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:
Azza Abouzied, Daniel J. Abadi, Kamil Bajda-Pawlikowski, Avi Silberschatz. Integration of Large-Scale Data Processing Systems and Traditional Parallel Database Technology. *PVLDB*, 12(12): 2290-2299, 2019.
DOI: https://doi.org/10.14778/3352063.3352145

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 [28]. The MapReduce programming model for expressing data transformations, along with the underlying system that supported fault tolerant, 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 [35, 47], 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 [29].

For example, MapReduce provided fault-tolerant, parallel execution of only two simple functions[1]: *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 replicated to a distributed file system. The model presents

[1]This limitation is not shared by Dryad. Nonetheless, Hadoop implemented MapReduce instead of Dryad.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352145

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 operators 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 fit 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 [40, 43], 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 handled 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 designed to be a hybrid system capable of achieving these advantages of Hadoop, while also achieving the high performance and efficiency of traditional parallel database systems on structured SQL queries.

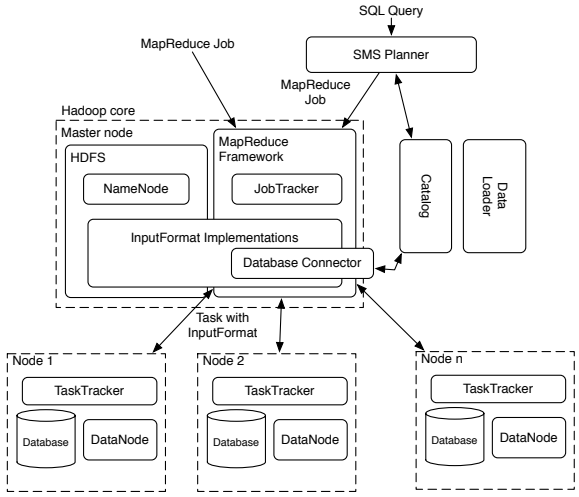

Figure 1: The HadoopDB System Architecture [18]

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 shortly after the initial release and its associated performance gains[2] 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

[2]See also Figure 3 that we will discuss below in which HadoopDB-VW leverages a column-oriented DBMS.

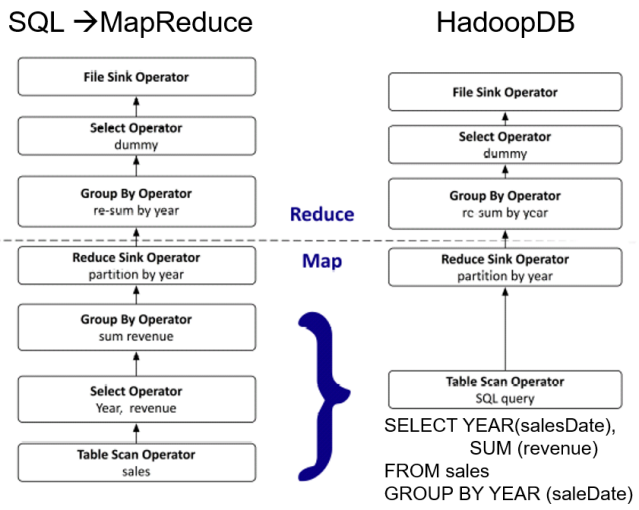

Figure 2: Pushing Map functions into the DBMS

requests the total revenue per year from a sales table. The left side of the figure shows how this query may be converted 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 into the underlying database systems, without forcing the user to be aware of where processing was being performed or how data was sharded.

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 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 frequent machine failure. In traditional clusters containing fewer 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 (instead of hours or days). 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 MapReduce model to implement mid-query fault tolerance and thereby scale to very large deployments.

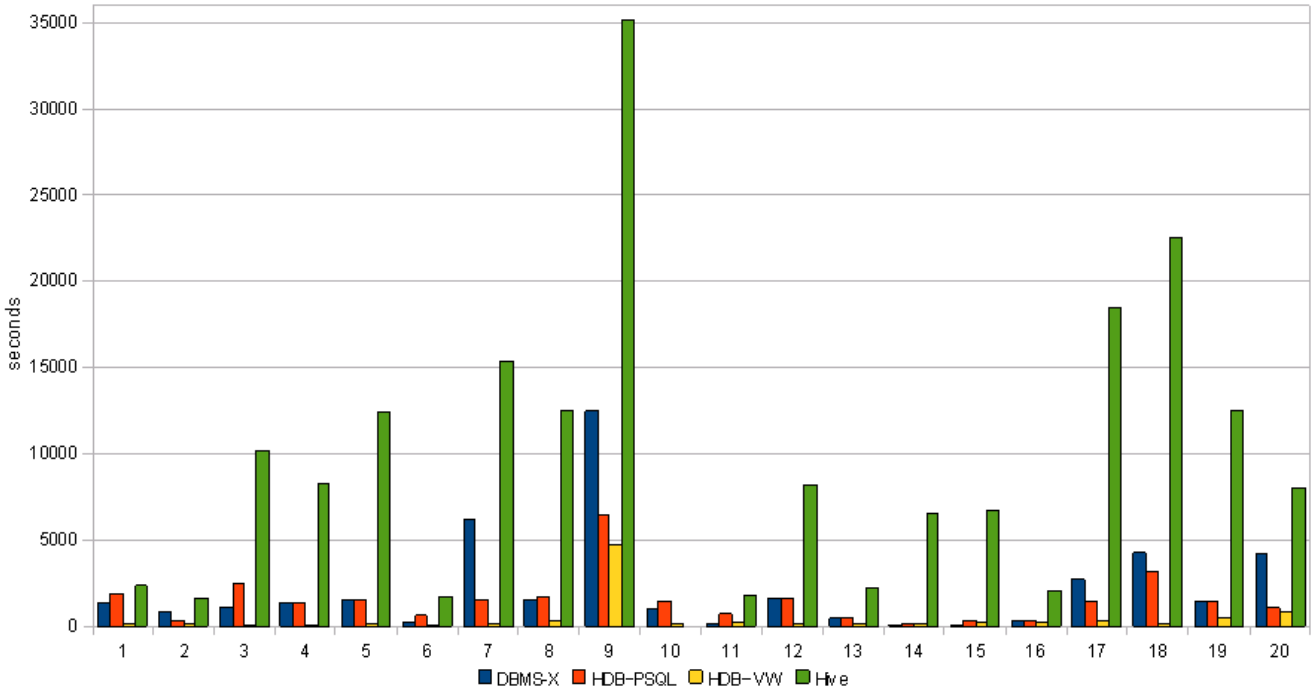

Figure 3: HadoopDB performance on TPC-H queries vs. Hive and commercial system DBMS-X [22]

4. ***Performance.*** Across a variety of data processing tasks, HadoopDB outperformed simple SQL-into-MapReduce translation layers (such as Hive), often by orders of magnitude [18, 22]. Figure 3 shows a performance benchmark comparison on TPC-H (see Bajda-Pawlikowski et. al for more details on this benchmark, experimental set-up, and results [22]) in which the performance of HadoopDB is compared with Hive and a commercial parallel row-oriented database system (that is anonymized as required by their license agreement and therefore called DBMS-X). Two versions of HadoopDB are compared: one in which PostgreSQL (a row-store) is used as the single-node database system on each node, and one in which VectorWise (an optimized column-store) is used instead of PostgreSQL. HadoopDB clearly outperformed Hive, and the column-oriented version of HadoopDB was even able to outperform DBMS-X.

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. ***Split MapReduce/Database Joins.*** When it was possible to use a broadcast join, HadoopDB was improved to choose optimally among multiple implementations: (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 `IN` clause, which was pushed down and performed by the underlying database system.

2. ***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.

3. ***Pre-join partial aggregations.*** HadoopDB transformed aggregation operators into multiple stages of 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 from 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 had not already been loaded into a database system was just for convenience: the user did not want to go to the effort of understanding all the data semantics and attribute types 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, non-relational data is frequently found: 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 [44] — 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. This universal relation contained one column corresponding to 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 relation is queryable via SQL, where NULL is returned for any column value that is undefined for a specific entity.

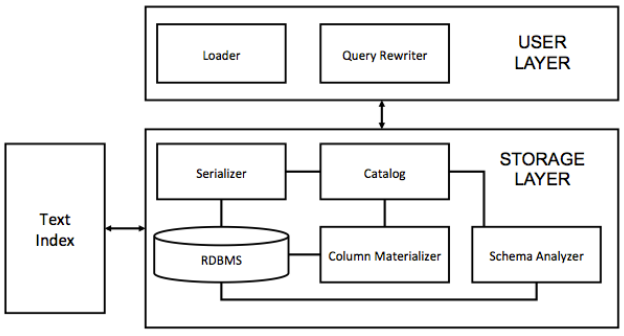

Figure 4: Sinew architecture [44]

As data is loaded into the database system, it starts off being serialized using a Sinew-specific semi-structured serialization format into 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 analyzer (see Figure 4) tracks a

query workload and the statistical properties of each column and decides which columns—if they were to be migrated from "virtual columns" in the column-reservoir to physical columns within the RDBMS table—will result in performance improvement. Dense (i.e. frequently appearing) attributes and those with a cardinality that significantly differs from the RDBMS optimizer's default assumption are typically good candidates for migration. The primary impetus in selecting which columns to migrate to physical columns is to reduce the system cost of on-the-fly materialization of virtual columns at query time. Migrating dense columns will usually lead to overall improved system performance, but if the RDBMS is a row-store, the schema analyzer has to balance the potential benefits with the associated system overhead of maintaining tables with many attributes.

Once a column is chosen for migration, a "column materializer" (Figure 4) incrementally migrates the column into a physical column in the DBMS, similar to the "invisible loading" technique described above, such that this migration does not need to occur all at once. This allows migration 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 attempted to optimize the initial version of a generated schema. 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 these 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 facilitates query processing. Meanwhile, for nested and other types of semi-structured data, Sinew used a single universal relation per entity instead of considering whether decomposition of this initial relation would lead to improved semantics 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 analyses raw nested files and uses the statistical properties of particular attribute expressions to automatically generate a normalized schema that can be used in HadoopDB (or any other relational database system) [30].

3.5 HadoopDB for Graph Datasets

HadoopDB was originally designed for relational datasets, and the work on Sinew and automatic schema generation 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 [33]. The basic idea was to replace the single-node database systems from the original

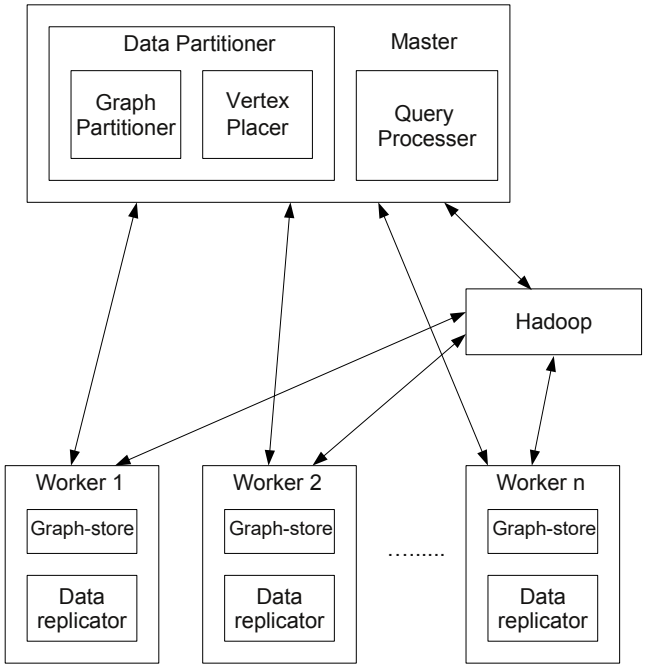

Figure 5: Using HadoopDB for graph datasets

HadoopDB architecture with single-node optimized graph database systems[3] as shown in Figure 5.

We also found that the way graph data is partitioned across the single-node graph database systems is important. In 2011, existing distributed graph analysis systems generally hash partitioned graph vertices across the machines in a cluster. This partitioning algorithm worked well for simple index lookup queries, where queries focus on small numbers of vertices or edges. However, for more involved queries, such as sub-graph pattern matching, efficiency was far from optimal. This is because many graph processing algorithms require traversing the graph along its edges. If data is hash-partitioned, neighboring vertices will usually be found on different physical machines, and traversing the edge between vertices requires communication across machines. The (potentially multiple) 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.

An additional complication relative to HadoopDB was how data is replicated. The original HadoopDB architecture replicated entire shards. A replica of a shard contained **all** data within that shard, and no data from any other shard. The original partitioning of data across shards was disjoint. Thus, a data item could be found only inside the shard that it was partitioned to, along with all the replicas of that shard. This disjoint partitioning and complete shard replication allowed management and accounting of the replicas

[3]Our initial prototype used RDF-3X as the single-node graph database system on each HadoopDB node.

to be straightforward. However, for graph data, we found that more complicated replication algorithms could improve performance. 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 caused two problems:

1. It is challenging to generate a balanced partitioning across nodes.

2. The large number of vertices they are connected to results in a likelihood that a non-trivial percentage of them are stored in a different partition.

Instead, we allowed additional replication at the borders of partitioned subgraphs such that the same vertex can be replicated to multiple partitions, thereby allowing well-connected vertices to be stored within the same partition of a larger percentage of its neighbors.

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 an initially disjoint partitioning of the graph by vertex. 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 and are good candidates for extra replication across partitions. Each partition is then loaded into the optimized graph database system on each node and indexed as appropriate.

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. Query plans are a composition of parallel operators that work in isolation across the single node database systems and cross-partition steps implemented within Hadoop.

This work continued with follow-up research on graph query optimization [34], graph compression [37], and dynamic graph partitioning and replication [32].

4. THE INDUSTRIAL EVOLUTION

The original HadoopDB codebase was released open source alongside 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 [14], split execution [13], invisible loading [12], Sinew [17], and application to graph data [16]—to raise over $16 million dollars in two rounds of financing [7]. Over the next two years, Hadapt 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. Along the way, 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 Hadapt engine, via extensions to Hadapt's SQL interface.

Consequently, Hadapt added a scalable full-text search capability by leveraging SOLR [5]. Each node ran a shard of SOLR in addition to the HadoopDB DBMS shard. 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 [15]. 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 [11] 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 [9], an independent company focused on developing and providing commercial support for its enterprise-grade distribution of Presto.

5. THE SQL/HADOOP ECOSYSTEM

During the time period in which 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 using a procedural interface via Map and Reduce functions. At the other extreme was Hive [45, 2] which provided a declarative SQL interface[4]. There were several popular interfaces in between these two extremes, such as Pig [39], SCOPE [26], 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 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 a performance advantage over the state of the art was by (optionally) storing structured data in columnar format and providing a query execution framework that was able to take advantage of columnar storage by keeping data in columnar form during certain query operators. It is well known that column-oriented storage and execution can improve performance of many classes of analytical query workloads—especially those 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 relational and nested data.

Parquet and ORCFile use PAX blocks [20] for columnar storage. In PAX, data is kept in columns within blocks, but a given block 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 in the same block. On the other hand, PAX reduces scan performance compared to pure column stores since not all data for a given column is placed contiguously on disk.

Parquet and ORCFile are not query execution engines. Rather, they are file formats that enable column-oriented storage and compression. In order to get the equivalent benefit that HadoopDB was able to achieve when leveraging an underlying column-oriented DBMS, Parquet and ORCFile need to be combined with a query execution engine that can leverage column-oriented storage. Indeed, Parquet and ORCFile were introduced exactly for this purpose: so that a new wave of SQL engines for Hadoop could benefit from column-oriented storage and processing just as HadoopDB had done in 2009.

[4]Hive converted all SQL expressions to Map and Reduce functions under the covers.

This next generation of Hadoop-based query engines used many of the ideas from HadoopDB in the way they were architected using 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 subsequent to the original HadoopDB paper.

As mentioned above, **Hive** existed solely as a language-based hybrid at the time the HadoopDB paper was published. Subsequently, its community transformed the project into a more systems-level hybrid. First, Hive evolved to support pluggable execution engines and proposed Apache Tez [41] as an alternative to MapReduce. Tez, which is similar to Dryad [35], 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 [48] is a distributed general-purpose processing engine comparable to Tez and Dryad in the way it expands data processing operators beyond 'Map' and 'Reduce'. Spark maintained the MapReduce system capability of providing fault tolerance during query processing, but provides users with more control in specifying the required level of fault tolerance. Compared to MapReduce, Spark achieves significant speedup for iterative data processing workloads and therefore became popular for machine learning, graph analytics, and ETL use cases. Spark supports data analytics via a variety of end user interfaces such as Scala, Python, and R. Analogously to how Hive and HadoopDB brought SQL to Hadoop, the Shark project [46] brought SQL to Spark. By using Spark instead of MapReduce, Shark was able to achieve higher performance than the original MapReduce-based version of Hive. In 2014, Shark was replaced by SparkSQL [21]. 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.

Flink is similar to Spark, but is less connected to the Hadoop ecosystem and more commonly used for continuous streaming use cases relative to Spark [25].

Drill [1] is a distributed SQL engine inspired by Google Dremel [38]. The project was created by MapR (now part of HPE) 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 [36] and **HAWQ** [27] went a step farther than HadoopDB. Like HadoopDB, they include specialized single-node query execution engines on each node in a Hadoop cluster, and execute query operators in parallel across these engines. However, in HadoopDB, only the lower parts of a query plan were pushed down into the single-node database systems, while communication across nodes was managed using Hadoop's MapReduce framework. By contrast, Im-

pala and Hawq built full parallel database execution engines within a Hadoop cluster, thereby allowing entire query plans to avoid MapReduce or the other execution frameworks available in Hadoop. This resulted in the query processing engine losing some of the fault tolerance, heterogeneous cluster tolerance, and thus scalability advantages of HadoopDB. Nonetheless, Impala and HAWQ were able to achieve low latency queries via fully pipelined relational operators and native integration with HDFS, which reduces the need for mid-query fault tolerance.

Presto [42] is another full parallel database execution engine that avoids the need for other Hadoop-based generic data processing engines during query processing. 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 an efficient execution engine employing vectorized columnar data processing, runtime query bytecode compilation, optimized data structures, and multi-threaded processing that leverages multi-core CPUs efficiently.

Presto includes native abilities to query multiple data sources including Object Stores, HDFS, NoSQL systems and RDBMSs. Presto's 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 Presto's development community is the ex-Hadapt team at Starburst Data [10] (see Section 4) that recently contributed a Cost-Based Optimizer [8].

By being complete implementations of parallel execution engines, Impala, HAWQ, and Presto are somewhat independent systems that integrate with Hadoop mostly at the storage level (although Impala and HAWQ both also integrate with Hadoop's resource management tools). To that end, they provide similar (albeit more native) functionality to a large number of commercial parallel database systems that have "connectors" to Hadoop that enable them to read data from HDFS. These systems thus serve as a SQL interface to data stored in HDFS, and use their own execution engines for non-trivial parts of query processing. Almost every parallel DBMS vendor now has a Hadoop connector. Some implementations that are more closely integrated with Hadoop include **Oracle Big Data SQL, Redshift Spectrum, Teradata QueryGrid, Vertica on Hadoop, and IBM Big SQL**.

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" [31]. While unified frameworks like Presto and central metadata repositories like Hive Metastore help ameliorate some of these issues, several end-user problems require novel solutions. These include (i) discoverability: allowing analysts to determine whether certain data exists and, if so, and how it is processed, (ii) minimizing wasted and duplicated effort involved in data cleaning, pre-processing, and data analysis, as well as sharing 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 data lake.

Much effort has been spent by Hadoop distributors and members of the ecosystem to integrate HadoopDB's performance improvements into the thriving current ecosystem of query processing tools that run on Hadoop. However, many of HadoopDB's usability innovations—such as incremental movement of data from HDFS into file formats that are optimized for query processing, as pioneered by the invisible loading [19], Sinew [44], and schema generation [30] projects—remain in their infancy in the Hadoop ecosystem. As Hadoop crosses the chasm of wide-spread adoption, we expect usability will become the next front of rapid innovation and competitive differentiation.

7. ACKNOWLEDGEMENTS

This work was sponsored by the National Science Foundation under grants IIS-1718581 and IIS-1910613. We thank Alessandro Andrioni, Milind Bhandarkar, Jorn Franke, Alan Gates, Marty Gubar, Christian Jensen, Julien Le Dem, Mark Lyons, Frank McSherry, and Carl Steinbach for their comments on earlier drafts of this paper.

8. REFERENCES

[1] Apache Drill. `https://drill.apache.org`.
[2] Apache Hive. `https://hive.apache.org/`.
[3] Apache ORC. `https://orc.apache.org`.
[4] Apache Parquet. `https://parquet.apache.org`.
[5] Apache SOLR. `https://lucene.apache.org/solr/`.
[6] Delta Lake. `https://delta.io/`.
[7] Hadapt Crunchbase profile. `https://www.crunchbase.com/organization/hadapt`.
[8] Introduction to Presto Cost-Based Optimizer. Presto Blog `https://prestosql.io/blog/2019/07/04/cbo-introduction.html`.
[9] Presto - Next Chapter. Starburst Blog `https://www.starburstdata.com/technical-blog/2017-12-13-presto-next-chapter/`.
[10] Starburst Data. `https://www.starburstdata.com`.
[11] Teradata Bets Big on Presto for Hadoop SQL. Datanami `https://www.datanami.com/2015/06/08/teradata-bets-big-on-presto-for-hadoop-sql/`.
[12] D. Abadi and A. Abouzied. Data loading systems and methods. US Patent 9336263, 2011.
[13] D. Abadi and K. Bajda-Pawlikowski. Query execution systems and methods. US Patent 8935232, 2011.
[14] D. Abadi, K. Bajda-Pawlikowski, A. Abouzied, and A. Silberschatz. Processing of data using a database system in communication with a data processing framework. US Patent 9495427, 2011.
[15] D. Abadi, K. Bajda-Pawlikowski, R. Schlussel, and P. Wickline. Systems and methods for fault tolerant, adaptive execution of arbitrary queries at low latency. US Patent 9934276, 2013.
[16] D. Abadi and J. Huang. Query execution systems and methods. US Patent 8886631, 2012.
[17] D. Abadi, D. Tahara, and T. Diamond. Schema-less access to stored data. US Patent 9471711, 2013.
[18] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS

technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[19] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. EDBT '13, pages 1–10, 2013.

[20] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. VLDB '01, pages 169–180, 2001.

[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. SIGMOD '15, pages 1383–1394, 2015.

[22] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. SIGMOD '11, pages 1165–1176, 2011.

[23] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. SIGMOD '18, pages 221–230, 2018.

[24] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, S. Shelukhin, P. Jayachandran, S. Seth, D. Jaiswal, S. Bouguerra, N. Bangarwa, S. Hariappan, A. Agarwal, J. Dere, D. Dai, T. Nair, N. Dembla, G. Vijayaraghavan, and G. Hagleitner. Apache hive: From mapreduce to enterprise-grade big data warehousing. SIGMOD '19, pages 1773–1786, 2019.

[25] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[26] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[27] L. Chang, Z. Wang, T. Ma, L. Jian, L. Ma, A. Goldshuv, L. Lonergan, J. Cohen, C. Welton, G. Sherry, and M. Bhandarkar. Hawq: A massively parallel processing sql engine in hadoop. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1223–1234, 2014.

[28] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI'04, pages 137–150, San Francisco, CA, 2004.

[29] D. J. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. The Database Column, 2008.

[30] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. SIGMOD '16, pages 295–310, 2016.

[31] J. Hellerstein, J. Gonzalez, V. Sreekanti, et al. Ground: A data context service. In *Proceedings of the 2017 Conference on Innovative Data Systems Research*, CIDR '17, 2017.

[32] J. Huang and D. J. Abadi. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *PVLDB*, 9(7):540–551, 2016.

[33] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(21):1123–1134, 2011.

[34] J. Huang, K. Venkatraman, and D. J. Abadi. Query optimization of distributed pattern matching. In *ICDE*, 2014.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. EuroSys '07, 2007.

[36] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[37] A. Maccioni and D. J. Abadi. Scalable pattern matching over compressed graphs via dedensification. KDD '16, pages 1755–1764, 2016.

[38] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.

[39] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. SIGMOD '08, pages 1099–1110, 2008.

[40] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. SIGMOD '09, pages 165–178, 2009.

[41] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. SIGMOD '15, pages 1357–1369, 2015.

[42] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: Sql on everything. ICDE '19, pages 1802–1813, 2019.

[43] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: Friends or foes? *Commun. ACM*, 53(1):64–71, Jan. 2010.

[44] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: A sql system for multi-structured data. SIGMOD '14, pages 815–826, 2014.

[45] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[46] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. SIGMOD '13, pages 13–24, 2013.

[47] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. OSDI'08, pages 1–14, 2008.

[48] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.