Foundations and Trends® in Databases
Vol. 5, No. 1 (2012) 1–104

DOI: 10.1561/1900000036

now
the essence of knowledge

Massively Parallel Databases and MapReduce Systems

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Herodotos Herodotou
Microsoft Research
herohero@microsoft.com

Contents

Abstract

Timely and cost-effective analytics over "big data" has emerged as a key ingredient for success in many businesses, scientific and engineering disciplines, and government endeavors. Web clicks, social media, scientific experiments, and datacenter monitoring are among data sources that generate vast amounts of raw data every day. The need to convert this raw data into useful information has spawned considerable innovation in systems for large-scale data analytics, especially over the last decade. This monograph covers the design principles and core features of systems for analyzing very large datasets using massively-parallel computation and storage techniques on large clusters of nodes. We first discuss how the requirements of data analytics have evolved since the early work on parallel database systems. We then describe some of the major technological innovations that have each spawned a distinct category of systems for data analytics. Each unique system category is described along a number of dimensions including data model and query interface, storage layer, execution engine, query optimization, scheduling, resource management, and fault tolerance. We conclude with a summary of present trends in large-scale data analytics.

S. Babu and H. Herodotou. *Massively Parallel Databases and MapReduce Systems.* Foundations and Trends® in Databases, vol. 5, no. 1, pp. 1–104, 2012.

DOI: 10.1561/1900000036.

1

Introduction

Organizations have always experienced the need to run data analytics tasks that convert large amounts of raw data into the information required for timely decision making. Parallel databases like Gamma [75] and Teradata [188] were some of the early systems to address this need. Over the last decade, more and more sources of large datasets have sprung up, giving rise to what is popularly called *big data*. Web clicks, social media, scientific experiments, and datacenter monitoring are among such sources that generate vast amounts of data every day.

Rapid innovation and improvements in productivity necessitate timely and cost-effective analysis of big data. This need has led to considerable innovation in systems for large-scale data analytics over the last decade. Parallel databases have added techniques like columnar data storage and processing [39, 133]. Simultaneously, new distributed compute and storage systems like MapReduce [73] and Bigtable [58] have been developed. This monograph is an attempt to cover the design principles and core features of systems for analyzing very large datasets. We focus on systems for large-scale data analytics, namely, the field that is called Online Analytical Processing (OLAP) as opposed to Online Transaction Processing (OLTP).

We begin in this chapter with an overview of how we have organized the overall content. The overview first discusses how the requirements of data analytics have evolved since the early work on parallel database systems. We then describe some of the major technological innovations that have each spawned a distinct category of systems for data analytics. The last part of the overview describes a number of dimensions along which we will describe and compare each of the categories of systems for large-scale data analytics.

The overview is followed by four chapters that each discusses one unique category of systems in depth. The content in the following chapters is organized based on the dimensions that will be identified in this chapter. We then conclude with a summary of present trends in large-scale data analytics.

1.1 Requirements of Large-scale Data Analytics

The Classic Systems Category: Parallel databases—which constitute the *classic* system category that we discuss—were the first systems to make parallel data processing available to a wide class of users through an intuitive high-level programming model. Parallel databases were based predominantly on the relational data model. The declarative SQL was used as the query language for expressing data processing tasks over data stored as tables of records.

Parallel databases achieved high performance and scalability by 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.

Three trends started becoming prominent in the early 2000s that raised questions about the superiority of classic parallel databases:

- More and more companies started to store as much data as they could collect. The classic parallel databases of the day posed major hurdles in terms of scalability and total cost of ownership as the need to process these ever-increasing data volumes arose.
- The data being collected and stored by companies was diverse in

structure. For example, it became a common practice to collect highly structured data such as sales data and user demographics along with less structured data such as search query logs and web page content. It was hard to fit such diverse data into the rigid data models supported by classic parallel databases.

- Business needs started to demand shorter and shorter intervals between the time when data was collected (typically in an OLTP system) and the time when the results of analyzing the data were available for manual or algorithmic decision making.

These trends spurred two types of innovations: (a) innovations aimed at addressing the deficiencies of classic parallel databases while preserving their strengths such as high performance and declarative query languages, and (b) innovations aimed at creating alternate system architectures that can support the above trends in a cost-effective manner. These innovations, together with the category of classic parallel database systems, give the four unique system categories for large-scale data analytics that we will cover. Table 1.1 lists the system categories and some of the systems that fall under each category.

1.2 Categorization of Systems

The Columnar Systems Category: Columnar systems pioneered the concept of storing tables by collocating entire columns together instead of collocating rows as done in classic parallel databases. Systems with columnar storage and processing, such as Vertica [133], have been shown to use CPU, memory, and I/O resources more efficiently in large-scale data analytics compared to row-oriented systems. Some of the main benefits come from reduced I/O in columnar systems by reading only the needed columns during query processing. Columnar systems are covered in Chapter 3.

The MapReduce Systems Category: MapReduce is a programming model and an associated implementation of a run-time system that was developed by Google to process massive datasets by harnessing a very large cluster of commodity nodes [73]. Systems in the classic

Category	Example Systems in this Category
Classic	Aster nCluster [25, 92], DB2 Parallel Edition [33], Gamma [75], Greenplum [99], Netezza [116], SQL Server Parallel Data Warehouse [177], Teradata [188]
Columnar	Amazon RedShift [12], C-Store [181], Infobright [118], MonetDB [39], ParAccel [164], Sybase IQ [147], VectorWise [206], Vertica [133]
MapReduce	Cascading [52], Clydesdale [123], Google MapReduce [73], Hadoop [192, 14], HadoopDB [5], Hadoop++ [80], Hive [189], JAQL [37], Pig [94]
Dataflow	Dremel [153], Dryad [197], Hyracks [42], Nephele [34], Pregel [148], SCOPE [204], Shark [195], Spark [199]

Table 1.1: The system categories that we consider, and some of the systems that fall under each category.

category 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. Popular systems in this category, such as Hadoop [14], store data in a standalone block-oriented distributed file-system, and run computational tasks in another distributed system that supports the MapReduce programming model. MapReduce systems are covered in Chapter 4.

The Dataflow Systems Category: Some deficiencies in MapReduce systems were identified as these systems were used for a large number of data analysis tasks. 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. Most of these problems can be addressed by replacing MapReduce with a more flexible dataflow-based execution model that can express a wide range of data access and communication

patterns. Various dataflow-based execution models have been used by the systems in this category, including directed acyclic graphs in Dryad [197], serving trees in Dremel [153], and bulk synchronous parallel processing in Pregel [148]. Dataflow systems are covered in Chapter 5.

Other System Categories: It became clear over time that new systems can be built by combining design principles from different system categories. For example, techniques used for high-performance processing in classic parallel databases can be used together with techniques used for fine-grained fault tolerance in MapReduce systems [5]. Each system in this *coalesced* category exposes a unified system interface that provides a combined set of features that are traditionally associated with different system categories. We will discuss coalesced systems along with the other system categories in the respective chapters.

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 saw the emergence of scalable distributed storage systems that provide various degrees of transactional capabilities. Support for transactions enables these systems to serve as the data store for online services while making the data available concurrently in the same system for analytics. The same need has led to the emergence of parallel database systems that support both OLTP and OLAP in a single system. We put both types of systems into the category called *mixed* systems because of their ability to run mixed workloads—workloads that contain transactional as well as analytics tasks—efficiently. We will discuss mixed systems in Chapter 6 as part of recent trends in massively parallel data analytics.

1.3 Categorization of System Features

We have selected eight key system features along which we will describe and compare each of the four categories of systems for large-scale data analytics.

Data Model and Interfaces: A *data model* provides the definition and logical structure of the data, and determines in which manner data

can be stored, organized, and manipulated by the system. The most popular example of a data model is the relational model (which uses a table-based format), whereas most systems in the MapReduce and Dataflow categories permit data to be in any arbitrary format stored in flat files. The data model used by each system is closely related to the *query interface* exposed by the system, which allows users to manage and manipulate the stored data.

Storage Layer: At a high level, a *storage layer* is simply responsible for persisting the data as well as providing methods for accessing and modifying the data. However, the design, implementation and features provided by the storage layer used by each of the different system categories vary greatly, especially as we start comparing systems across the different categories. For example, classic parallel databases use integrated and specialized data stores that are tightly coupled with their execution engines, whereas MapReduce systems typically use an independent distributed file-system for accessing data.

Execution Engine: When a system receives a query for execution, it will typically convert it into an *execution plan* for accessing and processing the query's input data. The *execution engine* is the entity responsible for actually running a given execution plan in the system and generating the query result. In the systems that we consider, the execution engine is also responsible for parallelizing the computation across large-scale clusters of machines, handling machine failures, and setting up inter-machine communication to make efficient use of the network and disk bandwidth.

Query Optimization: In general, *query optimization* is the process a system uses to determine the most efficient way to execute a given query by considering several alternative, yet equivalent, execution plans. The techniques used for query optimization in the systems we consider are very different in terms of: (i) the space of possible execution plans (e.g., relational operators in databases versus configuration parameter settings in MapReduce systems), (ii) the type of query optimization (e.g., cost-based versus rule-based), (iii) the type of cost modeling technique (e.g., analytical models versus models learned using machine-learning

techniques), and (iv) the maturity of the optimization techniques (e.g., fully automated versus manual tuning).

Scheduling: Given the distributed nature of most data analytics systems, *scheduling* the query execution plan is a crucial part of the system. Systems must now make several scheduling decisions, including scheduling where to run each computation, scheduling inter-node data transfers, as well as scheduling rolling updates and maintenance tasks.

Resource Management: *Resource management* primarily refers to the efficient and effective use of a cluster's resources based on the resource requirements of the queries or applications running in the system. In addition, many systems today offer elastic properties that allow users to dynamically add or remove resources as needed according to workload requirements.

Fault Tolerance: Machine failures are relatively common in large clusters. Hence, most systems have built-in *fault tolerance* functionalities that would allow them to continue providing services, possibly with graceful degradation, in the face of undesired events like hardware failures, software bugs, and data corruption. Examples of typical fault tolerance features include restarting failed tasks either due to application or hardware failures, recovering data due to machine failure or corruption, and using speculative execution to avoid stragglers.

System Administration: *System administration* refers to the activities where additional human effort may be needed to keep the system running smoothly while the system serves the needs of multiple users and applications. Common activities under system administration include performance monitoring and tuning, diagnosing the cause of poor performance or failures, capacity planning, and system recovery from permanent failures (e.g., failed disks) or disasters.

1.4 Related Work

This monograph is related to a few surveys done in the past. Lee and others have done a recent survey that focuses on parallel data processing with MapReduce [136]. In contrast, we provide a more comprehen-

sive and in-depth coverage of systems for large-scale data analytics, and also define a categorization of these systems. Empirical comparisons have been done in the literature among different systems that we consider. For example, Pavlo and others have compared the performance of both classic parallel databases and columnar databases with the performance of MapReduce systems [166].

Tutorials and surveys have appeared in the past on specific dimensions along which we describe and compare each of the four categories of systems for large-scale data analytics. Recent tutorials include one on data layouts and storage in MapReduce systems [79] and one on programming techniques for MapReduce systems [174]. Kossmann's survey on distributed query processing [128] and Lu's survey on query processing in classic parallel databases [142] are also related.

2

Classic Parallel Database Systems

The 1980s and early 1990s was a period of rapid strides in the technology for massively parallel processing. The initial drivers of this technology were scientific and engineering applications like weather forecasting, molecular modeling, oil and gas exploration, and climate research. Around the same time, several businesses started to see value in analyzing the growing volumes of transactional data. Such analysis led to a class of applications, called decision support applications, which posed complex queries on very large volumes of data. Single-system databases could not handle the workload posed by decision support applications. This trend, in turn, fueled the need for parallel database systems.

Three architectural choices were explored for building parallel database systems: *shared memory*, *shared disk*, and *shared nothing*. In the shared-memory architecture, all processors share access to a common central memory and to all disks [76]. This architecture has limited scalability because access to the memory quickly becomes a bottleneck. In the shared-disk architecture, each processor has its private memory, but all processors share access to all disks [76]. This architecture can become expensive at scale because of the complexity of connecting all processors to all disks.

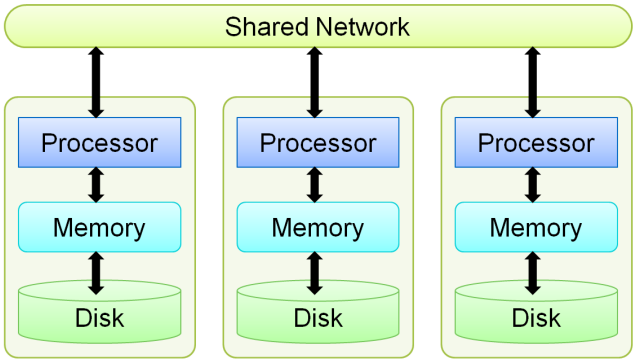

Figure 2.1: Shared-nothing architecture for parallel processing.

The shared-nothing architecture has proved to be the most viable at scale over the years. In the shared-nothing architecture, each processor has its own private memory as well as disks. Figure 2.1 illustrates the shared-nothing architecture used in parallel database systems. Note that the only resource shared among the processors is the communication network.

A number of research prototypes and industry-strength parallel database systems have been built using the shared-nothing architecture over the last three decades. Examples include Aster nCluster [25], Bubba [41], Gamma [75], Greenplum [99], IBM DB2 Parallel Edition [33], Netezza [116], Oracle nCUBE [48], SQL Server Parallel Data Warehouse [177], Tandem [85], and Teradata [188].

2.1 Data Model and Interfaces

Most parallel database systems support the relational data model. A relational database consists of *relations* (or, *tables*) that, in turn, consist of *tuples*. Every tuple in a table conforms to a *schema* which is defined by a fixed set of attributes [76].

This feature has both advantages and disadvantages. The simplicity of the relational model has historically played an important role in the success of parallel database systems. A well-defined schema helps with cost-based query optimization and to keep data error-free in the face

of data-entry errors by humans or bugs in applications. At the same time, the relational data model has been criticized for its rigidity. For example, initial application development time can be longer due to the need to define the schema upfront. Features such as support for JSON and XML data as well schema evolution reduce this disadvantage [71].

Structured Query Language (SQL) is the declarative language most widely used for accessing, managing, and analyzing data in parallel database systems. Users can specify an analysis task using an SQL query, and the system will optimize and execute the query. As part of SQL, parallel database systems also support (a) user-defined functions, user-defined aggregates, and stored procedures for specifying analysis tasks that are not easily expressed using standard relational-algebra operators, and (b) interfaces (e.g., ODBC, JDBC) for accessing data from higher-level programming languages such as C++ and Java or graphical user interfaces.

2.2 Storage Layer

The relational data model and SQL query language have the crucial benefit of data independence: SQL queries can be executed correctly irrespective of how the data in the tables is physically stored in the system. There are two noteworthy aspects of physical data storage in parallel databases: (a) *partitioning*, and (b) *assignment*. Partitioning a table S refers to the technique of distributing the tuples of S across disjoint fragments (or, partitions). Assignment refers to the technique of distributing these partitions across the nodes in the parallel database system.

2.2.1 Table Partitioning

Table partitioning is a standard feature in database systems today [115, 155, 185, 186]. For example, a sales records table may be partitioned *horizontally* based on value ranges of a date column. One partition may contain all sales records for the month of January, another partition may contain all sales records for February, and so on. A table can also be partitioned *vertically* with each partition containing a subset of

Uses of Table Partitioning in Database Systems
Efficient pruning of unneeded data during query processing
Parallel data access (partitioned parallelism) during query processing
Reducing data contention during query processing and administrative tasks. Faster data loading, archival, and backup
Efficient statistics maintenance in response to insert, delete, and update rates. Better cardinality estimation for subplans that access few partitions
Prioritized data storage on faster/slower disks based on access patterns
Fine-grained control over physical design for database tuning
Efficient and online table and index defragmentation at the partition level

Table 2.1: Uses of table partitioning in database systems

columns in the table. Vertical partitioning is more common in columnar database systems compared to the classic parallel database systems. We will cover vertical partitioning in Chapter 3. Hierarchical combinations of horizontal and vertical partitioning may also be used.

Table 2.1 lists various uses of table partitioning. Apart from giving major performance improvements, partitioning simplifies a number of common administrative tasks in database systems [9, 201]. The growing usage of table partitioning has been accompanied by efforts to give applications and users the ability to specify partitioning conditions for tables that they derive from base data. SQL extensions from database vendors now enable queries to specify how derived tables are partitioned (e.g., [92]).

The many uses of table partitioning have created a diverse mix of partitioning techniques used in parallel database systems. We will illustrate these techniques with an example involving four tables: $R(a, rdata)$, $S(a, b, sdata)$, $T(a, tdata)$, $U(b, udata)$. Here, a is an integer attribute and b is a date attribute. These two attributes will be

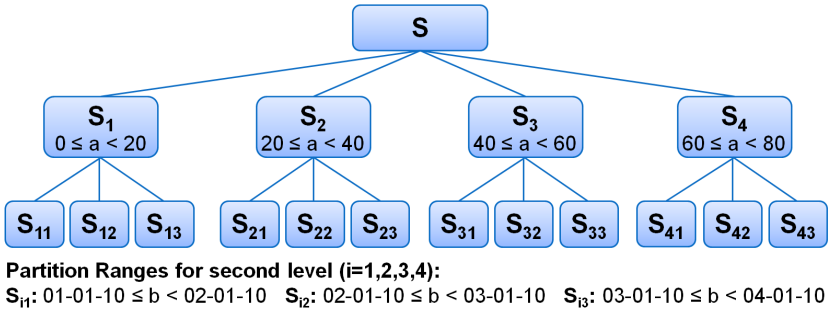

Figure 2.2: A hierarchical partitioning of table S

used as join keys in our examples. *rdata*, *sdata*, *tdata*, and *udata* are respectively the data specific to each of the tables R, S, T, and U.

Figure 2.2 shows an example partitioning for table S. S is *range-partitioned* on ranges of attribute a into four partitions S_1-S_4. Partition S_1 consists of all tuples in S with $0 \leq a < 20$, S_2 consists of all tuples in S with $20 \leq a < 40$, and so on. Each of S_1-S_4 is further range-partitioned on ranges of attribute b. Thus, for example, partition S_{11} consists of all tuples in S with $0 \leq a < 20$ and $01-01-2010 \leq b < 02-01-2010$.

Range partitioning is one among multiple partitioning techniques that can be used [53, 76, 108]:

- *Hash partitioning*, where tuples are assigned to tables 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. For example, a list partitioning for the example table S may specify that all tuples with $a \in 1, 2, 3$ should be in partition S_1.

- *Random (round-robin) partitioning*, where tuples are assigned to tables in a random (round-robin) fashion.

- *Block partitioning*, where each consecutive block of tuples (or bytes) written to a table forms a partition. For example, par-

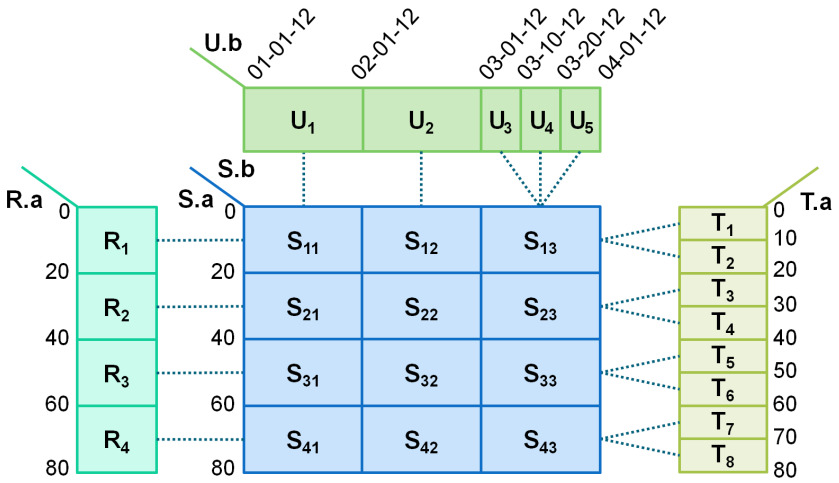

Figure 2.3: Partitioning of tables R, S, T, U. Dotted lines show partitions with potentially joining records

tition S_1 may consist of the first 1000 tuples inserted into S, S_2 may consist of the next 1000 tuples inserted into S, and so on.

Figure 2.3 shows how the partitioning of table S can be interpreted as a two-dimensional partitioning. The figure also shows partitions for tables R, T, and U. The dotted lines in the figure show the join relationships between pairs of partitions. These relationships become important when we talk about assignment in §2.2.2.

Figure 2.3 also shows how the sizes of partitions in a table may not be uniform. Such skewed partition sizes can arise for a number of reasons. Hash or equi-range partitioning produces skewed partition sizes if the attribute(s) used in the partitioning function has a skewed distribution. Skewed partition sizes may also come from data loading and archival needs. For example, Table U in Figure 2.3 is partitioned using unequal ranges on b. 10-day ranges are used for recent partitions of U. Older data is accessed less frequently, so older 10-day partitions of U are merged into monthly ones to improve query performance and archival efficiency.

2.2.2 Partition Assignment

Partition assignment is the task of deciding which node or nodes in the parallel database system should store each partition of the tables in the database. Three factors are usually considered as part of partition assignment: degree of declustering, collocation, and replication.

Degree of Declustering

The degree of declustering for a table specifies the number of nodes that store one or more partitions of the table. With *full declustering*, the degree of declustering is equal to the number of nodes in the system [75]. That is, tuples in a fully declustered table are spread over all nodes in the system. Otherwise, the table is said to be *partially declustered*.

Partial declustering is typically accompanied by the creation of *nodegroups* [33]. (Nodegroups are called *relation clusters* in [114].) A nodegroup is a collection of nodes in the parallel database system. Each nodegroup can be referenced by name. Consider the following example (with syntax borrowed from [33]) for a parallel database system with 10 nodes numbered 1-10:

CREATE NODEGROUP GROUP_A ON NODES(1 TO 10);
CREATE NODEGROUP GROUP_B ON NODES(2, 4, 6, 8, 10);
CREATE TABLE R (a integer, $rdata$ char[100]) IN GROUP_A PARTITIONING KEY (a) USING HASHING;
CREATE TABLE S (a integer, b integer, $sdata$ char[100]) IN GROUP_B PARTITIONING KEY (a) USING HASHING;
CREATE TABLE T (a integer, $tdata$ char[100]) IN GROUP_B PARTITIONING KEY (a) USING HASHING;

In this example, the database administrator chose to create two nodegroups—GROUP_A and GROUP_B—for the cluster of 10 nodes in the parallel database system. Group A consists of all 10 nodes while Group B consists of 5 nodes. Partitions of table R are stored on Group_A, so table R is fully declustered. Partitions of tables S and T are stored on Group_B, so these tables are partially declustered. Full declustering can benefit query processing over very large tables. For example, table R may be very large. In this case, a query that does

grouping and aggregation on $R.a$ can be processed in parallel using all 10 nodes, without needing to move any data among the nodes.

Collocation

It is sometimes beneficial to have selective overlap among the nodes on which the partitions of two or more tables are stored. Consider the example in §2.2.2 where partitions of tables S and T are both stored on the nodegroup Group_B. Also, note that both tables are hash partitioned on the respective attribute a. If the same hash function and number of partitions are chosen for both tables, then there will be a one-to-one correspondence between the partitions of both tables that will join with one another. In this case, it is possible to *collocate* the joining partitions of both tables. That is, any pair of joining partitions with be stored on the same node of the nodegroup.

The advantage of collocation is that tables can be joined without the need to move any data from one node to another. However, collocation of joining tables can be nontrivial when there are complex join relationships. Consider our four example tables and their join relationships shown in Figure 2.3. In this context, consider the following example three-way-join query that joins tables R, S, and U.

```
Select  *
From    R, S, U
Where   R.a = S.a and S.b = U.b
```

Suppose each partition can be stored only on one node in the parallel database system. In this case, the only way to collocate all pairs of joining partitions in the three tables is to store all partitions on a single node of the system. Such an assignment—where all three tables have a degree of declustering equal to one—would be a terrible waste of the resources in the parallel database.

Replication

As we saw in the above example, the flexibility of assignment is limited if tuples in a table are stored only once in the system. This problem can

be addressed by replicating tuples on multiple nodes. Replication can be done at the level of table partitions, which will address the problem in the above example. For example, if table U is small, then partitions of U can be replicated on all nodes where partitions of S are stored.

Replication can be done at the table level such that different replicas are partitioned differently. For example, one replica of the table may be hash partitioned while another may be range partitioned. 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 [151, 169].

2.3 Execution Engine

To execute a SQL query quickly and efficiently, a parallel database system has to break the query into multiple tasks that can be executed across the nodes in the system. The system's execution engine is responsible for orchestrating this activity.

In most parallel database systems, each submitted query is handled by a *coordinator task*. The coordinator first invokes a query optimizer in order to generate a execution plan for the query. An execution plan in a parallel database system is composed of operators that support both intra-operator 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. The coordinator task is responsible for checking whether the plan completed successfully, and if so, transferring the results produced by the plan to the user or application that submitted the query.

In this section, we will describe the components of a parallel execution plan. The next two sections will discuss respectively the techniques

used to select a good execution plan and to schedule the tasks in the plan.

2.3.1 Parallel Query Execution

Consider a query that joins two tables R and S based on the equi-join condition $R.a = S.a$. In §2.2.2, we introduced a *collocated join* operator that can perform the join if tables R and S are both partitioned and the partitions are assigned such that any pair of joining partitions is stored on the same node. A collocated join operator is often the most efficient way to perform the join because it performs the join in parallel while avoiding the need to transfer data between nodes. However, a collocated join is only possible if the partitioning and assignment of the joining tables is planned in advance.

Suppose the tables R and S are partitioned identically on the joining key, but the respective partitions are not collocated. In this case, a *directed join* operator can be used to join the tables. The *directed join* operator transfers 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.

Directed joins are also possible when the tables are not partitioned identically. For example, directed joins are possible for $R \bowtie S$, $T \bowtie S$, and $U \bowtie S$ in Figure 2.3 despite the complex partitioning techniques used. Compared to a collocated join, a directed join incurs the cost of transferring one of the tables across the network.

If the tables R and S are not partitioned identically on the joining attribute, then two other types of parallel join operators can be used to perform the join: *repartitioned join* operator and *broadcast join* (or *fragment-and-replicate join*) operator. The repartitioned join operator 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 the tables across the network.

The broadcast join operator transfers one table (say, R) in full to every node where any partition of the other table is stored. The join

is then performed locally. This operator incurs a data transfer cost of $size\_of(R) \times d$, where $size\_of(R)$ is the size of R and d is the degree of declustering of S. Broadcast join operators are typically used when one table is very small.

As discussed in §2.2.1, partition sizes may be skewed due to a number of reasons. The most common case is when one or both tables contain joining keys with 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.

Suppose a join key with value v has a skewed distribution among tuples of a table R that needs to be joined with a table S. In the regular repartitioned join, all tuples in R and S with join key equal to v will be processed by a single node in the parallel database system. Instead, the tuples in R with join key equal to v can be further partitioned across multiple nodes. The correct join result will be produced as long as the tuples in S 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 [77].

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* [76].

Two other forms of parallelism are also employed commonly in execution plans in parallel database systems: *pipelined parallelism* and *independent 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. This form of parallelism is called pipelined parallelism.

Independent parallelism refers to the parallel execution of independent operators in a query plan. For example, consider the following query that joins the four tables R, S, T, and U:

```
Select  *
From    R, S, T, U
Where   R.a = S.a and S.a = T.a and S.b = U.b
```

This query can be processed by a plan where R is joined with T, S is joined with U, and then the results of both these joins are joined together to produce the final result. In this plan, $R \bowtie T$ and $S \bowtie U$ can be executed independently in parallel.

2.3.2 Abstractions for Data Transfer

In a parallel query execution plan, interprocess data transfer may be needed between producer and consumer operators or between the different phases of a single operator (e.g., between the partition and join phases of a repartitioned join operator). Straightforward techniques for interprocess data transfer include materializing all intermediate tuples to disk or directly using the operating system's interprocess communication (IPC) mechanisms for each point-to-point data transfer. Such techniques, while easy to implement, have some drawbacks. They may incur high overhead or complicate the problem of finding good execution plans and schedules.

Parallel database systems have developed rich abstractions for interprocess data transfers in parallel query execution plans. We will describe three representative abstractions: (a) *split tables* from Gamma [75], (b) *table queues* from IBM DB2 Parallel Edition [33], and (c) *exchange operators* from Volcano [97].

Tuples output by an operator in a query plan in Gamma are routed to the correct destination by looking up a split table [75]. Thus, a split table maps tuples to destination tasks. For example, a split table that is used in conjunction with a repartitioned join of tables R and S may route tuples from both tables based on a common hash partitioning function.

IBM DB2 Parallel Edition's table queues are similar to split tables but provide richer functionality [33]. Table queues provide a buffering mechanism for tuples output by a producer task before the tuples are processed by a consumer task. Based on the rate at which the consumer can process tuples, tuple queues enable the rate at which the producer

produces tuples to be controlled. A table queue can have multiple producer tasks as well as consumer tasks. Furthermore, table queues can implement multiple communication patterns such as broadcasting versus directed (i.e., does a tuple produced by a producer task go to all consumer tasks or to a specific one?).

The exchange operator has a more ambitious goal beyond serving as the abstraction of interprocess data transfer in Volcano's query execution plans [97]. Exchange is a meta-operator that encapsulates all issues related to parallel execution (e.g., partitioning, producer-consumer synchronization) and drives parallel execution in Volcano. The benefit is that query execution plans in Volcano can leverage different forms of parallelism while still using the simpler implementations of relational operators from centralized databases.

2.4 Query Optimization

The following steps are involved in the execution of a SQL query in a parallel database system:

- *Plan selection:* choosing an efficient execution plan for the query.
- *Scheduling:* Generating executable tasks for the operators and data transfers in the plan, and choosing the nodes where these tasks will run.
- *Resource allocation:* Determining how much CPU, memory, and I/O (both local I/O and network I/O) resources to allocate to the executable tasks in the plan.

Plan selection, which is commonly referred to as query optimization, is the focus of this section. Scheduling and resource allocation will be covered respectively in the next two sections.

Plan selection for a query mainly involves the following steps:

- Ordering and choosing join operators.
- Access path selection for the tables.

- Choosing the appropriate cost metric (e.g., completion time versus total resource usage) based on which the best plan for the query can be determined.

The early approaches to plan selection in a parallel database system used a two-phase approach [113, 104]. First, a cost-based query optimizer—like the System R optimizer developed for centralized database systems [26, 173]—was used to find the best serial plan [113]. Problems like join ordering and access path selection were addressed as part of the selection of the best serial plan. Then, a post-optimization phase was applied to transform this serial plan into a parallel execution plan. Decisions made in the post-optimization phase included which attributes to repartition tables on such that data transfer costs were minimized [104].

While the two-phase approach showed initial promise—especially in shared-memory systems—it quickly became apparent that this approach can miss good plans in shared-nothing systems. For example, consider the four-way join example query introduced in §2.3.1. The best serial plan for the query may use the left-deep serial join order: $((R \bowtie S) \bowtie T) \bowtie U$. Since a serial plan assumes that the plan will run on a single node where all the data resides, the plan's cost is influenced primarily by factors like sort orders and the presence of indexes. However, the best join order for parallel execution could be different and bushy: $(R \bowtie T) \bowtie (S \bowtie U)$. Such a plan can be very efficient if R and T (and respectively, S and U) are stored in a collocated fashion. Recall that collocated joins are very efficient because they can leverage parallelism without incurring data transfer costs for the join processing.

As the above example shows, parallel database systems select query execution plans from a large plan space. The different forms of parallelism—partitioned, pipelined, and independent—motivate searching for the best plan from a larger set of operators as well as join trees compared to query execution in a centralized database system. The use of *semi-joins* and *partition-wise joins* further increases the size of the plan space for parallel execution plans [63, 108].

Semi-joins can act as size reducers (similar to a selection) such that the total size of tuples that need to be transferred is reduced. The use of

semi-joins is based on the following set of equivalences for the equi-join of two tables R and S on the join key a:

$$\begin{aligned} R \bowtie_a S &\equiv (R \ltimes_a S) \bowtie_a S \\ &\equiv R \bowtie_a (S \ltimes_a R) \\ &\equiv (R \ltimes_a S) \bowtie_a (S \ltimes_a R) \end{aligned}$$

The use of a semijoin is beneficial if the cost to produce and transfer it is less than the cost of transferring the whole table to do the join. However, now the plan space is significantly larger because of the many combinations in which joins and semi-join reducers can be used in a plan [63].

Recall from §2.2.1 that many types of partitioning techniques are used in parallel database systems. These techniques further increase the search space during plan selection over partitioned tables. Consider an example query Q_1 over the partitioned tables R, S, and T shown in Figure 2.3.

```
Q1: Select  *
    From    R, S, T
    Where R.a = S.a and S.a = T.a and S.b ≥ 02-15-10 and T.a < 25
```

Use of filter conditions for partition pruning: The partitions T_4-T_8 and S_{11}, S_{21}, S_{31}, S_{41} can be *pruned* from consideration because it is clear from the partitioning conditions that tuples in these partitions will not satisfy the filter conditions. Partition pruning can speed up query performance drastically by eliminating unnecessary table and index scans as well as reducing data transfers, memory needs, disk spills, and resource-contention-related overheads.

Use of join conditions for partition pruning: Based on a transitive closure of the filter and join conditions, partition pruning can also eliminate partitions S_{32}, S_{33}, S_{42}, S_{43}, R_3, R_4, and U_1.

The use of partition pruning will generate a plan like Q_1P_1 shown in Figure 2.4. In such plans, the leaf operators logically append together (i.e., UNION ALL) the unpruned partitions for each table. Each unpruned partition is accessed using regular table or index scans. The appended partitions are joined using one of the parallel join operators such as the collocated or repartitioned join operators.

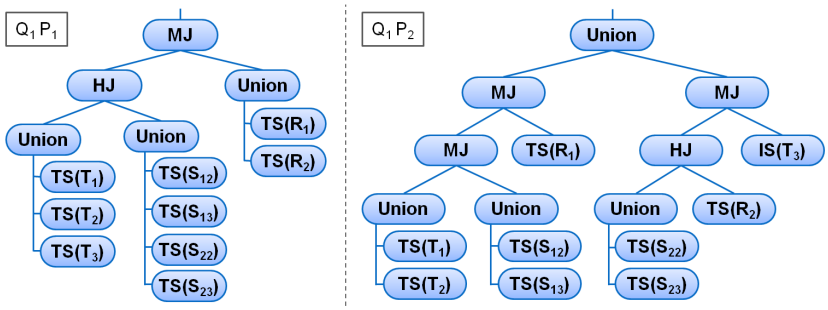

Figure 2.4: Q_1P_1 and Q_1P_2 are plans that can be generated by optimizers in parallel database systems for our example query Q_1. IS and TS are respectively index and table scan operators. HJ and MJ are respectively parallel join operators based on hash and merge. Union is a bag union operator.

Partition-aware join path selection: Depending on the data properties and data layout in the parallel database system, a plan like Q_1P_2 shown in Figure 2.4 can significantly outperform plan Q_1P_1 [108]. Q_1P_2 exploits certain properties arising from partitioning in the given setting:

- Tuples in partition R_1 can join only with $S_{12} \cup S_{13}$ and $T_1 \cup T_2$. Similarly, tuples in partition R_2 can join only with $S_{22} \cup S_{23}$ and T_3. Thus, the full $R \bowtie S \bowtie T$ join can be broken up into smaller and more efficient partition-wise joins where the best join order for $R_1 \bowtie (S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ can be different from that for $R_2 \bowtie (S_{22} \cup S_{23}) \bowtie T_3$. One likely reason is change in the data properties of tables S and T over time, causing variations in statistics across partitions.

- The best choice of join operators for $R_1 \bowtie (S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ may differ from that for $R_2 \bowtie (S_{22} \cup S_{23}) \bowtie T_3$, e.g., due to storage or physical design differences across partitions (e.g., index created on one partition but not on another).

Apart from the large size of the plan space, an additional challenge while selecting parallel query plans comes from the cost model used to estimate the cost of plans. The cost metric introduced by the classic System R query optimizer measures the total amount of work done during plan execution [173].

In a serial plan execution, the total work also corresponds to the latency of plan execution. (Latency measures the time to complete the plan execution.) However, the total work and latency metrics are different in the execution of a parallel plan. Unfortunately, the latency metric violates a fundamental assumption made in the dynamic-programming-based plan selection algorithm introduced by the System R query optimizer [93]. (Example 3 in [93] gives an illustration of how the latency metric violates the principle of optimality needed by the dynamic-programming-based plan selection algorithm in System R.) The optimal parallel execution plan for the latency metric may not be composed of optimal subplans for the same metric.

Because of the complexity posed by the plan space and cost metrics, a variety of plan selection algorithms have been proposed for parallel database systems. Deterministic algorithms proposed include those based on dynamic programming [93], flow optimization [74], as well as greedy approaches [143]. Randomized algorithms proposed include those based on simulated annealing [134] and genetic search [179].

Finally, errors can arise while estimating plan execution costs because of reasons such as unknown data properties or uncertain resource availability. This problem is more acute in parallel database systems compared to centralized database systems. Adaptive plan selection has been proposed as a way to address this problem [103].

2.5 Scheduling

As part of the execution of a parallel query plan, the plan is broken down into a set of executable tasks. These tasks are scheduled to run on the nodes of the parallel database system. The tasks usually form a directed acyclic graph (DAG) based on producer-consumer relationships. Subject to the possibility of pipelined parallel execution, a task is ready to run after all its ancestor tasks in the DAG are complete.

Two common approaches are used to decide which node to schedule a task on. In the first approach, this decision is made during plan selection. The decision is often simplified because shared-nothing parallel database systems are predominantly based on *function shipping*

as opposed to *data shipping*. The goal of function shipping is to run the computation as close to where the data is stored so that little data movement is incurred. Most systems that use function shipping try to run any computation on the node where the data needed by the computation resides. Some parallel database systems—e.g., Netezza [116]—can run computations on special-purpose *database accelerator cards* that access data while data is being moved from disk to CPU.

A disadvantage of this approach is that it increases the complexity of an already large search space for parallel execution plans. Systems like the IBM DB2 Parallel Edition which use this approach use heuristics to limit the size of the search space [33]. For example, for all the possible subsets of nodes that can be used to run tasks of a join, only a limited subset is considered: all the nodes, the nodes on which each of the joining tables is partitioned, and a few others [33].

The second approach first determines the execution plan and then uses a scheduling algorithm to determine the schedule. The scheduling algorithm may be *static* or *dynamic*. Static scheduling algorithms determine the schedule of all tasks in the plan before running any one of the tasks. The problem of finding the best static schedule is NP-Hard and approximation algorithms have been proposed for the problem [61].

The *shelf algorithm* was proposed for static scheduling in [187]. The basic approach here begins with a fixed number of shelves, each of which will be assigned one task initially. Each shelf will eventually contain one or more tasks that are to be executed concurrently by all nodes. For each of the remaining tasks that have not been assigned a shelf, the task is either assigned an existing shelf or a new shelf without violating the precedence constraints inherent in the task DAG. When a task is added to an existing shelf, it will be competing for resources with the other tasks in the shelf. As tasks are assigned, the total number of shelves may increase. The initial number of shelves is determined by the height of the task DAG since each task along the longest path in the DAG must be processed one after the other due to precedence constraints.

Static scheduling algorithms cannot react to issues that arise during task execution. For example, the load on a node may increase, which can make the execution time of tasks on that node much larger than

expected [51]. Dynamic scheduling algorithms can react to such problem by migrating task execution from one node to another at run-time. For example, the algorithms proposed in [144] and [145] migrate tasks from loaded nodes to idle nodes at run-time.

2.6 Resource Management

Resource management in parallel database systems happens at two levels: at the level of individual nodes and at the level of the entire system. Most parallel database systems are implemented by building on the software of a centralized database system at each node. Thus, many configuration parameters can be set independently—e.g., size of memory buffers, maximum number of concurrent processes (*MPL*), and the size of log files—at each node. This feature allows the database administrator to tune the performance of each individual node based on the differences in hardware capacities, database partition sizes, and workloads across the nodes in the system [33].

Resource management at the level of the entire system can be done through techniques such as *workload differentiation*—e.g., identifying which queries are short-running and which ones are long-running, and allocating resources appropriately to meet the respective requirements of these query classes [129, 130]—and *admission control* which limits the number of queries that can be running concurrently in the system so that every query gets some guaranteed fraction of the total resources during execution [46, 163].

Another important dimension of resource management is how the system can scale up in order to continue to meet performance requirements as the system load increases. The load increase may be in terms of the size of input data stored in the system and processed by queries, the number of concurrent queries that the system needs to run, the number of concurrent users who access the system, or some combination of these factors. The objective of shared-nothing parallel systems is to provide linear scalability. For example, if the input data size increases by 2x, then a linearly-scalable system should be able to maintain the current query performance by adding 2x more nodes.

Partitioned parallel processing is the key enabler of linear scalability in parallel database systems. One challenge here is that data may have to be repartitioned in order to make the best use of resources when nodes are added to or removed from the system. Most commercial parallel database systems support online reorganization where database administration utilities like repartitioning can be done incrementally without the need to quiesce query processing [33].

2.7 Fault Tolerance

One of the disadvantages of shared-nothing parallel processing is that there are many more causes of failures in these systems compared to centralized systems. Failures cause two important effects that the parallel database system needs to deal with. The first effect is *task failure* where a task that was assigned some work in a query plan fails to complete the work correctly. Task failure can happen due to many root causes. For example, a node may crash due to a hardware problem, a software bug, or due to a misconfiguration done by a database administrator, killing all the tasks that were running on that node. Or, a task may be killed by the operating system because the task's memory usage exceeded the memory allocated to the task.

Parallel database systems have traditionally had a coarse-grained approach to deal with task failures. The predominant approach is to convert task failures into query failures, and then resubmit the query. It is common to have the coordinator task for the query run two-phase commit across all nodes where tasks for the query ran. If all tasks completed without failure, then the two-phase commit will declare the query as successfully completed; otherwise, the query has failed and will have to be rerun [33].

We will see in Chapters 4 and 5 how MapReduce and Dataflow systems implement fine-grained fault tolerance at the task level. For example, the failure of a few tasks in a MapReduce job does not automatically lead to a job failure. The execution of the MapReduce job can be completed by rerunning successfully only the few tasks that may have failed during the job execution. Such fine-grained fault tolerance

is essential when jobs are long-running. However, when most queries are short-running, the additional complexity that comes with supporting fine-grained fault tolerance may not be desirable over the simpler alternative of resubmitting failed queries.

It is possible to implement fine-grained fault tolerance techniques in parallel database systems. For example, *Osprey* is a shared nothing parallel database system whose design is motivated by MapReduce systems [45]. Osprey divides running queries into subqueries, and replicates data such that each subquery can be rerun on a different node if the node initially responsible fails or returns too slowly.

Osprey is implemented using a middleware approach with only a small amount of custom code to handle cluster coordination. An independent database system is run on each node in the system. Tables are partitioned among nodes and each partition is replicated on several nodes. A coordinator node acts as a standard SQL interface to users and applications. The coordinator node transforms an input SQL query into a set of subqueries that are then executed on the worker nodes. Each subquery represents only a small fraction of the total execution of the query. Worker nodes are assigned a new subquery as they finish their current one. In this greedy approach, the amount of work lost due to node failure is small (at most one subquery's work), and the system is automatically load balanced because slow nodes will be assigned fewer subqueries.

The second type of failure causes data to become inaccessible by tasks, or worse, become lost irretrievably. These failures can have more serious consequences. Hardware problems such as errors in magnetic media (*bit rot*), erratic disk-arm movements or power supplies, and bit flips in CPU or RAM due to alpha particles can cause bits of data to change from what they are supposed to be [29]. Or, network partitions can arise during query execution which make it impossible for a task (say, a task running as part of a repartitioned join operator) running on one node to retrieve its input data from another node.

Data replication is used to prevent data availability problems. Parallel database systems replicate table partitions on more than one node. If one node is down, then the table partitions stored on that node can be

retrieved from other nodes. Several techniques have been proposed for data availability in the face of node failures, e.g., mirrored disks, data clustering, disk arrays with redundant check information, and chained declustering [43, 165, 114].

2.8 System Administration

Recall from §1.3 that system administration refers to the activities where additional human effort may be needed to keep the system running smoothly while the system serves the needs of multiple users and applications. It is typical to have dedicated *database administrators (DBAs)* for system administration in deployments of parallel database systems. DBAs have access to good tools provided by the database vendors and other companies for a variety of administrative tasks like performance monitoring, diagnosing the cause of poor query or workload performance, and system recovery from hardware failures (e.g., [60, 78]).

Performance tuning and capacity planning for parallel database systems can be challenging because of the large number of tuning choices involved. For example, the space of tuning choices in classic parallel database systems includes:

- Applying hints to change the query plan when the database query optimizer picks a poor plan due to limited statistics, errors in the cost model, or contention for system resources.
- Choosing the right set of indexes in order to balance the reduction in query execution times with the increase in data load and update times.
- Manipulating the database schema using techniques such as denormalization in order to make query execution more efficient.
- Controlling the layout of data through appropriate choices of declustering, partitioning, and replication.
- Picking hardware with the right amount of processing, memory, disk, and network bandwidth.

- Choosing the settings of server parameters like buffer pool size and join heap size.
- Finding the right time to schedule maintenance tasks like index rebuilds and storage defragmentation so that efficient query performance can be achieved while reducing resource contention between query execution and administrative tasks.
- Setting the thresholds that control the policies for admission control and load balancing.

A parallel database system may not have a unique workload running all the time. This factor increases the complexity of performance tuning and capacity planning. For example, a common scenario is to have several applications issuing complex queries against the database during the day. At night, there may be a batch window of time when the database is updated, e.g., through insertion of new data to the fact tables and updates to the dimension tables.

If this overall workload is treated as a single set of queries and updates—without accounting for the temporal separation between the queries and the updates—then it is quite possible that no suitable set of indexes can be found to improve the overall workload performance. While some indexes may speed up the queries during the day, the update cost incurred in the night for these indexes may far outweigh their benefits [8].

3

Columnar Database Systems

Columnar parallel database systems store entire columns of tables together on disk unlike the row-at-a-time storage of classic parallel database systems. Columnar systems excel at data-warehousing-type applications, where data is added in bulk but typically not modified much (if at all), and the typical access pattern is to scan through large parts of the data to perform aggregations and joins.

Research on columnar database systems has a long history dating back to the 1970s. Decomposing records into smaller subrecords and storing them in separate files was studied by Hoffer and Severance in [112]. A fully decomposed storage model (*DSM*) where each column is stored in a separate file was studied by Copeland and Khoshafian in [68].

MonetDB, which has been in development at CWI since the early nineties, was one of the first database system prototypes that fully embraced columnar storage as its core data model [39]. Sybase IQ was launched in 1996 as a commercial columnar database system [147]. The 2000s saw a number of new columnar database systems such as C-Store [181], Infobright [118], ParAccel [164], VectorWise [206], Vertica [133], and Amazon RedShift [12].

3.1 Data Model and Interfaces

Similar to classic parallel databases, columnar database systems use the popular relational data model where data consists of tables containing tuples with a fixed set of attributes. Columnar database systems also share the same query interface, namely SQL, along with various standards like JDBC and ODBC. The main difference between columnar database systems and the classic parallel database systems discussed in Chapter 2 is in the storage data layout used for tables.

3.2 Storage Layer

In a pure columnar data layout, each column is stored contiguously at a separate location on disk. It is common to use large disk pages or read units in order to amortize disk head seeks when scanning multiple columns on hard drives [2].

3.2.1 Column-oriented Data Layout

One common layout is to store a table of n attributes in n files. Each file corresponds to a column and stores tuples of the form $\langle k, v \rangle$ [39]. (Each of these files is called a *binary association table*, or *BAT*, in MonetDB [117].) Here, the key k is the unique identifier for a tuple, and v is the value of the corresponding attribute in the tuple with identifier k. Tuple identifiers may be generated automatically by the columnar database system based on the insertion order of tuples. An entire tuple with tuple identifier k can be reconstructed by bringing together all the attribute values stored for k.

It is possible to reduce the storage needs by eliminating the explicit storage of tuple identifiers [133, 181]. Instead, the tuple identifier can be derived implicitly based on the position of each attribute value in the file. Vertica stores two files per column [133]. One file contains the attribute values. The other file is called a *position index*. For each disk block in the file containing the attribute values, the position index stores some metadata such as the start position, minimum value, and maximum value for the attribute values stored in the disk block. The

position index helps with tuple reconstruction as well as eliminating reads of disk blocks during query processing. The position index is usually orders of magnitude smaller in size compared to the total size of the attribute values. Furthermore, removing the storage of tuple identifiers leads to more densely packed columnar storage [2, 133].

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.

When different projections can have different sort orders, and tuple identifiers are not stored as part of the projections, a *join index* is needed for reconstructing tuples across projections [181]. A join index from a projection P_1 to a projection P_2 in a table T contains entries of the form $\langle k_1, k_2 \rangle$. Here, k_1 is the identifier of the tuple in P_1 corresponding to the tuple with identifier k_2 in P_2. Note that k_1 and k_2 represent different attribute sets corresponding to the same tuple in table T. A join index from a projection P_1 to a projection P_2 can avoid explicitly storing the tuple identifier k_1, and instead derive the identifier from the position in the join index.

In practice and in experiments with early prototypes of Vertica, it was found that the overheads of join indices far exceeded the benefits of using them [133]. These indices were complex to implement and the run-time cost of reconstructing full tuples during parallel query execution was high. In addition, explicitly storing tuple identifiers consumed significant disk space for large tables. Thus, Vertica does not use join indices. Instead, Vertica implements the concept of *super projections* [133]. A super projection contains every column of the table.

A projection is not limited to contain the attributes from a single table only. Instead, the concept of denormalization can be used to create projections that contain attributes from multiple joining tables. C-Store proposed the creation of *per-join projections* that contain attributes from a table F as well as attributes from tables $D_1, ..., D_n$ with which

F has a many-one joining relationship [181]. Commonly, F is a fact table and $D_1, ..., D_n$ are the corresponding dimension tables. Pre-join projections can eliminate the need for joins at run-time.

However, in deployments of Vertica, it has been found that *pre-join projections* are uncommon because of three reasons: (i) the performance of joins in columnar database systems are very good; (ii) most customers are unwilling to slow down bulk data loads in order to optimize such joins; and (iii) joins done during data load offer fewer optimization opportunities than joins done at query time because the database knows nothing apriori about the data in the load stream [133].

3.2.2 Column-oriented Compression

An important advantage of columnar data layouts is that columns can be stored densely on disk, especially by using light-weight compression schemes. The intuitive argument for why columnar layouts compress data well is that compression algorithms perform better on data with low information entropy (high data value locality) [2].

A spectrum of basic and hybrid compression techniques are supported by columnar databases. These include [133, 181]:

- *Run Length Encoding (RLE):* Here, 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 low cardinality columns that are sorted.
- *Delta Value:* Here, each attribute value is stored as a difference from the smallest value. While such an approach leads to variable-sized data representation, it is useful when the differences can be stored in fewer bytes than the original attribute values. A block-oriented version of this compression scheme would store each attribute value in a data block as a difference from the smallest value in the data block. This type of compression is best for many-valued, unsorted integer or integer-based columns.
- *Compressed Delta Range:* Here, 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:* Here, 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. A block-oriented version of this compression scheme would create a dictionary per block. The *Blink* system introduced a compression scheme called *frequency partitioning* that partitions the domain underlying each column, based upon the frequency with which values occur in that column at load time [32]. Blink creates a separate dictionary for each partition. Since each dictionary needs to only represent the values of its partition, each dictionary can use shorter codes for the actual values. Dictionary-based compression is a general-purpose scheme, but it is good for few-valued, unsorted columns [133, 181].

- *Bitmap:* Here, 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 [181]. For example, given a column of integers 0,0,1,1,2,1,0,2,1, bitmap-based compression will encode this column as three pairs: (0, 110000100), (1, 001101001), and (2,000010010). RLE can be further applied to compress each bitmap.

Apart from the above compression schemes, hybrid combinations of these schemes are also possible. For example, the *Compressed Common Delta* scheme used in Vertica builds a dictionary of all the deltas in each block [133]. This type is best for sorted data with predictable sequences and occasional sequence breaks (e.g., timestamps recorded at periodic intervals or primary keys).

3.2.3 Updates

One of the disadvantages of columnar storage is that write operations cause two types of overheads. First, tuples inserted into a table have to be split into their component attributes and each attribute (or group of attributes in the case of projections) must be written separately. Thus, tuple insertions cause more logical writes in columnar database systems than in the row-based storage of classic parallel database systems.

The more serious overhead is that the densely-packed and compressed data layout in columnar database systems makes insertions to tuples within a disk block nearly impossible. A variety of schemes have been proposed in columnar database systems to reduce the overheads caused by writes. The main theme in these schemes is to buffer the writes temporarily, and then to apply them in bulk to update the columnar data layout.

C-Store and Vertica pioneered the use of a two-storage scheme consisting of a read-optimized store (RS) and a write-optimized store (WS), along with a *tuple mover* that is responsible for moving tuples from the write-optimized store to the read-optimized store [133, 181]. All write operations—inserts, deletes, and updates—are first done to the WS which is memory-resident. RS, as the name implies, is optimized for read and supports only a very restricted form of write, namely the batch movement of records from WS to RS. Queries access data in both storage systems.

Inserts are sent to WS, while deletes are marked in RS for later purging by the tuple mover [133, 181]. Updates are implemented as an insert and a delete. In order to support a high-speed tuple mover, C-Store and Vertica borrow the *LSM-tree* concept [160]. Basically, the tuple mover supports a mergeout process that moves tuples from WS to RS in bulk by an efficient method of merging ordered WS data with large RS blocks, resulting in a new copy of RS when the operation completes.

Systems like *SAP HANA* [88] and *VectorWise* [206] use data structures based on deltas. For example, in HANA, every table has a delta storage which is designed to strike a good balance between high update rates and good read performance. Dictionary compression is used here with the dictionary stored in a *cache-sensitive B+-Tree (CSB+-Tree)* [88]. The delta storage is merged periodically into the main data storage.

3.2.4 Partitioning

Data partitioning techniques used in parallel columnar database systems are fundamentally similar to the techniques used in the classic

parallel database systems described in Chapter 2. The same is true for the techniques used in declustering, assignment, and replication.

3.3 Execution Engine

The columnar data layout gives rise to a distinct space of execution plans in columnar parallel database systems compared to classic parallel database systems. In the MonetDB system, a new *columnar algebra* was developed to represent operations on columnar data layouts [117]. Expressions in relational algebra and SQL are converted into columnar algebra and then compiled into efficient executable plans over a columnar data layout.

We will describe three aspects of the execution plan space in columnar parallel database systems that provide opportunities for highly efficient execution: (a) operations on compressed columns, (b) vectorized operations, and (c) late materialization.

3.3.1 Operations on Compressed Columns

It is highly desirable to have an operator operate on the compressed representation of its input whenever possible in order to avoid the cost of decompression; at least until query results need to be returned back to the user or application that issued the query. The ability to operate directly on the 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-based compression technique described in §3.2.2. 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.

It is indeed possible for complex operators like range filters, aggregations, and joins to operate directly on compressed data. Some of the interesting possibilities arise because of novel dictionary-based compression techniques (recall §3.2.2). All the tuples in a block may be eliminated from consideration if the code for the constant in a filter predicate of the form "attribute = constant" cannot be found in that

block's dictionary. For example, a predicate "customerID = 42" cannot be true for any tuple in any block not having the code for 42 in its dictionary for the attribute customerID.

Order-preserving dictionary-based compression techniques are used in systems like Blink [32] and MonetDB [117]. Here, encoded values are assigned in each dictionary in an order-preserving way so that range and equality predicates can be applied directly to the encoded values. For example, Blink converts complex predicates on column values, such as LIKE predicates, into IN-lists in the code space by evaluating the predicate on each element of the dictionary. Due to order-preserving dictionary coding, all the standard predicates $(=, \neq, \leq, \geq)$ map to integer comparisons between codes, irrespective of the data type. As a result, even predicates containing arbitrary conjunctions and disjunctions of atomic predicates can be evaluated in Blink using register-wide mask and compare instructions provided by processors. Data is decompressed only when necessary, e.g., when character or numeric expressions must be calculated.

3.3.2 Vectorized Processing

While vectorized processing is not unique to columnar database systems, columnar layouts lend themselves naturally to operators processing 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 the 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.

In addition, other overheads in operators such as function calls, type casting, various metadata handling costs, etc., can all be reduced by processing inputs in large chunks at a time. For example, SAP HANA accelerates data scans significantly by using SIMD algorithms working directly on the compressed data [88].

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 [40]. 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. Like SAP HANA, X100 shows significant performance increases when vectorized processing is combined with just-in-time light-weight compression.

3.3.3 Late Materialization

Tuple reconstruction is expensive in columnar database systems since information about a logical tuple is stored in multiple locations on disk, yet most queries access more than one attribute from a tuple [2]. 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 [4].

For example, MonetDB uses late tuple reconstruction [117]. 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.

3.4 Query Optimization

Despite the very different data layout, standard cost-based query optimization techniques from the classic parallel database systems apply well in the columnar setting. At the same time, some additional factors have to be accounted for in columnar database systems:

- Picking operator implementations that can operate on the compressed data as discussed in §3.3.1.
- Late materialization of tuples as discussed in §3.3.3.

- Choosing the best projection(s) as input to the scan operations in the leaves of the query plan. Important factors that impact this decision include the attributes in the projection, the size of the projection, the types of compression used in the projection for the attributes of interest, and the sort order of the projection. For example, Vertica's optimizer tends to join projections with highly compressed and sorted predicate and join columns first; to make sure that not only fast scans and merge joins on compressed columns are applied first, but also that the cardinality of the data for later joins is reduced [133].

3.5 Scheduling

All the scheduling challenges listed for classic parallel database systems in Chapter 2 exist in columnar parallel database systems. Furthermore, since most popular columnar systems are of recent origin, they incorporate scheduling policies aimed at recent hardware trends such as nodes with a large number of CPU cores [133, 206]. For example, systems like VectorWise and Vertica have execution engines that are multi-threaded and pipelined: more than one operator can be running at any time, and more than one thread can be executing the code for any individual operator.

3.6 Resource Management

Query execution over a columnar data layout poses some unique challenges. While this layout allows for a more fine-grained data access pattern, it can result in performance overheads to allocate the memory per column. This overhead can be significant when a large number of columns are handled, e.g., when constructing a single result row consisting of hundreds of columns [88].

Vectorized and pipelined execution engines used in columnar database systems cause sharing of physical resources, especially memory, among multiple operators. Thus, careful resource management is needed in order to avoid overheads such as unnecessary spills to disk. Systems like Vertica partition query execution plans into multiple zones

that cannot all be executing at the same time. Downstream operators are able to reclaim resources previously used by upstream operators, allowing each operator more memory than if a pessimistic assumption were to be made that all operators would need their resources at the same time [133]. Furthermore, during query compile time, each operator can be assigned a memory budget based on a user-defined workload management policy, the resources available, and what each operator is going to do.

Memory management becomes further complicated in systems that focus on providing optimal performance for concurrent scan-intensive queries that are common in analytical workloads. For example, the X100 project uses *cooperative scans* [184, 207] where an *Active Buffer Manager (ABM)* determines the order to fetch tuples at run-time depending on the interest of all concurrent queries. The ABM can cause table scans to return tuples out of order. The ABM is a complex component since it has to strike a balance between optimizing both the average query latency and throughput. Thus, the product version of VectorWise uses a less radical, but still highly effective, variant of intelligent data buffering [206].

Further resource management challenges arise in columnar database systems that have separate write-optimized and read-optimized stores [133, 181]. Recall from §3.2.3 that a tuple mover is responsible for moving tuples in bulk from the write-optimized store to the read-optimized store. The tuple mover must balance its work so that it is not overzealous (which can cause unnecessary resource contention and also create many small files in the read-optimized store) but also not too lazy (resulting in unnecessary spills and many small files in the write-optimized store) [133].

3.7 Fault Tolerance

The techniques to deal with failures—e.g., query restarts on failure or replication to avoid data loss—remain the same across classic parallel database systems and columnar database systems. For example, Vertica replicates data to provide fault tolerance. Each projection must

have at least one buddy projection containing the same columns. The horizontal partitioning and assignment ensures that no row is stored on the same node by a projection and its buddy projection. When a node is down, the buddy projection is employed for data access as well as to create a new replica of the projection [133].

C-Store and Vertica provide the notion of *K-safety*: with K or fewer nodes down, the cluster is guaranteed to remain available. To achieve K-safety, the database projection design must ensure at least K+1 copies of each segment are present on different nodes such that a failure of any K nodes leaves at least one copy available. The failure of K+1 nodes does not guarantee a database shutdown. Only when node failures actually cause data to become unavailable will the database shut down until the failures can be repaired and consistency restored via recovery. A Vertica cluster will also perform a safety shutdown if N/2 nodes are lost where N is the number of nodes in the cluster. The agreement protocol requires a $N/2 + 1$ quorum to protect against network partitions and avoid a split brain effect where two halves of the cluster continue to operate independently [133, 181].

3.8 System Administration

System administration for columnar database systems is mostly similar to that for classic parallel database systems discussed in §2.8. However, some notable differences arise due to differences in the space of critical tuning decisions. For example, recall that in systems like C-Store and Vertica, the choice of data layout includes selecting the right columnar projections and sort order per projection. Two other important choices are with respect to the type of compression for each column and the scheduling of tuple movement between the read-optimized and write-optimized stores. Index selection is usually unimportant.

4

MapReduce Systems

MapReduce is a relatively young framework—both a programming model and an associated run-time system—for large-scale data processing [73]. *Hadoop* is the most popular open-source implementation of a MapReduce framework that follows the design laid out in the original paper [72]. 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.

Even though the MapReduce programming model is highly flexible, it has been found to be too low-level for routine use by practitioners such as data analysts, statisticians, and scientists [159, 189]. 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* (with a spreadsheet interface). The typical Hadoop stack is shown in Figure 4.1.

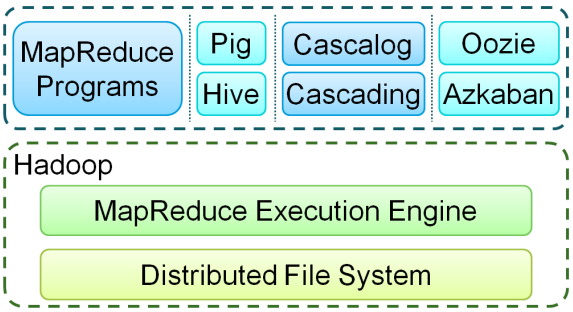

Figure 4.1: Typical Hadoop software stack.

4.1 Data Model and Interfaces

MapReduce systems typically process data directly from files, permitting data to be in any arbitrary format. Hence, MapReduce systems are capable of processing unstructured, semi-structured, and structured data alike.

The MapReduce programming model consists of two functions: $map(k_1, v_1)$ and $reduce(k_2, list(v_2))$ [192]. Users can implement their own processing logic by specifying a customized $map()$ and a $reduce()$ 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 $\langle k_2, v_2 \rangle$. 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. $reduce(k_2, list(v_2))$ outputs zero or more key-value pairs of the form $\langle k_3, v_3 \rangle$. The MapReduce programming model also allows other functions 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.

A given MapReduce program may be expressed in one among a variety of programming languages like Java, C++, Python, or Ruby; and then connected to form a workflow using a workflow scheduler such as Oozie [161]. Alternatively, the MapReduce jobs can be generated

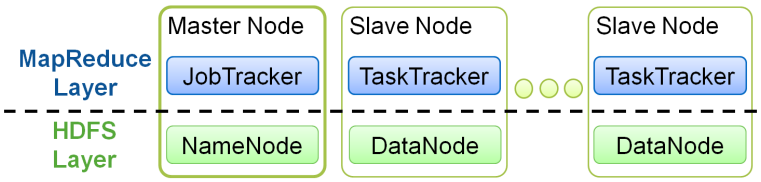

Figure 4.2: Hadoop architecture.

automatically using compilers for higher-level languages like Pig Latin [159], HiveQL [189], JAQL [36], and Cascading [52].

4.2 Storage Layer

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 [176]. Alternatively, a Hadoop cluster can process data from other file systems like the MapR File System [149], CloudStore (previously Kosmos File System) [127], Amazon Simple Storage Service (S3) [13], and Windows Azure Blob Storage [50].

HDFS is designed to be resilient to hardware failures and focuses more on batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. 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 4.2). 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.

The modularity of the storage architecture enables higher-level systems to introduce their own features or extensions on top of the distributed file system. For example, Hive organizes and stores the data into partitioned tables [189]. Hive tables are analogous to ta-

bles in relational databases and are represented using HDFS directories. Partitions are then created using subdirectories whereas 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. Hive's Metastore inspired the creation of HCatalog, a new table and storage management service for data created using Hadoop [24]. HCatalog provides a unified table abstraction and interoperability across data processing tools such as MapReduce, Pig, and Hive.

Indexing in HDFS: Hadoop++ provides indexing functionality for data stored in HDFS by means of an approach called *Trojan Indexes* [80]. This approach is based on user-defined functions so no changes are required to the underlying Hadoop system. The indexing information is stored as additional metadata in the HDFS blocks that are read by map tasks. The information is added when tables are written to HDFS so that no overhead is caused during query processing.

Index creation in Hadoop++ can increase the data loading time. This problem is addressed by HAIL [81] which also 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. For fault tolerance purposes, HDFS maintains k replicas for every HDFS block. ($k = 3$ by default.) HAIL builds a different clustered index in each replica. The most suitable index for a query is selected at run-time, and the corresponding replica of the blocks are read by the map tasks in HAIL.

Data collocation: A serious limitation of MapReduce systems is that HDFS does not support the ability to collocate data. Because of this limitation, query processing systems on top of MapReduce—e.g., Hive, Pig, and JAQL—cannot support collocated join operators. Recall from Chapter 2 that collocated joins are one of the most efficient ways to do large joins in parallel database systems. Hadoop++ and CoHadoop [84] provide two different techniques to collocate data in MapReduce systems.

Hadoop++ uses the same "trojan" approach as in trojan indexes in order to co-partition and collocate data at load time [80]. Thus,

blocks of HDFS can now contain data from multiple tables. With this approach, joins can be processed at the map side rather than at the reduce side. Map-side joins avoid the overhead of sorting and shuffling data.

CoHadoop adds a file-locator attribute to HDFS files and implements 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 very efficiently using a map-side join operator that behaves like the collocated join operator in parallel database systems. CoHadoop relies on applications to set the locator attributes for the files that they create.

Data layouts: It is also possible to implement columnar data layouts in HDFS. Systems like *Llama* [140] and *CIF* [89] use a pure column-oriented design while *Cheetah* [64], Hadoop++ [80], and *RCFile* [105] use a hybrid row-column design based on *PAX* [10]. Llama partitions attributes into vertical groups like the projections in C-Store and Vertica (recall §3.2.1). 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 to reduce read I/O costs, but may incur run-time costs for tuple reconstruction.

CIF uses a similar design of storing columns in separate files, but its design is different from Llama in many ways. First, CIF partitions the table horizontally and stores each horizontal partition in a separate HDFS directory for independent access in map tasks. Second, CIF uses an extension of HDFS to enable collocation of columns corresponding to the same tuple on the same node. Third, CIF supports some late materialization techniques to reduce tuple reconstruction costs [89].

Cheetah, Hadoop++, and RCFile use a layout where a set of tuples is stored per HDFS block, but a columnar format is used within the HDFS 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 com-

pression, how they treat replicas of the same block, etc. For example, Hadoop++ can use different layouts in different replicas, and choose the best layout at query processing time. Queries that require a large fraction of the columns should use a row-oriented layout, while queries that access fewer columns should use a column-oriented layout [122].

Almost all the above data layouts are inspired by similar data layouts in classic parallel database systems and columnar database systems. *HadoopDB* [5] takes this concept to the extreme by installing a centralized database system on each node of the cluster, and using 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. HadoopDB introduces some advanced partitioning capabilities such as reference-based partitioning which enable multi-way joins to be performed in a collocated fashion [30]. The HadoopDB architecture is further discussed in §4.3.

HDFS alternatives: A number of new distributed file systems are now viable alternatives to HDFS and offer full compatibility with Hadoop MapReduce. The MapR File System [149] and Ceph [191] 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. In addition, MapR allows for mutable data access and is NFS mountable. The Quantcast File System (QFS) [162], which evolved from the Kosmos File System (KFS) [127], 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.

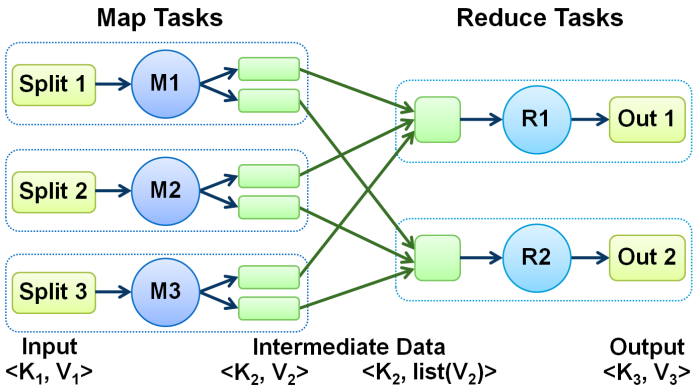

Figure 4.3: Execution of a MapReduce job.

4.3 Execution Engine

MapReduce execution engines represent a new data-processing framework that has emerged in recent years to deal with data at massive scale [72]. Users specify computations over large datasets in terms of Map and Reduce functions, and the underlying run-time system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disk bandwidth.

As shown in Figure 4.2, a Hadoop MapReduce cluster employs a master-slave architecture where one master component (called *JobTracker*) manages a number of slave components (called *TaskTrackers*). Figure 4.3 shows how a MapReduce job is executed on the cluster. 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* 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 *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.

Higher-level systems like Pig [94] and Hive [189] use the underlying MapReduce execution engine in similar ways to process the data. In particular, both Pig and Hive will compile the respective Pig Latin and HiveQL 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. A typical query will result in a directed acyclic graph (DAG) of multiple MapReduce jobs that are executed on the cluster.

The execution of Pig Latin and HiveQL queries as MapReduce jobs results in Pig and Hive being well suited for batch processing of data, similar to Hadoop. These systems thrive when performing long sequential scans in parallel, trying to utilize the entire cluster to the fullest. In addition, both Pig and Hive are read-based, and therefore not appropriate for online transaction processing which typically involves a high percentage of random write operations. Finally, since MapReduce is the basic unit of execution, certain optimization opportunities like better join processing can be missed (discussed further in §4.4).

4.3.1 Alternative MapReduce Execution Engines

As discussed earlier in §4.2, HadoopDB is a hybrid system that combines the best features of parallel database systems and MapReduce systems [5]. HadoopDB runs a centralized database system on each node and uses Hadoop primarily as the execution plan scheduling and fault-tolerance layer. HadoopDB introduced the concept of *split query execution* where a query submitted by a user or application will be converted into an execution plan where some parts of the plan would run as queries in the database and other parts would run as map and reduce tasks in Hadoop [30]. The best such splitting of work will be identified during plan generation. For example, if two joining tables are stored partitioned and collocated, then HadoopDB can perform an efficient collocated join like in parallel database systems.

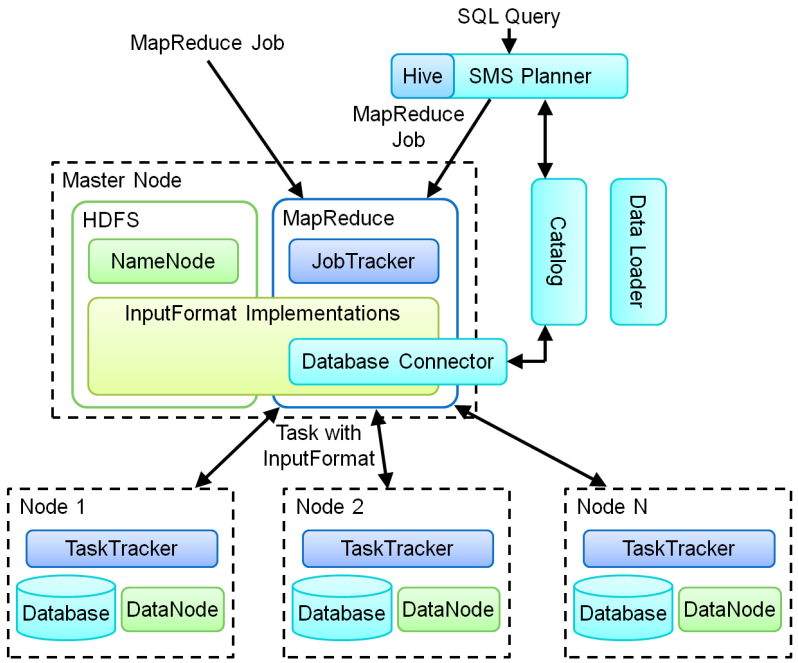

Figure 4.4: Architecture of the HadoopDB system, which has a hybrid design that combines MapReduce systems and centralized databases.

Figure 4.4 shows the architecture of HadoopDB. The database connector is an interface between the TaskTrackers in Hadoop and the individual database systems. The database connector can connect to a database, execute a SQL query, and return the query result in the form of key-value pairs. Metadata about the databases is stored in the system catalog. The catalog maintains system metadata such as connection parameters, schema and statistics of the tables stored, locations of replicas, and data partitioning properties.

The SMS (SQL to MapReduce to SQL) planner extends Hive and produces split-execution query plans that can exploit features provided by the available database systems. The data loader globally repartitions tables based on a partition key, breaks down single-node partitions further into smaller partitions, and bulk loads the single-node databases with these smaller partitions.

Clydesdale is a system that was built to demonstrate that a simple architecture that leverages existing techniques from parallel database systems can provide substantial performance benefits for query processing in MapReduce systems [123]. Clydesdale is aimed at workloads where the data fits a star schema. The fact table is stored on HDFS using CIF [89]. Copies of the dimension tables are also stored in HDFS, but are replicated to the local storage on each node in the cluster.

SQL queries submitted to Clydesdale are converted into a query plan composed of MapReduce jobs. Join processing is done in the map phase such that map tasks apply predicates to the dimension tables and build hash tables in the setup phase. Then, the map tasks join the tuples in the fact table by probing the hash tables. The reduce phase is responsible for grouping and aggregation. Clydesdale draws on several strategies for performance improvement: careful use of multi-core parallelism, employing a tailored star-join plan instead of joining tables in a pairwise manner, and columnar storage. All these techniques together give considerable improvement over processing the queries directly on a MapReduce system.

Sailfish [170] is an alternative MapReduce framework for large scale data processing whose design is centered around aggregating intermediate data, i.e., data produced by map tasks and consumed later by reduce tasks. In particular, the output of map tasks (which consists of key/value pairs) is first partitioned by key and then aggregated on a per-partition basis using a new file system abstraction, called *I-files* (Intermediate data files). I-files support batching of data written by multiple writers and read by multiple readers. This intermediate data is sorted and augmented with an index to support key-based retrieval. Based on the distribution of keys across the I-files, the number of reduce tasks as well as the key-range assignments to tasks can be determined dynamically in a data-dependent manner.

4.4 Query Optimization

Being a much newer technology, MapReduce engines significantly lack principled optimization techniques compared to database systems.

Hence, the MapReduce stack (see Figure 4.1) can be poorer in performance compared to a database system running on the same amount of cluster resources [90, 166]. In particular, the reliance on manual tuning and absence of cost-based optimization can result in missed opportunities for better join processing or partitioning. A number of ongoing efforts are addressing this issue through optimization opportunities arising at different levels of the MapReduce stack.

For higher levels of the MapReduce stack that have access to declarative semantics, many optimization techniques inspired by database query optimization and workload tuning have been proposed. Hive and Pig employ *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 [157], reducing the number of MapReduce jobs in a workflow [137], and handling data skew in sorts and joins. The *AQUA* system supports System-R-style join ordering [193]. Improved data layouts [122, 140] and indexing techniques [80, 81] inspired by database storage have also been proposed, as discussed in §4.2.

Lower levels of the MapReduce stack deal with workflows of MapReduce jobs. A MapReduce job may contain *black-box* map and reduce functions expressed in programming languages like Java, Python, and R. Many heavy users of MapReduce, ranging from large companies like Facebook and Yahoo! to small startups, have observed that MapReduce jobs often contain black-box UDFs to implement complex logic like statistical learning algorithms or entity extraction from unstructured data [146, 159]. One of the optimization techniques proposed for this level—exemplified by *HadoopToSQL* [121] and *Manimal* [49]—does static code analysis of MapReduce programs to extract declarative constructs like filters and projections. These constructs are then used for database-style optimization such as projection pushdown, column-based compression, and use of indexes.

Finally, the performance of MapReduce jobs is directly affected by various configuration parameter settings like degree of parallelism and use of compression. Choosing such settings for good job performance is a nontrivial problem and a heavy burden on users [27]. Starfish has

introduced a cost-based optimization framework for MapReduce systems for determining configuration parameter settings as well as the cluster resources to meet desired requirements on execution time and cost for a given analytics workload [110]. Starfish's approach is based on: (i) collecting monitoring information in order to learn the run-time behavior of workloads (profiling), (ii) deploying appropriate models to predict the impact of hypothetical tuning choices on workload behavior, and (iii) using efficient search strategies to find tuning choices that give good workload performance [107, 109].

4.5 Scheduling

The primary goal of scheduling in MapReduce is to maximize data-locality; that is, to schedule the MapReduce tasks to execute on nodes where data reside or as close to those nodes as possible. The original scheduling algorithm in Hadoop was integrated within the JobTracker and was called FIFO (First In, First Out). In FIFO scheduling, the JobTracker pulled jobs from a work queue in order of arrival and scheduled all the tasks from each job for execution on the cluster. This scheduler had no concept of priority or size of the job, but offered simplicity and efficiency.

As MapReduce evolved into a multi-tenant data-processing platform, more schedulers were created in an effort to maximize the workload throughput and cluster utilization. The two most prominent schedulers today are the *Fair Scheduler* [16] and the *Capacity Scheduler* [15], developed by Facebook and Yahoo!, respectively. The core idea behind the Fair Scheduler is to assign resources to jobs such that on average over time, each job gets an equal share of the available resources. Users may assign jobs to pools, with each pool allocated a guaranteed minimum number of Map and Reduce slots. Hence, this scheduler lets short jobs finish in reasonable time while not starving long jobs. The Capacity Scheduler shares similar principles with the Fair Scheduler, but focuses on enforcing cluster capacity sharing among users, rather than among jobs. In capacity scheduling, several queues are created, each with a configurable number of map and reduce slots. Queues that

contain jobs are given their configured capacity, while free capacity in a queue is shared among other queues. Within a queue, scheduling operates based on job priorities.

There exists an extensive amount of research work improving the scheduling policies in Hadoop [168]. Delay scheduling [198] is an extension to the Fair Scheduler that temporarily relaxes fairness to improve data locality by asking jobs to wait for a scheduling opportunity on a node with local data. Dynamic Proportional Share Scheduler [172] supports capacity distribution dynamically among concurrent users, allowing them to adjust the priority levels assigned to their jobs. However, this approach does not guarantee that a job will complete by a specific deadline. Deadline Constraint Scheduler [124] addresses the issue of deadlines but focuses more on increasing system utilization. Finally, Resource Aware Scheduler [196] considers resource availability on a more fine-grained basis to schedule jobs.

One potential issue with the Hadoop platform is that by dividing the tasks across multiple nodes, it is possible for a few slow nodes to rate-limit the rest of the job. To overcome this issue, Hadoop uses a process known as *speculative execution*. In particular, Hadoop schedules redundant copies of some tasks (typically towards the end of a job) for execution across several nodes that are not currently busy. Whichever copy of a task finishes first becomes the definitive copy and the other copies are abandoned.

As discussed earlier, a typical HiveQL or Pig Latin query is parsed, possibly optimized using a rule-based or cost-based approach, and converted into a *MapReduce workflow*. A MapReduce workflow is a directed acyclic graph (DAG) of multiple MapReduce jobs that read one or more input datasets and write one or more output datasets. The jobs in a workflow can exhibit *dataflow dependencies* because of producer-consumer relationships and hence, must be scheduled serially in order of their dependencies. Jobs without such dataflow dependencies can be scheduled to be run concurrently and are said to exhibit *cluster resource dependencies* [139]. The scheduling of jobs within a workflow is typically handled by the higher-level system itself (i.e., Hive and Pig).

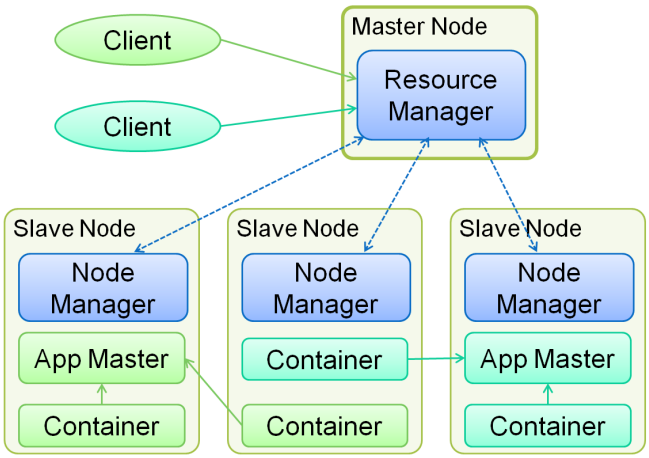

Figure 4.5: Hadoop NextGen (YARN) architecture.

4.6 Resource Management

In the original version of Hadoop, each node in a cluster is statically assigned a predefined number of map and reduce *slots* for running map and reduce tasks concurrently [192]. Hence, the allocation of cluster resources to jobs is done in the form of these slots. This static allocation of slots has the obvious drawback of lowered cluster utilization since slot requirements vary during the MapReduce job life cycle. Typically, there is a high demand for map slots when the job starts, whereas there is a high demand for reduce slots towards the end. However, the simplicity of this scheme simplified the resource management performed by the JobTracker, in addition to the scheduling decisions.

Overall, the Job Tracker in Hadoop has the dual role of managing the cluster resources as well as scheduling and monitoring MapReduce jobs. *Hadoop NextGen* (also known as *MapReduce 2.0* or *YARN*) [22] separates the above two functionalities into two separate entities: a global *ResourceManager* is responsible for allocating resources to running applications, whereas a per-application *ApplicationMaster* manages the application's life-cycle (see Figure 4.5). There is also a per-machine *NodeManager* that manages the user processes on that node.

The ApplicationMaster is a framework-specific library that negotiates resources from the ResourceManager and works with the NodeManager(s) to execute and monitor the tasks. In the YARN design, MapReduce is just one application framework; the design permits building and deploying distributed applications using other frameworks as well.

The resource allocation model in YARN addresses the static allocation deficiencies of the previous Hadoop versions by introducing the notion of *resource containers*. A container represents a specification of node resources—called *attributes*—such as CPU, memory, disk bandwidth, and network bandwidth. In this model, only a minimum and a maximum for each attribute are defined, and ApplicationMasters can request containers with attribute values as multiples of the minimum. Hence, different ApplicationMasters have the ability to request different container sizes at different times, giving rise to new research challenges on how to efficiently and effectively allocate resources among them.

Hadoop also supports dynamic node addition as well as decommissioning of failed or surplus nodes. When a node is added in the cluster, the TaskTracker will notify the JobTracker of its presence and the JobTracker can immediately start scheduling tasks on the new node. On the other hand, when a node is removed from the cluster, the JobTracker will stop receiving heartbeats from the corresponding TaskTracker and after a small period of time, it will stop scheduling tasks on that node.

Even though newly-added nodes can be used almost immediately for executing tasks, they will not contain any data initially. Hence, any map task assigned to the new node will most likely not access local data, introducing the need to *rebalance* the data. HDFS provides a tool for administrators that rebalances data across the nodes in the cluster. In particular, the HDFS Rebalancer analyzes block placement and moves data blocks across DataNodes in order to achieve a uniform distribution of data, while maintaining data availability guarantees.

Hadoop On Demand (HOD) [17] is a system for provisioning and managing virtual clusters on a larger shared physical cluster. Each virtual cluster runs its own Hadoop MapReduce and Hadoop Distributed File System (HDFS) instances, providing better security and performance isolation among the users of each cluster. Under the covers,

HOD uses the Torque resource manager [178] to do the node allocation. On each node, HOD will automatically prepare the configuration files and start the various Hadoop components. HOD is also adaptive in that it can shrink a virtual cluster when the workload changes. In particular, HOD can automatically deallocate nodes from a cluster after it detects no jobs were running for a given time period. This behavior permits the most efficient use of the overall physical cluster resources.

4.7 Fault Tolerance

Fault tolerance features are built in all the layers of the Hadoop stack, including HDFS, MapReduce, and higher-level systems like Pig and Hive. HDFS is designed to reliably store very large files across machines in a large cluster by replicating all blocks of a file across multiple machines [176]. For the common case, when the replication factor is three, HDFS's default placement policy is to put one replica on one node in the local rack and the other two replicas on two different nodes in a different rack. In case of a failure—a DataNode becoming unavailable, a replica becoming corrupted, or a hard disk on a DataNode failing—the NameNode will initiate re-replication of the affected blocks in order to ensure that the replication factor for each block is met. Data corruption is detected via the use of checksum validation, which is performed by the HDFS client by default each time the data is accessed.

The primary way that the Hadoop MapReduce execution engine achieves fault tolerance is through restarting tasks [192]. If a particular task fails or a TaskTracker fails to communicate with the JobTracker after some period of time, then the JobTracker will reschedule the failed task(s) for execution on different TaskTrackers. Failed map tasks are automatically restarted (in other nodes) to process their part of the data again (typically a single file block). Intermediate job data are persisted to disk so that failed reduce tasks can also be restarted without requiring the re-execution of map tasks.

4.8 System Administration

System administration for MapReduce systems is a less established area compared to database administration as done by DBAs. MapReduce system deployments are administered predominantly by cluster operations personnel, some of whom may specialize per system layer (e.g., HDFS operations staff). Some notable differences between parallel database systems and MapReduce systems make it harder to tune MapReduce systems for high performance. For example, it is common in MapReduce systems to interpret data (lazily) at processing time, rather than (eagerly) at loading time. Hence, properties of the input data (e.g., schema) may not be known.

Furthermore, there are many possible ways in which a MapReduce job J in a workload that runs on a MapReduce system could have been generated. A user could have generated J by writing the map and reduce functions in some programming language like Java or Python. J could have been generated by query-based interfaces like Pig or Hive that convert queries specified in some higher-level language to a workflow of MapReduce jobs. J could have been generated by program-based interfaces like Cascading or *FlumeJava* [55] that integrate MapReduce job definitions into popular programming languages. This spectrum of choice in MapReduce job generation increases the complexity of administrative tasks like system monitoring, diagnosing the cause of poor performance, performance tuning, and capacity planning.

5

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 restrictive programming and execution models pose problems from an ease of use as well as a performance perspective for many of these tasks. Consequently, recent *Dataflow* systems are extending the MapReduce framework with a more generalized dataflow-based execution model. In particular, *Spark* [200], *Hyracks* [42], and *Nephele* [34] have been developed to generalize the MapReduce execution model by supporting new primitive operations in addition to Map and Reduce. For example, in the case of Spark, data after every step is stored as *Resilient Distributed Datasets (RDDs)* which can reside in memory rather than be persisted to disk.

A number of systems in this category aim at replacing MapReduce altogether with flexible dataflow-based execution models that can express a wide range of data access and communication patterns. Various dataflow-based execution models have been proposed, including directed acyclic graphs in *Dryad* [119], serving trees in *Dremel* [153], and bulk synchronous parallel processing in *Pregel* [148].

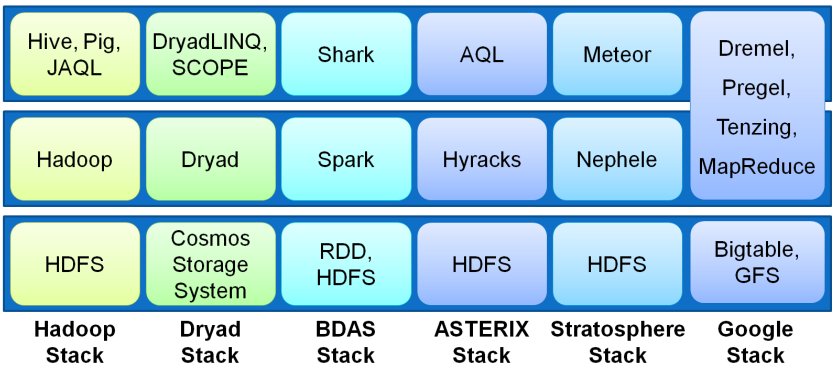

Figure 5.1: Typical dataflow software stacks along with the Hadoop stack.

Figure 5.1 summarizes some common dataflow software stacks comprised of various systems at the storage, execution, and query levels. HDFS is the most popular storage layer shared by most stacks, while the *Cosmos Storage System* [54] is used in the Dryad stack, and *GFS* [96] and *Bigtable* [58] are used in the Google stack. Dryad is the execution engine used predominantly by Microsoft with the higher-level languages *DryadLINQ* [120] and SCOPE [204] completing the Dryad stack. RDD, Spark, and Shark, developed at Berkeley's AMP Lab, complete the Berkeley Data Analytics Stack (BDAS) and have a strong emphasis on utilizing the memory on the compute nodes. ASTERIX and Stratosphere are both open-source platforms for large-scale data analytics with similar execution engines and interfaces. Finally, the Google stack contains more specialized systems including the original MapReduce, Tenzing, Dremel, and Pregel.

5.1 Data Model and Interfaces

Dataflow systems typically work with flexible data models supporting a wide range of data formats such as unstructured or semi-structured text, binary sequence files, JSON, XML, nested structured text, and relational data. The processing of data by the various systems in this category follows a common theme and consists of (a) a set of *computational vertices* that define local processing on a partition of the data,

and (b) a set of *communication edges* that define data transfers or transformations between the vertices. The main difference among the systems is how the vertices and edges are arranged for data processing:

- The *Directed Acyclic Graph (DAG)* pattern is the most popular one. Dryad, Spark, Hyracks, and Nephele are all DAG-based dataflow systems.
- The restricted graph form of a *tree* is used by Dremel in order to run aggregation queries over large datasets in near real time.
- The generalized graph form of a *directed graph* (i.e., with cycles) is used by Pregel to support large-scale iterative processing.

The query interfaces exposed by the various dataflow systems can be divided into three categories: (i) *domain-specific languages* that declare a programming language API and are used by Dryad, Hyracks, Nephele, and Pregel, (ii) *functional programming interfaces* that are used by DryadLINQ and Spark, and (iii) *declarative languages* that resemble SQL and are used by SCOPE, Shark, AQL, Meteor, Tenzing, and Dremel. Depending on their category, the various query interfaces provide interesting tradeoffs between expressiveness, declarativity, and optimizability. The lower-level, domain-specific languages exposed by systems like Dryad and Nephele, exhibit the highest level of expressiveness but at the expense of programming simplicity and optimization opportunities. At the other end of the spectrum, declarative languages like SCOPE have well-defined but constrained semantics that are amenable to similar optimizations as in relational databases (discussed in §5.4). Functional programming interfaces offered by DryadLINQ and Spark have many declarative constructs but also have a strong emphasis on user-defined functions, allowing the user to trade declarativity and optimizability for expressiveness.

Dryad's domain-specific language, implemented via a C++ library, is used to create and model a Dryad execution graph [197]. The graph consists of computational vertices and edges written using standard C++ constructs. Pregel [148] also exposes a C++ API for implementing arbitrary graph algorithms over various graph representations. On

the other hand, Hyracks [42] and Nephele [34] are implemented in Java and thus expose a Java-based API. Similar to Dryad, Hyracks allows users to express a computation as a DAG of data operators (vertices) and connectors (edges). 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.

Nephele uses the Parallelization Contracts (PACT) programming model [11], a generalization of the well-known MapReduce programming model. The PACT model extends MapReduce with more second-order functions (namely, Map, Reduce, Match, CoGroup, and Cross), as well as with "Output Contracts" that give guarantees about the behavior of a function. Complete PACT programs are workflows of user functions, starting with one or more data sources and ending with one or more data sinks.

DryadLINQ [120] is a hybrid of declarative and imperative language layers that targets the Dryad run-time and uses the Language INtegrated Query (LINQ) model [152]. 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. While DryadLINQ is implemented using .Net, Spark is implemented in Scala, a statically-typed high-level programming language for the Java Virtual Machine. Spark exposes a functional programming interface similar to DryadLINQ in Scala, Python, and Java.

SCOPE (Structured Computations Optimized for Parallel Execution) [204], Shark [195], the ASTERIX Query Language (AQL) [35], Meteor [138] and Tenzing [59] provide SQL-like declarative languages on top of Dryad, Spark, Hyracks, Nephele, and Google's MapReduce, respectively. In particular, SCOPE supports writing a program using traditional nested SQL expressions as well as a series of simple data transformations, while Shark has chosen to be compatible with Apache Hive using HiveQL. Meteor is an operator-oriented query language that uses a JSON-like data model to support the analysis of unstructured and semi-structured data. Dremel [153] also exposes a SQL-like inter-

face with extra constructs to query nested data since its data model is based on strongly-typed nested records. 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.

5.2 Storage Layer

There are four distinct storage layers used among the various dataflow systems for storing and accessing data. The most typical storage layer is an independent *distributed file system* such as GFS or HDFS. The second category includes distributed *wide columnar stores* like Bigtable and HBase, which operate on top of GFS and HDFS respectively. The third storage layer is an *in-memory data storage* layer that utilizes extra memory on the compute nodes, while the last one is a *nested columnar storage* layer.

Distributed File Systems: Most distributed file systems used in recent large-scale data analytics systems were influenced by or derived from the Google File System (GFS) [96]. In GFS, large files are broken into small pieces that are replicated and distributed across the local disks of the cluster nodes. The architecture of HDFS described in §4.2 is derived from the architecture of GFS.

The Pregel system can directly use GFS, whereas Hyracks, Nephele, and Spark use HDFS. In addition to GFS, Pregel can also process data stored in Bigtable. Dryad uses the Cosmos Storage System [54], an append-only distributed file system also inspired by GFS. Dryad also offers support for accessing files stored on the local NTFS file system as well as tables stored in SQLServer databases running on the cluster nodes [119].

The typical distributed file systems are optimized for large sequential reads and writes while serving single-writer, multiple-reader workloads. Therefore, they work best for batch processing systems like Hadoop, Hyracks, and Nephele. However, stores like GFS and HDFS are not suitable for systems that require concurrent write operations or applications that prefer a record-oriented abstraction of the data.

Wide columnar stores (and key-value stores in general) were designed as a layer on top of these distributed file systems to address the needs of such applications.

Wide Columnar Stores: Wide columnar (or column-family) stores employ a distributed, column-oriented data structure that accommodates multiple attributes per key. The most prominent example of a wide columnar store is Google's Bigtable [58], a distributed, versioned, column-oriented, key-value store that is well suited for sparse datasets. 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.

HBase [95] is the open-source equivalent of Bigtable, and it is built on top of HDFS supporting Hadoop MapReduce. Similar to Bigtable, HBase features compression, in-memory operation, and Bloom filters on a per-column basis. *Accumulo* [18] is also based on the Bigtable design and is built on top of Hadoop, Zookeeper, and Thrift. Compared to Bigtable and HBase, Accumulo features cell-based access control and a server-side programming mechanism that can modify key-value pairs at various points in the data management process.

Similar to HBase, *Cassandra* [20] also follows Bigtable's data model and implements tables as distributed multidimensional maps indexed by a key. Both systems offer secondary indexes, use data replication for fault tolerance both 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.

In-memory Data Storage: A Resilient Distributed Dataset (RDD) is

a distributed shared memory abstraction that represents an immutable collection of objects partitioned across a set of nodes [199]. 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.

Nested Columnar Storage: The demand for more interactive analysis of large datasets has led to the development of a new columnar storage format on top of a distributed file system that targets nested data. Dremel [153] uses this nested columnar data layout on top of GFS and Bigtable. The data model is based on strongly-typed nested records with a schema that forms a tree hierarchy, originating from Protocol Buffers [167]. 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 [153].

Several *data serialization formats* are now widely used for storing and transferring data, irrespective of the storage layer used. Protocol Buffers [167] are a method of serializing structured or semi-structured data in a compact binary format. This framework includes an Interface Definition Language (IDL) that describes the structure of the data and a program that generates source code in various programming languages to represent the data from that description. Apache Thrift [6] offers similar features to Protocol Buffers, with the addition of a concrete Remote Procedural Call (RPC) protocol stack to use with the defined data and services. Apache Avro [19] is also an RPC and serialization framework developed initially within the Apache Hadoop project.

Unlike Protocol Buffers and Thrift, Avro uses JSON for defining data types and protocols, and hence does not require a code-generation program. However, code generation is available in Avro for statically typed languages as an optional optimization.

5.3 Execution Engine

The dataflow-based execution models are what distinguish dataflow systems from MapReduce systems the most. Dryad, Spark, Hyracks, and Nephele all execute directed acyclic graphs of data processing units. Given the high degree of similarity among the execution engines of the aforementioned systems, we have chosen to only elaborate on Dryad's execution engine. The remaining two execution models are the serving trees used in Dremel and the bulk synchronous parallel processing used in Pregel.

A Dryad job is a *Directed Acyclic Graph (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 distributed file 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 5.2 shows the Dryad system architecture. The execution of a Dryad job is orchestrated by a centralized *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 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

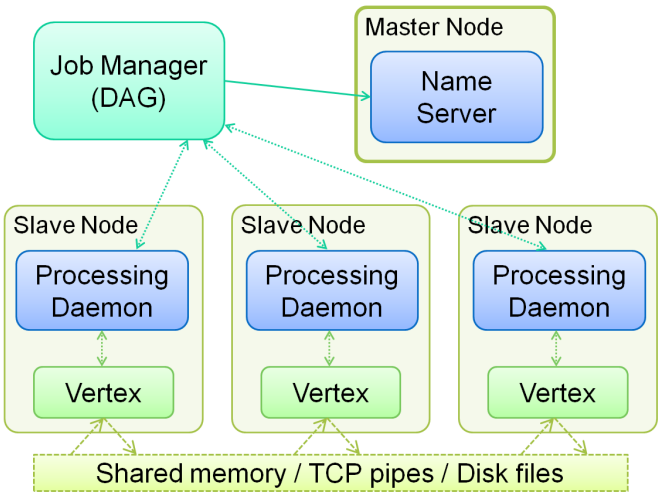

Figure 5.2: Dryad system architecture.

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 Processing 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.

Dremel [153]—with corresponding open-source systems, Cloudera Impala [65] and Apache Drill [21]—uses the concept of a *multi-level serving tree* borrowed from distributed search engines [70] to execute queries. 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.

Many practical applications involve iterative computations and

graph processing. While MapReduce and general dataflow systems can be used for such applications, they are not ideal for iterative and graph algorithms that often better fit a message passing model. For this reason, Pregel [148] introduced a new execution model inspired by the *Bulk Synchronous Parallel* model [190]. 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 dataflow 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.

5.4 Query Optimization

Dataflow systems combine benefits from both traditional parallel databases and MapReduce execution engines to deliver scalability and performance through various optimization techniques. *Static rule-based optimization* is fairly common in most systems that expose higher level interfaces, like DryadLINQ, SCOPE, and AQL. Lower-level systems like Dryad and Spark offer mechanisms for *dynamic optimization*, while some systems (e.g., SCOPE) support *cost-based optimization*.

Static Rule-based Optimization: Many of the traditional query rewrite optimization rules from database systems—removing unnecessary columns, pushing down selection predicates, and pre-aggregating when possible—are applicable to systems offering SQL-like interfaces. SCOPE, DryadLINQ, and AQL have their own rule-based optimizers for applying such rules to the query plan, optimizing locality, and improving performance [42, 197, 204]. DryadLINQ also allows users to specify various annotations that are treated as manual hints and are used to guide optimizations that the system is unable to perform automatically. For example, annotations can be used to declare a user-defined function as associative or commutative, enabling optimizations

such as eager aggregation [120]. 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 [34].

Dynamic Run-time Optimization: Both Dryad and Spark support dynamic optimizations for mutating the execution graph based on run-time information such as statistics of the dataset processed. In particular, Dryad supports a callback mechanism that can be used to implement various run-time optimization policies [119]. For example, after a set of vertices produces an intermediate data set to be consumed by the next set of vertices, Dryad can dynamically choose the degree of parallelism (i.e., change the number of vertices while preserving the graph topology) based on the amount of generated data. Another dynamic optimization supported by Dryad is the addition of vertices to create deeper aggregation trees. The new layer of vertices will process subsets of the data that are close in network topology (e.g., on the same node or rack) to perform partial aggregation and thus reduce the overall network traffic between racks [197].

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 [195]. 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.

Cost-based Query Optimization: The SCOPE optimizer is a transformation-based optimizer inspired by the Cascades framework [98] that translates input scripts into efficient execution plans. Hence, it can apply many of the traditional optimization rules from database systems such as column pruning and filter predicate push down. In addition to traditional optimization techniques, the SCOPE optimizer reasons about partitioning, grouping, and sorting properties in a single uniform framework, and seamlessly generates and optimizes both serial and parallel query plans [205]. The SCOPE optimizer considers such alternative plans from a large plan space, and chooses the plan with the lowest estimated cost based on data statistics and an internal cost

model. The cost model of the optimizer is similar to the analytical cost models used in classic parallel database systems.

5.5 Scheduling

The primary goal of scheduling in dataflow systems is identical to the goal in MapReduce systems, namely, schedule tasks on nodes so that high data locality can be obtained. All dataflow systems contain a scheduling component that implements one or more scheduling techniques for placing vertices close to their input data, rerunning failed vertices, and performing straggler mitigation.

Dryad offers an interesting choice between two scheduling approaches. On one hand, a Job Manager can contain its own internal scheduler that chooses which node each vertex should be executed on. This decision is based on the network topology exposed by the cluster's Name Server. One the other hand, the Job Manager can send its list of ready vertices and their constraints to a centralized scheduler that optimizes placement across multiple jobs running concurrently [119]. The Dryad Scheduler will then decide which physical resources to schedule work on and how to route data between computations. Note that the channel type can affect scheduling. In particular, TCP requires both vertices to run at the same time, while shared-memory requires both vertices to run in the same process. Finally, the Scheduler has grouping heuristics to ensure that each vertex has no more than a set number of inputs, or a set volume of input data to process. These heuristics help avoid overloading the vertex as well as the I/O system [204].

The schedulers in the other dataflow systems described in this chapter are more similar to the MapReduce schedulers. Spark, in particular, uses a FIFO scheduler with delay scheduling [198], even though other schedulers can be used [200]. Dremel's Query Dispatcher is in charge of scheduling and will also take into consideration priorities for each query as well as try to balance the load across the cluster [153]. Pregel offers support for preemptive scheduling and will sometimes kill vertex instances or move them to different nodes based on the overall cluster utilization [148].

5.6 Resource Management

The support for dynamic query optimization in some dataflow systems, like Dryad and Spark, has the potential to greatly improve the overall resource utilization of the cluster. An application, for example, can discover the size and placement of data at run-time, and modify the graph as the computation progresses to make efficient use of the available resources [119].

Dryad also offers techniques for improving resource utilization on individual nodes. Even though most vertices contain purely sequential code, Dryad supports an event-based programming style that uses asynchronous interfaces and a shared thread pool to run multiple vertices within the same process. The run-time automatically distinguishes between vertices which can use a thread pool and those that require a dedicated thread; therefore, encapsulated graphs that contain hundreds of asynchronous vertices are executed efficiently on a shared thread pool [197].

Spark also uses multi-threading to run multiple vertices within the same JVM process [200]. However, the main difference of Spark from Dryad is that it uses a memory abstraction—called Resilient Distributed Datasets—to explicitly store data in memory.

Hadoop NextGen (or YARN) is a new framework for cluster resource management and was discussed in detail in §4.6. Mesos [111] is a similar platform for sharing cluster resources between multiple different frameworks (like MapReduce and MPI). Mesos consists of a *Master* that manages the *Slave* daemons running on each cluster node as well as the cluster resources. Each framework consists of a *Scheduler* for making scheduling decisions and an *Executor* for running the framework's tasks on each cluster node. The Master and Slave in Mesos are equivalent to YARN's ResourceManager and NodeManager respectively, while the Scheduler in Mesos is similar to YARN's ApplicationMaster.

The main difference between YARN and Mesos lies within their resource models. In YARN, the ApplicationMaster requests containers with a given specification and locality preferences, and the ResourceManager grants the requests as resources become available. On the other hand, the Master in Mesos decides how many resources to offer to each

framework, and the framework's Scheduler decides which of the offered resources to use and how. Currently, Mesos supports control for CPU (in terms of number of cores) and memory. For example, the Master can offer 2 CPU cores and 4GB of memory on node N to a framework, and the framework's Scheduler can decide that it will use 1 CPU core and 1GB of memory for running task X, and 1 CPU core and 3GB of memory for running task Y. The Scheduler also has the option of rejecting a particular resource offer in anticipation of an offer for resources on a different cluster node.

5.7 Fault Tolerance

The fault-tolerance features offered by the dataflow systems are very similar to the ones offered by MapReduce systems in that vertices get re-executed in case of a failure. The channel type in Dryad, however, can affect the number of vertices to get re-executed. If files are used, then only the failed vertex gets re-executed. If TCP or shared-memory is used, then the set of all vertices involved in the communication will have to get re-executed. Spark, which handles data in memory, achieves fault tolerance through the notion of *lineage* in its Resilient Distributed Datasets (RDD). If a partition of an RDD is lost, then the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition [200].

In Dremel, the Query Dispatcher (the entity responsible for scheduling) provides fault tolerance when one server becomes much slower than others or a data replica becomes unreachable by taking advantage of the specialized aggregation tree topology [153]. Fault tolerance in Pregel is achieved through checkpointing [148]. At the beginning of each iteration, the state of each vertex is persisted to storage. Upon failure, a master coordinator reassigns graph partitions to the currently available set of workers, which reload their corresponding partition state from the most recently available checkpoint.

5.8 System Administration

The current state of system administration in dataflow systems is similar to the state of system administration in MapReduce systems as discussed in §4.8. However, from an architectural perspective, dataflow systems have more similarities to parallel database systems compared to the similarities between MapReduce systems and parallel database systems. Thus, it is conceivable that DBAs for parallel database systems have an easier time adapting their skills to administer dataflow systems compared to MapReduce systems.

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. The data storage and processing techniques 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* [82].

The velocity challenge refers to the short response-time requirements for collecting, storing, and processing data. Most of the systems that we covered in this monograph 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. There is an increasing appetite towards getting query results faster.

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. The storage and processing techniques that we have seen in this monograph (especially in Chapters 2 and 3) are predominantly aimed at handling data that can be represented using a relational model (rows and columns) and processed by query plan operators like filters, joins, and aggregation. However, the new and emerging data types cannot be captured easily in a relational data model, or analyzed easily by software that depends on running operators like filters, joins, and aggregation. Instead, the new and emerging data types need a variety of analytical techniques such as linear algebra, statistical machine learning, text search, signal processing, natural language processing, and iterative graph processing.

These challenges are shaping the new research trends in massively parallel data processing. We will conclude the monograph with a summary of the recent research trends aimed at addressing these challenges.

6.1 Mixed Systems

The need to reduce the gap between the generation of data and the generation of analytics results over this data has led to systems that can support both OLTP and OLAP workloads in a single system. On one hand, scalable distributed storage systems that provide various degrees of transactional capabilities are being developed. Support for transactions enables these systems to serve as the data store for online services while making the data available concurrently in the same system for analytics. The most prominent example here is Google's Bigtable system which is a distributed, versioned, and column-oriented system that stores multi-dimensional and sorted datasets [58]. Bigtable provides atomicity at the level of individual tuples.

Bigtable has motivated popular open-source implementations like *Accumulo* [18], *HBase* [23], and *Cassandra* [131] (recall §5.2), as well as follow-up systems from Google such as *Megastore* [31] and *Spanner* [69]. Megastore and Spanner provide more fine-grained transactional support compared to Bigtable without sacrificing performance require-

ments 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 supports transactions at a global scale across data centers.

Traditionally, parallel databases have used different systems to support OLTP and OLAP [88, 101, 125]. 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. For example, "available-to-promise" applications require OLTP-style queries while aggregating stock levels in real-time using OLAP-style queries to determine if an order can be fulfilled [101]. A recent benchmark tries to capture the current trend in database systems towards scenarios with mixed workloads [66].

Systems like *HYRISE*, *HyPer*, and *SAP HANA* aim to support OLTP and OLAP in a single system [88, 101, 125]. One of the challenges these systems face is that data layouts that are good for OLTP may not be good for OLAP, and vice versa. For example, HYRISE partitions tables into vertical groups of varying widths depending on how the columns of the tables are accessed. Smaller column groups are preferred for OLAP-style data access while wider column groups are preferred for OLTP-style data access (to reduce cache misses when performing single row retrievals). 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 [101].

HyPer complements columnar data layouts with sophisticated main-memory indexing structures based on hashing, balanced search trees (e.g., red-black trees), and radix trees [125]. 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.

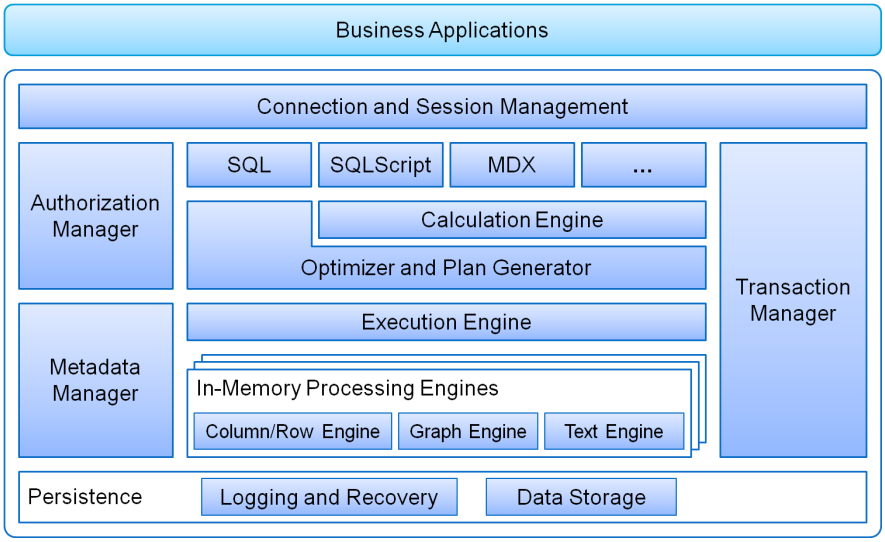

Figure 6.1: Overview of SAP HANA's architecture.

6.2 Memory-based Systems

Given the steadily increasing memory sizes in commodity servers, a number of memory-based systems are being developed such as SAP HANA. Figure 6.1 gives an overview of the general SAP HANA architecture [88]. A set of in-memory processing engines forms the core of this architecture. 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 [88].

Memory-based extensions and improvements on current systems have also been proposed. *M3R* (Main Memory MapReduce) [175] is a framework that extends Hadoop for running MapReduce jobs in memory. M3R caches input and output data in memory, performs in-memory shuffling, and always maps the same partition to the same location across all jobs in a sequence in order to allow for the re-use of already built memory structures. *PowerDrill* [102] is a column-oriented datastore similar to Dremel, but it relies on having as much data in memory as possible. PowerDrill uses two dictionaries as basic data structures for representing a data column and employs several optimizations for keeping the memory footprint of these structures small.

6.3 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. This data is analyzed for purposes like behavioral targeting, where personalized content is shown based on a user's past activity, and showing advertisements or recommendations based on the activity of her social friends [56]. 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*. A continuous query Q is a query that is issued once over data D that is constantly updated. Q runs continuously over D and lets users get new results as D changes, without

having to issue the same query repeatedly. Continuous queries arise naturally over activity and operational data because of two reasons: (i) the data is generated continuously in the form of append-only streams; (ii) 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* in 2010, Twitter released *Storm* in 2011, and Walmart Labs released *Muppet* in 2012 [132, 156, 182]. Also prominent are recent efforts to add continuous querying capabilities to the popular *Hadoop* platform for batch analytics. Examples include the *Oozie* workflow manager, *MapReduce Online*, and Facebook's real-time analytics system [44, 67, 161]. These platforms add to older research projects like *Aurora* [3], *Borealis* [1], *NiagaraCQ* [62], *STREAM* [28], and *TelegraphCQ* [57], as well as commercial systems like *Esper* [86], *Infosphere Streams* [38], *StreamBase* [183], and *Truviso* [91].

Some of the features of streaming systems have been added to the categories of systems discussed in the monograph. One example is incremental processing where a query over data D is processed quickly based on minimal additional processing done on top of a previous execution of the same query on a previous snapshot of D [87, 150, 154, 158]. *NOVA* is a workflow manager that supports incremental processing of continuously arriving data. NOVA is implemented on top of Pig and Hadoop without any modifications to these systems [158]. *REX* is a parallel query processing platform that supports recursive queries based on incremental refinement of results [154].

Another example is the use of techniques like one-pass processing and sampling to return quick, but approximate, answers for long-running queries over large datasets. For example, *BlinkDB* is a query processing framework for running queries interactively on large volumes of data [7]. BlinkDB uses pre-computed samples of data to enable quick retrieval of approximate query results. Dynamic MapReduce jobs that can terminate early after producing a query result sample are proposed in [100]. The *EARL* system produces query results quickly with

reliable accuracy estimates [135]. EARL uses uniform sampling and works iteratively to compute larger samples until the given accuracy level is reached (estimated through bootstrapping). MapReduce Online [67] can also generate approximate answers to MapReduce jobs using online aggregation [106] rather than sampling. In MapReduce Online, data produced by the Map tasks is pipelined directly to the Reduce tasks, allowing the latter to generate and refine an approximation of the final answer during the course of job execution.

6.4 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.

The analysis needs of such applications not only include processing the attribute values of the nodes in the graph, but also analyzing the way in which these nodes are connected. The relational data model can be a hindrance in representing graph data as well as expressing analysis tasks over this data especially when the data is distributed and has some complex structure. Graph databases—which use graph structures with nodes, edges, and their properties to represent and store data—are being developed to support such applications [141, 148, 194].

There are many techniques for how to store and process graphs. The effectiveness of these techniques depend on the amount of data—the number of nodes, edges, along with the size of data associated with them—and the types of analysis tasks, e.g., search and pattern-matching versus more complex analytics tasks such as finding strongly connected components, maximal independent sets, and shortest paths.

Many graph databases such as Pregel [148] use the Bulk Synchronous Parallel (BSP) computing model (recall §5.3). Like the map and reduce functions in MapReduce, Pregel has primitives that let neighboring nodes send and receive messages to one another, or change

the state of a node (based on the state of neighboring nodes). Graph algorithms are specified as a sequence of iterations built from such primitives. *GraphLab* uses similar primitives (called PowerGraph) but allows for asynchronous iterative computations [141]. *GraphX* 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 [194].

Techniques have also been proposed to support the iterative and recursive computational needs of graph analysis in the categories of systems that we have considered in this monograph. For example, *HaLoop* and *Twister* are designed to support iterative algorithms in MapReduce systems [47, 83]. 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*, a distributed framework for iterative workloads, enables faster convergence of iterative tasks by providing support for prioritized iteration [202]. Efficient techniques to run recursive algorithms needed in machine-learning tasks are supported by the Hyracks dataflow system [171].

6.5 Array Databases

For many applications involving time-series analysis, image or video analysis, climate modeling, and scientific simulation, the data is best represented as arrays. An array represents a homogeneous collection of data items sitting on a regular grid of one, two, or more dimensions. While array storage and processing can be simulated in the relational data model and SQL, this process is usually cumbersome and inefficient [203]. Array databases make arrays first-class citizens in the database so that flexible, scalable storage and processing of array-based data can be performed.

SciDB is a recent effort to build an array database [180]. *SciQL* is an array query language being added to the MonetDB system [126]. Both systems treat arrays at the same level at which relational database

systems treat tables, instead of the array-valued attribute type specified by the ISO SQL standard. Due to the massive array sizes and complex array-based queries observed in scientific and engineering applications, these systems employ a variety of techniques to optimize array storage and processing.

References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the 3rd Biennial Conf. on Innovative Data Systems Research*, 2005.

[2] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column Oriented Database Systems. *Proc. of the VLDB Endowment*, 2(2):1664–1665, 2009.

[3] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[4] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proc. of the 23rd IEEE Intl. Conf. on Data Engineering*, pages 466–475. IEEE, 2007.

[5] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proc. of the VLDB Endowment*, 2(1):922–933, 2009.

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

[7] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proc. of the 8th European Conf. on Computer Systems*, pages 29–42. ACM, 2013.

[8] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, pages 683–694. ACM, 2006.

[9] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 359–370. ACM, 2004.

[10] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving Relations for Cache Performance. *The VLDB Journal*, 1:169–180, 2001.

[11] Alexander Alexandrov, Max Heimel, Volker Markl, Dominic Battré, Fabian Hueske, Erik Nijkamp, Stephan Ewen, Odej Kao, and Daniel Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *Proc. of the VLDB Endowment*, 3(1-2):1625–1628, 2010.

[12] *Amazon RedShift*, 2013. `http://aws.amazon.com/redshift/`.

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

[14] *Apache Hadoop*, 2012. `http://hadoop.apache.org/`.

[15] *Apache Hadoop Capacity Scheduler*, 2013. `http://hadoop.apache.org/docs/r1.1.2/capacity_scheduler.html`.

[16] *Apache Hadoop Fair Scheduler*, 2013. `http://hadoop.apache.org/docs/r1.1.2/fair_scheduler.html`.

[17] *Apache Hadoop on Demand*, 2013. `http://hadoop.apache.org/docs/stable/hod_scheduler.html`.

[18] *Apache Accumulo*, 2013. `http://accumulo.apache.org/`.

[19] *Apache Avro*, 2013. `http://avro.apache.org/`.

[20] *Apache Cassandra*, 2013. `http://cassandra.apache.org/`.

[21] *Apache Drill*, 2013. `http://incubator.apache.org/drill/`.

[22] *Apache Hadoop NextGen MapReduce (YARN)*, 2013. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[23] *Apache HBase*, 2013. `http://hbase.apache.org/`.

[24] *Apache HCatalog*, 2013. `http://incubator.apache.org/hcatalog/`.

[25] *Aster Data nCluster*, 2012. `http://www.asterdata.com/product/ncluster_cloud.php`.

[26] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Trans. on Database Systems (TODS)*, 1(2):97–137, 1976.

[27] Shivnath Babu. Towards Automatic Optimization of MapReduce Programs. In *Proc. of the 1st Symp. on Cloud Computing*, pages 137–142. ACM, 2010.

[28] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *ACM SIGMOD Record*, 30(3):109–120, 2001.

[29] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. *ACM Trans. on Storage (TOS)*, 4(3):8, 2008.

[30] Kamil Bajda-Pawlikowski, Daniel J Abadi, Avi Silberschatz, and Erik Paulson. Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1165–1176. ACM, 2011.

[31] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of the 5th Biennial Conf. on Innovative Data Systems Research*, pages 223–234, 2011.

[32] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business Analytics in (a) Blink. *IEEE Data Engineering Bulletin*, 35(1):9–14, 2012.

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

[34] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proc. of the 1st Symp. on Cloud Computing*, pages 119–130. ACM, 2010.

[35] Alexander Behm, Vinayak R Borkar, Michael J Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[36] K. Beyer, V. Ercegovac, and E. Shekita. *Jaql: A JSON query language*. `http://www.jaql.org`.

[37] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *Proc. of the VLDB Endowment*, 4(12):1272–1283, 2011.

[38] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1093–1104. ACM, 2010.

[39] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, pages 479–490. ACM, 2006.

[40] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of the 2nd Biennial Conf. on Innovative Data Systems Research*, pages 225–237, 2005.

[41] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a Highly Parallel Database System. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):4–24, 2002.

[42] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A Flexible and Extensible Foundation for Data-intensive Computing. In *Proc. of the 27th IEEE Intl. Conf. on Data Engineering*, pages 1151–1162. IEEE, 2011.

[43] Andrea J. Borr. Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing. In *Proc. of the 7th Intl. Conf. on Very Large Data Bases*, pages 155–165, 1981.

[44] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand S. Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1071–1080. ACM, 2011.

[45] Bobby-Joe Breitkreutz, Chris Stark, Mike Tyers, et al. Osprey: A Network Visualization System. *Genome Biol*, 4(3):R22, 2003.

[46] Kurt P. Brown, Manish Mehta, Michael J. Carey, and Miron Livny. Towards Automated Performance Tuning for Complex Workloads. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pages 72–84. VLDB Endowment, 1994.

[47] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proc. of the VLDB Endowment*, 3(1-2):285–296, 2010.

[48] Ron Buck. The Oracle Media Server for nCUBE Massively Parallel Systems. In *Proc. of the 8th Intl. Parallel Processing Symposium*, pages 670–673. IEEE, 1994.

[49] Michael J. Cafarella and Christopher Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *Proc. of the 13th Intl. Workshop on the Web and Databases*, pages 10:1–10:6. ACM, 2010.

[50] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of the 23rd ACM Symp. on Operating Systems Principles*, pages 143–157. ACM, 2011.

[51] Michael J. Carey, Miron Livny, and Hongjun Lu. Dynamic Task Allocation in a Distributed Database System. In *Proc. of the 5th Intl. Conf. on Distributed Computing Systems*, pages 282–291. IEEE, 1985.

[52] *Cascading*, 2011. `http://www.cascading.org/`.

[53] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal Data Partitioning in Database Design. In *Proc. of the 1982 ACM SIGMOD Intl. Conf. on Management of Data*, pages 128–136, 1982.

[54] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. of the VLDB Endowment*, 1(2):1265–1276, 2008.

[55] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 363–375, 2010.

[56] Badrish Chandramouli, Jonathan Goldstein, and Songyun Duan. Temporal Analytics on Big Data for Web Advertising. In *Proc. of the 28th IEEE Intl. Conf. on Data Engineering*, pages 90–101. IEEE, 2012.

[57] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 668–668. ACM, 2003.

[58] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. on Computer Systems*, 26(2):4, 2008.

[59] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing: A SQL Implementation on the MapReduce Framework. *Proc. of the VLDB Endowment*, 4(12):1318–1327, 2011.

[60] Surajit Chaudhuri, Arnd Christian König, and Vivek R. Narasayya. SQLCM: A Continuous Monitoring Framework for Relational Database Engines. In *Proc. of the 20th IEEE Intl. Conf. on Data Engineering*, pages 473–484, 2004.

[61] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. Scheduling Problems in Parallel Query Optimization. In *Proc. of the 14th ACM Symp. on Principles of Database Systems*, pages 255–265. ACM, 1995.

[62] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390. ACM, 2000.

[63] Ming-Syan Chen and Philip S. Yu. Interleaving a Join Sequence with Semijoins in Distributed Query Processing. *IEEE Trans. on Parallel Distributed Systems*, 3(5):611–621, 1992.

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

[65] *Cloudera Impala*, 2013. `http://www.cloudera.com/content/cloudera/en/products/cdh/impala.html`.

[66] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The Mixed Workload CH-benCHmark. In *Proc. of the 4th Intl. Workshop on Testing Database Systems*. ACM, 2011.

[67] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proc. of the 7th USENIX Symp. on Networked Systems Design and Implementation*, volume 10. USENIX Association, 2010.

[68] George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In *Proc. of the 1985 ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–279, 1985.

[69] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation*, page 1. USENIX Association, 2012.

[70] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley Reading, 2010.

[71] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the Database Schema Evolution Process. *The VLDB Journal*, 22(1):73–98, 2013.

[72] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 137–149. USENIX Association, 2004.

[73] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[74] Amol Deshpande and Lisa Hellerstein. Flow Algorithms for Parallel Query Optimization. In *Proc. of the 24th IEEE Intl. Conf. on Data Engineering*, pages 754–763. IEEE, 2008.

[75] David J DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, H-I Hsiao, and Rick Rasmussen. The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):44–62, 1990.

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

[77] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proc. of the 18th Intl. Conf. on Very Large Data Bases*, pages 27–40. VLDB Endowment, 1992.

[78] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic Performance Diagnosis and Tuning in Oracle. In *Proc. of the 3rd Biennial Conf. on Innovative Data Systems Research*, pages 84–94, 2005.

[79] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient Big Data Processing in Hadoop MapReduce. *Proc. of the VLDB Endowment*, 5(12):2014–2015, 2012.

[80] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *Proc. of the VLDB Endowment*, 3(1-2):515–529, 2010.

[81] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only Aggressive Elephants are Fast Elephants. *Proc. of the VLDB Endowment*, 5(11):1591–1602, 2012.

[82] *3-D Data Management: Controlling Data Volume, Velocity and Variety*, 2013. Doug Laney, Research Note, META Group, February 2001.

[83] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *Proc. of the 19th Intl. Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[84] Mohamed Y Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. CoHadoop: Flexible Data Placement and its Exploitation in Hadoop. *Proc. of the VLDB Endowment*, 4(9):575–585, 2011.

[85] Susanne Englert, Jim Gray, Terrye Kocher, and Praful Shah. A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases. In *Proc. of the 1990 ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 245–246. ACM, 1990.

[86] *Esper*, 2013. `http://esper.codehaus.org/`.

[87] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning Fast Iterative Data Flows. *Proc. of the VLDB Endowment*, 5(11):1268–1279, 2012.

[88] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[89] Avrilia Floratou, Jignesh M Patel, Eugene J Shekita, and Sandeep Tata. Column-oriented Storage Techniques for MapReduce. *Proc. of the VLDB Endowment*, 4(7):419–429, 2011.

[90] Avrilia Floratou, Nikhil Teletia, David J DeWitt, Jignesh M Patel, and Donghui Zhang. Can the Elephants Handle the NoSQL Onslaught? *Proc. of the VLDB Endowment*, 5(12):1712–1723, 2012.

[91] Michael J Franklin, Sailesh Krishnamurthy, Neil Conway, Alan Li, Alex Russakovsky, and Neil Thombre. Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *Proc. of the 4th Biennial Conf. on Innovative Data Systems Research*. Citeseer, 2009.

[92] Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: A Practical Approach to Self-Describing, Polymorphic, and Parallelizable User-Defined Functions. *Proc. of the VLDB Endowment*, 2(2):1402–1413, 2009.

[93] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query Optimization for Parallel Execution. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 9–18. ACM, 1992.

[94] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *Proc. of the VLDB Endowment*, 2(2):1414–1425, 2009.

[95] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[96] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[97] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[98] Goetz Graefe. The Cascades Framework for Query Optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[99] *Greenplum*, 2012. `http://www.greenplum.com`.

[100] Raman Grover and Michael J. Carey. Extending Map-Reduce for Efficient Predicate-Based Sampling. In *Proc. of the 28th IEEE Intl. Conf. on Data Engineering*, pages 486–497. IEEE, 2012.

[101] Martin Grund, Philippe Cudré-Mauroux, Jens Krüger, Samuel Madden, and Hasso Plattner. An overview of HYRISE - a Main Memory Hybrid Storage Engine. *IEEE Data Engineering Bulletin*, 35(1):52–57, 2012.

[102] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. Processing a Trillion Cells per Mouse Click. *Proc. of the VLDB Endowment*, 5(11):1436–1446, 2012.

[103] Wook-Shin Han, Jack Ng, Volker Markl, Holger Kache, and Mokhtar Kandil. Progressive Optimization in a Shared-nothing Parallel Database. In *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*, pages 809–820. ACM, 2007.

[104] Waqar Hasan and Rajeev Motwani. Coloring Away Communication in Parallel Query Optimization. In *Proc. of the 21st Intl. Conf. on Very Large Data Bases*, pages 239–250. VLDB Endowment, 1995.

[105] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 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*, pages 1199–1208. IEEE, 2011.

[106] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online Aggregation. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182. ACM, 1997.

[107] Herodotos Herodotou and Shivnath Babu. Xplus: A SQL-Tuning-Aware Query Optimizer. *Proc. of the VLDB Endowment*, 3(1-2):1149–1160, 2010.

[108] Herodotos Herodotou, Nedyalko Borisov, and Shivnath Babu. Query Optimization Techniques for Partitioned Tables. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60. ACM, 2011.

[109] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *Proc. of the 2nd Symp. on Cloud Computing*. ACM, 2011.

[110] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of the 5th Biennial Conf. on Innovative Data Systems Research*, 2011.

[111] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Symp. on Networked Systems Design and Implementation*. USENIX Association, 2011.

[112] Jeffrey A. Hoffer and Dennis G. Severance. The Use of Cluster Analysis in Physical Data Base Design. In *Proc. of the 1st Intl. Conf. on Very Large Data Bases*, pages 69–86. ACM, 1975.

[113] Wei Hong and Michael Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[114] Hui-I Hsiao and David J. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proc. of the 6th IEEE Intl. Conf. on Data Engineering*, pages 456–465, 1990.

[115] IBM Corporation2. *Partitioned Tables*, 2007. `http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.partition.doc/doc/c0021560.html`.

[116] *IBM Netezza Data Warehouse Appliances*, 2012. `http://www-01.ibm.com/software/data/netezza/`.

[117] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[118] *Infobright*, 2013. `http://www.infobright.com/`.

[119] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[120] Michael Isard and Yuan Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, pages 987–994. ACM, 2009.

[121] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: A MapReduce Query Optimizer. In *Proc. of the 5th European Conf. on Computer Systems*, pages 251–264. ACM, 2010.

[122] Alekh Jindal, Jorge-Arnulfo Quian-Ruiz, and Jens Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *Proc. of the 2nd Symp. on Cloud Computing*. ACM, 2011.

[123] Tim Kaldewey, Eugene J Shekita, and Sandeep Tata. Clydesdale: Structured Data Processing on MapReduce. In *Proc. of the 15th Intl. Conf. on Extending Database Technology*, pages 15–25. ACM, 2012.

[124] Kamal Kc and Kemafor Anyanwu. Scheduling Hadoop Jobs to Meet Deadlines. In *Proc. of the 2nd IEEE Intl. Conf. on Cloud Computing Technology and Science*, pages 388–392. IEEE, 2010.

[125] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. *IEEE Data Engineering Bulletin*, 35(1):46–51, 2012.

[126] Martin L. Kersten, Ying Zhang, Milena Ivanova, and Niels Nes. SciQL, a Query Language for Science Applications. In *Proc. of the EDBT/ICDT Workshop on Array Databases*, pages 1–12. ACM, 2011.

[127] *Kosmos Distributed Filesystem*, 2013. `http://code.google.com/p/kosmosfs/`.

[128] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[129] Stefan Krompass, Umeshwar Dayal, Harumi A. Kuno, and Alfons Kemper. Dynamic Workload Management for Very Large Data Warehouses: Juggling Feathers and Bowling Balls. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, pages 1105–1115. VLDB Endowment, 2007.

[130] Stefan Krompass, Harumi A. Kuno, Janet L. Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Managing Long-running Queries. In *Proc. of the 13th Intl. Conf. on Extending Database Technology*, pages 132–143. ACM, 2009.

[131] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *Operating Systems Review*, 44(2):35–40, 2010.

[132] Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: MapReduce-Style Processing of Fast Data. *Proc. of the VLDB Endowment*, 5(12):1814–1825, 2012.

[133] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proc. of the VLDB Endowment*, 5(12):1790–1801, 2012.

[134] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. In *Proc. of the 19th Intl. Conf. on Very Large Data Bases*, pages 493–504. Morgan Kaufmann Publishers Inc., 1993.

[135] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. *Proc. of the VLDB Endowment*, 5(10):1028–1039, 2012.

[136] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel Data Processing with MapReduce: a Survey. *ACM SIGMOD Record*, 40(4):11–20, 2011.

[137] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. YSmart: Yet Another SQL-to-MapReduce Translator. In *Proc. of the 31st Intl. Conf. on Distributed Computing Systems*, pages 25–36. IEEE, 2011.

[138] Marcus Leich, Jochen Adamek, Moritz Schubotz, Arvid Heise, Astrid Rheinländer, and Volker Markl. Applying Stratosphere for Big Data Analytics. In *Proc. of the 15th USENIX Annual Technical Conference*, pages 507–510, 2013.

[139] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A Transformation-based Optimizer for MapReduce Workflows. *Proc. of the VLDB Endowment*, 5(11):1196–1207, 2012.

[140] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. 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*, pages 961–972. ACM, 2011.

[141] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. of the VLDB Endowment*, 5(8):716–727, 2012.

[142] Hongjun Lu. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, 1st edition, 1994.

[143] Hongjun Lu, Ming-Chien Shan, and Kian-Lee Tan. Optimization of Multi-Way Join Queries for Parallel Execution. In *Proc. of the 17th Intl. Conf. on Very Large Data Bases*, pages 549–560. VLDB Endowment, 1991.

[144] Hongjun Lu and Kian-Lee Tan. Dynamic and Load-balanced Task-Oriented Datbase Query Processing in Parallel Systems. In *Proc. of the 3rd Intl. Conf. on Extending Database Technology*, pages 357–372. ACM, 1992.

[145] Hongjun Lu and Kian-Lee Tan. Load Balanced Join Processing in Shared-Nothing Systems. *Journal of Parallel and Distributed Computing*, 23(3):382–398, 1994.

[146] Soren Macbeth. *Why YieldBot Chose Cascalog over Pig for Hadoop Processing*, 2011. `http://tech.backtype.com/52456836`.

[147] Roger MacNicol and Blaine French. Sybase IQ Multiplex-designed for Analytics. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 1227–1230. VLDB Endowment, 2004.

[148] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pages 135–146. ACM, 2010.

[149] *MapR File System*, 2013. `http://www.mapr.com/products/apache-hadoop`.

[150] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *Proc. of the 6th Biennial Conf. on Innovative Data Systems Research*, 2013.

[151] Manish Mehta and David J. DeWitt. Data Placement in Shared-Nothing Parallel Database Systems. *The VLDB Journal*, 6(1):53–72, 1997.

[152] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, pages 706–706. ACM, 2006.

[153] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. of the VLDB Endowment*, 3(1):330–339, 2010.

[154] Svilen R Mihaylov, Zachary G Ives, and Sudipto Guha. REX: Recursive, Delta-based Data-centric Computation. *Proc. of the VLDB Endowment*, 5(11):1280–1291, 2012.

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

[156] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *Proc. of the 2010 IEEE Intl. Conf. on Data Mining Workshops*. IEEE, 2010.

[157] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *Proc. of the VLDB Endowment*, 3(1):494–505, 2010.

[158] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B. N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: Continuous Pig/Hadoop Workflows. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1081–1090. ACM, 2011.

[159] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1099–1110. ACM, 2008.

[160] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[161] *Oozie: Workflow Engine for Hadoop*, 2010. `http://yahoo.github.com/oozie/`.

[162] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. *Proc. of the VLDB Endowment*, 6(11), 2013.

[163] HweeHwa Pang, Michael J. Carey, and Miron Livny. Multiclass Query Scheduling in Real-Time Database Systems. *IEEE Trans. on Knowledge and Data Engineering*, 7(4):533–551, 1995.

[164] *ParAccel Analytic Platform*, 2013. `http://www.paraccel.com/`.

[165] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of the 1985 ACM SIGMOD Intl. Conf. on Management of Data*, pages 109–116, 1988.

[166] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel Abadi, David DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, pages 165–178. ACM, 2009.

[167] *Protocol Buffers Developer Guide*, 2012. `https://developers.google.com/protocol-buffers/docs/overview`.

[168] B Thirumala Rao and LSS Reddy. Survey on Improved Scheduling in Hadoop MapReduce in Cloud Environments. *Intl. Journal of Computer Applications*, 34(9):29–33, 2011.

[169] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. Automating Physical Database Design in a Parallel Database. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 558–569, 2002.

[170] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A Framework for Large Scale Data Processing. In *Proc. of the 3rd Symp. on Cloud Computing*, page 4. ACM, 2012.

[171] Joshua Rosen, Neoklis Polyzotis, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Iterative MapReduce for Large Scale Machine Learning. *Computing Research Repository (CoRR)*, abs/1303.3517, 2013.

[172] Thomas Sandholm and Kevin Lai. Dynamic Proportional Share Scheduling in Hadoop. In *Proc. of the 15th IEEE Intl. Conf. on Data Mining*, pages 110–131. Springer, 2010.

[173] Patricia G. Selinger, Morton M Astrahan, Donald D. Chamberlin, Raymond A Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34. ACM, 1979.

[174] Kyuseok Shim. MapReduce Algorithms for Big Data Analysis. *Proc. of the VLDB Endowment*, 5(12):2016–2017, 2012.

[175] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3R: Increased Performance for In-memory Hadoop Jobs. *Proc. of the VLDB Endowment*, 5(12):1736–1747, 2012.

[176] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proc. of the 26th IEEE Symp. on Mass Storage Systems and Technologies*, pages 1–10. IEEE, 2010.

[177] *SQL Server Parallel Data Warehouse*, 2012. `http://www.microsoft.com/en-us/sqlserver/solutions-technologies/data-warehousing/pdw.aspx`.

[178] Garrick Staples. TORQUE Resource Manager. In *Proc. of the 20th ACM Intl. Conf. on Supercomputing*, page 8. ACM, 2006.

[179] Michael Stillger, Myra Spiliopoulou, and Johann Christoph Freytag. *Parallel Query Optimization: Exploiting Bushy and Pipeline Parallelisms with Genetic Programs.* Citeseer, 1996.

[180] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The Architecture of SciDB. In *Proc. of the 23rd Intl. Conf. on Scientific and Statistical Database Management*, pages 1–16. IEEE, 2011.

[181] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-Store: A Column-oriented DBMS. In *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005.

[182] *Storm*, 2013. `http://storm-project.net/`.

[183] *StreamBase*, 2013. `http://www.streambase.com/`.

[184] Michal Switakowski, Peter A. Boncz, and Marcin Zukowski. From Cooperative Scans to Predictive Buffer Management. *Proc. of the VLDB Endowment*, 5(12):1759–1770, 2012.

[185] Sybase, Inc. *Performance and Tuning: Optimizer and Abstract Plans*, 2003. `http://infocenter.sybase.com/help/topic/com.sybase.dc20023_1251/pdf/optimizer.pdf`.

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

[187] Kian-Lee Tan and Hongjun Lu. On Resource Scheduling of Multi-Join Queries. *Information Processing Letters*, 48(4):189–195, 1993.

[188] *Teradata*, 2012. `http://www.teradata.com`.

[189] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. of the VLDB Endowment*, 2(2):1626–1629, 2009.

[190] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[191] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*, pages 307–320. USENIX Association, 2006.

[192] Tom White. *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[193] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query Optimization for Massively Parallel Data Processing. In *Proc. of the 2nd Symp. on Cloud Computing*. ACM, 2011.

[194] Reynold Xin, Joseph Gonzalez, and Michael Franklin. GraphX: A Resilient Distributed Graph System on Spark. In *Proc. of the ACM SIGMOD GRADES Workshop*. ACM, 2013.

[195] Reynold Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and Rich Analytics at Scale. Technical Report UCB/EECS-2012-214, University of California, Berkeley, 2012.

[196] Mark Yong, Nitin Garegrat, and Shiwali Mohan. Towards a Resource Aware Scheduler in Hadoop. In *Proc. of the 2009 IEEE Intl. Conf. on Web Services*, pages 102–109. IEEE, 2009.

[197] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, pages 247–260. ACM, 2009.

[198] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. of the 5th European Conf. on Computer Systems*, pages 265–278. ACM, 2010.

[199] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 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*, pages 2–2. USENIX Association, 2012.

[200] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing*. USENIX Association, 2010.

[201] Bernhard Zeller and Alfons Kemper. Experience Report: Exploiting Advanced Database Optimization Features for Large-Scale SAP R/3 Installations. In *Proc. of the 28st Intl. Conf. on Very Large Data Bases*, pages 894–905. VLDB Endowment, 2002.

[202] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. PrIter: A Distributed Framework for Prioritized Iterative Computations. In *Proc. of the 2nd Symp. on Cloud Computing*, pages 13:1–13:14. ACM, 2011.

[203] Yi Zhang, Herodotos Herodotou, and Jun Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *Proc. of the 4th Biennial Conf. on Innovative Data Systems Research*, 2009.

[204] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal*, 21(5):611–636, 2012.

[205] Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *Proc. of the 26th IEEE Intl. Conf. on Data Engineering*, pages 1060–1071. IEEE, 2010.

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

[207] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, pages 723–734. VLDB Endowment, 2007.