERT` statements are used, the SQL-on-Hadoop engine stores the data in that file using the format belonging to that table. SQL-on-Hadoop engines, like all applications using HDFS, are free in selecting a file format. Nevertheless, regardless of the format used, the data is presented to the applications as tables consisting of records and columns—the file format used is hidden.

HDFS files may contain nested data, variable data, schema-less data, or self-describing data.

It's a different story when data has been inserted in HDFS without the use of a SQL-on-Hadoop engine, but through MapReduce or HBase, or when applications write directly to HDFS. For example, home-made applications using MapReduce may store the data with a hierarchical structure or may store data in a non-structured way, and others may have opted for storing the data as XML documents. If such files are retrieved by a SQL-on-Hadoop engine directly (without the use of another application), it must understand the format, and more importantly, it must be able to transform the storage format of the data to tables, columns, and records. Or in other words, non-SQL concepts must be transformed to SQL concepts.

The rest of this section describes how the following non-SQL concepts, which can be found in HDFS files, can be transformed to SQL concepts:

- Nested data
- Variable data
- Schema-less data
- Self-describing data

Note that in the explanations the SQL terms table, record, and column are used to describe concepts of HDFS. The reason that this classic terminology is used is to improve readability for readers with no HDFS background.

Nested Data – Data may be stored in HDFS with a *hierarchical structure,* using, for example, XML, JSON, or BSON. The effect is that a column of a record does not contain one atomic value, such as a number, string or date, but a set of values, or even an entire table. This explains the term *nested table*. In relational terminology a nested table does not conform to the *first normal form*. Such a table can be depicted as follows:

CUSTOMER_ID	LAST_NAME	FIRST_NAME	CUSTOMER_ORDERS
75295	Sylvian	David	CUSTOMER_ORDER_ID / ORDER_TIMESTAMP 203699 / 2008-01-16 306892 / 2008-07-21 477047 / 2008-12-09
103819	Scaggs	Boz	CUSTOMER_ORDER_ID / ORDER_TIMESTAMP 70675 / 2008-10-19 530223 / 2008-12-01
132171	Rundgren	Todd	CUSTOMER_ORDER_ID / ORDER_TIMESTAMP 210220 / 2008-04-21 485584 / 2008-10-14 718579 / 2008-11-23 741912 / 2008-12-24

Generally, there are three solutions for a SQL-to-Hadoop engine to process nested data. In the first solution, the nested data is transformed into a complex value with some internal structure, for example, as in the next table:

CUSTOMER_ID	LAST_NAME	FIRST_NAME	CUSTOMER_ORDERS
75295	Sylvian	David	{203699,2008-01-16},{306892,2008–07-21}, {477047,2008-12-09}
103819	Scaggs	Boz	{70675,2008-10-19},{530223,2008–12-01}
132171	Rundgren	Todd	{210220,2008-04-21},{485584,2008–10-14}, {718579,2008-12-24}

This means that the SQL-on-Hadoop engine leaves it to the application that receives this data to understand and unravel it. If the receiving application is written in Java that should not be a problem, but

if it's a reporting tool, there will be problems, because most of them don't support the functionality to process such complex values.

In the second solution, the SQL-to-Hadoop engine "flattens" the nested data somehow. For example, the first record in the table above becomes three separate records after flattening:

CUSTOMER_ID	CUSTOMER_ORDER_ID	ORDER_TIMESTAMP	LAST_NAME	FIRST_NAME
75295	203699	2008-01-16	Sylvian	David
75295	306892	2008-07-21	Sylvian	David
75295	477047	2008-12-09	Sylvian	David

With the third solution is that the SQL language itself is extended with nesting. In other words, the SQL-to-Hadoop engine understands and manipulates nested tables and can present them as nested tables to the applications. It involves enriching the SQL language with concepts to process and flatten these hierarchical structures. In such SQL-on-Hadoop engines these hierarchical structure are first-class citizens. Extended SQL interfaces are sometimes referred to as *SQL+* or *extended SQL*.

Note that if this option is selected, many reporting and analytical tools won't be able to access the hierarchical data, because they do not support extended SQL.

Variable Data – In SQL, all records in a table have the same set of columns and thus the same number of values. Tools accessing SQL expect that when a set of records is retrieved each returned record has the same set of columns. To summarize, SQL has been designed to work with *static data*.

HDFS files may contain *variable data* (sometimes referred to as *sparse data*). There are two forms of variable data with which a SQL-to-Hadoop engine may have to deal. First, HDFS allows records with an extra column to be inserted in an existing table, a column that hasn't been defined when creating the table. For example, HBase supports all forms of variable data. Here is an example of what such a table (with five records) may look like:

CUSTOMER_ID	CUSTOMER_ORDER_ID	ORDER_TIMESTAMP
75295	203699	2008-01-16
75295	306892	2008-07-21
75295	477047	2008-12-09

CUSTOMER_ID	CUSTOMER_ORDER_ID	ORDER_TIMESTAMP	ORDER_PROCESSED
463281	203643	2008-01-16	2008-01-20

CUSTOMER_ID	CUSTOMER_ORDER_ID	ORDER_TIMESTAMP	ORDER_CANCELLED
463246	285825	2008-01-19	2008-10-20

Here, the first three records have a value for each defined column. Then, a fourth record is inserted, which has a new column called `ORDER_PROCESSED`, and the fifth record has added an extra column as well called `ORDER_CANCELLED`.

The second form of variable data is the *repeating group,* which most SQL implementations don't support. In a column that holds a repeating group each record has a set of values. Below, a table is shown in which the column `TELEPHONE_NUMBERS` is a repeating group; each customer has a variable set of telephone number values.

CUSTOMER_ID	CUSTOMER_NAME	TELEPHONE_NUMBERS
463246	O'Keefe	{5157818, 2362436}
463249	Zappa	{1234567, 3262836, 4374777}
463350	Donahue	{3854757}

Both forms of variable data are not straightforward to transform. In the case of a repeating group, what should the result look like if all the records and all the columns are retrieved? One solution is to determine the maximum number of values in the repeating group first. In this example that's three. Then, the result of the query includes three telephone number columns. For the record with the maximum number of values, all these columns are filled with a value, and for the others the right-hand columns are filled with NULL values. Result:

CUSTOMER_ID	CUSTOMER_NAME	TELEPHONE_1	TELEPHONE_2	TELEPHONE_3
463246	O'Keefe	5157818	2362436	?
463249	Zappa	1234567	3262836	4374777
463350	Donahue	3854757	?	?

Another solution is to return the data "vertically" like this:

CUSTOMER_ID	CUSTOMER_NAME	TELEPHONE_NUMBER
463246	O'Keefe	5157818
463246	O'Keefe	2362436
463249	Zappa	1234567
463249	Zappa	3262836
463249	Zappa	4374777
463350	Donahue	3854757

A third solution is to return a three-part result. The first part consists of all the records with one telephone number and where each record consists of three columns (CUSTOMER_ID, CUSTOMER_NAME, TELEPHONE_1). This is followed by the second part that contains all the records with two telephone numbers and where each record consists of four columns (CUSTOMER_ID, CUSTOMER_NAME, TELEPHONE_1, TELEPHONE_2). And this is followed by the third part that contains all the records with three telephone numbers and five columns (CUSTOMER_ID, CUSTOMER_NAME, TELEPHONE_1, TELEPHONE_2, TELEPHONE_3). This concept is called a *multi-set result* in ODBC and JDBC.

What the best solution is, probably depends on the application needs. But whatever the result is, a SQL-on-Hadoop engine must be able to process variable data.

Schema-less Data – With nested and variable data, data does have a schema. HDFS may also contain *schema-less data*. With schema-less data, the schema of the data is not known to the data storage technology. For example, a popular form of schema-less big data is textual data.

Processing schema-less data is highly unusual in a SQL environment. When data is written to a database using SQL, a schema is always assigned to the data. This phenomenon is called *schema-on-write*. When schema-less data is stored, a schema must be assigned to the data when it's read. This is called *schema-on-read* and is the complete opposite of schema-on-write.

Here is an example of a record coming from a large weblog that contains schema-less data in the column WEBLOG:

ID	WEBLOG
30130	datestamp ip request 6/1/2012 11:10:19 AM 107.1.187.170 GET /x.php?u=http://studio-5.financialcontent.com/synacor?Page=QUOTE&Ticker=DDD HTTP/1.1 6/1/2012 5:53:49 AM 107.1.2.180 GET /tv/3/player/vendor/Chef%20Tips/player/fiveminute/content/steak/asset/gnrc_15879500 HTTP/1.1 6/1/2012 8:55:54 AM 107.34.51.63 GET /tv/3/search/content/The%20Andy%20Griffith%20Show/s/The%20Andy%20Griffith%20Show HTTP/1.1 6/1/2012 3:12:43 PM 107.5.115.117 GET /tv/3/search/content/Kathie%20Lee%20Gifford's%20epic%20'Today'%20gaffe/s/Kathie%20Lee%20Gifford's%20epic%20'Today'%20gaffe HTTP/1.1 6/1/2012 4:48:35 PM 108.225.132.245 GET /tv/3/search/content/Deadliest%20Catch/s/Deadliest%20Catch HTTP/1.1 6/1/2012 10:25:12 AM 108.246.20.125 GET /x.php?u=http://studio-5.financialcontent.com/synacor?Page=QUOTE&Ticker=DJ:DJI HTTP/1.1 6/1/2012 1:58:14 AM 108.246.25.117 GET /tv/3/player/vendor/Chef%20Tips/player/fiveminute/content/steak/asset/gnrc_15879500 HTTP/1.1

Schema-less data may be the hardest to transform to a more traditional relational form. Before the above value can be processed, it must be organized in columns, because a SQL query result requires a schema/structure. A SQL-on-Hadoop engine must offer features to assign a schema when the data is retrieved.

Self-Describing Data – With self-describing data each value in a column of a table can have a different structure. However, the structure (metadata) of each value is stored together with the data itself. In other words, data and metadata are stored together. This means that the data storage system understands the structure of the data. Here is an example of a table with three records in which the column called `VALUE` contains self-describing data. JSON is used to describe the structure of each value.

ID	VALUE

75295	{ "employee" : {
	"number" : "6",
	"name" : "Manzarek",
	"initials": "R",
	"street ": "Haseltine Lane"}
	}

103819	{ "employee" : {
	"number" : "7",
	"name" : "Metheny",
	"initials": "P",
	"street" : "Brownstreet"}
	}

132171	{ "employee" : {
	"number" : "15",
	"name" : "Metheny",
	"initials": "M"}
	}

When each value in the column has the same structure, transforming them all to one and the same relational structure is probably a matter of combining the logic used for transforming nested and variable data. For example, transforming the above table returns the following result:

ID	EMPLOYEE_NUMBER	EMPLOYEE_NAME	EMPLOYEE_INITIALS	EMPLOYEE_STREET
75295	6	Manzarek	R	Haseltine Lane
103819	7	Metheny	P	Brownstreet
132171	15	Metheny	M	?

When each value in a column has a different structure, transformation to SQL becomes more difficult. For example, how should a classic SQL-on-Hadoop engine transform the next table to a flat table?

ID	VALUE

75295	{ "employee" : {
	"number": "6",
	"name": { "lastname": "Manzarek",
	"initials": "R" },
	"address": { "street": "Haseltine Lane",
	"houseno": "80",
	"postcode": "1234KK",
	"town": "Stratford" }
	}
	}

103819	{ "employee" : {
	"number": "7",
	"name": { "lastname": "Metheny",
	"initials": "P" },
	"address": { "street": "Brownstreet",
	"houseno": "80",
	"province": "ZH",
	"town": "Boston" }
	}
	}

132171	{ "employee" : {
	"number": "15",
	"name": { "lastname": "Metheny",
	"initials": "M",
	"code": "45" }
	}
	}

Summary – Transforming non-SQL data, such as nested data, variable data, schema-less data, and self-describing data, to "flat" SQL data is a technological challenge for SQL-on-Hadoop engines.

6.2 Architectural Challenges

The second challenge facing SQL-on-Hadoop engines relates to the overall architecture of the engine. This primarily deals with how SQL queries can be executed fast even if the amount of data is massive. This section describes four requirements for the internal architecture of SQL-on-Hadoop engines:

- Concurrent queries/users
- Parallel execution of complex operations
- Running complex analytical functions
- Cost-based optimization

Single-query benchmarks are not very useful.

Managing Concurrent Queries – Many benchmarks are available with which vendors can show how fast their SQL-on-Hadoop engine is. Examples are TestDFSIO, TeraSort, NNBench, and MRBench. Regrettably, most of the available benchmarks show the performance of a single query. Being able to execute one query fast is very useful, but the real challenge is to concurrently run multiple queries fast, especially if these queries have different characteristics. The architecture of a SQL-on-Hadoop engine must be able to manage the processing of concurrent queries and the resources used by these queries.

SQL-on-Hadoop engines must be able to support a mixed query workload.

Practically, what this means is that the product must be able to support a *mixed query workload*. It must be possible to set parameters that determine how the engine should divide its resources over different queries and different types of queries. For example, queries from different applications may require different processing priorities, long-running queries should get less priority than simple queries being processed concurrently, and unplanned and resource-intensive queries may have to be cancelled or temporarily interrupted if they use too many resources. SQL-on-Hadoop engines require smart and advanced workload managers.

Parallel Execution of Complex Operations – To fully exploit the highly parallel hardware platforms on which Hadoop runs, a SQL-on-Hadoop engine must be able to parallelize all (or most) of the query processing. Retrieving all the data from the HDFS file system and then doing all the SQL execution on one node (possibly in memory) is not an optimal use of the Hadoop platform.

Running Complex Analytical Functions – The most important instrument Hadoop offers to speed up queries is parallelization. Simple SQL queries such as "For each month get the total revenues" are easy to parallelize. It's important that a SQL-on-Hadoop engine is also capable of executing complex analytical functions in parallel. For example, functions such as a market basket analysis or a Gaussian discriminative analysis are complex and may require the access of many records from many files and may include several complex calculations. A SQL-on-Hadoop engine must be able to parallelize the processing of these complex functions.

Cost-Based Optimization – Each SQL-on-Hadoop engine contains a *query optimizer*. This module is responsible for coming up with the fastest and most efficient strategy to execute queries. It must translate a SQL query to a *processing strategy* (sometimes called an *access plan*). Optimizers are usually categorized as rule-based or cost-based optimizers. A *rule-based optimizer* only looks at the query itself and the tables structures and uses certain rules to determine the best processing strategy.

Cost-based optimizers also check the query and the table structure, and in addition consult statistical information on the data in the tables. Examples of statistical information are the size of a table in bytes or number of records, the maximum and minimum values in each column, the distribution of values within a column, and the partitioning schema of data. Cost-based optimizers are likely to come up with a more efficient processing strategy although there is no 100% guarantee. For example, when an application asks for all the rows from a table where the value of a column is greater than 100, the optimizer doesn't even need to run the query when it knows that the maximum value in that column is 90. A rule-based optimizer wouldn't know this, and would access all the data.

Especially in big data systems where massive amounts of data are queried, a cost-based optimizer can improve query performance dramatically.

7 Use Cases of SQL-on-Hadoop

The first popular use case of Hadoop was running complex forms of analytics, such as data mining and predictive modeling, on big data in a somewhat batch-oriented and non-interactive style. In such an environment, queries that take twenty minutes to twenty hours are no exceptions. Still, analysts would

not be unhappy, because with other technologies these same queries may take several days. Hadoop fits this workload perfectly.

Today, many more use cases of Hadoop can be identified, such as analyzing schema-less data in weblog records, archiving data for compliancy reasons, studying DNA patterns, and geographical analysis of buyer consumption. And this list keeps getting longer. With all the available SQL-on-Hadoop engines, users can use almost any kind of tool to access and exploit the data stored in Hadoop. Undoubtedly, this will lead to even more use cases of Hadoop.

This section describes some of the more popular use cases of SQL-on-Hadoop to show the versatility of this technology. Note that more can be identified.

7.1 Interactive Reporting and Analysis

Interactive reporting and analytics is the classic use case of data warehouse environments. With interactive reporting, users repeatedly access their data to see what's happening with certain business processes. They see their data organized as cubes, dimensions, and hierarchies. They can filter their data, drill down to a more detailed level or, vice versa, do a roll-up, and they can apply aggregations and statistical functions, such as sum, average, and standard deviation. It's called interactive because the users are continuously sending new queries to the database.

Because most SQL database servers can support an interactive reporting workload, they have been used in almost all current data warehouse systems. However, some SQL-on-Hadoop engines have been designed to support interactive reporting and analysis as well. They can take over that interactive workload.

Some SQL-on-Hadoop engines have been designed to support interactive reporting and analysis.

This use case of SQL-on-Hadoop has two benefits:

- Imagine a Hadoop system containing a massive amount of data. It's probably too expensive to duplicate all that data in the data warehouse environment. By using a SQL-on-Hadoop engine, that data is still available for interactive reporting and analytics.

- Almost all the reporting tools can only process structured relational data, but not nested, variable, schema-less, or self-describing data. With a SQL-on-Hadoop engine all those forms of data do become available for interactive reporting and analytics as well. The only requirement is that the engine is able to transform the data to neat tables and columns.

7.2 Self-Service Business Intelligence

This second use case is very much like the first one. Lately, so-called *self-service Business Intelligence tools* have become popular in data warehouse environments. Examples are Tableau, Qlikview, Spotfire, and CXAIR. Most of these self-service BI tools use in-memory technology to speed up processing. They read data from a data warehouse or data mart into memory of the client machine. The data is organized in

memory using efficient compression and indexing technologies. Accessing this in-memory data is usually very fast.

Because of the in-memory approach, the underlying data storage technology is accessed infrequently, in fact, only when data is needed that hasn't been accessed by the tool. In this use case, as with the previous one, a SQL-on-Hadoop engine allows self-service BI tools to access big data stored in HDFS even when the it's nested, variable, schema-less, or self-describing.

7.3 Batch Reporting

Batch reporting is the most classic use case of reporting in the IT industry. Usually, on predefined days and times reports are created and the results are distributed to the business users. In the old days, these reports would be printed on green computer paper and sent to the users using the internal mail service. Nowadays, users receive them on their own machines and smartphones. The queries executed in batch reporting environments usually consist of simple joins and no complex analytical functions. This use case fits like a glove for SQL-on-Hadoop. Almost all these engines can support this type of workload.

7.4 Point Queries

A very popular type of query is the *point query*. With a point query an individual object is retrieved from a list of objects. For example, one customer record is selected from all the customers, or one order or one credit card payment is selected. Especially production systems execute many point queries. But also websites need to pick individual objects or small sets of objects from a database. In all situations, point queries are very simple and must be executed very fast, especially when customers are waiting.

Being a sequential file system, picking just one object using the HDFS interface requires a full scan of the file. HBase is much better suited for point queries. So, SQL-on-Hadoop engines that support HBase can deliver fast responses on these point queries, even if the number of records is enormous.

7.5 Operational Processing

Operational processing is a use case where *zero-latency data* is presented to users. It usually involves a data source that is continuously updated with new data and concurrently queried. Seconds and even micro-seconds count in operational processing. It could be the difference between success and failure. Operational reporting is the basis for popular trends such as *operational intelligence* or *real-time analytics*.

Most HDFS implementations can handle a massive data load workload. By placing a SQL-on-Hadoop engine on it, users are able to query the new data entered milliseconds ago. So, SQL-on-Hadoop opens the door to operational processing.

7.6 Investigative Analytics

With most forms of reporting and analytics, users know what they're looking for. With *investigative analytics* (or *data discovery*) they don't always know exactly what they are searching for, although they probably have a feeling or an inkling. This is the world of the *data scientist*.

But what is a data scientist and what does he do? For example, in an oil company, the ones responsible for analyzing soil test results to locate new oil fields or for analyzing new techniques to find new oil fields faster, can be classified as data scientists. Another clear example of a data scientist is an actuary working for an insurance company. Actuaries deploy mathematics, statistics, and financial theory to analyze the financial consequences of risk. Professors looking for cures for specific diseases by doing DNA research can also be classified as data scientists.

Examples of some of their queries are:

- What is a possible behavioral pattern of credit card usage that signifies a fraudulent action?
- What are other forms of data that can help us locate deeply buried oil fields more easily?
- How high is the financial risk if a person of 21 years with no job is given a mortgage?

The characteristics of this workload are: very complex queries, access to big data (structured and not structured), and an irregular workload. Most Hadoop implementations feel comfortable with this workload. Adding a SQL-on-Hadoop engine on top makes it easier for the data scientists to analyze the data freely.

7.7 Data Stream Processing

The *data stream processing* use case is almost the opposite of the batch reporting use case. With data stream processing, incoming data is continuously monitored and immediate actions are taken when some event occurs. For example, a server log is monitored and action is taken immediately when a component fails. It could also be the monitoring of a Weblog where real-time relevant data is pushed to an analytical dashboard.

Most SQL-on-Hadoop engines, in fact most SQL products, can't handle data stream processing. However, HDFS itself is more than capable to support this workload.

7.8 Storage of Cold Data Warehouse Data

Data stored in a data warehouse can be classified as cold, warm, or hot. Hot data is used almost every day, and cold data occasionally. Keeping cold data in a data warehouse slows down the majority of the queries. Also, it's expensive, because all the data is stored within an expensive data storage system. If the data warehouse database is a SQL database, it may be useful to store cold data outside that database in Hadoop HDFS. This storage form is less expensive and the data remains accessible. The resulting architecture is displayed in Figure 6.

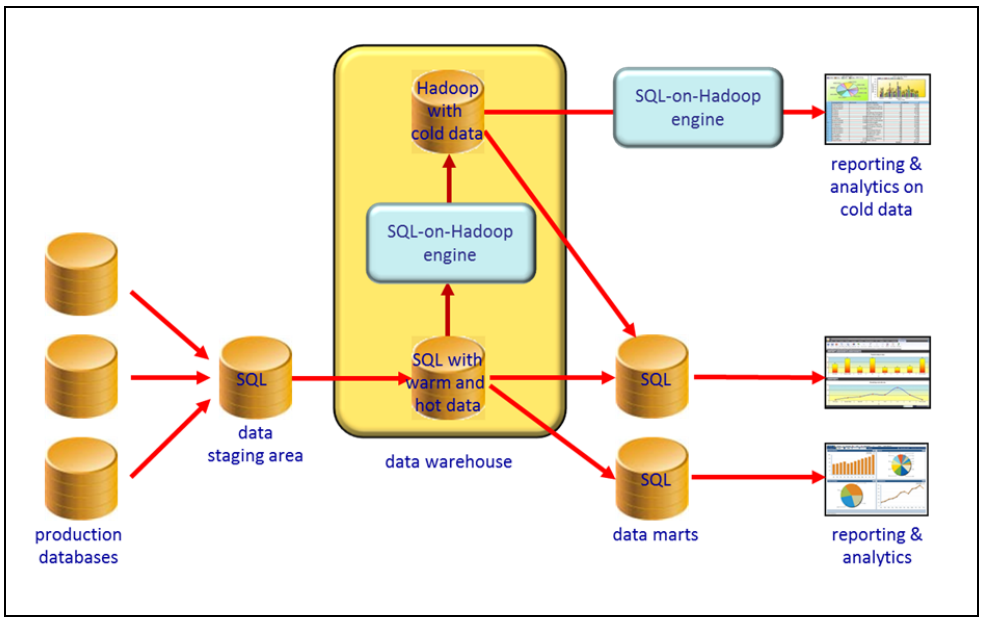

Figure 6 *In this solution cold data is stored using Hadoop and warm and hot data using a SQL database server. Together, the two databases form the data warehouse.*

This solution saves storage costs, it speeds up queries on the hot and warm data in the warehouse (less data), and it can handle larger data volumes by moving cold data to Hadoop.

To pump data from the SQL-part of the data warehouse to the Hadoop-part, a SQL-on-Hadoop engine is very useful. It makes copying of the data straightforward, because it comes down to simply copying the contents of one SQL table to another. And, with a SQL-on-Hadoop engine on the Hadoop files, reports can still get to the cold data easily. So, the cold data remains online available.

7.9 Storage of External Data

Analytics on internal data, such as data from ERP systems, call center log files, weblog files containing Website interactions, voice transcripts from customer calls, and personal spreadsheets can definitely lead to useful business insights. However, by enriching internal data with *external data*, analytical capabilities and the chance on finding valuable business insights increase dramatically. Nowadays, there is a lot of external data available of which social media data is the most well-known. By integrating internal customer data with, for example, Facebook data, a more detailed picture can be developed of what a customer thinks about the products and the company.

But it's not only social media data. Thousands and thousands of *open data sources* have become available for the public. There are open data sources that contain weather data, demographic data, energy consumption data, hospital performance data, public transport data, and the list goes on and on. Almost all these open data sources are available in the cloud through some API.

Because of the sheer size, storing external data in a classic SQL database may be too expensive. Storing it in HDFS makes a lot of sense, especially when the data is not highly structured.

By making external data stored in HDFS available via a SQL-on-Hadoop engine means that almost every user using his favorite reporting or analytical tool can exploit it. The SQL-on-Hadoop engine opens up the

external data to a large audience and thus increases its potential business value. A requirement for supporting this use case is evidently that the SQL-on-Hadoop engine knows how to transform all the data that has no relational structure.

7.10 Fast Staging Area

The workload of a *staging area* in a data warehouse architecture is usually straightforward: in (semi-)real-time new data entered in production systems is copied to a staging area and from there the data is copied onwards to an operational data store or data warehouse. Usually, this second step is done in batch. Eventually, after data has arrived in the data warehouse it can be deleted from the staging area; again, this a batch process. Ordinarily, no reports are executed on a staging area.

The workload of a staging area fits perfectly with Hadoop. HDFS is able to ingest large amounts of data in real-time fast, and, for example, MapReduce is great at extracting data using a batch-oriented approach.

Data extraction from a staging area is usually done using ETL tools. However, not all ETL tools support native MapReduce or one of the other Hadoop interfaces. Because all of them support SQL, they can use a SQL-on-Hadoop engine to extract the data periodically. If data must be heavily transformed before it's copied to the data warehouse, all the required processing can be executed by the SQL-on-Hadoop engine efficiently in parallel.

7.11 ETL (Pre)Processing Platform

The sheer data volume in a particular data warehouse may be too much for a SQL database. It may become too costly and queries may be too slow. In this situation, aggregating the data somewhat before it's stored in the data warehouse database may make sense; see Figure 7. This shrinks the size of the data warehouse database and speeds up query processing. In fact, besides doing data aggregation, other forms of processing may be applied as well.

However, if this solution is selected, another database must be available to hold all the detailed data (which is not the staging area) and some module must become responsible for aggregating and processing the data. Hadoop can be selected as the data store for all the detailed data and a SQL-on-Hadoop engine for access. In this architecture Hadoop contains the large data volumes—this is where data is pumped into first. Then, SQL-on-Hadoop is used to insert all the data efficiently and is used when all the data is aggregated and transformed before it's copied to the data warehouse database. In other words, the SQL-on-Hadoop engine is doing all the (pre)processing of data before it becomes available for reporting and analysis—SQL-on-Hadoop acts as an easy-to-use ETL engine.

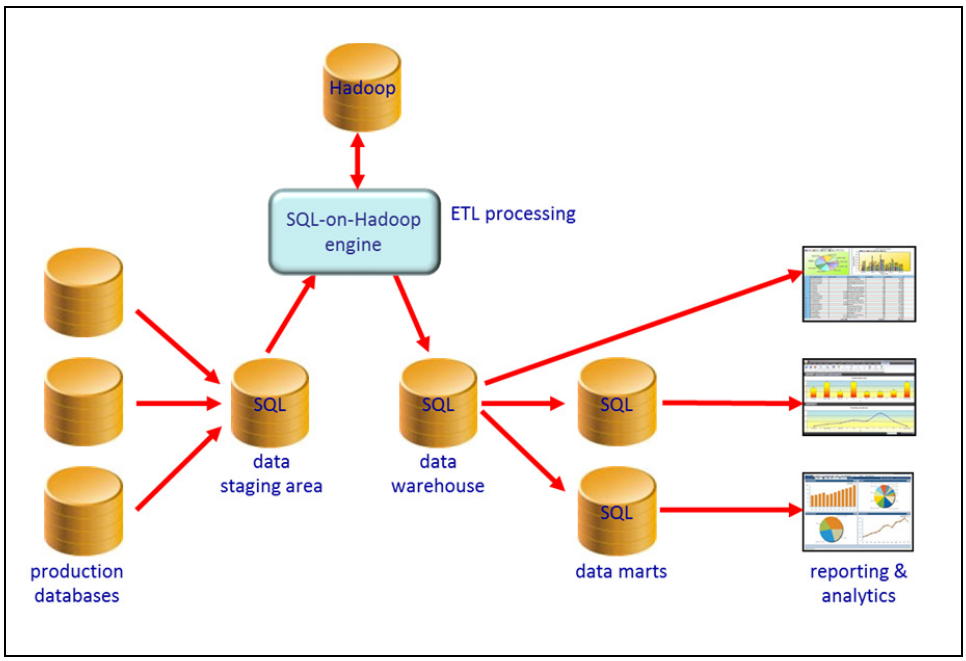

Figure 7 *Hadoop is used for fast ETL processing.*

7.12 New Use Cases and Non-Relational Data

Section 6 describes the non-SQL forms of data found in big data systems. For most of the use cases described in this section, it must be possible to support nested data, variable data, schema-less data, and self-describing data. In other words, it must be possible to do operational processing on structured data, as well as nested, variable, schema-less, and self-describing data; likewise, it must be possible to do interactive reporting on all forms of data.

8 Big Data: From Single Use Case to Multi Use Case

The Single Use Case of Big Data – Many organizations have implemented big data systems for just one particular *use case*. For example, a website owner runs a big data system with Hadoop with the single intention to analyze weblogs and find particular usage paths; an airline may use a big data system to store Twitter messages for monitoring customer comments live; and, a telephone company may deploy Hadoop purely for archiving call detail records.

But organizations are slowly starting to pass this single use case stage. They want to do more with their big data investments. They have discovered that big data can be used for other purposes as well, thus improving the ROI of their big data investment. Especially due to the SQL-on-Hadoop technology, so many more use cases exist, as shown in Section 7. They are evolving from a single-use case to multi-use cases.

The First Solution: Big Data Silos – Most SQL-on-Hadoop engines and data storage platforms have been designed for a subset of the use cases described in the previous section (see also Section 5, *Not All SQL Engines Are Created Equal*). For example, the stack consisting of Hive + MapReduce version 1 (MRv1) + HDFS was designed for non-interactive forms of analytics on structured and unstructured data. Now, that

Hive runs on MapReduce v2 (YARN) and will eventually use Tez as an execution engine instead of MapReduce, it will become more suitable for interactive reporting. However, its strength is still not handling complex forms of analytics on schema-less data or running data stream processing using for example Apache Storm. In other words, the current SQL-on-Hadoop engines to manipulate data in HDFS have been optimized for a few use cases.

So, whether a specific solution is right for a specific use case depends on many factors. The differences between SQL-on-Hadoop engines are bigger than one may think, some are optimized for batch analytics and investigative analytics whereas others are optimized for interactive reporting. In addition, the underlying file system influences for which use case the combination works well. For example, if a SQL-on-Hadoop engine is used, which has been designed for interactive reporting, but the file system underneath has been designed to support small and fast file access in-memory, then using the combination of the two is far from perfect. The file format has an impact as well. HDFS supports file formats in which data is stored in a more record-oriented format, and there are those in which data is stored in columnar fashion. To support a massive insert workload, the former is more suitable, while the second helps to speed up analytical queries.

An analogy is probably cars and tires. It doesn't make sense to place tires without grooves (slicks) on a powerful SUV. Slicks are optimized to drive fast on dry and fast tracks, while SUVs are designed to drive in rough countries. It would be far from a successful combination.

The consequence is that the stack of layers selected by many organizations was composed for a particular use case. The moment they have a completely different use case, suddenly their existing solution is not optimal anymore. It usually forces them to develop a second solution consisting of some other technologies. In the long run, this results in many data storage platforms: each one designed and optimized to support a limited number of use cases. In other words, this approach leads to *big data silos*; see Figure 8. Besides having to develop and manage all these silos, many replication solutions are needed to copy data from one database to another to keep them in synch.

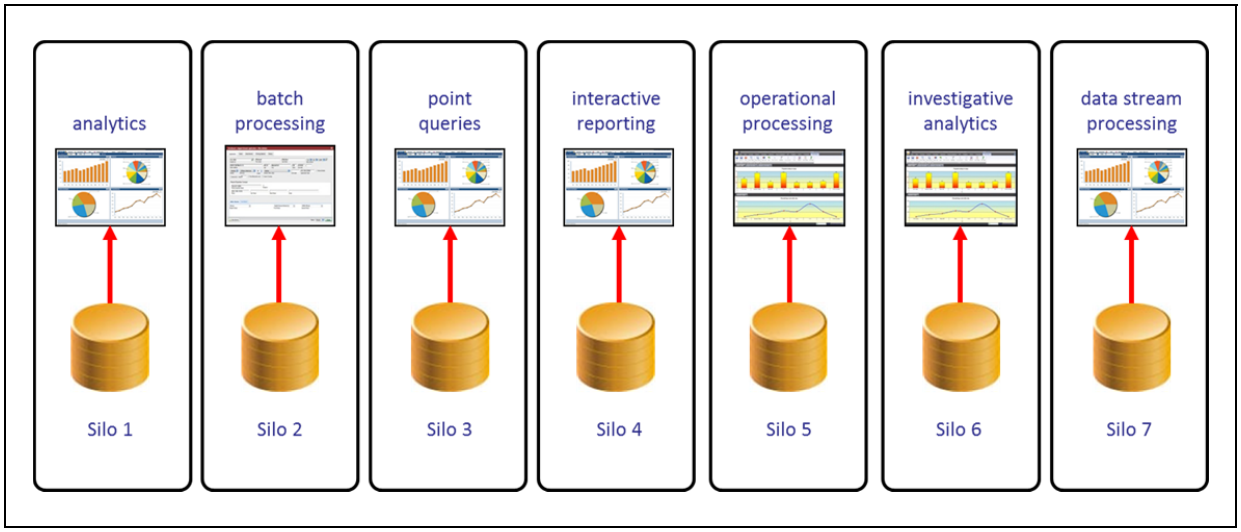

Figure 8 *Big data silos lead to data duplication, high data latency, complex data replication solutions, and data quality problems.*

The disadvantages of having all these big data silos are clear: high costs because of data duplication, high data latency, complex data replication solutions, and data quality problems.

The Integration Labyrinth – Big data silos, where each silo is built on one big data source, can be regarded as temporary solutions. History has shown that eventually the users of these silos want to combine data from multiple data sources. When this happens, applications have to be extended so that they access multiple data sources leading to a dedicated integration solution for each one of them. The result is an *integration labyrinth*; see Figure 9. For an organization to guarantee that all these integration solutions are correct, efficient, and lead to consistent results, is almost impossible.

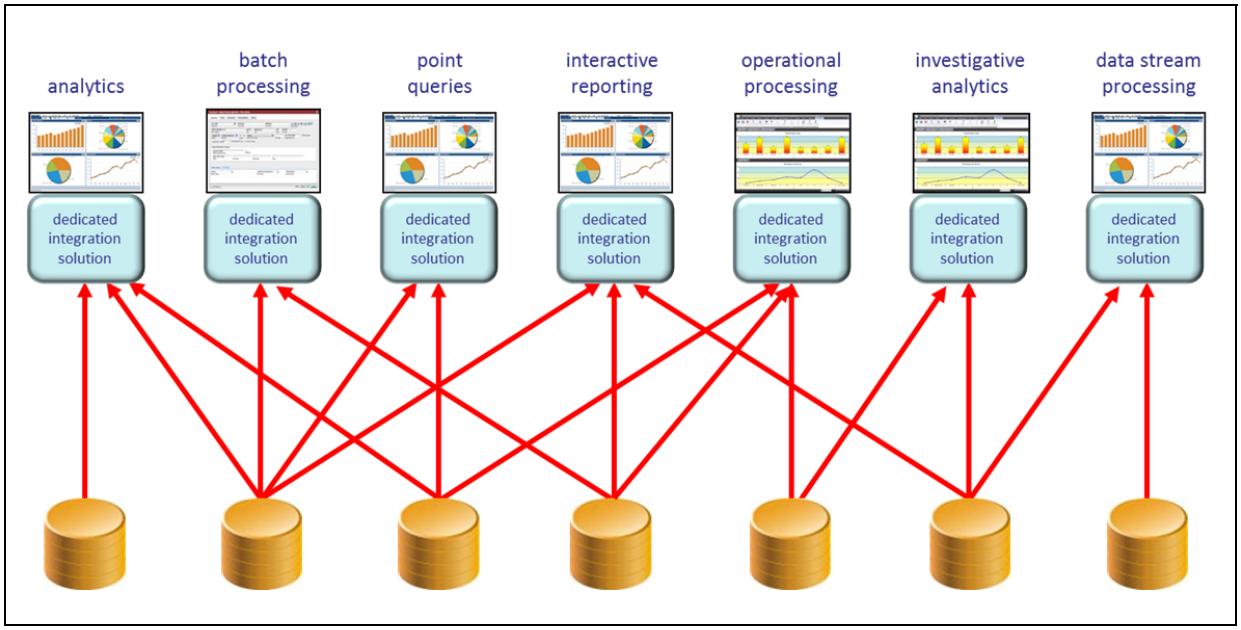

Figure 9 *Big data silos lead to an integration labyrinth.*

The Multi Use Case Solution – Big data silos, where one data source supports one use case, is not what organizations want, nor is any organization interested in an integration labyrinth. Big data systems *must* support multiple use cases; there should be (almost) no need to duplicate data.

9 One Platform to Rule Them All

In the famed book entitled *The Lord of the Rings* written by J.R.R. Tolkien, there was "one ring to rule them all". The word "all" here refers to all the other magic rings with power. The owner of that one ring controlled all the rings of power and would have the power to rule the world and would be invincible.

For building IT systems rings are not very helpful, but what is useful is *one platform to rule them all*. And here "all" refers to all the use cases.

Considering all the new requirements and use cases described in the previous sections, what is needed is one data management platform that supports all the current and future use cases, thus minimizing the need to duplicate all that big data and avoiding the development of big data silos and an integration labyrinth; see Figure 10.

One data management platform is needed that supports all current and future use cases.

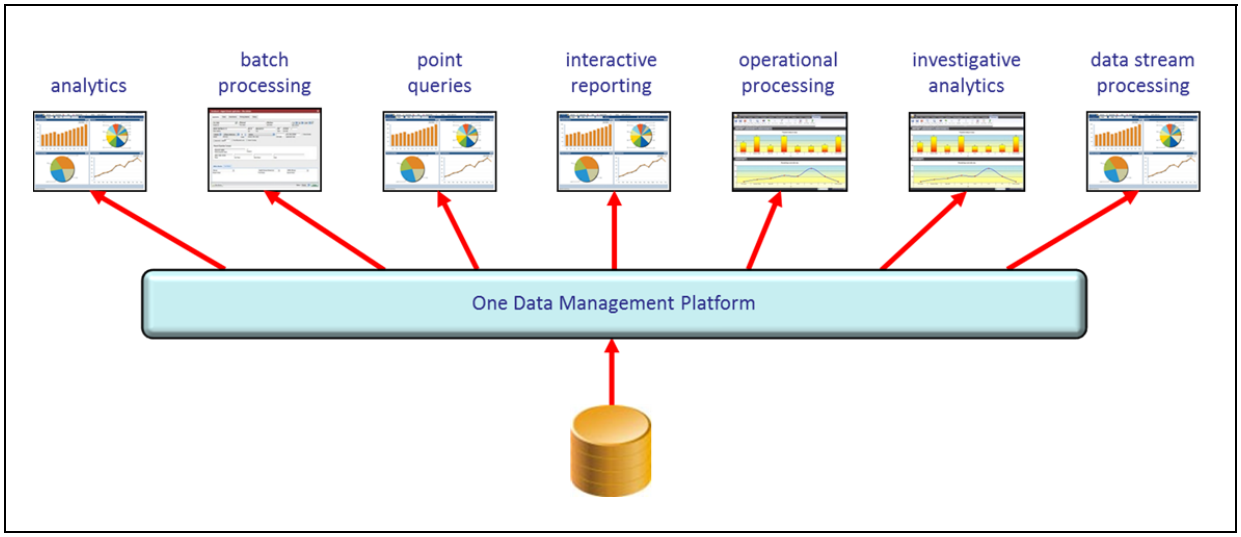

Figure 10 *To avoid an analytical labyrinth, one data management platform is needed that supports all the different forms of reporting, analytics, and operational data processing.*

The advantages of this one-platform architecture are obvious:

- No extra data storage costs due to duplication
- No risk of data inconsistency due to duplication failures
- No license fee costs for data duplication software
- No data latency problems because of duplication time
- More flexibility because no need to replicate data structure changes

Sounds good, but most platforms have been designed and optimized for a limited number of use cases. To return to the analogy with cars again, this is not much different from the car industry. Some cars, such as the Masarati Ghibli, have been designed to drive fast, while others such as the Dodge Grand Caravan have been designed to transport families comfortably, and a Freightliner truck has been manufactured to carry big and heavy loads. There are not that many cars that support most or all the use cases, and the same applies for data management platforms.

A data management platform must support a file system that is functionality rich enough to support a wide range of use cases, and the same should apply for the SQL-on-Hadoop engine on top. In an ideal situation it must be able to support analytics, batch processing, point queries, interactive reporting, operational processing, investigative analytics, and data stream processing. This minimizes the need to develop numerous big data silos.

The next section discusses why the MapR platform is such a platform, and Section 11 describes the SQL-on-Hadoop engine Apache Drill, which runs on MapR or any other Hadoop distribution and supports a wide range of use cases.

10 The MapR M7 Platform

Introduction to MapR – One platform that has been designed to support a wide range of use cases is the *MapR Distribution for Hadoop: M7 Enterprise Database Edition*. The MapR Distribution ships with over twenty Apache Hadoop projects. An application developed for any Apache Hadoop distribution can be ported to (and from) MapR without any changes. This is similar to other Hadoop systems such as Amazon Elastic MapReduce (EMR) which uses S3 for storage instead of HDFS; see also Section 4. Where the MapR Distribution for Hadoop distinguishes itself from other implementations is that it has been developed to be faster, more reliable, and more easily manageable than other distributions.

A key advantage of MapR M7 is the manageability of the platform. First of all, there is no need for database administration. There are no extra servers to administer that handle database operations. All the database operations are processed by the core MapR system. Plus, all high availability and disaster recovery (HA/DR) features apply to both Hadoop files and database tables, therefore, a separate HA/DR strategy is not required to keep database operations online. Secondly, MapR M7 offers zero downtime. The HA/DR features in MapR are built-in for production use. Other distributions require integration with third-party products for true HA/DR. Snapshots are easily taken and managed via the browser-based interface. When a node fails, instant recovery ensures a very short recovery time for database operations.

The M7 Integrated Database – MapR M7 ships both Apache HBase as well as an integrated, optimized database that is compatible with Apache HBase. One advantage it offers is that it leverages the full read and write capabilities of the MapR Data Platform. Although it's compatible with Apache HBase, its implementation is quite different. Because Apache HBase runs on Apache HDFS, the limitations of HDFS are imposed on HBase. For example, as indicated, Apache HDFS is an append-only file system. This means that when HBase needs to execute inserts, deletes, or updates, it must simulate these operations by creating new files. This adds tremendous read and write overhead, and has a serious impact on the performance and manageability of HBase applications.

What's unique in MapR is that it reads and writes directly to disk, it doesn't use HDFS via its own file system. So, the mentioned disadvantages don't apply to the MapR Distribution. Applications can do inserts, updates and deletes, and these are all executed very much like comparable operations in classic SQL database servers: existing records are replaced by new ones, or are completely deleted from the file. This involves a lot less work and, most importantly, it speeds up insert, update, and delete operations. Also, it requires no other software layers; see Figure 11. This is different from the Apache HBase architecture which relies on HDFS, and both need a Java Virtual Machine. This means the M7 database can run much faster with fewer resources. As a result, running a SQL-on-Hadoop engine on this database is very fast.

MapR M7 offers a continuous low latency which is important for operational processing.

A benefit of MapR M7 is its *continuous low latency,* which indicates how fast data can be transported from disk to the CPU to make it ready to be shipped to the application. Especially in an HDFS environment, where all data requests are executed by scanning complete files, it's important that read latency is as low as possible. Benchmarks with the well-known *YCSB benchmark* (Yahoo! Cloud Serving Benchmark) indicate that read latency in MapR is much lower than that of other distributions[2]. In addition, the read latency of MapR is much more consistent, because its optimized architecture eliminates the need for compaction and defragmentation, so there are no periods of intensive housekeeping to slow down the entire system. This low latency is especially relevant when running operational applications on Hadoop.

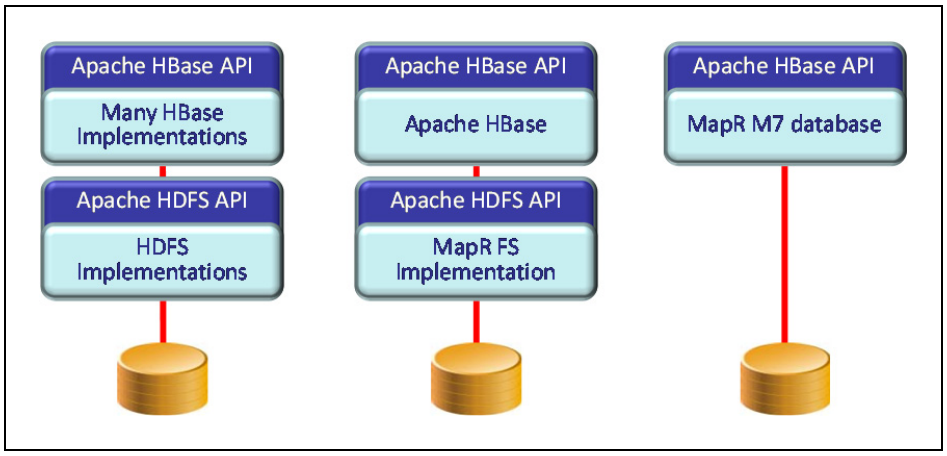

Figure 11 *The MapR M7 database implementation directly writes to and reads from the disks.*

Apache HBase – As a complete distribution for Hadoop, MapR M7 also includes Apache HBase. Applications built on Apache HBase can be run on the M7 integrated database, and vice versa. And if desired, applications can be run on both Apache HBase and the M7 database simultaneously on the same cluster.

"One Platform to Support Them All" – MapR has been designed from scratch to support a wide range of use cases along with supporting industry standard interfaces, such as NFS, HDFS, REST, LDAP, and ODBC. MapR supports most of the open source SQL-on-Hadoop engines (see also Figure 5) and a wide range of other applications. Note that MapR provides tight integration with industry leading non-open-source SQL technologies such as HP Vertica as well. It can operate as the sole data management platform for most forms of reporting and analytics and minimizes the need to duplicate data and replicate integration solutions; see Figure 10.

11 The Apache Drill SQL-on-Hadoop Engine

Hadoop is becoming a general-purpose platform for all big data applications—both analytic and operational. MapR is enabling such a platform with its unique enterprise capabilities. Being able to support multiple SQL-on-Hadoop options is a key element of the MapR platform.

[2] MapR Technologies, *NoSQL Performance Report, MapR M7 – Performance Comparison*, 2013, see http://www.mapr.com/sites/default/files/mapr_m7_performance-benchmark1.pdf

There are several SQL-on-Hadoop engines from various vendors each offering their own strengths, but to support a general purpose platform, you need a general purpose query layer which accounts for the flexible data model, nested data structures commonly found in big data applications. Apache Drill is trying to solve this problem and MapR is spearheading its development in the open source community.

Apache Drill is a real community-driven open source project.

Introduction to Apache Drill – The SQL-on-Hadoop engine called *Apache Drill* is an Apache community-driven open source project to which MapR is a key contributor along with others in the community. It operates on top of any standard implementation of HDFS and HBase, including for example the MapR Data Platform and the MapR M7 database, but also Apache HDFS and Amazon S3; see Figure 12.

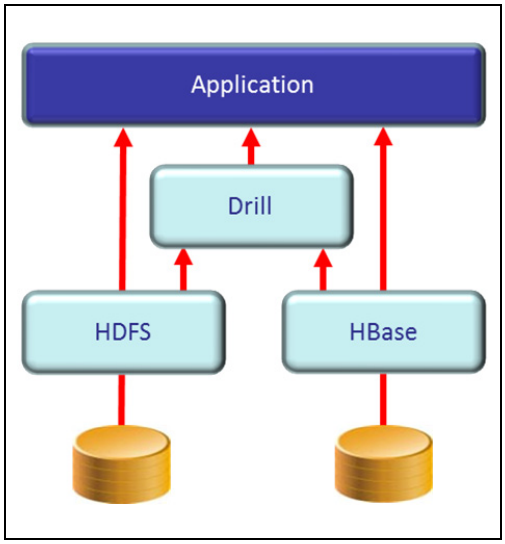

Figure 12 *Drill is a SQL-on-Hadoop engine that supports access to HDFS and HBase.*

The characteristics of Apache Drill are classified here in four categories: use cases, architecture, functionality, and non-SQL-to-SQL transformations.

Use Cases of Apache Drill:

- The Drill SQL query engine has been designed for a wide range of SQL use cases on big data, including interactive query environments (OLAP, self-service BI, data visualization), batch-oriented query environment (data mining), point-queries (retrieving individual objects), and investigative analytics (data science).
- Apache Drill has been designed from the start to support use cases that require processing of nested data, variable data, schema-less data, and self-describing data (see Section 6) without the need to define schemas around them.

Architecture of Apache Drill:

- **Zero-latency queries:** Due to its columnar format to store and process data and that it doesn't require centralized schemas to be pre-built, Apache Drill offers low-latency queries as soon as data arrives. Processing of the queries starts immediately by Drill making it suitable for interactive

processing. This is different from SQL-on-Hadoop engines that run on platforms such as MapReduce v1 (and MapReduce v2) where the processing is scheduled and therefore take longer.

- **Internal architecture:** Drill doesn't use MapReduce, it comes with its own processing architecture. The architecture is based on a set of hierarchically-organized modules called *drillbits*. Drillbits are responsible for executing SQL statements. A drillbit is installed on each node that holds data. A drillbit module is capable of executing SQL queries on the data that it manages. If data is stored across many nodes, all relevant drillbits are involved in the processing of the query thus parallelizing its execution. There is no master-slave architecture in Drill. When applications access Drill, they are "connected" to different drillbits, to avoid one drillbit module becoming responsible for the management of all the queries. Such a drillbit would become a bottleneck when many queries are executed by many applications. Query processing is distributed over as many drillbits as possible, always ensuring data locality.

  The architecture of Apache Drill is based distributes query processing over as many nodes as possible.

- **Extensible architecture:** Drill has been designed to be extensible. New data sources, new file formats, new operators, and also new query languages can be added easily. For instances, users can easily create new user-defined functions or build custom storage plugins for traditional data sources beyond Hadoop.

- **Availability:** Drill follows an optimistic query execution model. If a query fails when it tries to access data that's not available (for whatever technical reason), depending on internal settings, Drill will re-run the query automatically. The failing and re-running of queries is transparent to the users. To them it feels as if the query has executed without any hiccups.

- **Data storage platform independent:** Drill is not dependent on a specific data storage platform. Drill is a query engine capable of executing SQL statements on various data storage platforms, including Apache HDFS, MapR FS, Apache HBase, and the MapR M7 database. In fact, when existing data is copied from HDFS to HBase using SQL statements, an application using Drill can be migrated without any changes to the code.

  Drill is a query engine capable of executing SQL statements on various data storage platforms.

SQL Functionality of Apache Drill:

- **ANSI SQL dialect:** The SQL dialect supported by Drill is a classic dialect that includes all the standard features such as inner joins, left and right outer joins, aggregations (group-by), and statistical functions. Not implemented yet are windowing functions.

- **User-defined Functions:** Drill supports user-defined functions. Such functions can be used for analytical operations, but also for transforming schema-less data in schema-rich data. This makes schema-less data available for more classic reporting and analytical tools. The processing of UDFs can be distributed over many nodes.

- **Metadata store:** Although Drill doesn't require a metadata store to function, in instances where there are already pre-defined schemas, Drill uses the same *metadata store* as other SQL-on-Hadoop engines, such as Hive and Impala. So, table definitions entered by, for example, Hive can be read by Drill, and vice versa. This metadata store is accessible through the *HCatalog interface*.

Non-SQL-to-SQL Transformations by Apache Drill:

- **Nested data:** Drill processes nested data in its native format and offers specialized SQL extensions to work with nested data.

- **Variable data:** Many HBase files contain variable data. It's one of the strengths of HBase. Drill transforms all the variable data to classic SQL data types so that it any reporting or analytical tool can query the data.

- **Schema-less data:** As indicated, with user-defined functions schema-on-read on schema-less data can be implemented. Even when HDFS files with schema-less data have been created outside Drill, they can still be processed.

- **Self-describing data:** Maybe this is the most distinguishing factor of Drill. With Drill, SQL queries can be executed on self-describing data. Usually, when a query is executed on a table, the table schema is retrieved from the metadata store to determine how the data should be retrieved. Because there is no table schema for self-describing data and creating a process around building schemas for such data would be complex and difficult to maintain, the Drill optimizer just starts to read the data, and with each retrieved record a better understanding of the structure unfolds. It's possible that after so many records, Drill restarts the query because it understands the schema of the data better. This feature makes it possible to transform schema-less data to SQL structures and makes that data available for any type of reporting tool.

One of the most distinguishing factors of Drill is that it can execute SQL queries on self-describing data without having to rely on predefined, centralized schema.

12 Closing Remarks

Standards and Independency – Although the Apache Software Foundation is not a standardization committee, the Hadoop stack is slowly turning out to become an integrated set of de-facto standards. For each module, such as HDFS, HBase, and MapReduce, an interface has been defined; see Section 4. These interfaces can be implemented and optimized by vendors, as has been done by, for example, MapR Technologies, Amazon, and GridGain.

Standardization of interfaces has several benefits. First of all, organizations can assemble their own Hadoop stack based on their requirements with respect to, for example, performance, scalability, and use cases. The only requirement is that the vendors implement these interfaces correctly and fully. Secondly, it makes customers independent of a specific module—if a selected module doesn't meet all the requirements, it can be replaced by another one, and thus insuring the rest of the investment.

One Platform – Because the number of use cases for which organizations want to deploy Hadoop is growing, it's important that data management platforms can easily support all these different use cases. This is important if organizations want to avoid big data silos (leading to duplication of big data, expensive replication mechanisms and so on; see Section 8) and if they want to avoid an analytical labyrinth. Due to its unique internal architecture the MapR Distribution is capable of supporting most of them while retaining support for all the Hadoop interfaces. The effect is that there is less need to deploy different distributions for different use cases.

Comparison of Several SQL-on-Hadoop Engines – The previous section describes the Apache Drill SQL-on-Hadoop engine in some detail. This section contains some comparison material of a number of SQL-on-Hadoop engines.

Table 1 contains a high-level overview of several open sources SQL-on-Hadoop engines next to a classic SQL database server. Apache Drill is included, plus three open source SQL-on-Hadoop engines, Cloudera Impala, Apache Hive, and Shark. For completeness sake a column is added that indicates how a typical SQL database server scores. All of these engines are supported on the MapR Distribution for Hadoop.

The products are compared on five criteria: Completeness of their SQL dialect, the complexity of the SQL queries they can process well, the amount of data they can handle, the types of processing they support, and the types of data they can process. The more blue boxes that are drawn, the more complete the support is for a criteria.

	Apache Drill	Cloudera Impala	Apache Hive	Shark	Classic SQL
ANSI SQL complete ANSI SQL medium Minimum SQL					
Complex queries Medium queries Simple queries					
Big data Large data Small data					
Operational processing Interactive processing Batch processing					
Schema-less data Non-relational data Structured data					

Table 1 *A high-level comparison of several SQL-on-Hadoop engines plus a classic SQL database server.*

What can be derived from Table 1 is that there doesn't exist an SQL-on-Hadoop engine that is ideal for every possible use case. Some are perfect for interactive analytics while others work best for batch-oriented analytics. In addition, some are good at processing all the non-structured data, while others are optimized to work with structured relational data.

No SQL-on-Hadoop engine is ideal for every possible use case.

With respect to Apache Drill, its key strength is how it has been developed to work with all forms of data, including nested data, variable data, schema-less data, and self-describing data. This is especially

important for big data environments where IT or DBA's do not want to create or continuously maintain a centralized schema to enable self-service data exploration.

Maturity of Solutions – SQL-on-Hadoop engines differ in their maturity. This is especially relevant with respect to their query optimizers. Efficient query optimizers that are able to come up with the perfect processing strategy for every query are not born in development labs. They need many "hours in the saddle." It's when an optimizer is used over and over again in all kinds of situations, will the developers know how to improve and optimize it. This process make take a few years. Some of the SQL-on-Hadoop engines are still young and still have to proof themselves in large-scale, complex, and multi-user environments. It's important that organizations are aware of this and that they keep an alternative route open. Don't tie yourself into one solution that may cause problems later on.

About the Author Rick F. van der Lans

Rick F. van der Lans is an independent analyst, consultant, author, and lecturer specializing in data warehousing, business intelligence, database technology, and data virtualization. He works for R20/Consultancy (www.r20.nl), a consultancy company he founded in 1987.

Rick is chairman of the annual European Data Warehouse and Business Intelligence Conference (organized in London). He writes for the eminent B-eye-Network.com[3] and other websites. He introduced the business intelligence architecture called the *Data Delivery Platform* in 2009 in a number of articles[4] all published at BeyeNetwork.com.

He has written several books on SQL:

- Introduction to SQL, fourth edition
- SQL for MySQL Developers
- The SQL Guide to SQLite
- The SQL Guide to Ingres
- The SQL Guide to Pervasive PSQL
- The SQL Guide to Oracle

Published in 1987, his popular *Introduction to SQL*[5] was the first English book on the market devoted entirely to SQL. After more than twenty years, this book is still being sold, and has been translated in several languages, including Chinese, German, and Italian. His latest book[6] *Data Virtualization for Business Intelligence Systems* was published in 2012.

For more information please visit www.r20.nl, or email to rick@r20.nl. You can also get in touch with him via LinkedIn and via Twitter @Rick_vanderlans.

About MapR Technologies, Inc.

MapR Technologies delivers on the promise of Hadoop with a proven, enterprise-grade platform that supports a broad set of mission-critical and real-time production uses. MapR brings unprecedented dependability, ease-of-use and world-record speed to Hadoop, NoSQL, database and streaming applications in one unified Big Data platform. MapR is used across financial services, retail, media, healthcare, manufacturing, telecommunications and government organizations as well as by leading Fortune 100 and Web 2.0 companies.

Amazon, Cisco and Google are part of MapR's broad partner ecosystem. Investors include Lightspeed Venture Partners, Mayfield Fund, NEA, and Redpoint Ventures. Connect with MapR on Facebook, LinkedIn, and Twitter.

[3] See http://www.b-eye-network.com/channels/5087/articles/
[4] See http://www.b-eye-network.com/channels/5087/view/12495
[5] R.F. van der Lans, *Introduction to SQL; Mastering the Relational Database Language*, fourth edition, Addison-Wesley, 2007.
[6] R.F. van der Lans, *Data Virtualization for Business Intelligence Systems*, Morgan Kaufmann Publishers, 2012.