Palgrave Studies in Democracy, Innovation and Enterpreneurship for Growth

Analytics, Innovation and Excellence-Driven Enterprise Sustainability

Business Intelligence and Analytics: Big Systems for Big Data

Herodotos Herodotou

Cyprus University of Technology

herodotos.herodotou@cut.ac.cy

2016

Abstract

The amount of data collected by modern industrial, government, and academic organizations has been increasing exponentially and will continue to grow at an accelerating rate for the foreseeable future. At companies across all industries, servers are overflowing with usage logs, message streams, transaction records, sensor data, business operations records and mobile device data. Effectively analyzing these massive collections of data ("Big Data") can create significant value for the world economy by enhancing productivity, increasing efficiency, and delivering more value to consumers.

The need to convert raw data into useful information has led to the development of advanced and unique data storage, management, analysis, and visualization technologies, especially over the last decade. This monograph is an attempt to cover the design principles and core features of systems for analyzing very large datasets for business purposes. In particular, we organize systems into four main categories based on major and distinctive technological innovations. *Parallel databases* dating back to 1980s have added techniques like columnar data storage and processing, while new distributed platforms like *MapReduce* have been developed. Other innovations aimed at creating alternative system architectures for more generalized *dataflow* applications. Finally, the growing demand for *interactive analytics* has led to the emergence of a new class of systems that combine analytical and transactional capabilities.

Contents

1 Introduction

Modern industrial, government, and academic organizations are collecting massive amounts of data ("big data") at an unprecedented scale and pace. Many enterprises continuously collect records of customer interactions, product sales, results from advertising campaigns on the Web, and other types of information. Powerful telescopes in astronomy, particle accelerators in physics, and genome sequencers in biology are putting massive volumes of data into the hands of scientists (Cohen et al., 2009; Thusoo et al., 2009). The ability to perform timely and cost-effective analytical processing of such large datasets to extract *deep insights* is now a key ingredient for success. These insights can drive automated processes for advertisement placement, improve customer relationship management, and lead to major scientific breakthroughs (Frankel and Reid, 2008).

The set of techniques, systems, and tools that transform raw data into meaningful and useful information for business analysis purposes is collectively known as *Business Intelligence* (*BI*) (Chen et al., 2012). In addition to the underlying data processing and analytical techniques, BI includes business-centric practices and methodologies that can be applied to various high-impact applications such as e-commerce, market intelligence, healthcare, and security. The more recent explosion of data has lead to the development of advanced and unique data storage, management, analysis, and visualization technologies—termed *big data analytics*—in order to serve applications that are so large (from terabytes to exabytes) and complex (from sensor to social media data) that could not be served effectively with the previous technologies. Big data analytics can give organizations an edge over their rivals and lead to business rewards, including more potent promotion and enhanced revenue.

Existing database systems are adapting to the new status quo while large-scale data analytics systems, like MapReduce (Dean and Ghemawat, 2008) and Dryad (Isard et al., 2007), are becoming popular for analytical workloads on big data. Industry leaders such as Teradata, SAP, Oracle and EMC/Greenplum have addressed this explosion of data volumes by leveraging more powerful and parallel hardware in combination with sophisticated parallelization techniques in the underlying data management software. Internet services companies such as Twitter, LinkedIn, Facebook, Google, and others address the scalability challenge by leveraging a combination of new technologies in their clusters: key-value stores, columnar storage, and the MapReduce programming paradigm (Wu et al., 2012; Thusoo et al., 2010; Lee et al., 2012; Melnik et al., 2010). Finally, small and medium enterprises are slowly adopting the new technologies to satisfy their needs for identifying, developing and otherwise creating new strategic business opportunities.

This monograph is an attempt to cover the design principles and core features of systems for analyzing very large datasets for business purposes. We organize systems into four main categories—*Parallel Databases*, *MapReduce*, *Dataflow*, and *Interactive*

Analytics—each with multiple subcategories, based on some major and distinctive technological innovations. The categories loosely correspond to the chronological evolution of systems as the requirements for large-scale analytics have evolved over the last few decades. Table 1 lists all categories and subcategories we discuss along with some example systems for each subcategory.

1.1 Evolution of Data Analytics Systems

The need for improvements in productivity and decision making processes has led to considerable innovation in systems for large-scale data analytics. *Parallel databases* dating back to 1980s have added techniques like columnar data storage and processing (Boncz et al., 2006; Lamb et al., 2012), while new distributed platforms like *MapReduce* (Dean and Ghemawat, 2008) have been developed. Other innovations aimed at creating alternative system architectures for more generalized *dataflow* applications, including Dryad (Isard et al., 2007) and Stratosphere (Alexandrov et al., 2014). More recently, the growing demand for *interactive analytics* has led to the emergence of a new class of systems, like SAP HANA (Färber et al., 2012) and Spanner (Corbett et al., 2012), combine analytical and transactional capabilities.

Parallel Database Systems: *Row-based parallel databases* were the first systems to make parallel data processing available to a wide class of users through an intuitive high-level programming model, namely SQL. High performance and scalability were achieved through partitioning tables across the nodes in a shared-nothing cluster. Such a horizontal partitioning scheme enabled relational operations like filters, joins, and aggregations to be run in parallel over different partitions of each table stored on different nodes. On the other hand, *columnar databases* pioneered the concept of storing data tables as sections of columns rather than rows and performing vertical partitioning. Systems with columnar storage and processing have been shown to use CPU, memory, and I/O resources more efficiently in large-scale data analytics compared to row-oriented systems (Lamb et al., 2012). Some of the main benefits come from reduced I/O in columnar systems by (a) reading only the needed columns during query processing and (b) offering better compression. Row-based and columnar systems are discussed in Section 2.

MapReduce Systems: MapReduce is a programming model and an associated implementation developed by Google for processing massive datasets on large clusters of thousands of commodity servers (Dean and Ghemawat, 2008). Parallel databases have traditionally struggled to scale to such levels. MapReduce systems pioneered the concept of building multiple standalone scalable distributed systems and then composing two or more of these systems together in order to run analytic tasks on large datasets. Typical MapReduce systems such as Hadoop (White, 2010) store data in a standalone block-oriented *distributed file system* and run computational tasks in a *MapReduce execution engine*. The MapReduce model, although highly flexible, has been found to be too low-level for routine use by practitioners such as data analysts, statisticians, and scientists (Olston et al., 2008; Thusoo et al., 2009). As a result, the MapReduce framework has evolved rapidly over the past few years into a *MapReduce stack* that includes a number of higher-level layers added over the core MapReduce engine. Prominent examples of these higher-level layers include Hive (with an SQL-like declarative interface), Pig (with an interface that mixes declarative and procedural elements), Cascading (with a Java interface for specifying workflows), Cascalog (with a Datalog-inspired interface), and BigSheets (includes a spreadsheet interface). MapReduce systems are covered in Section 3.

Dataflow Systems: As MapReduce systems were being adopted for a large number of data analysis tasks, a number of shortcomings became apparent. The MapReduce programming model is too restrictive to express certain data analysis tasks easily, e.g., joining two datasets together. More importantly, the execution techniques used by MapReduce systems are suboptimal for many common types of data analysis tasks such as relational operations, iterative machine learning, and graph processing. Some of these problems have been addressed by replacing MapReduce with a more *generalized MapReduce* execution model that contains extra operators in addition to Map and Reduce (e.g., Hyracks (Borkar et al., 2011), Nephele (Battré et al., 2010)). A different class of dataflow systems such as Dryad (Isard et al., 2007) and Spark (Zaharia et al., 2012) use the *directed acyclic graph* model that can express a wide range of data access and communication patterns. Finally, *graph processing* systems like Pregel (Malewicz et al., 2010) are specialized in running iterative computations and other analytics tasks over data graphs. Dataflow systems are described in Section 4.

Systems for Interactive Analytics: The need to reduce the gap between the generation of data and the generation of analytics results over this data has required system developers to constantly raise the bar in large-scale data analytics. On one hand, this need has led to the emergence of scalable distributed storage and compute systems that support *mixed analytical and transactional* workloads, such as Spanner (Corbett et al., 2012) and Megastore (Baker et al., 2011). Support for transactions enables storage systems in particular to serve as the data store for online services while making the data available concurrently in the same system for analytics. The same need led to the emergence of *distributed SQL query engines* that run over distributed file systems and support ad-hoc analytics. For instance, Cloudera Impala (Wanderman-Milne and Li, 2014) enables users to issue low-latency SQL queries to data stored in HDFS (Shvachko et al., 2010) and Apache HBase (George, 2011) without requiring data movement or transformations. Finally, *stream processing systems* are driven by a data-centric model that allows for near real-time consumption and analysis of data. We discuss systems for interactive analytics in Section 5.

(Sub)Category	Example Systems
Parallel Databases	
Row-based Parallel Databases	Aster nCluster, DB2 Parallel Edition, Greenplum, Netezza, Teradata
Columnar Databases	C-Store, Infobright, MonetDB, ParAccel, Sybase IQ, VectorWise, Vertica
MapReduce	
Distributed File Systems	Ceph, GFS, HDFS, Kosmos, MapR, Quantcast
MapReduce Execution Engines	Google MapReduce, Hadoop, HadoopDB, Hadoop++
MapReduce-based Platforms	Cascading, Clydesdale, Hive, Jaql, Pig
Dataflow	
Generalized MapReduce	ASTERIX, Hyracks, Nephele, Stratosphere
Directed Acyclic Graph Systems	Dryad, DryadLINQ, SCOPE, Shark, Spark
Graph Processing Systems	GraphLab, GraphX, HaLoop, Pregel, PrIter, Twister
Interactive Analytics	
Mixed Analytical and Transactional	Bigtable, HBase, HyPer, HYRISE, Megastore, SAP HANA, Spanner
Distributed SQL Query Engines	Apache Drill, Cloudera Impala, Dremel, Presto, Stinger.next
Stream Processing Systems	Aurora, Borealis, Muppet, S4, Storm, STREAM

Table 1: The system categories, subcategories, and example systems (in alphabetical order) for large-scale data analytics.

2 Parallel Database Systems

Traditionally, Enterprise Data Warehouses (EDWs) and Business Intelligence (BI) tools built on top of database systems have been providing the means for retrieving and analyzing large amounts of data. In this monograph, we focus on Massive Parallel Processing (MPP) Database Management Systems (DBMSs) that run on clusters of commodity servers and provide support for big data analytics. As these systems were developed based on centralized DBMSs, they use the Structured Query Language (SQL) for accessing, managing, and analyzing data. Users can specify an analysis task using a SQL query, while the DBMS will optimize and execute the query.

In addition, database systems require that data conforms to a well-defined schema and is stored in a specialized data store. The storage format is the main differentiator between the two categories of parallel database systems we consider, namely row-oriented and column-oriented systems. For both categories, we concentrate on the technological innovations that differentiate them from earlier centralized database systems and from each other.

2.1 Row-based Parallel Databases

A number of research prototypes and industry-strength parallel database systems have been built using a shared-nothing architecture over the last three decades. Examples include *Gamma* (DeWitt et al., 1990), *Pivotal Greenplum Database* (Greenplum, 2013), *IBM DB2 Parallel Edition* (Baru et al., 1995), *Netezza* (IBM Netezza, 2012), and *Teradata* (Teradata, 2012). Given the parallel nature of the aforementioned systems, we focus primarily on two key system aspects: (i) parallel data storage and (ii) parallel query execution.

Parallel Data Storage: The relational data model and SQL query language have the crucial benefit of data independence; that is, SQL queries can be executed correctly irrespective of how the data in the tables is physically stored in the system. There are four noteworthy aspects of physical data storage in parallel databases: (a) *partitioning*, (b) *declustering*, (c) *collocation*, and (d) *replication*.

Table partitioning refers to the technique of distributing the tuples of a table across disjoint fragments (or, partitions) and is a standard feature in parallel database systems today (IBM Corporation, 2007; Morales, 2007; Talmage, 2009). The most common types of partitioning are:

- **Range partitioning**, where tuples are assigned to tables based on value ranges of one or more attributes.

- **Hash partitioning**, where tuple assignment is based on the result of a hash function applied to one or more attributes.
- **List partitioning**, where the unique values of one or more attributes in each partition are specified.
- **Random partitioning**, where tuples are assigned to partitions in a random fashion.
- **Round-robin partitioning**, where tuples are assigned to partitions in a round-robin fashion.
- **Block partitioning**, where each consecutive block of tuples (or bytes) written to a table forms a partition.

Benefits of partitioning range from more efficient loading and removal of data on a partition-by-partition basis to finer control over the choice of physical design, statistics creation, and storage provisioning based on the workload. Deciding how to partition tables, however, is now an involved process where multiple objectives—e.g., getting fast data loading along with good query performance—and constraints—e.g., on the maximum size or number of partitions per table—may need to be met (Herodotou et al., 2011). Various table partitioning schemes as well as techniques to find a good partitioning scheme automatically have been proposed as part of database physical design tuning (Agrawal et al., 2004; Rao et al., 2002).

The next task after table partitioning is deciding which node or nodes in the cluster should store each partition of the tables in the database. The number of nodes across which a table is distributed is called the *degree of declustering.* When that number equals to the number of nodes in the system, the table is said to be fully declustered; otherwise it is partially declustered (DeWitt et al., 1990). With partial declustering, nodes are typically grouped in sets—called nodegroups (Baru et al., 1995) or relation clusters (Hsiao and DeWitt, 1990)—that can be referenced by name. Each table is then assigned to one such group. Note that it is possible to have multiple tables assigned to the same group but one table cannot be assigned to multiple groups.

Having selective overlap among the nodes (or the group) on which the partitions of two or more tables are stored can be beneficial, especially for join processing. Consider two tables $R(a, b)$ and $S(a, c)$, where a is a common attribute. Suppose both tables are hash partitioned on the respective attribute a using the same hash function and the same number of partitions. Further, suppose the partitions of tables R and S are both stored on the same group of nodes. In this case, there will be a one-to-one correspondence between the partitions of both tables that can join with one another on attribute a. That is, any pair of joining partitions will be stored on the same node of the group. Under these conditions, the two tables R and S are said to be *collocated*. The advantage of collocation is that tables can be joined without the need to move any data from one node to another.

In addition to collocation, *data replication* can often provide performance benefits, both for join processing and for the concurrent execution of multiple queries. Replication is usually done at the table level in two scenarios. When a table is small, it can be replicated on all nodes in the cluster or a group. Such replication is common for dimension tables in star and snowflake schemas so that they can easily join with the partitions of the distributed fact table(s). Replication can also be done such that different replicas are partitioned differently. For example, one replica of the table may be hash partitioned while another may be range partitioned for speeding up multiple workloads with different access and join patterns. Apart from performance benefits, replication also helps reduce unavailability or loss of data when faults arise in the parallel database system (e.g., a node fails permanently or becomes disconnected temporarily from other nodes due to a network failure).

The diverse mix of partitioning, declustering, collocation, and replication techniques available can make it confusing for users of parallel database systems to identify the best data layout for their workload. This problem has motivated research on automated ways to recommend good data layouts based on the workload (Mehta and DeWitt, 1997; Rao et al., 2002) and on partition-aware optimization techniques to generate efficient plans for SQL queries over partitioned tables (Herodotou et al., 2011).

Parallel Query Execution: When a SQL query is submitted to the database system, the query optimizer is responsible for generating a parallel execution plan for the query. The plan is composed of operators that support both intra- and inter-operator parallelism, as well as mechanisms to transfer data from producer operators to consumer operators. The plan is broken down into schedulable tasks that are run on the nodes in the system. Upon completion of the plan, the results are transferred back to the user or application that submitted the query.

Parallel database systems employ multiple forms of parallelism in execution plans, including *join*, *partitioned*, *pipelined*, and *independent* parallelism. Join parallelism refers to the type of join used to execute table joins and depends primarily on the partitioning, declustering, collocation, and replication techniques used for storing the data. We discuss four main join types (illustrated in Figure 1) for joining two tables R and S based on the equi-join condition $R.a = S.a$.

- **Collocated join:** A collocated join can be used only when tables R and S are both partitioned on attribute a and the partitions are assigned such that any pair of joining partitions is stored on the same node. A collocated join operator

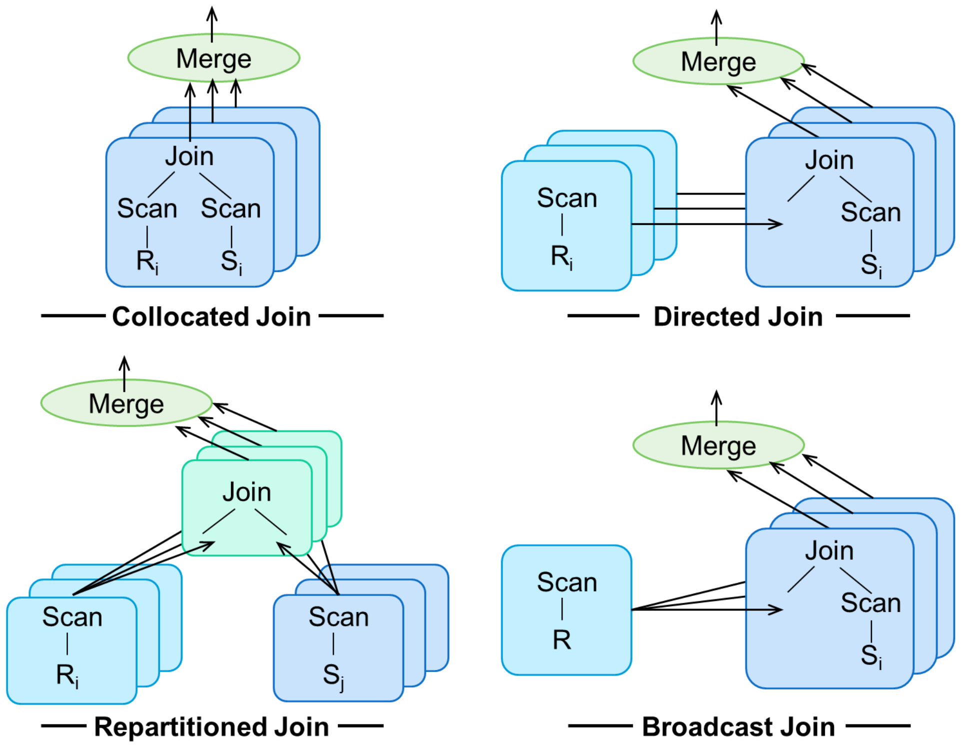

Figure 1: Parallel join types.

is often the most efficient way to perform the join because it performs the join in parallel on each node while avoiding the need to transfer data between nodes.

- **Directed join:** Suppose tables R and S are both partitioned on attribute a but the respective partitions are not collocated. In this case, a directed join can transfer each partition of one table (say, R) to the node where the joining partition of the other table is stored. Once a partition from R is brought to where the joining partition in S is stored, a local join can be performed. Compared to a collocated join, a directed join incurs the cost of transferring one of the tables across the network.

- **Repartitioned join:** If tables R and S are not partitioned on the joining attribute, then the repartitioned join is used. This join simply repartitions the tuples in both tables using the same partitioning condition (e.g., hash). Joining partitions are brought to the same node where they can be joined. This operator incurs the cost of transferring both tables across the network.

- **Broadcast join:** When tables R and S are not partitioned on the joining attribute but one of them (say, R) is very small, then the broadcast join will transfers R in full to every node where any partition of the other table (S) is stored. The join is then performed locally. This operator incurs a data transfer cost equal to the size of R times the degree of declustering of S.

A typical issue with join processing is the presence of skew in partition sizes. Hash or range partitioning can produce skewed partition sizes if the attribute used in the partitioning function has a skewed distribution. The load imbalance created by such skew can severely degrade the performance of join operators such as the repartitioned join. This problem can be addressed by identifying the skewed join keys and handling them in special ways. In particular, tuples in a table with a join key value v that has a skewed distribution can be further partitioned across multiple nodes. The correct join result will be produced as long as the tuples in the joining table with join key equal to v are replicated across the same nodes. In this fashion, the resources in multiple nodes can be used to process the skewed join keys (DeWitt et al., 1992).

While our discussion focused on the parallel execution of joins, the same principles apply to the parallel execution of other relational operators like filtering and group-by. The unique approach used here to extract parallelism is to partition the input into multiple fragments, and to process these fragments in parallel. This form of parallelism is called *partitioned parallelism* (DeWitt and Gray, 1992).

Another form of parallelism employed commonly in execution plans in parallel database systems is the *pipelined parallelism.* A query execution plan may contain a sequence of operators linked together by producer-consumer relationships where all operators can be run in parallel as data flows continuously across every producer-consumer pair. For example, suppose an execution plan contains three operators: a table scan S, a filter F, and a hash aggregator H. S starts scanning the table and places the tuples in F's input queue. At the same time, F reads from its input queue, performs the filtering, and writes to H's input queue. Finally, H starts building the hash table. Thus, S, F, and H can be working concurrently on stages from different iterations, thereby increasing performance.

Finally, *independent parallelism* refers to the parallel execution of independent operators in a query plan. For example, consider a query that joins together four tables R, S, T, and U. This query can be processed by an execution plan where R is joined with S, T is joined with U, and then the results from both joins are joined together to produce the final result $[(R \bowtie S) \bowtie (T \bowtie U)]$. In this plan, $R \bowtie S$ and $T \bowtie U$ can be executed independently in parallel.

2.2 Columnar Databases

Columnar systems excel at data-warehousing-type applications, where (i) data is loaded in bulk but typically not modified much and (ii) the typical access pattern is to scan through large parts of the data to perform aggregations and joins. The first columnar database systems that appeared in the 1990s were *MonetDB* (Boncz et al., 2006) and *Sybase IQ* (MacNicol and French, 2004). The 2000s saw a number of new columnar database systems such as *C-Store* (Stonebraker et al., 2005), *Infobright* (Infobright, 2013), *ParAccel* (ParAccel, 2013), *VectorWise* (Zukowski and Boncz, 2012), and *Vertica* (Lamb et al., 2012). Similar to the row-based databases discussed above, we focus on the data storage and query execution of columnar database systems.

Columnar Data Storage: In a pure columnar data layout, each table column is stored contiguously in a separate file on disk. Each file stores tuples of the form $\langle k, v \rangle$ (Boncz et al., 2006), where the key k is the unique identifier for a tuple and v is the corresponding value. An entire tuple with tuple identifier k can be reconstructed by bringing together all the attribute values stored for k. It is also possible to eliminate the explicit storage of tuple identifiers and derive them implicitly based on the position of each attribute value in the file (Lamb et al., 2012; Stonebraker et al., 2005).

Vertica stores two files per column (Lamb et al., 2012). One file contains the attribute values while the other file, called *position index*, stores corresponding metadata such as the start position, minimum value, and maximum value for the attribute values. The position index helps with tuple reconstruction as well as eliminating reads of disk blocks during query processing. Furthermore, removing the storage of tuple identifiers leads to more densely packed columnar storage (Abadi et al., 2009; Lamb et al., 2012).

C-Store introduced the concept of *projections*. A projection is a set of columns that are stored together. The concept is similar to a materialized view that projects some columns of a base table. However, in C-Store, all the data in a table is stored as one or more projections. That is, C-Store does not have an explicit differentiation between base tables and materialized views. Each projection is stored sorted on one or more attributes. Vertica implemented a similar concept later—called *super projections*—that contains every column of the table (Lamb et al., 2012).

An important advantage of columnar data layouts is that columns can be stored densely on disk using various *compression techniques* (Abadi et al., 2009; Lamb et al., 2012; Stonebraker et al., 2005):

- **Run Length Encoding (RLE):** Sequences of identical values in a column are replaced with a single pair that contains the value and number of occurrences. This type of compression is best for sorted, low cardinality columns.
- **Delta Value:** Each attribute value is stored as the difference from the smallest value, so it is useful when the differences can be stored in fewer bytes than the original attribute values. This type of compression is best for many-valued, unsorted integer or integer-based columns.
- **Compressed Delta Range:** Each value is stored as a delta from the previous one. This type of compression is best for many-valued float columns that are either sorted or confined to a range.

- **Dictionary:** The distinct values in the column are stored in a dictionary which assigns a short code to each distinct value. The actual values are replaced with the code assigned by the dictionary. Dictionary-based compression is a general-purpose scheme, but it is good for unsorted, low cardinality columns.

- **Bitmap:** A column is represented by a sequence of tuples $\langle v, b \rangle$ such that v is a value stored in the column and b is a bitmap indicating the positions in which the value is stored. RLE can be further applied to compress each bitmap.

Hybrid combinations of the above schemes are also possible. For example, the *Compressed Common Delta* scheme used in Vertica builds a dictionary of all the deltas in each block (Lamb et al., 2012). This type is best for sorted data with predictable sequences and occasional sequence breaks (e.g., timestamps recorded at periodic intervals or primary keys).

Columnar Query Execution: The columnar data layout gives rise to a distinct space of execution plans in columnar parallel database systems that provide opportunities for highly efficient execution: (a) *operations on compressed columns*, (b) *vectorized operations*, and (c) *late materialization.*

Given the typical use of compression in columnar systems, it is highly desirable to have (some) operators operate on the compressed representation of their input whenever possible, in order to avoid the cost of decompression. The ability to operate directly on compressed data depends on the type of the operator and the compression scheme used. For example, consider a filter operator whose filter predicate is on a column compressed using the Bitmap compression technique. This operator can do its processing directly on the stored unique values of the column and then only read those bitmaps from disk whose values match the filter predicate. Complex operators like range filters, aggregations, and joins can also operate directly on compressed data.

Columnar layouts encourage *vectorized processing* since it is more efficient for operators to process their input in large chunks at a time as opposed to one tuple at a time. A full or partial column of values can be treated as an array (or, a vector) on which SIMD (single instruction multiple data) instructions in CPUs can be evaluated. SIMD instructions can greatly increase performance when the same operations have to be performed on multiple data objects. The *X100* project (which was commercialized later as VectorWise) explored a compromise between the classic tuple-at-a-time pipelining and operator-at-a-time bulk processing techniques (Boncz et al., 2005). X100 operates on chunks of data that are large enough to amortize function call overheads, but small enough to fit in CPU caches and to avoid materialization of large intermediate results into main memory. X100 shows significant performance benefits when vectorized processing is combined with just-in-time light-weight compression.

Tuple reconstruction is expensive in columnar database systems since information about a tuple is stored in multiple locations on disk, yet most queries access more than one attribute from a tuple (Abadi et al., 2009). Further, most users and applications (e.g., using ODBC or JDBC) access query results tuple-at-a-time (not column-at-a-time). Thus, at some point in a query plan, data from multiple columns must be *materialized* as tuples. Many techniques have been developed to reduce such tuple reconstruction costs (Abadi et al., 2007). For example, MonetDB uses late tuple reconstruction (Idreos et al., 2012). All intermediate results are kept in a columnar format during the entire query evaluation. Tuples are constructed only just before sending the final result to the user or application. This approach allows the query execution engine to exploit CPU-optimized and cache-optimized vector-like operator implementations throughout the whole query evaluation. One disadvantage of this approach is that larger intermediate results may need to be materialized compared to the traditional tuple-at-a-time processing.

3 MapReduce Systems

MapReduce is both a programming model and an associated run-time system for large-scale data processing (Dean and Ghemawat, 2008). *Hadoop* is the most popular open-source implementation of a MapReduce framework that follows the design laid out in the original paper (Dean and Ghemawat, 2004). A number of companies use Hadoop in production deployments for applications such as Web indexing, data mining, report generation, log file analysis, machine learning, financial analysis, scientific simulation, and bioinformatics research. Infrastructure-as-a-Service cloud platforms like Amazon and Rackspace have made it easier than ever to run Hadoop workloads by allowing users to instantly provision clusters and pay only for the time and resources used.

A combination of features contributes to Hadoop's increasing popularity, including fault tolerance, data-local scheduling, ability to operate in a heterogeneous environment, handling of straggler tasks[1], as well as a modular and customizable architecture. In typical Hadoop deployments, data is stored in a block-oriented *distributed file system* (usually HDFS) and processed using either the *Hadoop MapReduce execution engine* directly or one of the many *MapReduce-based platforms* built on top of Hadoop (e.g., Hive, Pig, Jaql). The Hadoop ecosystem is shown in Figure 2.

3.1 Distributed Storage

The storage layer of a typical MapReduce cluster is an independent distributed file system. Typical Hadoop deployments use the *Hadoop Distributed File System (HDFS)* running on the cluster's compute nodes (Shvachko et al., 2010). Alternatively, a Hadoop cluster can process data from other file systems like the *MapR File System* (MapR, 2013), *Ceph* (Weil et al., 2006), *Amazon Simple Storage Service (S3)* (Amazon S3, 2013), and *Windows Azure Blob Storage* (Calder et al., 2011).

As HDFS focuses more on batch processing rather than interactive use, it emphasizes high throughput of data access rather than low latency. An HDFS cluster employs a master-slave architecture consisting of a single *NameNode* (the master) and multiple *DataNodes* (the slaves), usually one per node in the cluster (see Figure 3). The NameNode manages the file system namespace and regulates access to files by clients, whereas the DataNodes are responsible for serving read and write requests from the file system's clients. HDFS is designed to reliably store very large files across machines in a large cluster. Internally, a file is split into one or more blocks that are replicated for fault tolerance and stored in a set of DataNodes.

A number of other distributed file systems are viable alternatives to HDFS and offer full compatibility with Hadoop MapReduce. The *MapR File System* (MapR,

[1]A straggler is a task that performs poorly typically due to faulty hardware or misconfiguration.

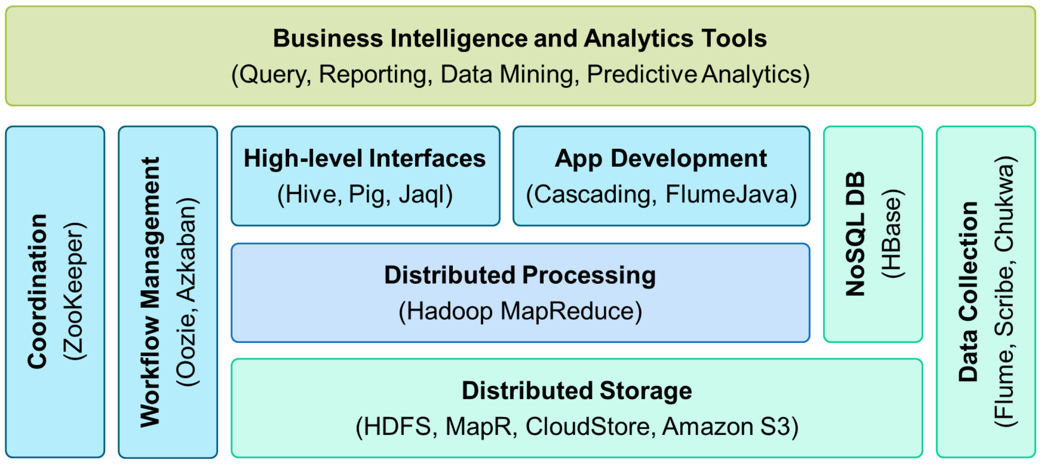

Figure 2: Hadoop ecosystem for big data analytics.

2013) and *Ceph* (Weil et al., 2006) have similar architectures to HDFS but both offer a distributed metadata service as opposed to the centralized NameNode on HDFS. In MapR, metadata is sharded across the cluster and collocated with the data blocks, whereas Ceph uses dedicated metadata servers with dynamic subtree partitioning to avoid metadata access hot spots. The *Quantcast File System (QFS)* (Ovsiannikov et al., 2013), which evolved from the *Kosmos File System (KFS)* (KFS, 2013), employs erasure coding rather than replication as its fault tolerance mechanism. Erasure coding enables QFS to not only reduce the amount of storage but to also accelerate large sequential write patterns common to MapReduce workloads.

Distributed file systems are primarily designed for accessing raw files and, therefore, lack any advanced features found in the storage layer of database systems. This limitation has inspired a significant amount of research for introducing (i) indexing, (ii) collocation, and (iii) columnar capabilities into such file systems.

Indexing: *Hadoop++* (Dittrich et al., 2010) provides indexing functionality for data stored in HDFS using the so-called *Trojan Indexes*. The indexing information is created during the initial loading of data onto HDFS and is stored as additional metadata in the data blocks. Hence, targeted data retrieval can be very efficient at the expense of increased data loading time. This problem is addressed by *HAIL* (Dittrich et al., 2012), which improves query processing speeds over Hadoop++. HAIL creates indexes during the I/O-bound phases of writing to HDFS so that it consumes CPU cycles that are otherwise wasted. In addition, HAIL builds a different clustered index in each replica maintained by HDFS for fault tolerance purposes. The most suitable index for a query is then selected at run-time, and the corresponding replicas are read during the MapReduce execution over HAIL.

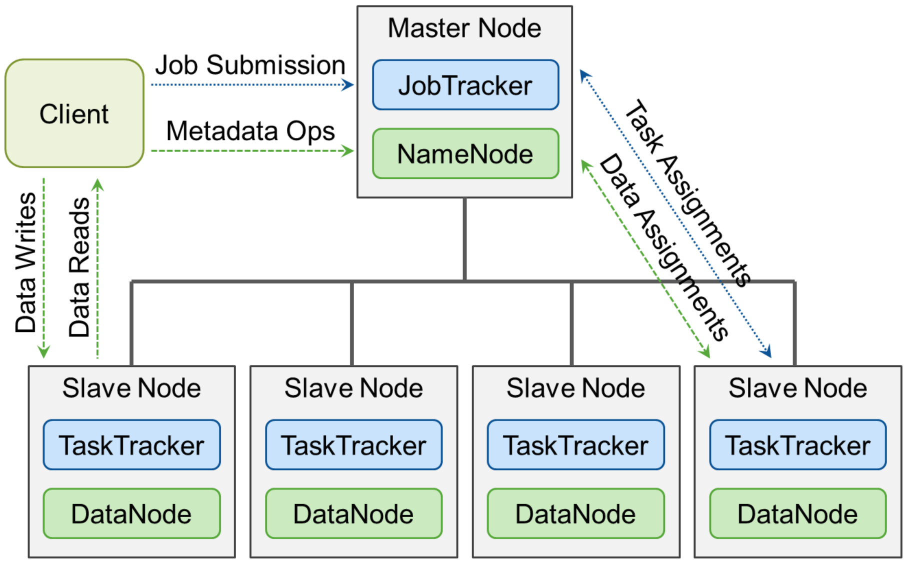

Figure 3: Hadoop architecture.

Collocation: In addition to indexing, Hadoop++ provides a data collocation technique in MapReduce systems. Specifically, Hadoop++ allows users to co-partition and collocate data at load time while writing metadata in the data blocks (Dittrich et al., 2010). Hence, blocks of HDFS can now contain data from multiple tables. With this approach, collocated joins can be processed at each node without the overhead of sorting and shuffling data across nodes. *CoHadoop* (Eltabakh et al., 2011) provides a different collocation strategy by adding a file-locator attribute to HDFS files and implementing a file layout policy such that all files with the same locator are placed on the same set of nodes. Using this feature, CoHadoop can collocate any related pair of files, e.g., every pair of joining partitions across two tables that are both hash-partitioned on the join key; or, a partition and an index on that partition. CoHadoop can then run joins in a similar manner as collocated joins in parallel database systems.

Columnar layouts: It is also possible to implement columnar data layouts in HDFS. *Llama* (Lin et al., 2011) and *CIF* (Floratou et al., 2011) use a pure column-oriented design, based on which they partitions attributes into vertical groups like the projections in C-Store and Vertica (recall Section 2.2). Each vertical group is sorted based on one of its component attributes. Each column is stored in a separate HDFS file, which enables each column to be accessed independently and, thus, reduces read I/O costs, but may incur run-time costs for tuple reconstruction. Unlike Llama, CIF uses an extension of HDFS to enable collocation of columns corresponding to the same tuple on the same node and supports some late materialization techniques for reducing tuple reconstruction costs (Floratou et al., 2011).

Cheetah (Chen, 2010), *RCFile* (He et al., 2011), and *Hadoop++* (Dittrich et al., 2010) use a hybrid row-column design based on *PAX* (Ailamaki et al., 2001). In particular, each file is horizontally partitioned into blocks but a columnar format is used within each block. Since HDFS guarantees that all the bytes of an HDFS block will be stored on a single node, it is guaranteed that tuple reconstruction will not require data transfer over the network. The intra-block data layouts used by these systems differ in how they use compression, how they treat replicas of the same block, and how they are implemented. For example, Hadoop++ can use different layouts in different replicas, and choose the best layout at query processing time.

3.2 MapReduce Execution Engines

MapReduce execution engines implement the MapReduce programming model for dealing with data at massive scale (Dean and Ghemawat, 2004). Users specify computations in terms of Map and Reduce functions while the underlying run-time system automatically parallelizes the computation across large-scale clusters of commodity servers, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disk bandwidth.

The MapReduce programming model consists of two functions: $map(k_1, v_1)$ and $reduce(k_2, list(v_2))$. Users can implement their own processing logic by specifying a customized *map*() and *reduce*() function written in a general-purpose language like Java or Python. The $map(k_1, v_1)$ function is invoked for every *key-value* pair $\langle k_1, v_1 \rangle$ in the input data to output zero or more key-value pairs of the form $\langle k_2, v_2 \rangle$ (see Figure 4). The $reduce(k_2, list(v_2))$ function is invoked for every unique key k_2 and corresponding values $list(v_2)$ in the map output, and outputs zero or more key-value pairs of the form $\langle k_3, v_3 \rangle$. The MapReduce programming model allows for other functions as well, such as (i) $partition(k_2)$, for controlling how the map output key-value pairs are partitioned among the reduce tasks, and (ii) $combine(k_2, list(v_2))$, for performing partial aggregation on the map side. The keys k_1, k_2, and k_3 as well as the values v_1, v_2, and v_3 can be of different and arbitrary types.

Hadoop MapReduce (White, 2010) is the most widely used implementation of a MapReduce execution engine. A Hadoop MapReduce cluster employs a master-slave architecture where one master node (called *JobTracker*) manages a number of slave nodes (called *TaskTrackers*), as seen in Figure 3. Hadoop launches a MapReduce job by first splitting (logically) the input dataset into data *splits*. Each data split is then scheduled to one TaskTracker node and is processed by a map task. A *Task Scheduler* resides in the JobTracker and is responsible for scheduling the execution of map tasks while taking data locality into account. Each TaskTracker has a predefined number of task execution *slots* for running map (reduce) tasks. If the job will execute more map (reduce) tasks than there are slots, then the map (reduce) tasks will run in multiple

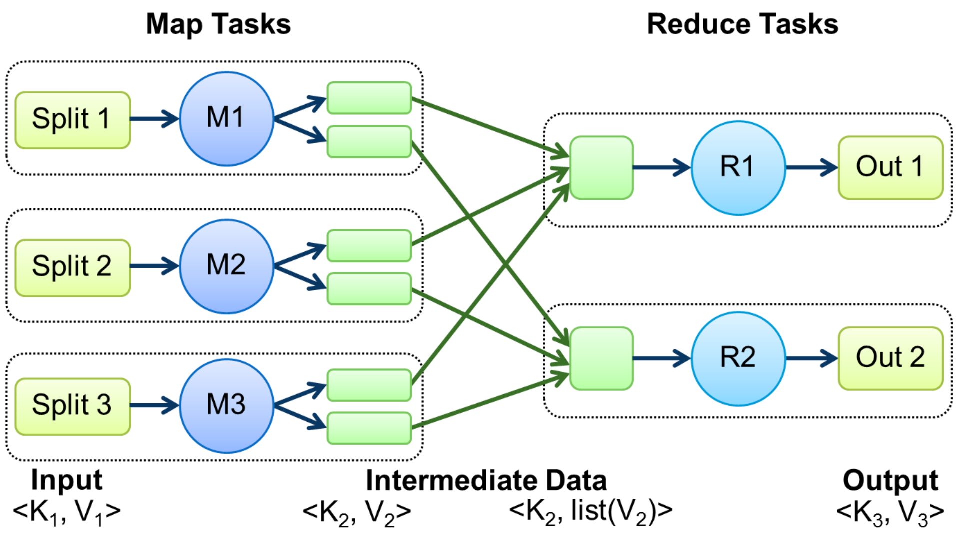

Figure 4: MapReduce job execution.

waves. When map tasks complete, the run-time system groups all intermediate key-value pairs using an external sort-merge algorithm. The intermediate data is then *shuffled* (i.e., transferred) to the TaskTrackers scheduled to run the reduce tasks. Finally, the reduce tasks will process the intermediate data to produce the results of the job.

HadoopDB (Abouzeid et al., 2009) is a hybrid system that combines features from parallel database systems with Hadoop. Specifically, HadoopDB runs a centralized database system on each node of the cluster and uses Hadoop primarily as the engine to schedule query execution plans as well as to provide fine-grained fault tolerance. The additional storage system provided by the databases gives HadoopDB the ability to overcome limitations of HDFS such as lack of collocation and indexing. In addition, HadoopDB includes some advanced partitioning capabilities such as reference-based partitioning, which enable multi-way joins to be performed in a collocated fashion.

HadoopDB introduced the concept of *split query execution* where a query submitted by a user or application will be converted into an execution plan consisting of some parts that would run as queries in the database and other parts that would run as map and reduce tasks in Hadoop (Bajda-Pawlikowski et al., 2011). The best such splitting of work will be identified during plan generation based on metadata stored in a system catalog. Metadata information includes connection parameters, schema and statistics of the tables stored, locations of replicas, and data partitioning properties.

3.3 MapReduce-based Platforms

The MapReduce model, although highly flexible, has been found to be too low-level for routine use by practitioners such as data analysts, statisticians, and scientists (Olston et al., 2008; Thusoo et al., 2009). As a result, the MapReduce framework has evolved into a *MapReduce ecosystem* shown at Figure 2, which includes a number of (i) *high-level interfaces* added over the core MapReduce engine, (ii) *application development* tools, (iii) *workflow management* systems, and (iv) *data collection* tools.

High-level Interfaces: The two most prominent examples of higher-level layers are *Apache Hive* (Thusoo et al., 2009) with an SQL-like declarative interface (called *HiveQL*) and *Apache Pig* (Olston et al., 2008) with an interface that mixes declarative and procedural elements (called *Pig Latin*). Both Hive and Pig will compile the respective HiveQL and Pig Latin queries into logical plans, which consist of a tree of logical operators. The logical operators are then converted into physical operators, which in turn are packed into map and reduce tasks for execution. The execution plan generated for a HiveQL or Pig Latin query is usually a workflow (i.e., a directed acyclic graph) of MapReduce jobs. Workflows may be ad-hoc, time-driven (e.g., run every hour), or data-driven. Yahoo! uses data-driven workflows to generate a reconfigured preference model and an updated home-page for any user within seven minutes of a home-page click by the user.

Similar to a data warehouse, Hive organizes and stores the data into partitioned tables (Thusoo et al., 2009). Hive tables are analogous to tables in relational databases and are represented using HDFS directories. Partitions are then created using subdirectories while the actual data is stored in files. Hive also includes a system catalog—called *Metastore*—containing schema and statistics, which are useful in data exploration and query optimization. In particular, Hive employs *rule-based* approaches for a variety of optimizations such as filter and projection pushdown, shared scans of input datasets across multiple operators from the same or different analysis tasks (Nykiel et al., 2010), reducing the number of MapReduce jobs in a workflow (Lee et al., 2011), and handling data skew in sorts and joins.

Application Development: *Cascading* (Cascading, 2011) and *FlumeJava* (Chambers et al., 2010) are software abstraction layers for MapReduce used to express data-parallel pipelines. They both offer program-based interfaces that integrate MapReduce job definitions into popular programming languages such as Java, JRuby, and Clojure. Hence, application developers can develop, test, and run efficient data-parallel pipelines without worrying about the underlying complexity of MapReduce jobs. To enable parallel operations to run efficiently, FlumeJava internally constructs an execution plan as a dataflow graph but defers its evaluation. When the final results are eventually needed, FlumeJava optimizes the execution plan and then executes the optimized operations on the underlying MapReduce primitives. Cascading and FlumeJava are most often used for log file analysis, bioinformatics, machine learning, and predictive analytics.

Workflow Management: A given MapReduce program may be expressed in one among a variety of programming languages like Java, C++, Python, or Ruby; may be generated by a query-based interface such as Hive or Pig; or may be generated by a program-based interface such as Cascading or JavaFlume. All these MapReduce programs can then be connected to form a *workflow* of MapReduce jobs using a workflow scheduler such as *Oozie* (Islam et al., 2012) and *Azkaban* (Sumbaly et al., 2013). Workflow schedulers ease construction of MapReduce workflows, which are typically defined as a collection of actions (e.g., native MapReduce jobs, Pig, Hive, and shell scripts) arranged in a control dependency DAG (Directed Acyclic Graph). The actions are then executed in sequence based on the dependencies described by the DAG.

Data Collection: MapReduce is designed to work on data stored in a distributed file system like HDFS. As a result, a number of distributed data collection systems have been built to copy data into distributed file systems, including *Flume* (Hoffman, 2015), *Scribe* (Thusoo et al., 2010), *Chukwa* (Rabkin and Katz, 2010) and *Kafka* (Sumbaly et al., 2013). The basic abstraction for most big data collection pipelines is the same: there is (i) a *source* that collects the data and inserts it into the system, (ii) a *sink* that delivers and stores the data into the file system, and (iii) a *channel* that acts as a conduit between the source and the sink allowing data to be streamed to a range of destinations. All systems are also designed to be scalable, reliable, extensible, and robust to failures of the network or any specific machine.

4 Dataflow Systems

The application domain for data-intensive analytics is moving towards complex data-processing tasks such as statistical modeling, graph analysis, machine learning, and scientific computing. While MapReduce can be used for these tasks, its programming model seems to be too restrictive in certain cases (e.g., joining two datasets together) and its execution model seems to be suboptimal for some common analysis tasks such as relational operations and graph processing. Consequently, *dataflow* systems such as Nephele (Battré et al., 2010) and Hyracks (Borkar et al., 2011) are extending the MapReduce framework with a more *generalized MapReduce execution model* that supports new primitive operations in addition to Map and Reduce A different class of dataflow systems such as Dryad (Isard et al., 2007) and Spark (Zaharia et al., 2012) aim at replacing MapReduce altogether with the *directed acyclic graph* model that can express a wide range of data access and communication patterns. Finally, *graph processing* systems like Pregel (Malewicz et al., 2010) use the bulk synchronous parallel processing model for running iterative computations and analysis over data graphs.

4.1 Generalized MapReduce Systems

Similar to MapReduce, *Nephele* (Battré et al., 2010) and *Hyracks* (Borkar et al., 2011) are two partitioned-parallel software systems designed to run data-intensive computations on large shared-nothing clusters of computers. However, they offer a more versatile execution model compared to MapReduce, with more data operators as well as data connectors. Nephele and Hyracks differ mainly on the type of operators and connectors that they support.

Nephele uses the *Parallelization Contracts (PACT)* programming model (Alexandrov et al., 2010), a generalization of the well-known MapReduce programming model. The PACT model extends MapReduce with a total of five second-order functions:

- **Map** is used to independently process each key-value pair.
- **Reduce** and **Combine** partition and group key-value pairs by their keys and process them together. They both assure that all pairs in a partition have the same key but Combine does not assure that all pairs with the same key are in the same partition.
- **Cross** is defined as the Cartesian product over its input sets (two or more). The user function is executed for each element of the Cartesian product.

- **CoGroup** partitions the key-value pairs of all input sets according to their keys. For each input, all pairs with the same key form one subset. Over all inputs, the subsets with same keys are grouped together and handed to the user function.
- **Match** is a relaxed version of the CoGroup contract and is equivalent to an inner equi-join.

In addition, the PACT model defines optional *output contracts* that give guarantees about the behavior of a function:

- **Same-Key:** Each key-value pair that is generated by the function has the same key as the key-value pair(s) that it was generated from.
- **Super-Key:** Each key-value pair that is generated by the function has a superkey of the key-value pair(s) that it was generated from.
- **Unique-Key:** Each key-value pair that is produced has a unique key.
- **Partitioned-by-Key:** Key-value pairs are partitioned by key. This property can be exploited when the contract is attached to a data source that supports partitioned storage.

Complete PACT programs are directed acyclic graphs (DAGs) of user functions, starting with one or more data sources and ending with one or more data sinks. Finally, Nephele uses certain declarative aspects of the second-order functions of the PACT programs to guide a series of transformation and optimization rules for generating an efficient parallel dataflow plan (Battré et al., 2010).

Nephele is the execution engine for *Stratosphere* (Alexandrov et al., 2014), a massively parallel data processing platform. In addition to Nephele and PACT, Stratosphere contains the *Sopremo* layer. A Sopremo program consists of a set of logical operators connected in a directed acyclic graph (DAG), akin to a logical query plan in relational DBMSs. Programs for the Sopremo layer can be written in *Meteor*, an operator-oriented query language that uses a JSON-like data model to support the analysis of unstructured and semi-structured data.

Similar to Nephele, Hyracks (Borkar et al., 2011) allows users to express a computation as a DAG of data operators and connectors. Operators process partitions of input data and produce partitions of output data, while connectors repartition operator outputs to make the newly-produced partitions available at the consuming operators. The most important Hyracks operators are:

- **Mapper:** Evaluates a user-defined function on each item in the input.

- **Sorter:** Sorts input records using user-provided comparator functions.
- **Joiner:** Binary-input operator that performs equi-joins.
- **Aggregator:** Performs aggregation using a user-defined aggregation function.

Hyracks is the lowest level of *ASTERIX* (Behm et al., 2011), a scalable platform for large-scale information storage, search, and analytics. The topmost layer of the ASTERIX stack is a parallel DBMS, with a full, flexible data model (ADM) and a query language (AQL) for describing, querying, and analyzing data. AQL is comparable to languages such as HiveQL and Pig Latin but supports both native storage and indexing of data as well as access to external data residing in a distributed file system (e.g., HDFS). In between these layers sits *Algebricks*, a model-agnostic, algebraic "virtual machine" for parallel query processing and optimization. Algebricks is the target for AQL query compilation, but it can also be the target for other declarative languages.

4.2 Directed Acyclic Graph Systems

The directed acyclic graph model replaces the MapReduce or MapReduce-based execution models in certain dataflow systems, such as *Dryad* (Isard et al., 2007) and *Spark* (Zaharia et al., 2012), offering a wider range of possible analytical tasks. Dryad is the execution engine used predominantly by Microsoft and utilized by the higher-level languages *DryadLINQ* (Isard and Yu, 2009) and *SCOPE* (Zhou et al., 2012). Spark, and its SQL-like interface *Shark* (Xin et al., 2013), have been developed at Berkeley's AMP Lab and have a strong emphasis on utilizing the memory on the compute nodes.

Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. A Dryad job has the form of a DAG, where each *vertex* defines the operations that are to be performed on the data and each *edge* represents the flow of data between the connected vertices. Vertices can have an arbitrary number of input and output edges. At execution time, vertices become processes communicating with each other through data channels (edges) used to transport a finite sequence of data records. The physical implementation of the channel abstraction is realized by shared memory, TCP pipes, or disk files. The inputs to a Dryad job are typically stored as partitioned files in the *Cosmos Storage System*. Each input partition is represented as a source vertex in the job graph and any processing vertex that is connected to a source vertex reads the entire partition sequentially through its input channel.

Figure 5 shows the Dryad system architecture. The execution of a Dryad job is orchestrated by a user-provided *Job Manager*. The primary function of the Job Manager is to construct the run-time DAG from its logical representation and execute it in the cluster. The Job Manager is also responsible for scheduling the vertices on

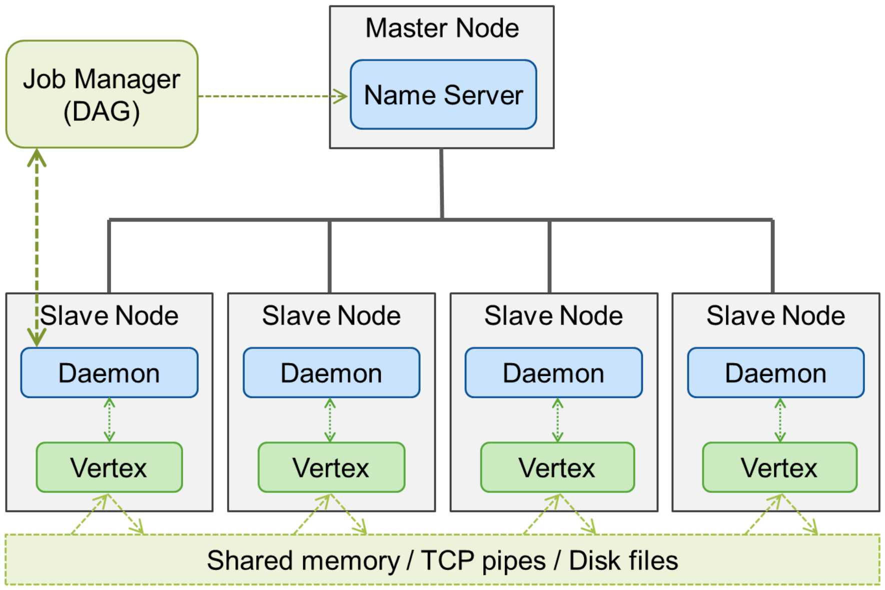

Figure 5: Dryad system architecture and execution.

the processing nodes when all the inputs are ready, monitoring progress, and re-executing vertices upon failure. A Dryad cluster has a *Name Server* that enumerates all the available compute nodes and exposes their location within the network so that scheduling decisions can take better account of locality. There is a processing *Daemon* running on each cluster node that is responsible for creating processes on behalf of the Job Manager. Each process corresponds to a vertex in the graph. The Daemon acts as a proxy so that the Job Manager can communicate with the remote vertices and monitor the state and progress of the computation.

DryadLINQ (Isard and Yu, 2009) is a hybrid of declarative and imperative language layer that targets the Dryad run-time and uses the *Language INtegrated Query (LINQ)* model (Meijer et al., 2006). DryadLINQ provides a set of .NET constructs for programming with datasets. A DryadLINQ program is a sequential program composed of LINQ expressions that perform arbitrary side-effect-free transformations on datasets. SCOPE (Zhou et al., 2012), on the other hand, offers a SQL-like declarative language with well-defined but constrained semantics. In particular, SCOPE supports writing a program using traditional nested SQL expressions as well as a series of simple data transformations.

Spark (Zaharia et al., 2012) is a similar DAG-based execution engine. However, the main difference of Spark from Dryad is that it uses a memory abstraction—called *Resilient Distributed Datasets (RDDs)*—to explicitly store data in memory. An RDD is a distributed shared memory abstraction that represents an immutable collection of objects partitioned across a set of nodes. Each RDD is either a collection backed by an external storage system, such as a file in HDFS, or a derived dataset created by applying various data-parallel operators (e.g., map, group-by, hashjoin) to other RDDs. The elements of an RDD need not exist in physical storage or reside in memory explicitly; instead, an RDD can contain only the lineage information necessary for computing the RDD elements starting from data in reliable storage. This notion of lineage is crucial for achieving fault tolerance in case a partition of an RDD is lost as well as managing how much memory is used by RDDs. Currently, RDDs are used by Spark with HDFS as the reliable back-end store.

Shark (Xin et al., 2013) is a higher-level system implemented over Spark and uses HiveQL as its query interface. Shark supports dynamic query optimization in a distributed setting via offering support for *partial DAG execution (PDE)*; a technique that allows dynamic alteration of query plans based on data statistics collected at run-time. Shark uses PDE to select the best join strategy at run-time based on the exact sizes of the join's input as well as to determine the degree of parallelism for operators and mitigate skew.

4.3 Graph Processing Systems

For a growing number of applications, the data takes the form of graphs that connect many millions of nodes. The growing need for managing graph-shaped data comes from applications such as: (a) identifying influential people and trends propagating through a social-networking community, (b) tracking patterns of how diseases spread, and (c) finding and fixing bottlenecks in computer networks. Graph processing systems such as *Pregel* (Malewicz et al., 2010), *GraphLab* (Low et al., 2012), and *GraphX* (Xin et al., 2013) use graph structures with nodes, edges, and their properties to represent and store data.

Many graph databases such as Pregel (Malewicz et al., 2010) use the *Bulk Synchronous Parallel (BSP)* computing model. A typical Pregel computation consists of: (i) initializing the graph from the input, (ii) performing a sequence of iterations separated by global synchronization points until the algorithm terminates, and (iii) writing the output. Similar to DAG-based systems, each vertex executes the same user-defined function that expresses the logic of a given algorithm. Within each iteration, a vertex can modify its state or that of its outgoing edges, receive messages sent to it in the previous iteration, send messages to other vertices (to be received in the next iteration), or even mutate the topology of the graph.

GraphLab (Low et al., 2012) uses similar primitives (called *PowerGraph*) but directly targets asynchronous, dynamic, graph-parallel computations in the shared-memory setting. In addition, GraphLab contains several performance optimizations such as using data versioning to reduce network congestion and pipelined distributed locking to mitigate the effects of network latency. GraphX (Xin et al., 2013) runs on Spark and introduces a new abstraction called *Resilient Distributed Graph (RDG)*.

Graph algorithms are specified as a sequence of transformations on RDGs, where a transformation can affect nodes, edges, or both, and yields a new RDG.

Techniques have also been proposed to support the iterative and recursive computational needs of graph analysis in MapReduce systems. For example, *HaLoop* and *Twister* are designed to support iterative algorithms in MapReduce systems (Bu et al., 2010; Ekanayake et al., 2010). HaLoop employs specialized scheduling techniques and the use of caching between each iteration, whereas Twister relies on a publish/subscribe mechanism to handle all communication and data transfers. *PrIter* (Zhang et al., 2011), a distributed framework for iterative workloads, enables faster convergence of iterative tasks by providing support for prioritized iteration. Instead of performing computations on all data records without discrimination, PrIter prioritizes the computations that help convergence the most, so that the convergence speed of iterative process is significantly improved.

5 Systems for Interactive Analytics

The need to reduce the gap between the generation of data and the generation of analytics results over large-scale data has lead to a new breed of systems for *interactive* (i.e., with low latency) analytics. We separate these systems into three distinct categories. The first category refers to distributed storage and processing systems that support *mixed analytical and transactional* workloads, such as Bigtable (Chang et al., 2008) and Megastore (Baker et al., 2011). Support for transactions enables storage systems in particular to serve as the data store for online services while making the data available concurrently in the same system for analytics. Second, *distributed SQL query engines* run over distributed file systems and support ad-hoc analytics. For instance, Cloudera Impala (Wanderman-Milne and Li, 2014) enables users to issue low-latency SQL queries to data stored in HDFS (Shvachko et al., 2010) and Apache HBase (George, 2011) without requiring data movement or transformation. Finally, *stream processing systems* such as S4 (Neumeyer et al., 2010) and Storm (Storm, 2013) are driven by a data-centric model that allows for near real-time consumption and analysis of data.

5.1 Mixed Analytical and Transactional Systems

Traditionally, parallel databases have used different systems to support OLTP and OLAP. OLTP workloads are characterized by a mix of reads and writes to a few tuples at a time, typically through index structures like B-Trees. OLAP workloads are characterized by bulk updates and large sequential scans that read only a few columns at a time. However, newer database workloads are increasingly a mix of the traditional OLTP and OLAP workloads, which led to the development of new systems that can support both. On one hand, multiple distributed storage systems like *Bigtable* (Chang et al., 2008) and *Megastore* (Baker et al., 2011) provide various degrees of transactional capabilities, enabling them to serve as the data store for online services while making the data available concurrently in the same system for analytics. On the other hand, processing systems like *SAP HANA* (Färber et al., 2012) and *HYRISE* (Grund et al., 2012) can execute both OLTP and OLAP workloads.

Mixed Storage Systems: The most prominent example of a mixed storage system is Google's Bigtable, which is a distributed, versioned, and column-oriented system that stores multi-dimensional and sorted datasets (Chang et al., 2008). Each Bigtable table is stored as a multidimensional sparse map, with rows and columns, where each cell contains a timestamp and an associated arbitrary byte array. A cell value at a given row and column is uniquely identified by the tuple <table, row, column-family:column, timestamp>. All table accesses are based on the aforementioned primary key, while secondary indices are possible through additional index tables. Bigtable provides atomicity at the level of individual tuples.

Bigtable has motivated popular open-source implementations like *HBase* (George, 2011) and *Cassandra* (Lakshman and Malik, 2010). Both systems offer compression, secondary indexes, use data replication for fault tolerance within and across data centers, and have support for Hadoop MapReduce. However, Cassandra has a vastly different architecture: all nodes in the cluster have the some role and coordinate their activities using a pure peer-to-peer communication protocol. Hence, there is no single point of failure. Furthermore, Cassandra offers a tunable level of consistency per operation, ranging from weak, to eventual, to strong consistency. HBase, on the other hand, offers strong consistency by design.

Bigtable also led to the development of follow-up systems from Google such as *Megastore* (Baker et al., 2011) and *Spanner* (Corbett et al., 2012). Megastore and Spanner provide more fine-grained transactional support compared to Bigtable without sacrificing performance requirements in any significant way. Megastore supports ACID transactions at the level of user-specified groups of tuples called entity groups, and looser consistency across entity groups. Spanner, on the other hand, supports transactions at a global scale across data centers.

Mixed Processing Systems: Systems like *SAP HANA*, *HYRISE*, and *HyPer*, aim to support OLTP and OLAP in a single system. SAP HANA (Färber et al., 2012) is an in-memory relational database management system that can handle both high transaction rates and complex query processing. Figure 6 gives an overview of the general SAP HANA architecture. At the core, SAP HANA has a set of in-memory processing engines, each specialized in a different category of data formats. Relational data resides in tables in column or row layout in the combined column and row engine and can be converted from one layout to the other to allow query expressions with tables in both layouts. Graph data (e.g., XML, JSON) and text data reside in the graph engine and the text engine, respectively; more engines are possible due to the extensible architecture.

All engines in SAP HANA keep all data in main memory as long as there is enough space available. All data structures are optimized for cache-efficiency instead of being optimized for organization in traditional disk blocks. Furthermore, the engines compress the data using a variety of compression schemes. When the limit of available main memory is reached, entire data objects, e.g., tables or partitions, are unloaded from main memory under the control of application semantics and reloaded into main memory when they are required again. While virtually all data is kept in main memory by the processing engines for performance reasons, data is stored by the persistence layer for backup and recovery in case of a system restart after an explicit shutdown or a failure (Färber et al., 2012).

HYRISE (Grund et al., 2012) is a main memory hybrid database system, which automatically partitions tables into vertical groups of varying widths depending on how the columns of the table are accessed. Smaller column groups are preferred

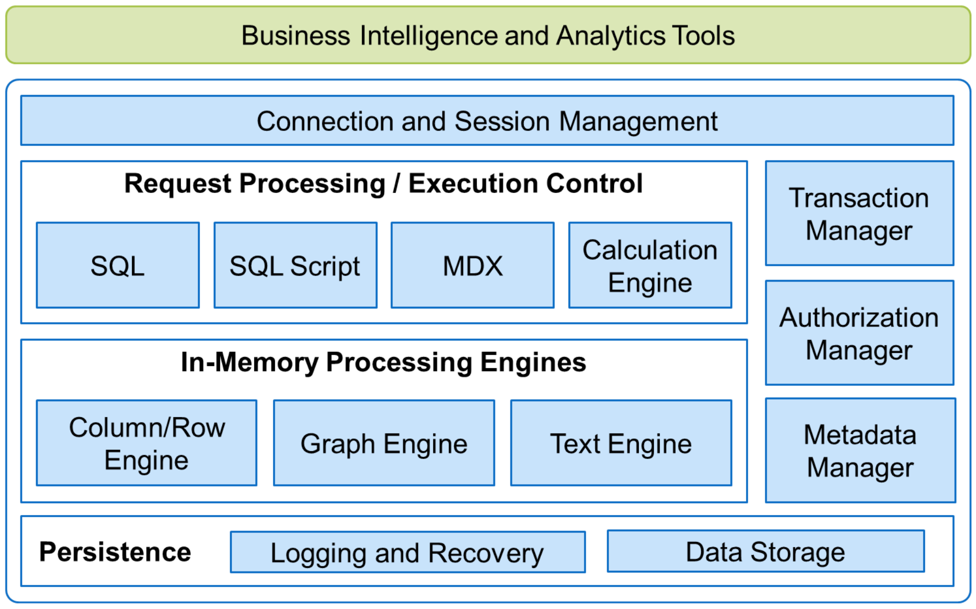

Figure 6: SAP HANA architecture.

for OLAP-style data access because, when scanning a single column, cache locality is improved when the values of that column are stored contiguously. On the other hand, wider column groups are preferred for OLTP-style data access because such transactions frequently insert, delete, update, or access many of the fields of a row, and co-locating those fields leads to better cache locality. Being an in-memory system, HYRISE identifies the best column grouping based on a detailed cost model of cache performance in mixed OLAP/OLTP settings.

HyPer (Kemper et al., 2012) is also a main memory database system that complements columnar data layouts with sophisticated main-memory indexing structures based on hashing, balanced search trees (e.g., red-black trees), and radix trees. Hash indexes enable exact match (e.g., primary key) accesses that are the most common in transactional processing, while the tree-structured indexes are essential for small-range queries that are also encountered here. Finally, HyPer uses adaptive compression techniques for separating cold (i.e. immutable) data for aggressive compression from the hot (i.e. mutable) working set data that remains uncompressed and readily available to mission-critical OLTP queries.

5.2 Distributed SQL Query Engines

The demand for more interactive analysis of large datasets has led to the development of new SQL-like query engines that run on top of distributed file systems and are optimized for ad-hoc analytics. *Dremel* (Melnik et al., 2010) is such a system that runs on top of GFS (Ghemawat et al., 2003) and Bigtable (Chang et al., 2008). Dremel

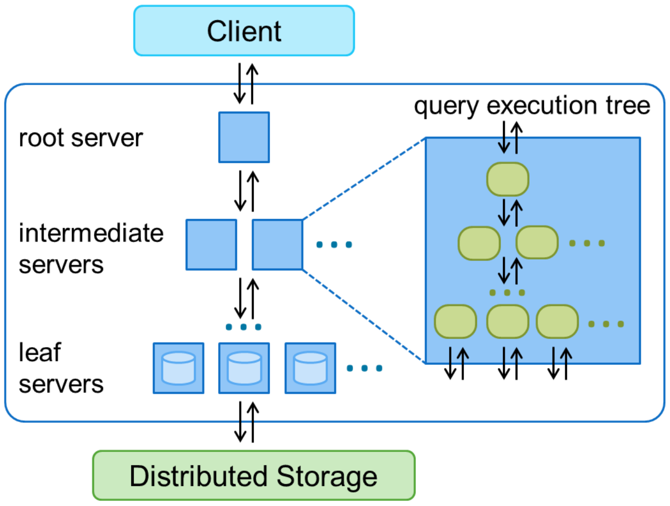

Figure 7: Dremel architecture and execution inside a server node.

exposes a SQL-like interface with extra constructs to query read-only data stored in a new columnar storage format that supports nested data. Each SQL statement in Dremel (and the algebraic operators it translates to) takes as input one or multiple nested tables and the input schema, and produces a nested table and its output schema. The two core technologies of Dremel are columnar storage for nested data and the tree architecture for query execution.

Dremel's data model is based on strongly-typed nested records with a schema that forms a tree hierarchy, originating from *Protocol Buffers* (Protocol Buffers, 2012). The key ideas behind the nested columnar format are: (i) a lossless representation of record structure by encoding the structure directly into the columnar format, (ii) fast encoding of column stripes by creating a tree of writers whose structure matches the field hierarchy in the schema, and (iii) efficient record assembly by utilizing finite state machines (Melnik et al., 2010).

Dremel—with corresponding open-source systems, *Cloudera Impala* (Wanderman-Milne and Li, 2014) and *Apache Drill* (Hausenblas and Nadeau, 2013)—uses the concept of a *multi-level serving tree* borrowed from distributed search engines (Croft et al., 2010) to execute queries. Figure 7 shows Dremel's architecture and execution inside a server node. When a root server receives an incoming query, it will rewrite the query into appropriate subqueries based on metadata information, and then route the subqueries down to the next level in the serving tree. Each serving level performs a similar rewriting and re-routing. Eventually, the subqueries will reach the leaf servers, which communicate with the storage layer or access the data from local disk. On the way up, the intermediate servers perform a parallel aggregation of partial results until the result of the query is assembled back in the root server.

Compared to Dremel that can query only single tables, Cloudera Impala supports both join and aggregate queries over multiple tables. Cloudera Impala can query data stored in HDFS or Apache HBase and uses the same metadata, SQL syntax (HiveQL), and user interface as Apache Hive; providing a unified platform for batch-oriented or real-time queries. Unlike Cloudera Impala that was developed to fit nicely with the Hadoop ecosystem, Apache Drill is meant to provide distributed query capabilities across multiple big data platforms including MongoDB, Cassandra, Riak and Splunk. Finally, *Presto* (Traverso, 2013) is a distributed SQL query engine developed at Facebook and, unlike Cloudera Impala and Apache Drill, supports standard ANSI SQL, including complex queries, aggregations, joins, and window functions.

5.3 Stream Processing Systems

Timely analysis of activity and operational data is critical for companies to stay competitive. Activity data from a company's Web-site contains page and content views, searches, as well as advertisements shown and clicked. A user's activity data, in combination with similar data from social friends, can be analyzed for various purposes like providing personalized content and recommendations as well as showing targeted advertisements (Chandramouli et al., 2012). Operational data includes monitoring data collected from Web applications (e.g., request latency) and cluster resources (e.g., CPU usage). Proactive analysis of operational data is used to ensure that Web applications continue to meet all service-level requirements.

The vast majority of analysis over activity and operational data involves *continuous queries* processed by stream processing systems. A continuous query is issued once over streaming data that is constantly updated and is run continuously. Hence, users get new results as the data changes, without having to issue the same query repeatedly. Continuous queries arise naturally over activity and operational data because (a) the data is generated continuously in the form of append-only streams; and (b) the data has a time component such that recent data is usually more relevant than older data.

The growing interest in continuous queries is reflected by the engineering resources that companies have recently been investing in building continuous query execution platforms. Yahoo! released *S4* (Neumeyer et al., 2010) in 2010, Twitter released *Storm* (Storm, 2013) in 2011, and Walmart Labs released *Muppet* in 2012 (Lam et al., 2012). In addition, systems such as *MapReduce Online* (Condie et al., 2010) and Facebook's real-time analytics system (Borthakur et al., 2011) are adding continuous querying capabilities to the popular Hadoop platform for batch analytics. These platforms add to older research projects like *Aurora* (Abadi et al., 2003), *Borealis* (Abadi et al., 2005), and *STREAM* (Babu and Widom, 2001), and as well as commercial systems like *Infosphere Streams* (Biem et al., 2010), and *Truviso* (Franklin et al., 2009).

S4 (Neumeyer et al., 2010) is a general-purpose, distributed, scalable platform that allows programmers to develop applications for processing continuous unbounded streams of data. S4 implements the *actors programming paradigm*. A user's program is defined in terms of *Processing Elements (PEs)* and *Adapters*, while the framework instantiates one PE for each unique key in the data stream. Each PE consumes the events and do one or both of the following: (a) emit one or more events which may be consumed by other PEs, (b) publish results. Execution-wise, S4 uses the push model for pushing events from one PE to the next. If a receiver buffer gets full, events are dropped to ensure the system will not get overloaded. Finally, S4 provides state recovery via uncoordinated checkpointing. When a node crashes, a new node takes over its task and restarts from a recent snapshot of its state. Events sent after the last checkpoint and before the recovery are lost.

Storm (Storm, 2013) is another platform for processing continuous unbounded streams of data but with a different programming paradigm and architecture compared to S4. A program in Storm is defined in terms of *spouts* (the sources) and *bolts* (the processing vertices) arranged in a specific topology. The number of bolts to instantiate is defined a-priori and each bolt will process a partition of the stream. Unlike S4, Storm uses a pull model where each bolt pulls events from its source, be it a spout or another bolt. Event loss can, therefore, happen only at ingestion time in the spouts when the external event rate is higher than what the system can process. Finally, the Storm provides guaranteed delivery of events based on which an event will either traverse the entire pipeline within a time interval or it will be declared as failed and can be replayed from the start by the spout.

6 Conclusions

A major part of the challenge in data analytics today comes from the sheer volume of data available for processing. Data volumes that many companies want to process in timely and cost-efficient ways have grown steadily from the multi-gigabyte range to terabytes and now to many petabytes. All data storage and processing systems that we presented in this monograph were aimed at handling such large datasets. This challenge of dealing with very large datasets has been termed the *volume* challenge. There are two other related challenges, namely, those of *velocity* and *variety* (Laney, 2001).

The velocity challenge refers to the short response-time requirements for collecting, storing, and processing data. Most of the systems in the MapReduce and Dataflow categories are batch systems. For latency-sensitive applications, such as identifying potential fraud and recommending personalized content, batch data processing is insufficient. The data may need to be processed as it streams into the system in order to extract the maximum utility from the data. Systems for interactive analytics are typically optimized for addressing the velocity challenge.

The variety challenge refers to the growing list of data types—relational, time series, text, graphs, audio, video, images, genetic codes—as well as the growing list of analysis techniques on such data. New insights are found while analyzing more than one of these data types together using a variety of analytical techniques such as linear algebra, statistical machine learning, text search, signal processing, natural language processing, and iterative graph processing.

Several higher-level systems and tools have been built on top of the systems described in this monograph for implementing these techniques, which drive automated processes for spam and fraud detection, advertisement placement, Web-site optimization, and customer relationship management. Business Intelligence (BI) tools like *SAS*, *SAP Business Objects*, *IBM Cognos*, *SPSS Modeler*, *Oracle Hyperion*, and *Microsoft BI*, provide support for reporting, online analytical processing, data mining, process mining, and predictive analytics based on data stored primarily in Data Warehouses. Other software platforms such as *Tableau* and *Spotfire* specialize in interactive data visualization of business data. In particular, these platforms query relational databases, cubes, cloud databases, and spreadsheets to generate a number of graph types that can be combined into analytic dashboards and applications. Both platforms also support visualizing large-scale data stored in distributed file systems such as HDFS. On the other hand, companies like *Datameer*, *Karmasphere*, and *Platforma* offer business intelligence solutions that specifically target the Hadoop ecosystem.

References

Abadi, D. J., Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik (2005). The Design of the Borealis Stream Processing Engine. In *Proc. of the 3rd Biennial Conf. on Innovative Data Systems Research (CIDR)*, pp. 277–289.

Abadi, D. J., P. A. Boncz, and S. Harizopoulos (2009). Column Oriented Database Systems. *Proc. of the VLDB Endowment 2*(2), 1664–1665.

Abadi, D. J., D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik (2003). Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal 12*(2), 120–139.

Abadi, D. J., D. S. Myers, D. J. DeWitt, and S. Madden (2007). Materialization Strategies in a Column-Oriented DBMS. In *Proc. of the 23rd IEEE Intl. Conf. on Data Engineering (ICDE)*, pp. 466–475. IEEE.

Abouzeid, A., K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz (2009). HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proc. of the VLDB Endowment 2*(1), 922–933.

Agrawal, S., V. Narasayya, and B. Yang (2004). Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 359–370. ACM.

Ailamaki, A., D. J. DeWitt, M. D. Hill, and M. Skounakis (2001). Weaving Relations for Cache Performance. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 169–180. Morgan Kaufmann Publishers Inc.

Alexandrov, A., R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. (2014). The Stratosphere Platform for Big Data Analytics. *The VLDB Journal 23*(6), 939–964.

Alexandrov, A., M. Heimel, V. Markl, D. Battré, F. Hueske, E. Nijkamp, S. Ewen, O. Kao, and D. Warneke (2010). Massively Parallel Data Analysis with PACTs on Nephele. *Proc. of the VLDB Endowment 3*(1-2), 1625–1628.

Amazon S3 (2013). *Amazon Simple Storage Service (S3)*. `http://aws.amazon.com/s3/`.

Babu, S. and J. Widom (2001). Continuous Queries over Data Streams. *ACM SIGMOD Record 30*(3), 109–120.

Bajda-Pawlikowski, K., D. J. Abadi, A. Silberschatz, and E. Paulson (2011). Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1165–1176. ACM.

Baker, J., C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh (2011). Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of the 5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pp. 223–234.

Baru, C. K., G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson (1995). DB2 Parallel Edition. *IBM Systems Journal 34*(2), 292–322.

Battré, D., S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke (2010). Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proc. of the 1st Symposium on Cloud Computing (SoCC)*, pp. 119–130. ACM.

Behm, A., V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras (2011). ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models. *Distributed and Parallel Databases 29*(3), 185–216.

Biem, A., E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran (2010). IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1093–1104. ACM.

Boncz, P., T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner (2006). MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 479–490. ACM.

Boncz, P. A., M. Zukowski, and N. Nes (2005). MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of the 2nd Biennial Conf. on Innovative Data Systems Research (CIDR)*, pp. 225–237.

Borkar, V., M. Carey, R. Grover, N. Onose, and R. Vernica (2011). Hyracks: A Flexible and Extensible Foundation for Data-intensive Computing. In *Proc. of the 27th IEEE Intl. Conf. on Data Engineering (ICDE)*, pp. 1151–1162. IEEE.

Borthakur, D., J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. S. Aiyer (2011). Apache Hadoop Goes Realtime at Facebook. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1071–1080. ACM.

Bu, Y., B. Howe, M. Balazinska, and M. Ernst (2010). HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proc. of the VLDB Endowment 3*(1-2), 285–296.

Calder, B., J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. (2011). Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of the 23rd ACM Symp. on Operating Systems Principles (SOSP)*, pp. 143–157. ACM.

Cascading (2011). *Cascading: Application Platform for Enterprise Big Data.* `http://www.cascading.org/`.

Chambers, C., A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum (2010). FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 363–375. IEEE.

Chandramouli, B., J. Goldstein, and S. Duan (2012). Temporal Analytics on Big Data for Web Advertising. In *Proc. of the 28th IEEE Intl. Conf. on Data Engineering (ICDE)*, pp. 90–101. IEEE.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber (2008). Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. on Computer Systems (TOCS) 26*(2), 4:1–4:26.

Chen, H., R. H. Chiang, and V. C. Storey (2012). Business Intelligence and Analytics: From Big Data to Big Impact. *MIS quarterly 36*(4), 1165–1188.

Chen, S. (2010). Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *Proc. of the VLDB Endowment 3*(2), 1459–1468.

Cohen, J., B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton (2009). MAD Skills: New Analysis Practices for Big Data. *Proc. of the VLDB Endowment 2*(2), 1481–1492.

Condie, T., N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears (2010). MapReduce Online. In *Proc. of the 7th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pp. 313–328. USENIX Association.

Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford (2012). Spanner: Google's Globally-distributed Database. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pp. 261–264. USENIX Association.

Croft, W. B., D. Metzler, and T. Strohman (2010). *Search Engines: Information Retrieval in Practice*. Addison-Wesley Reading.

Dean, J. and S. Ghemawat (2004). MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pp. 137–149. USENIX Association.

Dean, J. and S. Ghemawat (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*(1), 107–113.

DeWitt, D. J., S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen (1990). The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering (TKDE) 2*(1), 44–62.

DeWitt, D. J. and J. Gray (1992). Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM 35*(6), 85–98.

DeWitt, D. J., J. F. Naughton, D. A. Schneider, and S. Seshadri (1992). Practical Skew Handling in Parallel Joins. In *Proc. of the 18th Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 27–40. VLDB Endowment.

Dittrich, J., J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad (2010). Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *Proc. of the VLDB Endowment 3*(1-2), 515–529.

Dittrich, J., J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad (2012). Only Aggressive Elephants are Fast Elephants. *Proc. of the VLDB Endowment 5*(11), 1591–1602.

Ekanayake, J., H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox (2010). Twister: A Runtime for Iterative MapReduce. In *Proc. of the 19th Intl. Symposium on High Performance Distributed Computing (HPDC)*, pp. 810–818. ACM.

Eltabakh, M. Y., Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson (2011). CoHadoop: Flexible Data Placement and its Exploitation in Hadoop. *Proc. of the VLDB Endowment 4*(9), 575–585.

Färber, F., S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner (2012). SAP HANA Database: Data Management for Modern Business Applications. *ACM SIGMOD Record 40*(4), 45–51.

Färber, F., N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees (2012). The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin 35*(1), 28–33.

Floratou, A., J. M. Patel, E. J. Shekita, and S. Tata (2011). Column-oriented Storage Techniques for MapReduce. *Proc. of the VLDB Endowment 4*(7), 419–429.

Frankel, F. and R. Reid (2008). Big Data: Distilling Meaning from Data. *Nature 455*(7209), 30.

Franklin, M. J., S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre (2009). Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *Proc. of the 4th Biennial Conf. on Innovative Data Systems Research (CIDR)*.

George, L. (2011). *HBase: The Definitive Guide*. O'Reilly Media.

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003). The Google File System. *ACM SIGOPS Operating Systems Review 37*(5), 29–43.

Greenplum (2013). *Pivotal Greenplum Database*. `http://www.pivotal.io/big-data/pivotal-greenplum-database`.

Grund, M., P. Cudré-Mauroux, J. Krüger, S. Madden, and H. Plattner (2012). An Overview of HYRISE - A Main Memory Hybrid Storage Engine. *IEEE Data Engineering Bulletin 35*(1), 52–57.

Hausenblas, M. and J. Nadeau (2013). Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data 1*(2), 100–104.

He, Y., R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu (2011). RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proc. of the 27th IEEE Intl. Conf. on Data Engineering (ICDE)*, pp. 1199–1208. IEEE.

Herodotou, H., N. Borisov, and S. Babu (2011). Query Optimization Techniques for Partitioned Tables. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 49–60. ACM.

Hoffman, S. (2015). *Apache Flume: Distributed Log Collection for Hadoop.* Packt Publishing Ltd.

Hsiao, H.-I. and D. J. DeWitt (1990). Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proc. of the 6th IEEE Intl. Conf. on Data Engineering (ICDE)*, pp. 456–465. IEEE.

IBM Corporation (2007). *IBM Knowledge Center: Partitioned Tables.* IBM Corporation. `http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.partition.doc/doc/c0021560.html`.

IBM Netezza (2012). *IBM Netezza Data Warehouse Appliances.* `http://www-01.ibm.com/software/data/netezza/`.

Idreos, S., F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten (2012). MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin 35*(1), 40–45.

Infobright (2013). *Infobright - Analytic Database for the Internet of Things.* `http://www.infobright.com/`.

Isard, M., M. Budiu, Y. Yu, A. Birrell, and D. Fetterly (2007). Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review 41*(3), 59–72.

Isard, M. and Y. Yu (2009). Distributed Data-Parallel Computing Using a High-Level Programming Language. In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 987–994. ACM.

Islam, M., A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur (2012). Oozie: Towards a Scalable Workflow Management System for Hadoop. In *Proc. of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, pp. 4. ACM.

Kemper, A., T. Neumann, F. Funke, V. Leis, and H. Mühe (2012). HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. *IEEE Data Engineering Bulletin 35*(1), 46–51.

KFS (2013). *Kosmos Distributed Filesystem.* `http://code.google.com/p/kosmosfs/`.

Lakshman, A. and P. Malik (2010). Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*(2), 35–40.

Lam, W., L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan (2012). Muppet: MapReduce-Style Processing of Fast Data. *Proc. of the VLDB Endowment 5*(12), 1814–1825.

Lamb, A., M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear (2012). The Vertica Analytic Database: C-store 7 Years Later. *Proc. of the VLDB Endowment 5*(12), 1790–1801.

Laney, D. (2001). 3D Data Management: Controlling Data Volume, Velocity and Variety. *Application Delivery Strategies 6*, 1–4.

Lee, G., J. Lin, C. Liu, A. Lorek, and D. Ryaboy (2012). The Unified Logging Infrastructure for Data Analytics at Twitter. *Proc. of the VLDB Endowment 5*(12), 1771–1780.

Lee, R., T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang (2011). YSmart: Yet Another SQL-to-MapReduce Translator. In *Proc. of the 31st Intl. Conf. on Distributed Computing Systems (ICDCS)*, pp. 25–36. IEEE.

Lin, Y., D. Agrawal, C. Chen, B. C. Ooi, and S. Wu (2011). Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 961–972. ACM.

Low, Y., J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein (2012). Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. of the VLDB Endowment 5*(8), 716–727.

MacNicol, R. and B. French (2004). Sybase IQ Multiplex-designed for Analytics. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 1227–1230. VLDB Endowment.

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski (2010). Pregel: A System for Large-scale Graph Processing. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 135–146. ACM.

MapR (2013). *MapR File System.* `http://www.mapr.com/products/apache-hadoop`.

Mehta, M. and D. J. DeWitt (1997). Data Placement in Shared-Nothing Parallel Database Systems. *The VLDB Journal 6*(1), 53–72.

Meijer, E., B. Beckman, and G. Bierman (2006). LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 706–706. ACM.

Melnik, S., A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis (2010). Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. of the VLDB Endowment 3*(1), 330–339.

Morales, T. (2007). *Oracle Database VLDB and Partitioning Guide 11g Release 1 (11.1)*. Oracle Corporation. `http://docs.oracle.com/cd/B28359_01/server.111/b32024.pdf`.

Neumeyer, L., B. Robbins, A. Nair, and A. Kesari (2010). S4: Distributed Stream Computing Platform. In *Proc. of the 2010 IEEE Intl. Conf. on Data Mining Workshops*. IEEE.

Nykiel, T., M. Potamias, C. Mishra, G. Kollios, and N. Koudas (2010). MRShare: Sharing Across Multiple Queries in MapReduce. *Proc. of the VLDB Endowment 3*(1), 494–505.

Olston, C., B. Reed, U. Srivastava, R. Kumar, and A. Tomkins (2008). Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1099–1110. ACM.

Ovsiannikov, M., S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly (2013). The Quantcast File System. *Proc. of the VLDB Endowment 6*(11), 1092–1101.

ParAccel (2013). *ParAccel Analytic Platform*. `http://www.paraccel.com/`.

Protocol Buffers (2012). *Protocol Buffers Developer Guide*. `https://developers.google.com/protocol-buffers/docs/overview`.

Rabkin, A. and R. Katz (2010). Chukwa: A System for Reliable Large-scale Log Collection. In *Proc. of the 24th USENIX Intl. Conf. on Large Installation System Administration (LISA)*, pp. 1–15. USENIX Association.

Rao, J., C. Zhang, N. Megiddo, and G. M. Lohman (2002). Automating Physical Database Design in a Parallel Database. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 558–569. ACM.

Shvachko, K., H. Kuang, S. Radia, and R. Chansler (2010). The Hadoop Distributed File System. In *Proc. of the 26th IEEE Symp. on Mass Storage Systems and Technologies (MSST)*, pp. 1–10. IEEE.

Stonebraker, M., D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. (2005). C-Store: A Column-oriented DBMS. In *Proc. of the 31st Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 553–564. VLDB Endowment.

Storm (2013). *Storm: Distributed and Fault-tolerant Realtime Computation*. `http://storm-project.net/`.

Sumbaly, R., J. Kreps, and S. Shah (2013). The Big Data Ecosystem at LinkedIn. In *Proc. of the 2013 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1125–1134. ACM.

Talmage, R. (2009). *Partitioned Table and Index Strategies Using SQL Server 2008.* Microsoft. `http://msdn.microsoft.com/en-us/library/dd578580.aspx`.

Teradata (2012). *Teradata Enterprise Data Warehouse.* `http://www.teradata.com`.

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy (2009). Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. of the VLDB Endowment 2*(2), 1626–1629.

Thusoo, A., Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu (2010). Data Warehousing and Analytics Infrastructure at Facebook. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1013–1020. ACM.

Traverso, M. (2013). *Presto: Interacting with Petabytes of Data at Facebook.* `https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920`.

Wanderman-Milne, S. and N. Li (2014). Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin 37*(1), 31–37.

Weil, S. A., S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn (2006). Ceph: A Scalable, High-performance Distributed File System. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pp. 307–320. USENIX Association.

White, T. (2010). *Hadoop: The Definitive Guide.* Yahoo! Press.

Wu, L., R. Sumbaly, C. Riccomini, G. Koo, H. J. Kim, J. Kreps, and S. Shah (2012). Avatara: OLAP for Web-scale Analytics Products. *Proc. of the VLDB Endowment 5*(12), 1874–1877.

Xin, R. S., J. E. Gonzalez, M. J. Franklin, and I. Stoica (2013). GraphX: A Resilient Distributed Graph System on Spark. In *Proc. of the 1st Intl. Workshop on Graph Data Management Experiences and Systems*, pp. 2:1–2:6. ACM.

Xin, R. S., J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica (2013). Shark: SQL and Rich Analytics at Scale. In *Proc. of the 2013 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 13–24. ACM.

Zaharia, M., M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica (2012). Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pp. 15–28. USENIX Association.

Zhang, Y., Q. Gao, L. Gao, and C. Wang (2011). PrIter: A Distributed Framework for Prioritized Iterative Computations. In *Proc. of the 2nd Symposium on Cloud Computing (SoCC)*, pp. 13:1–13:14. ACM.

Zhou, J., N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib (2012). SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal 21*(5), 611–636.

Zukowski, M. and P. Boncz (2012). VectorWise: Beyond Column Stores. *IEEE Data Engineering Bulletin 35*(1), 21–27.