Integration of Large-Scale Data Processing Systems and Traditional Parallel Database Technology

Azza Abouzied  
New York University Abu Dhabi  
azza@nyu.edu

Daniel J. Abadi  
University of Maryland, College Park  
abadi@cs.umd.edu

Kamil Bajda-Pawlikowski  
Starburst Data  
kamil@starburstdata.com

Avi Silberschatz  
Yale University  
avi@cs.yale.edu

ABSTRACT
In 2009 we explored the feasibility of building a hybrid SQL data analysis system that takes the best features from two competing technologies: large-scale data processing systems (such as Google MapReduce and Apache Hadoop) and parallel database management systems (such as Greenplum and Vertica). We built a prototype, HadoopDB, and demonstrated that it can deliver the high SQL query performance and efficiency of parallel database management systems while still providing the scalability, fault tolerance, and flexibility of large-scale data processing systems. Subsequently, HadoopDB grew into a commercial product, Hadapt, whose technology was eventually acquired by Teradata. In this paper, we provide an overview of HadoopDB’s original design, and its evolution during the subsequent ten years of research and development effort. We describe how the project innovated both in the research lab, and as a commercial product at Hadapt and Teradata. We then discuss the current vibrant ecosystem of software projects (most of which are open source) that continued HadoopDB’s legacy of implementing a systems level integration of large-scale data processing systems and parallel database technology.

PVLDB Reference Format:

1. INTRODUCTION
In the first few years of this century, several papers were published on large-scale data processing systems: systems that partition large amounts of data over potentially thousands of machines and provide a straightforward language in which to express complex transformations and analyses of this data. The key feature of these systems is that the user does not have to be explicitly aware of how data is partitioned or how machines work together to process the transformations or analyses, yet these systems provide fault-tolerant, parallel processing of user programs. Most notable of these efforts was a paper published in 2004 by Dean and Ghemawat, that described Google’s MapReduce framework for data processing on large clusters [26]. The MapReduce programming model for expressing data transformations, along with the underlying system that supported fault tolerance, parallel processing of these transformations, was at the time widely used across Google’s many business operations, and subsequently became widely used across hundreds of thousands of other businesses, through the open-source Hadoop implementation. Today, companies that package, distribute, support, and train companies to use Hadoop combine to form a multi-billion dollar industry.

MapReduce, along with other large-scale data processing systems such as Microsoft’s Dryad/LINQ project [33, 46], were originally designed for processing unstructured data. One of their most famous use cases within Google and Microsoft was the creation of the indexes needed to power their respective Internet search capabilities—which requires processing large amounts of unstructured text found in Web pages. The success of these systems in processing unstructured data led to a natural desire to also use them for processing structured data. However, the final result was a major step backward relative to the decades of research in parallel database systems that provide similar capabilities of parallel query processing over structured data [27].

For example, MapReduce provided fault-tolerant, parallel execution of only two simple functions: Map, which reads key-value pairs within a partition of a distributed file in parallel, applies a filter or transform to these local key-value pairs, and then outputs the result as key-value pairs; and Reduce, which reads the key-value pairs output by the Map function (after the system partitions the pairs across machines by hashing the keys), and performs some arbitrary per-key computation such as applying an aggregation function over all values associated with the same key. After performing the reduce function, the results are materialized and repackaged to a distributed file system. The model presents several inefficiencies for parallel structured query processing, such as: (1) Complex SQL queries can require a large number of operators. Although it is possible to express these op-
erators as a sequence of Map and Reduce functions, database systems are most efficient when they can pipeline data between operators. The forced materialization of intermediate data by MapReduce—especially when data is replicated to a distributed file system after each Reduce function—is extremely inefficient and slows down query processing. (2) MapReduce naturally provides support for one type of distributed join operation: the partitioned hash join. In parallel database systems, broadcast joins and co-partitioned joins—when eligible to be used—are frequently chosen by the query optimizer, since they can improve performance significantly. Unfortunately, no implementation of broadcast and co-partitioned joins fits naturally into the MapReduce programming model. (3) Optimizations for structured data at the storage level—such as column-orientation, compression in formats that can be operated on directly (without decompression), and indexing—were hard to leverage via the execution framework of the MapReduce model.

Even as studies continued to find that Hadoop performed poorly on structured data processing tasks when compared to shared-nothing parallel DBMSs [38, 42], widely respected technical teams—such as the team at Facebook—continued to use Hadoop for traditional SQL data analysis workloads. Although it is impossible to fully explain the reasoning behind the continued popularity of Hadoop for structured data processing, possible explanations include the following:

- The parallel database industry had no free and open source equivalent to the thriving Hadoop community.
- Hadoop’s adoption at well-known large Web companies, such as Yahoo, Facebook, and Twitter gave it an additional level of credibility in terms of scalability and ability to run over massive, heterogeneous, shared nothing clusters of commodity servers.
- Hadoop had a level of fault tolerance that was unmatched by even the most fault tolerant parallel database system. For example, Hadoop was able to handle server failure in the middle of query processing without having to restart a query. As clusters scale, this level of fault tolerance becomes increasingly important.
- Many workloads contained a mix of structured and unstructured data processing. Using a single system for all types of query processing, that had the ability to parallelize user defined functions over unstructured data was convenient and typically reduced data transfer and management costs.

The reasons behind Hadoop’s popularity thus included both technical and non-technical considerations. However, the technical reasons that contributed to the rise of Hadoop often tended to be under-appreciated. HadoopDB was architected with the intention of taking Hadoop’s technical contributions seriously. For example, Hadoop’s level of fault tolerance during run-time query processing, its ability to handle heterogeneous clusters, and its ability to parallelize user defined functions were legitimate reasons behind Hadoop’s success. HadoopDB was therefore designed as a hybrid system that was able to achieve these advantages of Hadoop, yet at the same time achieve the high performance and efficiency of traditional parallel database systems on structured SQL queries.

In the next section, we give a technical overview of HadoopDB, according to the way it was described in the original paper. In Section 3 we describe how the project evolved in the research lab over the past decade after the original paper was published, as HadoopDB expanded to handle additional use-cases and workloads. In Section 4, we discuss the commercialization of HadoopDB and how the commercial product impacted the development of additional features and capabilities. In Section 5, we describe several other integration efforts of large-scale data processing systems with parallel database technology and the current commercial landscape and open-source software tools in this area. In Section 6, we conclude with a few ideas for future work.

2. HADOOPDB

HadoopDB placed a local DBMS on every node in the data processing clusters (Figure 1). Structured data were stored in tables, sharded across these database systems. Data was indexed within each DBMS as appropriate. The original HadoopDB implementation used PostgreSQL as the database system on each node; however, the HadoopDB paper pointed out that improved performance could be gained via using column-store systems. This capability of using an underlying column-store system was added to the codebase the following year and its associated performance gains1 were published in SIGMOD 2011 [22] as part of a larger HadoopDB follow-up paper. Regardless of whether HadoopDB used an underlying row-store or column-store, storing data in systems optimized for managing structured data enabled significant speedup in the map functions of MapReduce tasks over structured data. HadoopDB pushed filtering, projection, transformation, and even some join and partial aggregations into the database systems on each node.

Figure 2 shows an example of how query processing work is pushed into the single node database systems. A SQL query is submitted to the system; in our example this query requests the total revenue per year from a sales table. The left side of the figure shows how this query may be converted

1See also Figure 3 that we will discuss below in which HadoopDB-VW leverages a column-oriented DBMS.
to Map and Reduce operators in a MapReduce job. The leaf operator is a scan of the sales table. For each tuple, a projection operator (called Select) extracts the relevant attributes for this query (year and revenue). An aggregation operator (called Group By) groups all tuples with the same year and sums the total revenue within each group. Each of these operators — the scanning, the projecting of relevant attributes, and the aggregating — can be done on a per-partition basis in an embarrassingly parallel fashion. Therefore, systems that automate the conversion of our example SQL query to MapReduce tasks (such as the original version of Hive) will perform all these tasks inside Map functions. For efficiency, all three of these operations can be combined into a single Map task (instead of requiring one Map task per SQL operator), since each of these operators occur consecutively.

The aggregation that is done inside the Map task is done on a per-partition basis. However, our example query required a global aggregation of revenue per year. Therefore, additional aggregation is necessary to complete the processing of this query, since each partition may contain tuples associated with the same year. A Reduce task is used for this global aggregation. Prior to the Reduce task, the aggregated data produced by the Map task is partitioned by year (so that all Map output associated with the same year end up on the same physical machine), and then the aggregation is performed again in a Reduce task (which sums each intermediate sum associated with the same year). In truth, the aggregation done by the Reduce task is the only one that is necessary. The “pre-aggregation” done in the Map task could have been dropped without changing the semantics of the query. However, network is often a bottleneck in large-scale data processing systems, and pre-aggregating data enables a reduction of data that must be shipped over the network. Therefore, the pre-aggregation in the Map phase is added by systems such as Hive as an optimization.

The right side of Figure 2 shows how the same query is performed in HadoopDB. The query processing operators that were performed in the Map task in the query plan from the left side of the Figure are converted into a single operator that is pushed down into the shards of the database system and performed there as a series of traditional relational operators. The results of this query are then partitioned by year and the final aggregation performed inside a Reduce task, in the same way as the left side of the figure.

At a high level, HadoopDB was able to achieve the following desirable properties in a data processing framework:

1. **Flexibility with minimal cognitive overload.** Querying data in HadoopDB could be done in SQL, MapReduce, or combinations thereof. HadoopDB automatically pushed as much processing as possible of structured data into the underlying database systems, without forcing the user to be aware of where processing was being performed or how data was shardsed.

2. **Ability to run in a heterogeneous environment.** By leveraging the Hadoop framework for scheduling operators within a query plan, HadoopDB overcame one of the key scalability hurdles found in parallel database systems that must run over thousands of machines. In such environments, it is impossible to achieve performance homogeneity across machines. Even if every single machine within the cluster consists of the exact same hardware, it is usually the case that at that level of scale that at least one of the machines is running slower than the others—either because of malfunctioning hardware that allows the machine to stay alive, but at reduced performance, or because of some software issue that prevents the operating system or database system from giving the query the same level of resources as the other nodes in the cluster. In these heterogeneous conditions, there is a danger that the run-time of a task is lower-bounded by the run-time of the slowest node in the cluster. HadoopDB leveraged Hadoop to side-step such stragglers, by preemptively re-scheduling query operators on other nodes (usually a replica) within a cluster.

3. **Fault-tolerance.** Another scalability hurdle in parallel database systems is mid-query fault tolerance. In traditional clusters containing less than a hundred machines, failure is a rare event—usually occurring less frequently than one failed machine per day. Therefore, parallel database systems typically did not support mid-query fault tolerance. If any machine involved in processing a query failed in the middle of (or shortly after) its task, the query would abort and start from scratch using a different replica of the data. As long as queries take orders of magnitude less time to execute than the mean time to (at least one) machine failure in the cluster, this lack of support for mid-query fault tolerance is not problematic. However, as the data scales, and the number of machines in a cluster scales accordingly, two things change: First, queries tend to take longer to run since it is generally impossible to get perfectly linear speedup on all queries. Second, the mean time to (at least one) machine failure decreases: for very large clusters of thousands of cheap, commodity machines, it is not uncommon for at least one machine to be failing on the order of minutes. This combination of increased query latency with lower mean time to failure results in a requirement for mid-query fault tolerance. HadoopDB leveraged Hadoop’s checkpointing of intermediate data to disk after Map tasks, along with the determinism of Map and Reduce tasks in the
3. **RESEARCH GROWTH**

After the initial prototype and paper, HadoopDB continued to develop — both in the research lab and in industry. In this section, we discuss how it developed in the research lab, and in the next section we discuss how it was commercialized and developed in the real world.

3.1 **Split Execution**

We further improved the performance of HadoopDB for join and aggregation queries using strategies designed specifically for HadoopDB’s split execution environment [22], where the goal is to push as much query execution as possible into the higher performing underlying database systems. Examples of these improvements include:

1. **Referential Pre-partitioning.** In cases where a chain of tables can be related to each other via primary-key / foreign-key references, we extended HadoopDB to aggressively pre-partition these tables in order to ensure that joins can be computed locally within the database systems at each node.

2. **Split MapReduce/Database Joins.** HadoopDB was improved to choose optimally among multiple implementations of the broadcast join: (i) a Map-side join where at each node, the smaller table is read from HDFS and transformed into an in-memory hash table, which is then probed by tuples accessed sequentially from the larger table within a Map task, or (ii) the smaller table is inserted into a temporary table within each HadoopDB node’s underlying database system and the join is pushed entirely into the DBMS. Furthermore, Hadoop leveraged semi-joins, especially when the projected column from the semi-join is small, and implemented them via selection predicates using the SQL \texttt{IN} clause, which was pushed down and performed by the underlying database system.

3. **Post-join partial aggregations.** When a query involves both joining tables and aggregating results of the join in a way that requires multiple MapReduce iterations—such as when the join key is different from the group-by aggregation key—HadoopDB computed partial aggregations at the end of the first Reduce task. Such partial aggregations saved significant IO (and runtime) costs by preventing unnecessary intermediate data from being written to HDFS.
4. **Pre-join partial aggregations.** HadoopDB transformed aggregation operators into partial aggregation operators and computed these partial aggregations before a join when the product of the cardinalities of the group-by and join-key columns was smaller than the cardinality of the entire table.

### 3.2 Invisible Loading

A key selling point of Hadoop was its low time-to-first analysis: as soon as data is produced, the data could be dumped into Hadoop’s distributed file system and be immediately available for analysis via MapReduce jobs. In contrast, database systems typically require data to be loaded and tuned prior to being available for SQL queries. This initial data preparation for database systems usually involves a non-trivial human cost of data modeling and schematizing in addition to the tunable computational costs of copying, clustering and indexing the data. Hadoop and NoSQL systems allowed users, especially those who were not entirely familiar with the data, to trade cumulative long-term performance benefits for quick initial analysis.

This presented another opportunity for a systems-level hybrid between large scale data processing systems and traditional database systems. We extended HadoopDB with an **Invisible Loading** (IL) feature that achieves the low time-to-first analysis of MapReduce jobs over a distributed file system while still yielding the long-term performance benefits of database systems [19]. IL seamlessly moves data from a file system into a database system (i) with minimal human intervention: users only need to write their MapReduce jobs using a fixed parsing API and optionally use a library of standard processing operators (e.g. filter, join, group by, etc.) and (ii) without any human detection: IL piggybacks on running MapReduce jobs and moves data into the database systems without a visible increase in response time by incrementally loading vertical and horizontal partitions of the data into the database system and then incrementally reorganizing the data.

In more detail, the important technical features of invisible loading include:

1. **Laziness and opportunism.** Only data that is actually accessed by MapReduce tasks get loaded into the database system. As data is accessed by a MapReduce job, invisible loading takes advantage of the data already being in cache to opportunistically load it into the database system. This results in the more frequently accessed attributes to be more fully loaded into the database system.

2. **No requirement of complete schema knowledge.** MapReduce users generally write MapReduce jobs over datasets for which they do not have complete knowledge of the semantics or types of the various attributes within the records/key-value pairs that these jobs access. The invisible loading technique therefore did not require users to specify the schema of a file a priori. Rather, the code simply injects itself into the data parsing logic contained within Map tasks to infer the parts of the schema that are present in those tasks.

3. **Utilization of column-stores to align multiple attributes.** The above described features of laziness, opportunism and incomplete schema knowledge implies that some columns will be loaded before others. Invisible loading leveraged column-oriented storage to enable increased flexibility for incremental loading of different attributes from the same HDFS file.

4. **Incremental data reorganization.** Invisible loading included an implementation of an Incremental Merge Sort (IMS), that gradually sorted columns on which users often executed selection predicates. IMS had the added advantage of only performing a fixed amount of work per MapReduce job, which kept overhead costs low in comparison with other adaptive organization techniques of the time.

5. **A polymorphic library of data access and processing operators.** These operators were designed to work correctly over data spread arbitrarily across the file system and the underlying database systems of HadoopDB.

From a performance perspective, the overhead of incrementally loading data from the file system to the database system was barely detectable by the end user. The primary detectable side-effects were the performance improvements that resulted in an increased amount of data being inside the higher performing database systems [19].

### 3.3 Sinew

The prerequisite to use invisible loading is that data found on the file system is relational and straightforward to load into a relational database system. The reason why the data is not already in a database system is just for convenience — the user did not want to go to the effort of understanding all the attribute semantics and types of the data to the point where a schema can be declared, or spend the time waiting for the data to be loaded into a database system. But if a user had the prerequisite time and effort, the data could be loaded up front.

However, given Hadoop’s usage as a data lake that stores arbitrary data, much of the data stored in Hadoop is not relational: key-value, nested and other semi-structured and self-describing data types are common. While the invisible loading technique was a good solution for relational data, a different technique was necessary for “multi-structured” data.

This observation lead to the development of Sinew [43] — a technique similar in motivation to invisible loading, but designed for multi-structured data. The basic idea was to present to the user a “universal relation” for each entity-set, with one column for each attribute that is defined in at least one entity within the entity-set. If an entity contains a nested object, the nested keys are flattened and referenceable as distinct columns using a dot-delimited name with the nested key preceded by the key of the parent object. The less structured an entity-set, the more sparse is its universal relation. This universal relation is queryable via SQL, where NULL is returned for any column value that is undefined for a specific entity.

As data is loaded into the database system, it starts off being serialized using a Sinew-specific semi-structured serialization format which is stored within a binary column in the DBMS called a ”column reservoir”. This column reservoir is parsed on the fly at query time to extract the relevant columns for a given input query. A schema analyser
optimized relational schemas for datasets commonly found
significant effort was spent in the automatic generation of
relation) is rarely optimal for data analysis. Consequently,
careful consideration of the optimal schema for a dataset
position of this initial relation would lead to improved se-
cation does not need to occur all at once. This allows mi-
original loading" technique described above, such that this migra-
mentation of virtual columns at query time. Migrating
dense columns will usually lead to overall improved system
performance, but if the RDMS is a row-store, the schema
alyzer has to balance the potential benefits with the as-
sociated system overhead of maintaining tables with many attributes.
Once a column is chosen for migration, a "column mate-
ializer" (Figure 4) incrementally migrates the column into
a physical column in the DBMS, similar to the "invisible
loading" technique described above, such that this migra-
does not need to occur all at once. This allows mi-
gation to have no discernible effect on system performance
(aside from the performance benefit of reduced parsing costs
to access the column at query time).

3.4 Automatic Schema Generation
Neither the work on invisible loading, nor Sinew at-
tended to optimize the initial version of a generated
chema. The work on invisible loading inferred the schema
of a dataset based on how it was used. As mentioned above,
the invisible loading technique intercepted Map tasks and
looked for parsing logic within those tasks. It used this
parsing logic to detect the schema of the raw files. In many
cases, the way data was organized in raw files is based solely
on the whims of the processes that generated the data, and
little concern is made for organizing data in a way that fa-
cilitates query processing. Meanwhile, for nested and other
types of semi-structured data, Sinew used a single universal
relation per entity instead of considering whether decom-
position of this initial relation would lead to improved se-
manics and performance. In practice, we found that more
careful consideration of the optimal schema for a dataset
could lead to many benefits — especially for nested data
sets, for which the initial data organization (or the universal
relation) is rarely optimal for data analysis. Consequently,
significant effort was spent in the automatic generation of
optimized relational schemas for datasets commonly found
in Hadoop. This work culminated in a technique that anal-
yses raw nested files and uses the statistical properties of
pecific attribute expressions to automatically generate a
ormalized schema that can be used in HadoopDB (or any
other relational database system) [28].

3.5 HadoopDB for Graph Datasets
HadoopDB was originally designed for relational datasets,
and the work on Sinew and automatic schema genera-
tion extended its applicability to nested and other semi-
structured datasets. However, in 2011, we discovered that
the HadoopDB architecture is well-suited for an additional
use-case: graph datasets. In particular, we found that
HadoopDB could scale graph analysis algorithms beyond
existing limitations at the time. The basic idea was to re-
place the single-node database systems from the original
HadoopDB architecture with single-node optimized graph
database systems\(^2\) as shown in Figure 5.

\(^2\)Our initial prototype used RDF-3X as the single-node
graph database system on each HadoopDB node.
rounds of communication over the network needed for non-trivial graph processing queries can quickly become a performance bottleneck, leading to high query latencies.

Therefore, instead of using hash partitioning, we used a graph partitioning algorithm. This enabled vertices that are close to each other in the graph to be stored on the same machine, which resulted in a smaller amount of network communication at query time. For subgraph matching queries, entire subgraphs could be matched in parallel across the single-node high performance graph database systems in the HadoopDB architecture.

An additional complication relative to HadoopDB was how data is replicated. HadoopDB replicated entire shards, which allowed management and accounting of the replicas to be straightforward. However, for graph data, since a graph partitioning algorithm was being used, we found that performance could be improved if there was some overlap in the subgraphs that were produced from the partitioning algorithm. The intuition is that, in practice, some vertices are much more broadly connected than other vertices. Placing such well-connected vertices on only a single machine made it challenging to generate a balanced partitioning across nodes. Instead, we allowed additional replication at the edges of partitioned subgraphs, and we introduced a method for automatic decomposition of queries into chunks that can be performed independently, with zero communication across partitions. These chunks were then reconstructed using the Hadoop MapReduce framework.

The architecture of this approach is shown in Figure 5. Graph vertices are loaded into the system by feeding them into the data partitioner module which performs a partitioning of the RDF graph by vertex that is initially disjoint. The output of the partitioning algorithm is then used to assign vertices to worker machines according a placement algorithm that determines which vertices were on the boundary of the graph partitioning output are good candidates for extra replication across partitions. Each partition is then loaded into the optimized graph database system on each node.

The master node serves as the interface for graph analysis queries. It accepts queries and decomposes them into operators that can be performed in isolation across the single node graph database systems, and ships these operators to the worker nodes. All cross-partition operations are performed using the Hadoop processing framework, just as in the original HadoopDB model.

This work continued with follow-up research on query optimization, compression, and dynamic graph partitioning and replication.

4. THE INDUSTRIAL EVOLUTION

The original HadoopDB codebase was released open source along with the original VLDB publication. Since several thousand downloads shortly followed its release, it became clear that there was significant commercial interest in the HadoopDB technology. This created an opportunity to develop the prototype into a production-ready system that could be deployed by real enterprises. Two of the authors of the original HadoopDB paper (Daniel Abadi and Kamil Bajda-Pawlikowski) joined forces with a student at the Yale School of Management (Justin Borgman) to start a company called Hadapt whose goal was to transform HadoopDB from a research idea into a usable system. The expectation was that commercialization, in addition to demonstrating the applicability of the original design, would inspire further research within the context of the HadoopDB project.

Hadapt was founded in 2010. It used the success of the initial prototype, along with the patents associated with the research innovations described in Section 3—patents on the HadoopDB architecture, split execution, invisible loading, Sinew, and application to graph data—to raise over $16 million dollars in two rounds of financing. Over the next two years, hired approximately 30 employees, mostly engineers who worked on improving and expanding the HadoopDB-based technology.

Hadapt succeeded in taking the HadoopDB technology from academic prototype to production-ready software. In so doing the company gained customers who requested additional features that had not been anticipated in the original HadoopDB design. Most of these features related to the data lake nature of Hadoop deployments. These customers found the Hadoop distributed file system (HDFS) to be cost-efficient in storing all kinds of data (structured, semi-structured, and unstructured). Although HadoopDB was able to achieve high performance and fault tolerance when querying structured and semi-structured data, it lacked sufficient native capabilities to interact with the unstructured data that sat in the same file system. Hadapt customers wanted to access unstructured data directly from the Hadoop engine, via extensions to Hadapt’s SQL interface.

Consequently, Hadapt added a scalable full-text search capability by leveraging SOLR. Each node ran a shard of SOLR in addition to the DBMS shard from the original HadoopDB design. Hadapt supported indexing columns in a table via SOLR, thereby enabling full-text search using SQL syntax extensions, typically in the WHERE clause of SQL queries. Later, as the product expanded its applications to more latency-sensitive BI analytics, Hadapt allowed customers to trade fault tolerance for reduced query runtime using the Hadapt ”Interactive Query” capability.

Along the way, native support for modern HDFS file formats such as ORC and Parquet was introduced. Finally, Hadapt built a cost-based optimizer that leveraged table and column statistics to reorder joins and choose appropriate distribution methods.

In 2014 Hadapt was acquired by Teradata and became a new division called the Teradata Center for Hadoop. Hadapt’s impact began with a significant improvement of the Teradata QueryGrid for Hadoop. By applying the principles of split execution from HadoopDB, leaf parts of the query plan were pushed down from Teradata into Hadoop. This optimization substantially reduced expensive CPU cycles on Teradata clusters and minimized network traffic.

Later, in order to further extend the reach of QueryGrid, Teradata decided to back the open source project Presto that supported many data sources beyond Hadoop. The ex-Hadapt team embarked on a multi-year roadmap to contribute to Presto and bring an enterprise-ready distribution of the project to the market. What followed were many enhancements that descended from Hadapt, especially in the areas of security, performance, ANSI SQL compatibility, BI tool support, and data source connectivity.

As a result of these efforts, Presto experienced an unprecedented global growth in popularity in both on-premise and cloud deployments. In late 2017 many of the original Hadapt team members founded Starburst Data, an independent
company focused on developing and providing commercial support for its enterprise-grade distribution of Presto.

5. THE SQL/MR ECOSYSTEM

Around the time when HadoopDB was developed, much debate and research effort focused on improving the interface to large-scale data processing systems. At one extreme was the original MapReduce paper which expressed all transformations in Map and Reduce functions. At the other extreme was Hive [44, 2] which provided a SQL interface\(^3\). There were several popular interfaces within these two extremes, such as Pig [37], SCOPE [25], and MapReduce extensions in commercial parallel database systems such as Greenplum and Aster Data.

HadoopDB aimed to be a hybrid not at the language or interface level but at the systems level. To achieve this, it integrated certain features of MapReduce-style systems (fault tolerance, handling heterogeneous commodity clusters, ability to parallelize user-defined functions) with some capabilities of parallel database systems (storage-level optimizations, efficient query processing). Perhaps the most important contribution of HadoopDB was breaking down the illusory divide between the two technologies and embracing the important technical advantages of both.

HadoopDB led the way in bringing systems implementation techniques from the parallel database systems community into the large-scale data processing community. Several subsequent projects continued in this direction aiming for even higher performance and efficiency in analyzing structured data.

One way that HadoopDB gained performance advantage over the state of the art was by (optionally) storing structured data in columnar format. It is well known that column-oriented storage can improve performance of query workloads that scan large amounts of data but analyze only a subset of attributes from a given table per query. HadoopDB implemented this approach by placing single-node column-oriented database systems on each machine. The Hadoop community subsequently also introduced columnar storage capabilities in native HDFS file formats. The two most widely-used options are ORCFile [3] and Parquet [4]. They both support semi-structured (nested) data.

Parquet and ORCFile use PAX pages [20] for columnar storage. In PAX, data is kept in columns within pages, but a given page may consist of multiple columns from the same table. This makes tuple reconstruction faster since all data needed to perform this operation can be found on the same page. At the same time, though, PAX reduces scan performance compared to pure column stores since not all data for a given column is placed contiguously on disk. While Parquet and ORCFile maintain some of the other benefits of columnar storage such as column-oriented compression, they are not query execution engines. Rather, they are file formats that were introduced so that a new wave of SQL engines for Hadoop could benefit from column-oriented storage and processing just as HadoopDB did earlier.

This next generation of Hadoop-based query engines followed HadoopDB’s lead in systems-level integration of low-latency parallel database techniques within large-scale data processing systems. We now give several examples of such SQL engine projects and work subsequent to the HadoopDB paper.

As mentioned above, Hive existed solely as a language-based hybrid at the time the HadoopDB paper was published. Subsequently, its community, mostly centered around a Hadoop distributor called Hortonworks, transformed the project into a more systems-level hybrid. First, Hive evolved to support pluggable execution engines and proposed Apache Tez [40] as an alternative to MapReduce. Tez, which is similar to Dryad [33], represents data processing as DAGs, allowing more efficient execution of SQL operators. Next, Hive gained an additional processing layer called LLAP (Live Long and Process) [24] that introduced per-node daemons responsible for local query execution and caching hot data. In essence, LLAP instances served a similar purpose in Hive as local DBMS servers in HadoopDB. To further improve the performance of complex queries, Hive incorporated Apache Calcite [23] that provided cost-based optimization using statistics kept in the Hive Metastore. Finally, transactional table support using ORC ACID completed Hive’s journey towards Big Data Warehousing on Hadoop.

Spark [47] is a distributed general-purpose processing engine comparable to Tez and Dryad in expanding data processing operators beyond ‘Map’ and ‘Reduce’. Similar to Hadoop, Spark made fault tolerance during query processing a key property but provided users with more control in specifying the required level of fault tolerance. Compared to MapReduce, Spark achieved significant speedup for iterative data processing workloads and therefore become popular in machine learning, graph analytics, and ETL. Spark supports data analytics via a variety of end user interfaces such as Scala, Python, and R. Just as Hive and HadoopDB brought SQL to Hadoop, the Shark project [45] brought SQL to Spark. By using Spark instead of MapReduce, Shark was able to achieve higher performance than the original Hive. In 2014, Shark was replaced by SparkSQL [21], a brand new module inside Spark. SparkSQL leveraged a new DataFrame API, featured a query optimizer called Catalyst, and introduced a number of execution engine improvements collectively referred to as Project Tungsten. More recently, Databricks open-sourced Delta [6], a transactional table storage for Spark built on top of Parquet.

Drill [1] is a distributed SQL engine inspired by Google Dremel [36]. The project was created by MapR but never got embraced by the leading Hadoop vendors. Apache Drill provided connectivity to data sources beyond HDFS, including plain JSON files, MongoDB, HBase, and Object stores.

Impala [34], an engine built by Cloudera, went a step farther than HadoopDB. Impala, like HadoopDB, delegated execution of query operators to the single-node database systems deployed on each Hadoop machine. However, in HadoopDB only the lower parts of the query plan were pushed down into the database systems and communication across nodes was managed using Hadoop’s MapReduce framework. By contrast, Impala built a full parallel database execution engine within a Hadoop cluster thereby allowing entire query plans to avoid MapReduce. In so doing Cloudera’s SQL engine lost much of the fault tolerance advantages of MapReduce. Nonetheless, Impala was able to achieve low latency queries via fully pipelined relational operators and runtime code generation with LLVM.
Presto [41] is another full parallel database execution engine that avoids MapReduce entirely. The project was originally created at Facebook as the successor to Hive to allow for concurrent interactive queries over large datasets. Similar to Impala, Presto fully pipelines query execution for performance and therefore does not support mid-query fault tolerance. The open source project features a highly efficient execution engine employing vectorized columnar data processing, runtime query bytecode compilation, optimized data structures, and multi-threaded processing leveraging multi-core CPUs efficiently.

Presto has several well-known users, such as Airbnb, Comcast, GrubHub, Facebook, FINRA, LinkedIn, Lyft, Netflix, Twitter, and Uber. It achieves low-latency queries, high concurrency, and native ability to query multiple data sources including Object Stores, HDFS, NoSQL and RDBMS. In contrast to the SQL-on-Hadoop projects discussed earlier, Presto is often referred to as an SQL-on-Anything engine. Its inherent separation of compute and storage makes Presto well-suited for deployments in the cloud and in cloud-like environments such as Kubernetes.

Among the key members of the Presto development community is the ex-Hadapt team at Starburst Data [10] which recently contributed the Cost-Based Optimizer [8].

6. CONCLUSION

Hellerstein et al. note that the “unfortunate consequence of the disaggregated nature of contemporary data systems is the lack of a standard mechanism to assemble a collective understanding of the origin, scope, and usage of the data they manage” [30]. While unified frameworks like Presto and central metadata repositories like Hive Metastore might help ameliorate some of these issues by providing a single catalog for many datasets, several end-user problems remain that require novel, usable solutions. These include (i) discoverability: allowing analysts to determine whether certain data exists and, if so, and how it is processed, (ii) minimizing wasted effort by sharing data cleaning, pre-processing and data analysis efforts, as well as learned insights, (iii) better governance through tracking data from source to use, and (iv) better automation of schema inference, data cleaning, repair, and pattern finding. These problems require human-in-the-loop tools that visualize metadata, processes, provenance and results across an entire organization’s data lake access.

7. ACKNOWLEDGEMENTS

This work was sponsored by the National Science Foundation under grants IIS-1718581 and IIS-1910613.

8. REFERENCES
