The VLDB Journal (2014) 23:355–380
DOI 10.1007/s00778-013-0319-9

REGULAR PAPER

A survey of large-scale analytical query processing in MapReduce

Christos Doulkeridis · Kjetil Nørvåg

Received: 28 September 2012 / Revised: 28 February 2013 / Accepted: 13 May 2013 / Published online: 8 June 2013

Abstract Enterprises today acquire vast volumes of data from different sources and leverage this information by means of data analysis to support effective decision-making and provide new functionality and services. The key requirement of data analytics is scalability, simply due to the immense volume of data that need to be extracted, processed, and analyzed in a timely fashion. Arguably the most popular framework for contemporary large-scale data analytics is MapReduce, mainly due to its salient features that include scalability, fault-tolerance, ease of programming, and flexibility. However, despite its merits, MapReduce has evident performance limitations in miscellaneous analytical tasks, and this has given rise to a significant body of research that aim at improving its efficiency, while maintaining its desirable properties. This survey aims to review the state of the art in improving the performance of parallel query processing using MapReduce. A set of the most significant weaknesses and limitations of MapReduce is discussed at a high level, along with solving techniques. A taxonomy is presented for categorizing existing research on MapReduce improvements according to the specific problem they target. Based on the proposed taxonomy, a classification of existing research is provided focusing on the optimization objective. Concluding, we outline interesting directions for future parallel data processing systems.

C. Doulkeridis (✉)
Department of Digital Systems, University of Piraeus, 18534 Piraeus, Greece
e-mail: cdoulk@unipi.gr

K. Nørvåg
Department of Computer and Information Science, Norwegian University of Science and Technology, 7491 Trondheim, Norway
e-mail: Kjetil.Norvag@idi.ntnu.no

Keywords MapReduce · Survey · Data analysis · Query processing · Large-scale · Big Data

1 Introduction

In the era of "Big Data", characterized by the unprecedented volume of data, the velocity of data generation, and the variety of the structure of data, support for large-scale data analytics constitutes a particularly challenging task. To address the scalability requirements of today's data analytics, parallel shared-nothing architectures of commodity machines (often consisting of thousands of nodes) have been lately established as the de-facto solution. Various systems have been developed mainly by the industry to support Big Data analysis, including Google's MapReduce [32,33], Yahoo's PNUTS [31], Microsoft's SCOPE [112], Twitter's Storm [70], LinkedIn's Kafka [46], and WalmartLabs' Muppet [66]. Also, several companies, including Facebook [13], both use and have contributed to Apache Hadoop (an open-source implementation of MapReduce) and its ecosystem.

MapReduce has become the most popular framework for large-scale processing and analysis of vast data sets in clusters of machines, mainly because of its simplicity. With MapReduce, the developer gets various cumbersome tasks of distributed programming for free without the need to write any code; indicative examples include machine to machine communication, task scheduling to machines, scalability with cluster size, ensuring availability, handling failures, and partitioning of input data. Moreover, the open-source Apache Hadoop implementation of MapReduce has contributed to its widespread usage both in industry and academia. As a witness to this trend, we counted the number of papers related to MapReduce and cloud computing published yearly in major database conferences.[1] We report a significant increase from 12 in 2008 to 69 papers in 2012.

Despite its popularity, MapReduce has also been the object of severe criticism [87,98], mainly due to its performance limitations, which arise in various complex processing tasks. For completeness, it should be mentioned that MapReduce has been defended in [34]. Our article analyzes the limitations of MapReduce and surveys existing approaches that aim to address its shortcomings. We also describe common problems encountered in processing tasks in MapReduce and provide a comprehensive classification of existing work based on the problem they attempt to address.

Scope and Aim of this Survey. This survey focuses primarily on query processing aspects in the context of data analytics over massive data sets in MapReduce. A broad coverage of existing work in the area of improving analytical query processing using MapReduce is provided. In addition, this survey offers added value by means of a comprehensive classification of existing approaches based on the problem they try to solve. The topic is approached from a data-centric perspective, thus highlighting the importance of typical data management problems related to efficient parallel processing and analysis of large-scale data.

This survey aims to serve as a useful guidebook of problems and solving techniques in processing data with MapReduce, as well as a point of reference for future work in improving the MapReduce execution framework or introducing novel systems and frameworks for large-scale data analytics. Given the already significant number of research papers related to MapReduce-based processing, this work also aims to provide a clear overview of the research field to the new researcher who is unfamiliar with the topic, as well as record and summarize the already existing knowledge for the experienced researcher in a meaningful way. Last but not least, this survey provides a comparison of the proposed techniques and exposes their potential advantages and disadvantages as much as possible.

Related Work. Probably the most relevant work to this article is the recent survey on parallel data processing with MapReduce [68]. However, our article provides a more in-depth analysis of limitations of MapReduce and classifies existing approaches in a comprehensive way. Other related work includes the tutorials on data layouts and storage in MapReduce [35] and on programming techniques for MapReduce [93]. A comparison of parallel DBMSs versus MapReduce that criticizes the performance of MapReduce is provided in [87,98]. The work in [57] suggests five design factors that improve the overall performance of Hadoop, thus making it more comparable to parallel database systems.

[1] The count is based on articles that appear in the proceedings of ICDE, SIGMOD, VLDB, thus includes research papers, demos, keynotes, and tutorials.

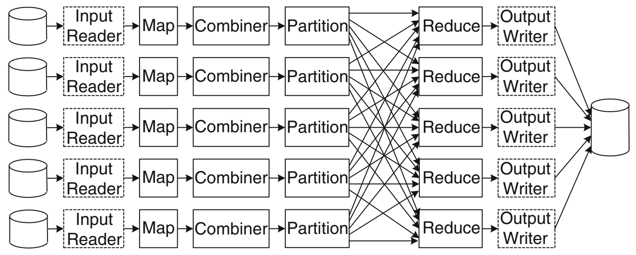

Fig. 1 MapReduce dataflow

Of separate interest is the survey on systems for large-scale data management in the cloud [91] and Cattell's survey on NoSQL data stores [25]. Furthermore, the tutorials on Big Data and cloud computing [10] and on the I/O characteristics of NoSQL databases [92] as well as the work of Abadi on limitations and opportunities for cloud data management [1] are also related.

Organization of this Paper. The remainder of this article is organized as follows: Sect. 2 provides an overview of MapReduce focusing on its open-source implementation Hadoop. Then, in Sect. 3, an outline of weaknesses and limitations of MapReduce are described in detail. Section 4 organizes existing approaches that improve the performance of query processing in a taxonomy of categories related to the main problem they solve and classifies existing work according to optimization goal. Finally, Sect. 5 identifies opportunities for future work in the field.

2 MapReduce basics

2.1 Overview

MapReduce [32] is a framework for parallel processing of massive data sets. A *job* to be performed using the MapReduce framework has to be specified as two phases: the *map* phase as specified by a Map function (also called *mapper*) takes key/value pairs as input, possibly performs some computation on this input, and produces intermediate results in the form of key/value pairs; and the *reduce* phase which processes these results as specified by a Reduce function (also called *reducer*). The data from the map phase are *shuffled*, i.e., exchanged and merge-sorted, to the machines performing the reduce phase. It should be noted that the shuffle phase can itself be more time-consuming than the two others depending on network bandwidth availability and other resources.

In more detail, the data are processed through the following 6 steps [32] as illustrated in Fig. 1:

1. *Input reader:* The input reader in the basic form takes input from files (large blocks) and converts them to

key/value pairs. It is possible to add support for other input types, so that input data can be retrieved from a database or even from main memory. The data are divided into *splits*, which are the unit of data processed by a *map task*. A typical split size is the size of a block, which for example in HDFS is 64 MB by default, but this is configurable.
2. *Map function:* A map task takes as input a key/value pair from the input reader, performs the logic of the Map function on it, and outputs the result as a new key/value pair. The results from a map task are initially output to a main memory buffer, and when almost full *spill* to disk. The spill files are in the end merged into one sorted file.
3. *Combiner function:* This optional function is provided for the common case when there is (1) significant repetition in the intermediate keys produced by each map task, and (2) the user-specified Reduce function is commutative and associative. In this case, a Combiner function will perform partial reduction so that pairs with same key will be processed as one group by a reduce task.
4. *Partition function:* As default, a hashing function is used to partition the intermediate keys output from the map tasks to reduce tasks. While this in general provides good balancing, in some cases it is still useful to employ other partitioning functions, and this can be done by providing a user-defined Partition function.
5. *Reduce function:* The Reduce function is invoked once for each distinct key and is applied on the set of associated values for that key, i.e., the pairs with same key will be processed as one group. The input to each reduce task is guaranteed to be processed in increasing key order. It is possible to provide a user-specified comparison function to be used during the sort process.
6. *Output writer:* The output writer is responsible for writing the output to stable storage. In the basic case, this is to a file, however, the function can be modified so that data can be stored in, e.g., a database.

As can be noted, for a particular job, only a Map function is strictly needed, although for most jobs a Reduce function is also used. The need for providing an Input reader and Output writer depends on data source and destination, while the need for Combiner and Partition functions depends on data distribution.

2.2 Hadoop

Hadoop [104] is an open-source implementation of MapReduce, and without doubt, the most popular MapReduce variant currently in use in an increasing number of prominent companies with large user bases, including companies such as Yahoo! and Facebook.

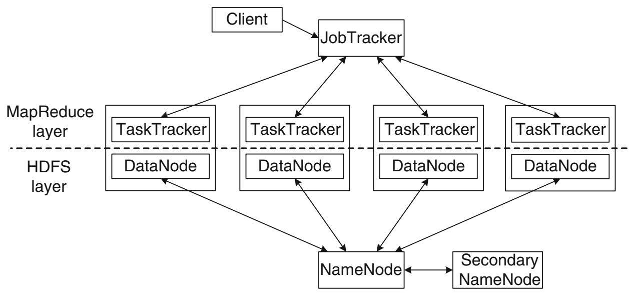

Fig. 2 Hadoop architecture

Hadoop consists of two main parts: the Hadoop distributed file system (HDFS) and MapReduce for distributed processing. As illustrated in Fig. 2, Hadoop consists of a number of different daemons/servers: NameNode, DataNode, and Secondary NameNode for managing HDFS, and JobTracker and TaskTracker for performing MapReduce.

HDFS is designed and optimized for storing very large files and with a streaming access pattern. Since it is expected to run on commodity hardware, it is designed to take into account and handle failures on individual machines. HDFS is normally not the primary storage of the data. Rather, in a typical workflow, data are copied over to HDFS for the purpose of performing MapReduce, and the results then copied out from HDFS. Since HDFS is optimized for streaming access of large files, random access to parts of files is significantly more expensive than sequential access, and there is also no support for updating files, only append is possible. The typical scenario of applications using HDFS follows a write-once read-many access model.

Files in HDFS are split into a number of large blocks (usually a multiple of 64 MB) which are stored on *DataNodes*. A file is typically distributed over a number of DataNodes in order to facilitate high bandwidth and parallel processing. In order to improve reliability, data blocks in HDFS are replicated and stored on three machines, with one of the replicas in a different rack for increasing availability further. The maintenance of file metadata is handled by a separate *NameNode*. Such metadata includes mapping from file to block and location (DataNode) of block. The NameNode periodically communicates its metadata to a *Secondary NameNode* which can be configured to do the task of the NameNode in case of the latter's failure.

MapReduce Engine. In Hadoop, the *JobTracker* is the access point for clients. The duty of the JobTracker is to ensure fair and efficient scheduling of incoming MapReduce jobs, and assign the tasks to the *TaskTrackers* which are responsible for execution. A TaskTracker can run a number of tasks depending on available resources (for example two map tasks and two reduce tasks) and will be allocated a new task by the JobTracker when ready. The relatively small size of each task compared to the large number of tasks in

Table 1 Weaknesses in MapReduce and solving techniques at a high level

Weakness	Technique
Access to input data	Indexing and data layouts
High communication cost	Partitioning and colocation
Redundant and wasteful processing	Result sharing, batch processing of queries and incremental processing
Recomputation	Materialization
Lack of early termination	Sampling and sorting
Lack of iteration	Loop-aware processing, caching, pipelining, recursion, incremental processing
Quick retrieval of approximate results	Data summarization and sampling
Load balancing	Pre-processing, approximation of the data distribution and repartitioning
Lack of interactive or real-time processing	In-memory processing, pipelining, streaming and pre-computation
Lack of support for *n*-way operations	Additional MR phase(s), re-distribution of keys and record duplication

total helps to ensure load balancing among the machines. It should be noted that while the number of map tasks to be performed is based on the input size (number of splits), the number of reduce tasks for a particular job is user-specified.

In a large cluster, machine failures are expected to occur frequently, and in order to handle this, regular heartbeat messages are sent from TaskTrackers to the JobTracker periodically and from the map and reduce tasks to the TaskTracker. In this way, failures can be detected, and the JobTracker can reschedule the failed task to another TaskTracker. Hadoop follows a speculative execution model for handling failures. Instead of fixing a failed or slow-running task, it executes a new equivalent task as backup. Failure of the JobTracker itself cannot be handled automatically, but the probability of failure of one particular machine is low so that this should not present a problem in general.

The Hadoop Ecosystem. In addition to the main components of Hadoop, the Hadoop ecosystem also contains other libraries and systems. The most important in our context are HBase, Hive, and Pig. *HBase* [45] is a distributed column-oriented store, inspired by Google's Bigtable [26] that runs on top of HDFS. Tables in HBase can be used as input or output for MapReduce jobs, which is especially useful for random read/write access. *Hive* [99,100] is a data warehousing infrastructure built on top of Hadoop. Queries are expressed in an SQL-like language called HiveQL, and the queries are translated and executed as MapReduce jobs. *Pig* [86] is a framework consisting of the Pig Latin language and its execution environment. Pig Latin is a procedural scripting language making it possible to express data workflows on a higher level than a MapReduce job.

3 Weaknesses and limitations

Despite its evident merits, MapReduce often fails to exhibit acceptable performance for various processing tasks. Quite often this is a result of weaknesses related to the nature of MapReduce or the applications and use-cases it was originally designed for. In other cases, it is the product of limitations of the processing model adopted in MapReduce. In particular, we have identified a list of issues related to large-scale data processing in MapReduce/Hadoop that significantly impact its efficiency (for a quick overview, we refer to Table 1):

– *Selective access to data*: Currently, the input data to a job is consumed in a brute-force manner, in which the entire input is scanned in order to perform the map-side processing. Moreover, given a set of input data partitions stored on DataNodes, the execution framework of MapReduce will initiate map tasks on all input partitions. However, for certain types of analytical queries, it would suffice to access only a subset of the input data to produce the result. Other types of queries may require focused access to a few tuples only that satisfy some predicate, which cannot be provided without accessing and processing all the input data tuples. In both cases, it is desirable to provide a selective access mechanism to data, in order to prune local non-useful data at a DataNode from processing as well as prune entire DataNodes from processing. In traditional data management systems, this problem is solved by means of *indexing*. In addition, since HDFS blocks are typically large, it is important to optimize their internal organization (*data layout*) according to the query workload to improve the performance of data access. For example, queries that involve only few attributes benefit from a columnar layout that allows fetching of specific columns only, while queries that involve most of the attributes perform better when row-wise storage is used.
– *High communication cost*: After the map tasks have completed processing, some selected data are sent to reduce tasks for further processing (shuffling). Depending on the query at hand and on the type of processing that takes place during the map phase, the size of the output of the map phase can be significant and its transmission may delay the overall execution time of the job. A typical example of such a query is a join, where it is not possible for a map task to eliminate input data from being sent to reduce tasks, since this data may be joined with other data consumed by other map tasks. This problem is also present in distributed data management systems, which

address it by careful *data partitioning* and *placement* of partitions that need to be processed together at the same node.

- *Redundant and wasteful processing*: Quite often multiple MapReduce jobs are initiated at overlapping time intervals and need to be processed over the same set of data. In such cases, it is possible that two or more jobs need to perform the same processing over the same data. In MapReduce, such jobs are processed independently, thus resulting in redundant processing. As an example, consider two jobs that need to scan the same query log, one trying to identify frequently accessed pages and another performing data mining on the activity of specific IP addresses. To alleviate this shortcoming, jobs with similar subtasks should be identified and *processed together in a batch*. In addition to sharing common processing, *result sharing* is a well-known technique that can be employed to eliminate wasteful processing over the same data.
- *Recomputation*: Jobs submitted to MapReduce clusters produce output results that are stored on disk to ensure fault-tolerance during the processing of the job. This is essentially a check-pointing mechanism that allows long-running jobs to complete processing in the case of failures, without the need to restart processing from scratch. However, MapReduce lacks a mechanism for management and future reuse of output results. Thus, there exists no opportunity for reusing the results produced by previous queries, which means that a future query that requires the result of a previously posed query will have to resolve in *recomputing* everything. In database systems, query results that are expected to be useful in the future are materialized on disk and are available at any time for further consumption, thus achieving significant performance benefits. Such a materialization mechanism that can be enabled by the user is missing from MapReduce and would boost its query processing performance.
- *Lack of early termination*: By design, in the MapReduce execution framework, map tasks must access input data in its entirety before any reduce task can start processing. Although this makes sense for specific types of queries that need to examine the complete input data in order to produce any result, for many types of queries only a subset of input data suffices to produce the complete and correct result. A typical example arises in the context of rank-aware processing, as in the case of top-k queries. As another prominent example, consider sampling of input data to produce a small and fixed-size set of representative data. Such query types are ubiquitous in data analytics over massive data sets and cannot be efficiently processed in MapReduce. The common conclusion is the *lack of an early termination mechanism* in MapReduce processing, which would allow map tasks to cease processing, when a specific condition holds.
- *Lack of iteration*: Iterative computation and recursive queries arise naturally in data analysis tasks, including PageRank or HITS computation, clustering, social network analysis, recursive SQL queries, etc. However, in MapReduce, the programmer needs to write a sequence of MapReduce jobs and coordinate their execution, in order to implement simple iterative processing. More importantly, a non-negligible performance penalty is paid, since data must be reloaded and reprocessed in each iteration, even though quite often a significant part of the data remains unchanged. In consequence, iterative data analysis tasks cannot be processed efficiently by the MapReduce framework.
- *Quick retrieval of approximate results*: Exploratory queries are a typical requirement of applications that involve analysis of massive data sets, as in the case of scientific data. Instead of issuing exploratory queries to the complete data set that would entail long waiting times and potentially non-useful results, it would be extremely useful to test such queries on small representative parts of the data set and try to draw conclusions. Such queries need to be processed fast and if necessary, return approximate results, so that the scientist can review the result and decide whether the query makes sense and should be processed on the entire data set. However, MapReduce does not provide an explicit way to support *quick retrieval of indicative results* by processing on representative input samples.
- *Load balancing*: Parallel data management systems try to minimize the runtime of a complex processing task by carefully partitioning the input data and distributing the processing load to available machines. Unless the data is distributed in a fair manner, the runtime of the slowest machine will easily dominate the total runtime. For a given machine, its runtime is dependent on various parameters (including the speed of the processor and the size of the memory), however, our main focus is on the effect induced by data assignment. Part of the problem is partitioning the data fairly, such that each machine is assigned an equi-sized data partition. For real data sets, data skew is commonly encountered in practice, thus plain partitioning schemes that are not data-aware, such as those used by MapReduce, easily fall short. More importantly, even when the data is equally split to the available machines, equal runtime may not always be guaranteed. The reason for this is that some partitions may require complex processing, while others may simply contain data that need not be processed for the query at hand (e.g., are excluded based on some predicate). Providing advanced load balancing mechanisms that aim to increase the efficiency of query processing by assigning equal shares of useful work to the available machines is a weakness of MapReduce that has not been addressed yet.

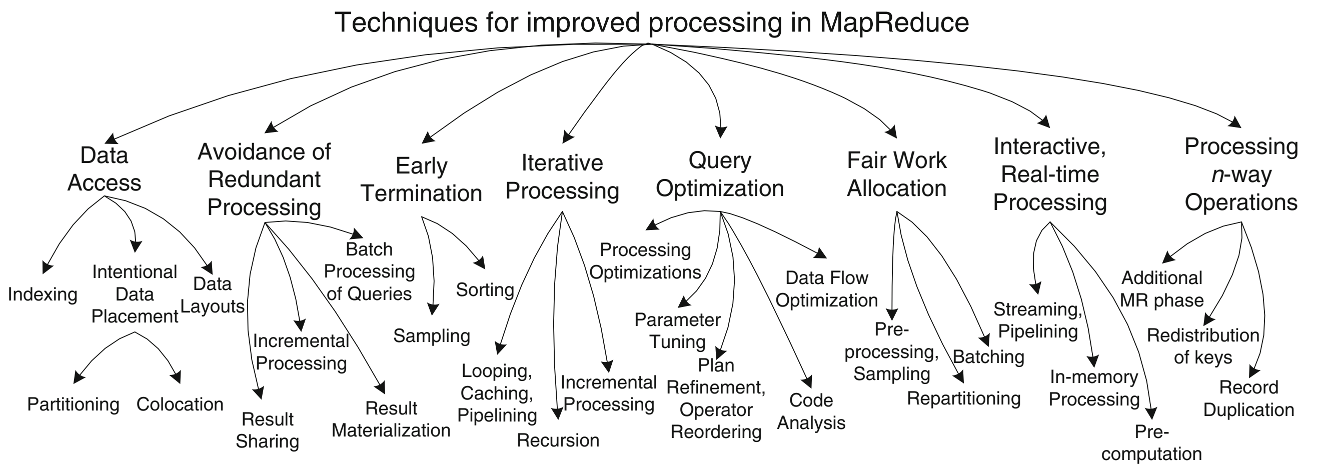

Fig. 3 Taxonomy of MapReduce improvements for efficient query processing

- *Lack of interactive or real-time processing*: MapReduce is designed as a highly fault-tolerant system for batch processing of long-running jobs on very large data sets. However, its very design hinders efficient support for *interactive* or *real-time processing*, which requires fast processing times. The reason is that to guarantee fault-tolerance, MapReduce introduces various overheads that negatively impact its performance. Examples of such overheads include frequent writing of output to disk (e.g., between multiple jobs), transferring big amounts of data in the network, limited exploitation of main memory, delays for job initiation and scheduling, extensive communication between tasks for failure detection. However, numerous applications require fast response times, interactive analysis, online analytics, and these requirements are hardly met by the performance of MapReduce.
- *Lack of support for n-way operations*: Processing n-way operations over data originating from multiple sources is not naturally supported by MapReduce. Such operations include (among others) join, union, intersection, and can be binary operations ($n = 2$) or multi-way operations ($n > 2$). Taking the prominent example of a join, many analytical tasks typically require accessing and processing data from multiple relations. In contrast, the design of MapReduce is not flexible enough to support n-way operations with the same simplicity and intuitiveness as data coming from a single source (e.g., single file).

4 MapReduce improvements

In this section, an overview is provided of various methods and techniques present in the existing literature for improving the performance of MapReduce. All approaches are categorized based on the introduced improvement. We organize the categories of MapReduce improvements in a taxonomy, illustrated in Fig. 3.

Table 2 classifies existing approaches for improved processing based on their optimization objectives. We determine the primary objective (marked with ♠ in the table), and then, we also identify secondary objectives (marked with ◇).

4.1 Data access

Efficient access to data is an essential step for achieving improved performance during query processing. We identify three subcategories of data access, namely indexing, intentional data placement, and data layouts.

4.1.1 Indexing

Hadoop++ [36] is a system that provides indexing functionality for data stored in HDFS by means of User-defined Functions (UDFs), i.e., without modifying the Hadoop framework at all. The indexing information (called Trojan Indexes) is injected into logical input splits and serves as a cover index for the data inside the split. Moreover, the index is created at load time, thus imposing no overhead in query processing. Hadoop++ also supports joins by co-partitioning data and colocating them at load time. Intuitively, this enables the join to be processed at the map side, rather than at the reduce side (which entails expensive data transfer/shuffling in the network). Hadoop++ has been compared against HadoopDB and shown to outperform it [36].

HAIL [37] improves the long index creation times of Hadoop++, by exploiting the n replicas (typically $n = 3$) maintained in Hadoop by default for fault-tolerance and by building a different clustered index for each replica. At query time, the most suitable index to the query is selected, and the particular replica of the data is scanned during the map phase.

Table 2 Classification of existing approaches based on optimization objective (♠ indicates primary objective, while ◇ indicates secondary objectives)

Approach	Data access	Avoidance of redundant processing	Early termination	Iterative processing	Query optimization	Fair work allocation	Interactive, real-time processing
Hadoop++ [36]	♠	◇	◇		◇		
HAIL [37]	♠	◇	◇		◇		
CoHadoop [41]	♠	◇					
Llama [74]	♠	◇					
Cheetah [28]	♠	◇					
RCFile [50]	♠	◇					
CIF [44]	♠	◇					
Trojan layouts [59]	♠	◇			◇		
MRShare [83]		♠			◇		
ReStore [40]		♠			◇		
Sharing scans [11]		♠			◇		
Silva et al. [95]		♠					
Incoop [17]	◇	♠		◇			◇
Li et al. [71,72]		♠					
Grover et al. [47]		◇	♠		◇		
EARL [67]		◇	♠	◇			
Top-*k* queries [38]	◇	◇	♠			◇	
RanKloud [24]	◇	◇	♠			◇	
HaLoop [22,23]		◇		♠			
MapReduce online [30]		◇		♠			
NOVA [85]				♠	◇		
Twister [39]				♠			
CBP [75,76]				♠			
Pregel [78]				♠			
PrIter [111]				♠			
PACMan [14]				♠			
REX [82]		◇		♠	◇		
Differential dataflow [79]		◇		♠			
HadoopDB [2]	◇	◇	◇		♠		
SAMs [101]			◇		♠	◇	
Clydesdale [60]	◇	◇			♠		
Starfish [51]					♠		
AQUA [105]		◇			♠		
YSmart [69]		◇			♠		
RoPE [8]					♠		
SUDO [109]					♠		
Manimal [56]	◇				♠		
HadoopToSQL [54]					♠		
Stubby [73]					♠		
Hueske et al. [52]					♠		
Kolb et al. [62]						♠	
Ramakrishnan et al. [88]						♠	
Gufler et al. [48]						♠	
SkewReduce [64]						♠	
SkewTune [65]						♠	

Table 2 continued

Approach	Data access	Avoidance of redundant processing	Early termination	Iterative processing	Query optimization	Fair work allocation	Interactive, real-time processing
Sailfish [89]						♠	
Themis [90]						♠	
Dremel [80]	◇	◇					♠
Hyracks [19]							♠
Tenzing [27]		◇			◇		♠
PowerDrill [49]	◇	◇					♠
Shark [42,107]				◇			♠
M3R [94]		◇		◇			♠
BlinkDB [7,9]							♠

As a result, HAIL improves substantially the performance of MapReduce processing, since the probability of finding a suitable index for efficient data access is increased. In addition, the creation of the indexes occurs during the data upload phase to HDFS (which is I/O bound), by exploiting "unused CPU ticks", and thus, it does not affect the upload time significantly. Given the availability of multiple indexes, choosing the optimal execution plan for the map phase is an interesting direction for future work. HAIL is shown in [37] to improve index creation times and the performance of Hadoop++. Both Hadoop++ and HAIL support joins with improved efficiency.

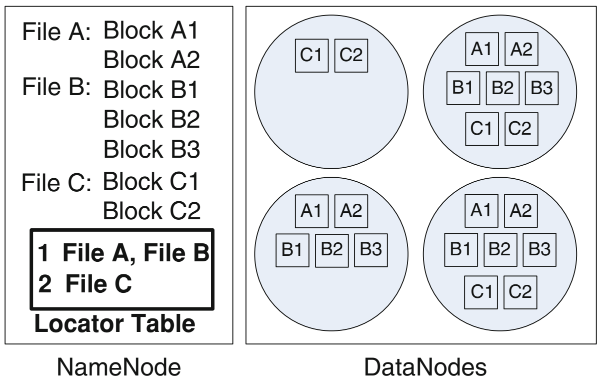

Fig. 4 The way files are colocated in CoHadoop [41]

4.1.2 Intentional data placement

CoHadoop [41] colocates and copartitions data on nodes intentionally, so that related data are stored on the same node. To achieve colocation, CoHadoop extends HDFS with a file-level property (locator), and files with the same locator are placed on the same set of DataNodes. In addition, a new data structure (locator table) is added to the NameNode of HDFS. This is depicted in the example of Fig. 4 using a cluster of five nodes (one NameNode and four DataNodes) where 3 replicas per block are kept. All blocks (including replicas) of files A and B are colocated on the same set of DataNodes, and this is described by a locator present in the locator table (shown in bold) of the NameNode. The contributions of CoHadoop include colocation and copartitioning, and it targets a specific class of queries that benefit from these techniques. For example, joins can be efficiently executed using a map-only join algorithm, thus eliminating the overhead of data shuffling and the reduce phase of MapReduce. However, applications need to provide hints to CoHadoop about related files, and therefore, one possible direction for improvement is to automatically identify related files. CoHadoop is compared against Hadoop++, which also supports copartitioning and colocating data at load time, and demonstrates superior performance for join processing.

4.1.3 Data layouts

A nice detailed overview of the exploitation of different data layouts in MapReduce is presented in [35]. We provide a short overview in the following.

Llama [74] proposes the use of a columnar file (called CFile) for data storage. The idea is that data are partitioned in vertical groups, each group is sorted based on a selected column and stored in column-wise format in HDFS. This enables selective access only to the columns used in the query. In consequence, more efficient access to data than traditional row-wise storage is provided for queries that involve a small number of attributes.

Cheetah [28] also employs data storage in columnar format and also applies different compression techniques for different types of values appropriately. In addition, each cell is further compressed after it is created using GZIP. Cheetah employs the PAX layout [12] at the block level, so each block contains the same set of rows as in row-wise storage, only inside the block column layout is employed. Compared to Llama, the important benefit of *Cheetah* is that all data that belong to a record are stored in the same block, thus avoiding expensive network access (as in the case of CFile).

RCFile [50] combines horizontal with vertical partitioning to avoid wasted processing in terms of decompression of unnecessary data. Data are first partitioned horizontally, and then, each horizontal partition is partitioned vertically. The benefits are that columns belonging to the same row are located together on the same node thus avoiding network access, while compression is also possible within each vertical partition thus supporting access only to those columns that participate in the query. Similarly to Cheetah, *RCFile* also employs PAX, however, the main difference is that RCFile does not use block-level compression. RCFile has been proposed by Facebook and is extensively used in popular systems, such as Hive and Pig.

CIF [44] proposed a column-oriented, binary storage format for HDFS aiming to improve its performance. The idea is that each file is first horizontally partitioned in splits, and each split is stored in a subdirectory. The columns of the records of each split are stored in individual files within the respective subdirectory, together with a file containing metadata about the schema. When a query arrives that accesses some columns, multiple files from different subdirectories are assigned in one split and records are reconstructed following a lazy approach. *CIF* is compared against RCFile and shown to outperform it. Moreover, it does not require changes to the core of Hadoop.

Trojan data layouts [59] also follow the spirit of PAX, however, data inside a block can have any data layout. Exploiting the replicas created by Hadoop for fault-tolerance, different Trojan layouts are created for each replica, thus the most appropriate layout can be used depending on the query at hand. It is shown that queries that involve almost all attributes should use files with row-level layout, while selective queries should use column layout. Trojan data layouts have been shown to outperform PAX-based layouts in Hadoop [59].

4.2 Avoiding redundant processing

MRShare [83] is a sharing framework that identifies different queries (jobs) that share portions of identical work. Such queries do not need to be recomputed each time from scratch. The main focus of this work is to save I/O, and therefore, sharing opportunities are identified in terms of sharing scans and sharing map-output. MRShare transforms sets of submitted jobs into groups and treat each group as a single job, by solving an optimization problem with objective to maximize the total savings. To process a group of jobs as a single job, MRShare modifies Hadoop to (a) tag map output tuples with tags that indicate the tuple's originating jobs, and (b) write to multiple output files on the reduce side. Open problems include extending MRShare to support jobs that use multiple inputs (e.g., joins) as well as sharing parts of the Map function.

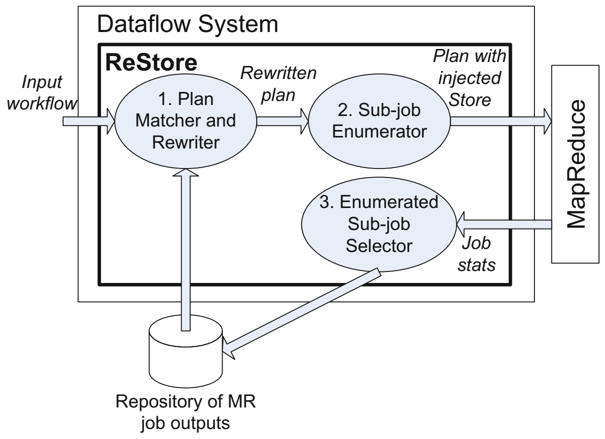

Fig. 5 System architecture of ReStore [40]

ReStore [40] is a system that manages the storage and reuse of intermediate results produced by workflows of MapReduce jobs. It is implemented as an extension to the Pig dataflow system. ReStore maintains the output results of MapReduce jobs in order to identify reuse opportunities by future jobs. To achieve this, ReStore maintains together with a file storing a job's output the physical execution plan of the query and some statistics about the job that produced it, as well as how frequently this output is used by other workflows. Figure 5 shows the main components of ReStore: (1) the plan matcher and rewriter, (2) the subjob enumerator, and (3) the enumerated subjob selector. The figure shows how these components interact with MapReduce as well as the usage of the repository of MapReduce job outputs. The plan matcher and rewriter identifies outputs of past jobs in the repository that can be used to answer the query and rewrites the input job to reuse the discovered outputs. The subjob enumerator identifies the subsets of physical operators that can be materialized and stored in the DFS. Then, the enumerated subjob selector chooses which outputs to keep in the repository based on the collected statistics from the MapReduce job execution. The main difference to MRShare is that ReStore allows individual queries submitted at different times to share results, while MRShare tries to optimize sharing for a batch of queries executed concurrently. On the downside, ReStore introduces some overhead to the job's execution time and to the available storage space, when it decides to store the results of subjobs, especially for large-sized results.

Sharing scans from a common set of files in the MapReduce context has been studied in [11]. The authors study the problem of scheduling sharable jobs, where a set of files needs to be accessed simultaneously by different jobs. The aim is to maximize the rate of processing these files by aggressively sharing scans, which is important for jobs whose execution time is dominated by data access, yet the input data cannot be cached in memory due to its size. In MapReduce, such jobs have typically long execution times, and therefore, the proposed solution is to reorder their execution to amortize expensive data access across multiple jobs. Traditional scheduling algorithms (e.g., shortest job first) do not work well for sharable jobs, and thus, new techniques suited for the effective scheduling of sharable workloads are proposed. Intuitively, the proposed scheduling policies prioritize scheduling non-sharable scans ahead of ones that can share I/O work with future jobs, if the arrival rate of sharable future jobs is expected to be high.

The work by Silva et al. [95] targets cost-based optimization of complex scripts that share common subexpressions, and the approach is prototyped in Microsoft's SCOPE [112]. Such scripts, if processed naively, may result in executing the same subquery multiple times, leading to redundant processing. The main technical contribution is dealing with competing physical requirements from different operators in a DAG, in a way that leads to a globally optimal plan. An important difference to existing techniques is that locally optimizing the cost of shared subexpressions does not necessarily lead to an overall optimal plan, and thus, the proposed approach considers locally suboptimal local plans that can generate a globally optimal plan.

Many MapReduce workloads are incremental, resulting in the need for MapReduce to run repeatedly with small changes in the input. To process data incrementally using MapReduce, users have to write their own application-specific code. *Incoop* [17] allows existing MapReduce programs to execute transparently in an incremental manner. Incoop detects changes to the input and automatically updates the output. In order to achieve this, a HDFS-like file system called Inc-HDFS is proposed, and techniques for controlling granularity are presented that make it possible to avoid redoing the full task when only parts of it need to be processed. The approach has a certain extra cost in the case where no computations can be reused.

Support for incremental processing of new data is also studied in [71,72]. The motivation is to support one-pass analytics for applications that continuously generate new data. An important observation of this work is that support for incremental processing requires non-blocking operations and avoidance of bottlenecks in the processing flow, both computational and I/O-specific. The authors' main finding is to abandon the sort-merge implementation for data partitioning and parallel processing, for purely hash-based techniques which are non-blocking. As a result, the sort cost of the map phase is eliminated, and the use of hashing allows fast in-memory processing of the Reduce function.

4.3 Early termination

Sampling based on predicates raises the issue of *lack of early termination* of map tasks [47], even when a job has read enough input to produce the required output. The problem addressed by [47] is how to produce a fixed-size sample (that satisfies a given predicate) of a massive data set using MapReduce. To achieve this objective, some technical issues need to be addressed on Hadoop, and thus, two new concepts are defined. A new type of job is introduced, called *dynamic*, which is able to dynamically control its data access. In addition, the concept of *Input Provider* is introduced in the execution model of Hadoop. The Input Provider is provided by the job together with the Map and Reduce logic. Its role is to make dynamic decisions about the access to data by the job. At regular intervals, the JobClient provides statistics to the Input Provider, and based on this information, the Input Provider can respond in three different ways: (1) "end of input", in which case the running map tasks are allowed to complete, but no new map tasks are invoked and the shuffle phase is initiated, (2) "input available", which means that additional input needs to be accessed, and (3) "input unavailable", which indicates that no decision can be made at this point and processing should continue as normal until the next invocation of Input Provider.

EARL [67] aims to provide early results for analytical queries in MapReduce, without processing the entire input data. EARL utilizes uniform sampling and works iteratively to compute larger samples, until a given accuracy level is reached, estimated by means of bootstrapping. Its main contributions include incremental computation of early results with reliable accuracy estimates, which is increasingly important for applications that handle vast amounts of data. Moreover, EARL employs delta maintenance to improve the performance of re-executing a job on a larger sample size. At a technical level, EARL modifies Hadoop in three ways: (1) reduce tasks can process input before the completion of processing in the map tasks by means of pipelining, (2) map tasks remain active until explicitly terminated, and (3) an inter-communication channel between map tasks and reduce tasks is established for checking the satisfaction of the termination condition. In addition, a modified reduce phase is employed, in which incremental processing of a job is supported. Compared to [47], EARL provides an error estimation framework and focuses more on using truly uniform random samples.

The lack of early termination has been recognized in the context of rank-aware processing in MapReduce [38], which is also not supported efficiently. Various individual techniques are presented to support early termination for top-k processing, including the use of sorted access to data, intelligent data placement using advanced partitioning schemes tailored to top-k queries, and the use of synopses for the data stored in HDFS that allow efficient identification of blocks with promising tuples. Most of these techniques can be combined to achieve even greater performance gains.

RanKloud [24] has been proposed for top-k retrieval in the cloud. RanKloud computes statistics (at runtime) during scanning of records and uses these statistics to compute a threshold (the lowest score for top-k results) for early termination. In addition, a new partitioning method is proposed, termed uSplit, that aims to repartition data in a utility-sensitive way, where utility refers to the potential of a tuple to be part of the top-k. The main difference to [38] is that RanKloud cannot guarantee retrieval of k results, while [38] aims at retrieval of the exact result.

4.4 Iterative processing

The straightforward way of implementing iteration is to use an outsider driver program to control the execution of loops and launch new MapReduce jobs in each iteration. For example, this is the approach followed by *Mahout*. In the following, we review iterative processing using looping constructs, caching and pipelining, recursive queries, and incremental iterations.

4.4.1 Loop-aware processing, caching and pipelining

HaLoop [22,23] is a system designed for supporting iterative data analysis in a MapReduce-style architecture. Its main goals include avoidance of processing invariant data at each iteration, support for termination checking without the need for initiating an extra devoted job for this task, maintaining the fault-tolerant features of MapReduce, and seamless integration of existing MapReduce applications with minor code changes. However, several modifications need to be applied at various stages of MapReduce. First, an appropriate programming interface is provided for expressing iteration. Second, the scheduler is modified to ensure that tasks are assigned to the same nodes in each iteration, thus enabling the use of local caches. Third, invariant data is cached and does not need to be reloaded at each iteration. Finally, by caching the reduce task's local output, it is possible to support comparisons of results of successive iterations in an efficient way and allow termination when convergence is identified. Figure 6 shows how these modifications are reflected to specific components in the architecture. It depicts the new loop control module as well as the modules for local caching and indexing in the HaLoop framework. The loop control is responsible for initiating new MapReduce steps (loops) until a user-specified termination condition is fulfilled. HaLoop uses three types of caches: the map task and reduce task input caches, as well as the reduce task output cache. In addition, to improve performance, cached data are indexed. The task scheduler is modified to become loop-aware and exploits

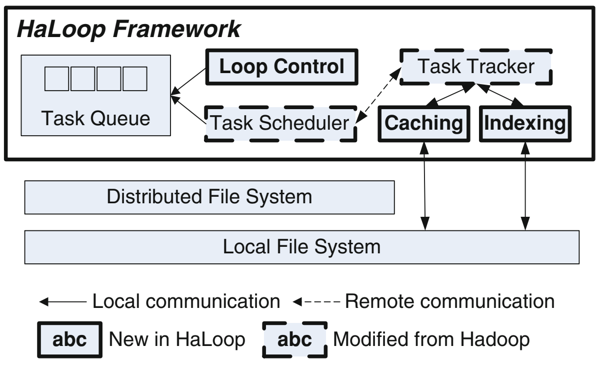

Fig. 6 Overview of the HaLoop framework [22,23]

local caches. Also, failure recovery is achieved by coordination between the task scheduler and the task trackers.

MapReduce online [30] proposes to overcome the built-in feature of materialization of output results of map tasks and reduce tasks, by providing support for pipelining of intermediate data from map tasks to reduce tasks. This is achieved by applying significant changes to the MapReduce framework. In particular, modification to the way data is transferred between map task and reduce tasks as well as between reduce tasks and new map tasks is necessary, and also the TaskTracker and JobTracker have to be modified accordingly. An interesting observation is that pipelining allows the execution framework to support continuous queries, which is not possible in MapReduce. Compared to HaLoop, it lacks the ability to cache data between iterations for improving performance.

NOVA [85] is a workflow manager developed by Yahoo! for incremental processing of continuously arriving data. NOVA is implemented on top of Pig and Hadoop, without changing their interior logic. However, NOVA contains modules that communicate with Hadoop/HDFS, as in the case of the Data Manager module that maintains the mapping from blocks to the HDFS files and directories. In summary, NOVA introduces a new layer of architecture on top of Hadoop to support continuous processing of arriving data.

Twister [39] introduces an extended programming model and a runtime for executing iterative MapReduce computations. It relies on a publish/subscribe mechanism to handle all communication and data transfer. On each node, a daemon process is initiated that is responsible for managing locally running map and reduce tasks, as well as for communication with other nodes. The architecture of Twister differs substantially from MapReduce, with indicative examples being the assumption that input data in local disks are maintained as native files (differently than having a distributed file system), and that intermediate results from map processes are maintained in memory, rather than stored on disk. Some limitations include the need to break large data sets to multiple files, the assumption that map output fits in the distributed memory, and no intra-iteration fault-tolerance.

CBP [75,76] (Continuous Bulk Processing) is an architecture for maintaining state during bulk processing, thus enabling the implementation of applications that support incremental analytics. CBP introduces a new processing operator, called *translate*, that has two distinguishing features: it takes state as an explicit input and supports group-based processing. In this way, CBP achieves maintenance of persistent state, thus re-using work that has been carried out already (incremental processing), while at the same time it drastically reduces data movement in the system.

Pregel [78] is a scalable system for graph processing where a program is a sequence of iterations (called supersteps). In each superstep, a vertex sends and receives messages, updates its state as well as the state of its outgoing edges, and modifies the graph topology. The graph is partitioned, and the partitions are assigned to machines (workers), while this assignment can be customized to improve locality. Each worker maintains in memory the state of its part of the graph. To aggregate global state, tree-based aggregation is used from workers to a master machine. However, to recover from a failure during some iteration, Pregel needs to re-execute all vertices in the iteration. Furthermore, it is restricted to graph data representation, for example it cannot directly support non-graph iterative data mining algorithms, such as clustering (e.g., k-means) or dimensionality reduction (e.g., multi-dimensional scaling).

PrIter [111] is a distributed framework for faster convergence of iterative tasks by support for prioritized iteration. In PrIter, users can specify the priority of each processing data point, and in each iteration, only some data points with high priority values are processed. The framework supports maintenance of previous states across iterations (not only the previous iteration's), prioritized execution, and termination check. A StateTable, implemented as an in-memory hash table, is kept at reduce side to maintain state. Interestingly, a disk-based extension of PrIter is also proposed for cases that the maintained state does not fit in memory.

In many iterative applications, some data are used in several iterations, thus caching this data can avoid many disk accesses. *PACMan* [14] is a caching service that increases the performance of parallel jobs in a cluster. The motivation is that clusters have a large amount of (aggregated) memory, and this is underutilized. The memory can be used to cache input data, however, if there are enough slots for all tasks to run in parallel, caching will only help if all tasks have their input data cached, otherwise those that have not will be stragglers and caching is of no use. To facilitate this, PACMan provides coordinated management of the distributed caches, using cache-replacement strategies developed particularly for this context. The improvements depend on the amount of data re-use, and it is therefore not beneficial for all application workloads.

4.4.2 Recursive queries

Support for recursive queries in the context of MapReduce is studied in [3]. Examples of recursive queries that are meaningful to be implemented in MapReduce include PageRank computation, transitive closure, and graph processing. The authors explore ways to implement recursion. A significant observation is that recursive tasks usually need to produce some output before completion, so that it can be used as feedback to the input. Hence, the blocking property of map and reduce (a task must complete its work before delivering output to other tasks) violates the requirements of recursion.

In [21], recursive queries for machine learning tasks using Datalog are proposed over a data-parallel engine (Hyracks [19]). The authors argue in favor of a global declarative formulation of iterative tasks using Datalog, which offers various optimization opportunities for physical dataflow plans. Two programming models (Pregel [78] and iterative MapReduce) are shown that they can be captured in Datalog, thus demonstrating the generality of the proposed framework.

4.4.3 Incremental processing

REX [82] is a parallel query processing platform that also supports recursive queries expressed in extended SQL, but focuses mainly on incremental refinement of results. In REX, incremental updates (called programmable deltas) are used for state refinement of operators, instead of accumulating the state produced by previous iterations. Its main contribution is increased efficiency of iterative processing, by only computing and propagating deltas describing changes between iterations, while maintaining state that has not changed. REX is based on a different architecture than MapReduce, and also uses cost-based optimization to improve performance. REX is compared against HaLoop and shown to outperform it in various setups.

An iterative data flow system is presented in [43], where the key novelty is support for incremental iterations. Such iterations are typically encountered in algorithms that entail sparse computational dependencies (e.g., graph algorithms), where the result of each iteration differs only slightly from the previous result.

Differential dataflow [79] is a system proposed for improving the performance of incremental processing in a parallel dataflow context. It relies on *differential computation* to update the state of computation when its inputs change, but uses a partially ordered set of versions, in contrast to a totally ordered sequence of versions used by traditional incremental computation. This results in maintaining the differences for multiple iterations. However, the state of each version can be reconstructed by indexing the related set of updates, in contrast to consolidating updates and discarding them. Thus, differential dataflow supports more effective re-use of any previous state, both for changes due to an updated input and due to iterative execution.

4.5 Query optimization

4.5.1 Processing optimizations

HadoopDB [2] is a hybrid system, aiming to exploit the best features of MapReduce and parallel DBMSs. The basic idea behind HadoopDB is to install a database system on each node and connect these nodes by means of Hadoop as the task coordinator and network communication layer. Query processing on each node is improved by assigning as much work as possible to the local database. In this way, one can harvest all the benefits of query optimization provided by the local DBMS. Particularly, in the case of joins, where Hadoop does not support data colocation, a significant amount of data needs to be repartitioned by the join key in order to be processed by the same Reduce process. In HadoopDB, when the join key matches the partitioning key, part of the join can be processed locally by the DBMS, thus reducing substantially the amount of data shuffled. The architecture of HadoopDB is depicted in Fig. 7, where the newly introduced components are marked with bold lines and text. The database connector is an interface between database systems and TaskTrackers. It connects to the database, executes a SQL query, and returns the result in the form of key-value pairs. The catalog maintains metadata about the databases, such as connection parameters as well as metadata on data sets stored, replica locations, data partitioning properties. The data loader globally repartitions data based on a partition key during data upload, breaks apart single node data into smaller partitions, and bulk loads the databases with these small partitions. Finally, the SMS planner extends Hive and produces query plans that can exploit features provided by the available database systems. For example, in the case of a join, some tables may be colocated, and thus, the join can be pushed entirely to the database engine (similar to a map-only job).

Situation-Aware Mappers (SAMs) [101] have been recently proposed to improve the performance of query processing in MapReduce in different ways. The crucial idea behind this work is to allow map tasks (mappers) to communicate and maintain global state by means of a distributed meta-data store, implemented by the distributed coordination service Apache ZooKeeper,[2] thus making globally coordinated optimization decisions. In this way, MapReduce is enhanced with dynamic and adaptive features. Adaptive Mappers can avoid frequent checkpointing by taking more input splits. Adaptive Combiners can perform local aggregation by maintaining a

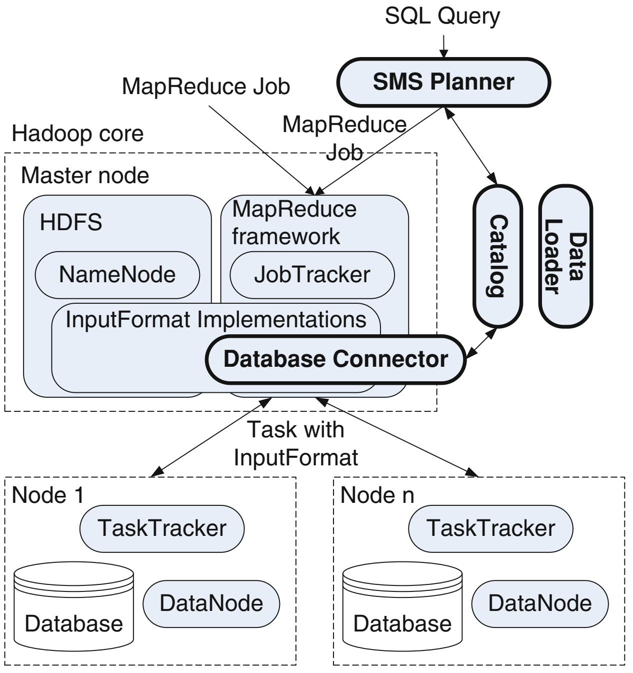

Fig. 7 The architecture of HadoopDB [2]

hash table in the map task and using it as a cache. Adaptive Sampling creates local samples, aggregates them, and produces a global histogram that can be used for various purposes (e.g., determine the satisfaction of a global condition). Adaptive Partitioning can exploit the global histogram to produce equi-sized partitions for better load balancing.

Clydesdale [60] is a system for structured data processing, where the data fits a star schema, which is built on top of MapReduce without *any* modifications. Clydesdale improves the performance of MapReduce by adopting the following optimization techniques: columnar storage, replication of dimension tables on local storage of each node which improves the performance of joins, re-use of hash tables stored in memory by scheduling map tasks to specific nodes intentionally, and block iteration (instead of row iteration). However, Clydesdale is limited by the available memory on each individual node, i.e., when the dimension tables do not fit in memory.

4.5.2 Configuration parameter tuning

Starfish [51] introduces cost-based optimization of MapReduce programs. The focus is on determining appropriate values for the configuration parameters of a MapReduce job, such as number of map and reduce tasks, amount of allocated memory. Determining these parameters is both cumbersome and unclear for the user, and most importantly they significantly affect the performance of MapReduce jobs, as indicated in [15]. To address this problem, a Profiler is intro-

[2] http://zookeeper.apache.org/.

duced that collects estimates about the size of data processed, the usage of resources, and the execution time of each job. Further, a What-if Engine is employed to estimate the benefit from varying a configuration parameter using simulations and a model-based estimation method. Finally, a cost-based optimizer is used to search for potential configuration settings in an efficient way and determine a setting that results in good performance.

4.5.3 Plan refinement and operator reordering

AQUA [105] is a query optimizer for Hive. One significant performance bottleneck in MapReduce is the cost of storing intermediate results. In AQUA, this problem is addressed by a two-phase optimization where join operators are organized into a number of groups which each can be executed as one job, and then, the intermediate results from the join groups are joined together. AQUA also provides two important query plan refinements: sharing table scan in map phase, and concurrent jobs, where independent subqueries can be executed concurrently if resources are available. One limitation of AQUA is the use of pairwise joins as the basic scheduling unit, which excludes the evaluation of multi-way joins in one MapReduce job. As noted in [110], in some cases, evaluating a multi-way join with one MapReduce job can be much more efficient.

YSmart [69] is a correlation-aware SQL-to-Map- Reduce translator, motivated by the slow speed for certain queries using Hive and other "one-operation-to-one-job" translators. Correlations supported include multiple nodes having input relation sets that are not disjoint, multiple nodes having both non-disjoint input relations sets and same partition key, and occurrences of nodes having same partition key as child node. By utilizing these correlations, the number of jobs can be reduced thus reducing time-consuming generation and reading of intermediate results. YSmart also has special optimizations for self-join that require only one table scan. One possible drawback of YSmart is the goal of reducing the number of jobs, which does not necessarily give the most reduction in terms of performance. The approach has also limitations in handling theta-joins.

RoPE [8] collects statistics from running jobs and use these for re-optimizing future invocations of the same (or similar) job by feeding these statistics into a cost-based optimizer. Changes that can be performed during re-optimization include changing degree of parallelism, reordering operations, choosing appropriate implementations, and grouping operations that have little work into a single physical task. One limitation of RoPE is that it depends on the same jobs being executed several times and as such is not beneficial for queries being executed the first time. This contrasts to other approach like, e.g., Shark [107].

In MapReduce, user-defined functions such as Map and Reduce are treated like "black boxes". The result is that in the data-shuffling stage, we must assume conservatively that all data-partition properties are lost after applying the functions. In *SUDO* [109], it is proposed that these functions expose data-partition properties, so that this information can be used to avoid unnecessary reordering operations. Although this approach in general works well, it should be noted that it can also in some situations introduce serious data skew.

4.5.4 Code analysis and query optimization

The aim of *HadoopToSQL* [54] is to be able to utilize SQL's features for indexing, aggregation, and grouping from MapReduce. HadoopToSQL uses static analysis to analyze the Java code of a MapReduce query in order to completely or partially transform it to SQL. This makes it possible to access only parts of data sets when indexes are available, instead of a full scan. Only certain classes of MapReduce queries can be transformed, for example there can be no loops in the Map function. The approach is implemented on top of a standard database system, and it is not immediately applicable in a standard MapReduce environment without efficient indexing. This potentially also limits the scalability of the approach to the scalability of the database system.

Manimal [56] is a framework for automatic analysis and optimization of data-centric MapReduce programs. The aim of this framework is to improve substantially the performance of MapReduce programs by exploiting potential query optimizations as in the case of traditional RDBMSs. Exemplary optimizations that can be automatically detected and be enforced by Manimal include the use of local indexes and the delta-compression technique for efficient representation of numerical values. Manimal shares many similarities with HadoopToSQL, as they both rely on static analysis of code to identify optimization opportunities. Also, both approaches are restricted to detecting certain types of code. Some differences also exist, and the most important being that the actual execution in HadoopToSQL is performed by a database system, while the execution in Manimal is still performed on top of MapReduce.

4.5.5 Data flow optimization

An increasing number of complex analytical tasks are represented as a workflow of MapReduce jobs. Efficient processing of such workflows is an important problem, since the performance of different execution plans of the same workflow varies considerably. *Stubby* [73] is a cost-based optimizer for MapReduce workflows that searches the space of the full plan of the workflow to identify optimization opportunities. The input to Stubby is an annotated MapReduce workflow, called *plan*, and its output is an equivalent, yet optimized plan. Stubby works by transforming a plan to another equivalent plan, and in this way, it searches the space of potential transformations in a selective way. The space is searched using two traversals of the workflow graph, where in each traversal different types of transformations are applied. The cost of a plan is derived by exploiting the What-If engine proposed in previous work [51].

Hueske et al. [52] study optimization of data flows, where the operators are User-defined Functions (UDFs) with unknown semantics, similar to "black boxes". The aim is to support reorderings of operators (together with parallelism) whose properties are not known in advance. To this end, automatic static code analysis is employed to identify a limited set of properties that can guarantee safe reorderings. In this way, typical optimizations carried out on traditional RDBMSs, such as selection and join reordering as well as limited forms of aggregation push-down, are supported.

4.6 Fair work allocation

Since reduce tasks work in parallel, an overburdened reduce task may stall the completion of a job. To assign the work fairly, techniques such as pre-processing and sampling, repartitioning, and batching are employed.

4.6.1 Pre-processing and sampling

Kolb et al. [62] study the problem of load balancing in MapReduce in the context of entity resolution, where data skew may assign workload to reduce tasks unfairly. This problem naturally occurs in other applications that require pairwise computation of similarities when data is skewed, and thus, it is considered a built-in vulnerability of MapReduce. Two techniques (*BlockSplit* and *PairRange*) are proposed, and both rely on the existence of a pre-processing phase (a separate MapReduce job) that builds information about the number of entities present in each block.

Ramakrishnan et al. [88] aim at solving the problem of reduce keys with large loads. This is achieved by splitting large reduce keys into several medium-load reduce keys, and assigning medium-load keys to reduce tasks using a bin-packing algorithm. Identifying such keys is performed by sampling before the MapReduce job starts, and information about reduce-key load size (large and medium keys) is stored in a *partition file*.

4.6.2 Repartitioning

Gufler et al. [48] study the problem of handling data skew by means of an adaptive load balancing strategy. A cost estimation method is proposed to quantify the cost of the work assigned to reduce tasks, in order to ensure that this is performed fairly. By means of computing local statistics in the map phase and aggregating them to produce global statistics, the global data distribution is approximated. This is then exploited to assign the data output from the map phase to reduce tasks in a way that achieves improved load balancing.

In *SkewReduce* [64], mitigating skew is achieved by an optimizer utilizing user-supplied cost functions. Requiring users to supply this is a disadvantage and is avoided in their follow-up approach *SkewTune* [65]. SkewTune handles both skew caused by an uneven distribution of input data to operator partitions (or tasks) as well as skew caused by some input records (or key-groups) taking much longer to process than others, with no extra user-supplied information. When SkewTune detects a straggler and its reason is skew, it repartitions the straggler's remaining unprocessed input data to one or additional slots, including ones that it expects to be available soon.

4.6.3 Batching at reduce side

Sailfish [89] aims at improving the performance of MapReduce by reducing the number of disk accesses. Instead of writing to intermediate files at map side, data are shuffled directly and then written to a file at reduce side, one file per reduce task (batching). The result is one intermediate file per reduce task, instead of one per map task, giving a significant reduction in disk seeks for the reduce task. This approach also has the advantage of giving more opportunities for auto-tuning (number of reduce tasks). Sailfish aims at applications where the size of intermediate data is large, and as also noted in [89], when this is not the case other approaches for handling intermediate data might perform better.

Themis [90] considers medium-sized Hadoop-clusters where failures will be less common. In case of failure, jobs are re-executed. Task-level fault-tolerance is eliminated by aggressively pipelining records without unnecessarily materializing intermediate results to disk. At reduce side, batching is performed, similar to how it is done in Sailfish [89]. Sampling is performed at each node and is subsequently used to detect and mitigate skew. One important limitation of Themis is its dependence on controlling access to the host machine's I/O and memory. It is unclear how this will affect it in a setting where machines are shared with other applications. Themis also has problems handling stragglers since jobs are not split into tasks.

4.7 Interactive and real-time processing

Support for interactive and real-time processing is provided by a mixture of techniques such as streaming, pipelining, in-memory processing, and pre-computation.

4.7.1 Streaming and pipelining

Dremel [80] is a system proposed by Google for interactive analysis of large-scale data sets. It complements MapReduce by providing much faster query processing and analysis of data, which is often the output of sequences of MapReduce jobs. Dremel combines a nested data model with columnar storage to improve retrieval efficiency. To achieve this goal, Dremel introduces a lossless representation of record structure in columnar format, provides fast encoding and record assembly algorithms, and postpones record assembly by directly processing data in columnar format. Dremel uses a multi-level tree architecture to execute queries, which reduces processing time when more levels are used. It also uses a query dispatcher that can be parameterized to return a result when a high percentage but not all (e.g., 99 %) of the input data has been processed. The combination of the above techniques is responsible for Dremel's fast execution times. On the downside, for queries involving many fields of records, Dremel may not work so well due to the overhead imposed by the underlying columnar storage. In addition, Dremel supports only tree-based aggregation and not more complex DAGs required for machine learning tasks. A query processing framework sharing the aims of Dremel, *Impala* [63], has recently been developed in the context of Hadoop and extends Dremel in the sense it can also support multi-table queries.

Hyracks [19] is a platform for data-intensive processing that improves the performance of Hadoop. In Hyracks, a job is a dataflow DAG that contains operators and connectors, where operators represent the computation steps while connectors represent the redistribution of data from one step to another. Hyracks attain performance gains due to design and implementation choices such as the use of pipelining, support of operators with multiple inputs, additional description of operators that allow better planning and scheduling, and a push-based model for incremental data movement between producers and consumers. One weakness of Hyracks is the lack of a mechanism for restarting only the necessary part of a job in the case of failures, while at the same time keeping the advantages of pipelining.

Tenzing [27] is a SQL query execution engine built on top of MapReduce by Google. Compared to Hive [99,100] or Pig [86], Tenzing additionally achieves low latency and provides a SQL92 implementation with some SQL99 extensions. At a higher level, the performance gains of Tenzing are due to the exploitation of traditional database techniques, such as indexes as well as rule-based and cost-based optimization. At the implementation level, Tenzing keeps processes (workers) running constantly in a worker pool, instead of spawning new processes, thus reducing query latency. Also, hash table-based aggregation is employed to avoid the sorting overhead imposed in the reduce task. The performance of sequences of MapReduce jobs is improved by implementing streaming between the upstream reduce task and the downstream map task as well as memory chaining by colocating these tasks in the same process. Another important improvement is a block-based shuffle mechanism that avoids the overheads of row serialization and deserialization encountered in row-based shuffling. Finally, when the processed data are smaller than a given threshold, local execution on the client is used to avoid sending the query to the worker pool.

4.7.2 In-memory processing

PowerDrill [49] uses a column-oriented datastore and various engineering choices to improve the performance of query processing even further. Compared to Dremel that uses streaming from DFS, PowerDrill relies on having as much data in memory as possible. Consequently, PowerDrill is faster than Dremel, but it supports only a limited set of selected data sources, while Dremel supports thousands of different data sets. PowerDrill uses two dictionaries as basic data structures for representing a data column. Since it relies on memory storage, several optimizations are proposed to keep the memory footprint of these structures small. Since it works in memory, PowerDrill is constrained by the available memory for maintaining the necessary data structures.

Shark [42,107] (Hive on *Spark* [108]) is a system developed at UC Berkeley that improves the performance of Hive by exploiting in-memory processing. Shark uses a main memory abstraction called resilient distributed dataset (RDD) that is similar to shared memory in large clusters. Compared to Hive, the improvement in performance mainly stems from inter-query caching of data in memory, thus eliminating the need to read/write repeatedly on disk. In addition, other improving techniques are employed including hash-based aggregation and dynamic partial DAG execution. Shark is restricted by the size of available main memory, for example when the map output does not fit in memory.

M3R [94] (Main Memory MapReduce) is a framework that runs MapReduce jobs in memory much faster than Hadoop. Key technical changes of M3R for improving performance include sharing heap state between jobs, elimination of communication overheads imposed by the JobTracker and heartbeat mechanisms, caching input and output data in memory, in-memory shuffling, and always mapping the same partition to the same location across all jobs in a sequence thus allowing re-use of built memory structures. The limitations of M3R include lack of support for resilience and constraints on the type of supported jobs imposed by memory size.

4.7.3 *Pre-computation*

BlinkDB [7,9] is a query processing framework for running interactive queries on large volumes of data. It extends Hive and Shark [42] and focuses on quick retrieval of approximate results, based on pre-computed samples. The performance and accuracy of BlinkDB depends on the quality of the samples, and amount of recurring *query templates*, i.e., set of columns used in WHERE and GROUP-BY clauses.

4.8 Processing n-way operations

By design, a typical MapReduce job is supposed to consume input from one file, thereby complicating the implementation of n-way operations. Various systems, such as Hyracks [19], have identified this weakness and support n-way operators by design, which naturally take multiple sources as input. In this way, operations such as joins of multiple data sets can be easily implemented. In the sequel, we will use the join operator as a typical example of n-way operation, and we will examine how joins are supported in MapReduce.

Basic methods for processing joins in MapReduce include (1) distributing the smallest operand(s) to all nodes, and performing the join by the Map or Reduce function, and (2) map-side and reduce-side join variants [104]. The first approach is only feasible for relatively small operands, while the map-side join is only possible in restricted cases when all records with a particular key end up at the same map task (for example, this is satisfied if the operands are outputs of jobs that had the same number of reduce task and same keys, *and* the output files are smaller than one HDFS block). The more general method is reduce-side join, which can be seen as a variant of traditional parallel join where input records are tagged with operands and then shuffled so that all records with a particular key end up at the same reduce task and can be joined there.

More advanced methods for join processing in MapReduce are presented in the following, categorized according to the type of join.

4.8.1 *Equi-join*

Map-Reduce-Merge [29] introduces a third phase to MapReduce processing, besides map and reduce, namely *merge*. The *merge* is invoked after reduce and receives as input two pairs of key/values that come from two distinguishable sources. The *merge* is able to join reduced outputs, and thus, it can be effectively used to implement join of data coming from heterogeneous sources. The paper describes how traditional relational operators can be implemented in this new system and also focuses explicitly on joins, demonstrating the implementation of three types of joins: sort-merge joins, hash joins, and block nested-loop joins.

Map-Join-Reduce [58] also extends MapReduce by introducing a third phase called *join*, between map and reduce. The objective of this work is to address the limitation of the MapReduce programming model that is mainly designed for processing homogeneous data sets, i.e., the same logic encoded in the Map function is applied to all input records, which is clearly not the case with joins. The main idea is that this intermediate phase enables joining multiple data sets, since *join* is invoked by the runtime system to process joined records. In more details, the Join function is run together with Reduce in the same ReduceTask process. In this way, it is possible to pipeline join results from *join* to reduce, without the need of checkpointing and shuffling intermediate results, which results in significant performance gains. In practice, two successive MapReduce jobs are required to process a join. The first job performs filtering, joins the qualified tuples, and pushes the join results to reduce tasks for partial aggregation. The second job assembles and combines the partial aggregates to produce the final result.

The work in [5,6] on optimizing joins shares many similarities in terms of objectives with Map-Join-Reduce. The authors study algorithms that aim to minimize the communication cost, under the assumption that this is the dominant cost during query processing. The focus is on processing multi-way joins as a single MapReduce process, and the results show that in certain cases this works better than having a sequence of binary joins. In particular, the proposed approach works better for (1) analytical queries where a very large fact table is joined with many small dimension tables and (2) queries involving paths in graphs with high out-degree.

The work in [18] provides a comparison of join algorithms for log processing in MapReduce. The main focus of this work is on two-way equijoin processing of relations L and R. The compared algorithms include Repartition Join, Broadcast Join, Semi-Join, and Per-Split Semi-Join. Repartition Join is the most common join strategy in which L and R are partitioned on join key and the pairs of partitions with common key are joined. Broadcast Join runs as a map-only job that retrieves the smaller relation, e.g., $R(|R| << |L|)$, over the network to join with local splits of L. Semi-Join operates in three MapReduce jobs, and the main idea is to avoid sending records of R over the network that will not join with L. Finally, Per-Split Semi-Join also has three phases and tries to send only those records of R that will join with a particular split L_i of L. In addition, various pre-processing techniques, such as partitioning, replication, and filtering, are studied that improve the performance. The experimental study indicates that all join strategies are within a factor of 3 in most cases in terms of performance for a high available network bandwidth (in more congested networking environments, this factor is expected to increase). Another reason that this factor is relatively small is that MapReduce has inherent overheads related to input record parsing, checksum validation, and task initialization.

Llama [74] proposes the *concurrent join* for processing multi-way joins in a single MapReduce job. Differently than the existing approaches, Llama relies on columnar storage of data which allows processing as many join operators as possible in the map phase. Obviously, this approach significantly reduces the cost of shuffling. To enable map-side join processing, when data is loaded in HDFS, it is partitioned into vertical groups and each group is stored sorted, thus enabling efficient join processing of relations sorted on key. During query processing, it is possible to scan and re-sort only some columns of the underlying relation, thus improving substantially the cost of sorting.

4.8.2 Theta-join

Processing of theta-joins using a single MapReduce job is first studied in [84]. The basic idea is to partition the join space to reduce tasks in an intentional manner that (1) balances input and output costs of reduce tasks, while (2) minimizing the number of input tuples that are duplicated at reduce tasks. The problem is abstracted to finding a mapping of the cells of the *join matrix* to reduce tasks that minimizes the job completion time. A cell $M(i, j)$ of the join matrix M is set to true if the ith tuple from S and the jth tuple from R satisfy the join condition, and these cells need to be assigned to reduce tasks. However, this knowledge is not available to the algorithm before seeing the input data. Therefore, the proposed algorithm (*1-Bucket-Theta*) uses a matrix to regions mapping that relies only on input cardinalities and guarantees fair balancing to reduce tasks. The algorithm follows a randomized process to assign an incoming tuple from $S(R)$ to a random row (column). Given the mapping, it then identifies a set of intersecting regions of $R(S)$, and outputs multiple tuples having the corresponding regions IDs as keys. This is a conservative approach that assumes that all intersecting cells are true, however, the reduce tasks can then find the actual matches and ignore those tuples that do not match. To improve the performance of this algorithm, additional statistics about the input data distribution are required. Approximate equi-depth histograms of the inputs are constructed using two MapReduce jobs and can be exploited to identify empty regions in the matrix. Thus, the resulting join algorithms, termed *M-Bucket*, can improve the runtime of any theta-join.

Multi-way theta-joins are studied in [110]. Similar to [5, 6], the authors consider processing a multi-way join in a single MapReduce job and identify when a join should be processed by means of a single or multiple MapReduce jobs. In more detail, the problem addressed is given a limited number of available processing units, how to map a multi-way theta-join to MapReduce jobs and execute them in a scheduled order, such that the total execution time is minimized. To this end, they propose rules to decompose a multi-way theta-join query and a suitable cost model that estimates the minimum cost of each decomposed query plan, and selecting the most efficient one using one MapReduce job.

4.8.3 Similarity join

Set-similarity joins in MapReduce are first studied in [102], where the PPJoin+ [106] algorithm is adapted for MapReduce. The underlying data can be strings or sets of values, and the aim is to identify pairs of records with similarity above a certain user-defined threshold. All proposed algorithms are based on the *prefix filtering* technique. The proposed approach relies on three stages, and each stage can be implemented by means of one or two MapReduce jobs. Hence, the minimum number of MapReduce jobs to compute the join is three. In a nutshell, the first stage produces a list of join tokens ordered by frequency count. In the second stage, the record ID and its join tokens are extracted from the data, distributed to reduce tasks, and the reduce tasks produce record ID pairs of similar records. The third stage performs duplicate elimination and joins the records with IDs produced by the previous stage to generate actual pairs of joined records. The authors study both the case of self-join as well as binary join.

V-SMART-Join [81] addresses the problem of computing all-pair similarity joins for sets, multi-sets, and vector data. It follows a two-stage approach consisting of a joining phase and a similarity phase; in the first phase, partial results are computed and joined, while the second phase computes the exact similarities of the candidate pairs. In more detail, the joining phase uses two MapReduce jobs to transform the input tuples to join tuples, which essentially are multiple tuples for each multi-set (one tuple for each element in the multi-set) enhanced with some partial results. The similarity phase also consists of two jobs. The first job builds an inverted index and scans the index to produce candidate pairs, while the second job computes the similarity between all candidate pairs. V-SMART-Join is shown to outperform the algorithm of Vernica et al. [102] and also demonstrates some applications where the amount of data that need to be kept in memory is so high that the algorithm in [102] cannot be applied.

SSJ-2R and *SSJ-2* (proposed in [16]) are also based on prefix filtering and use two MapReduce phases to compute the similarity self-join in the case of document collections. Document representations are characterized by sparseness, since only a tiny sub-set of entries in the lexicon occur in any given document. We focus on SSJ-2R which is the most efficient of the two and introduces novelty compared to existing work. SSJ-2R broadcasts a remainder file that contains frequently occurring terms, which is loaded in memory in order to avoid remote access to this information from different reduce tasks. In addition, to handle large remainder files, a partitioning technique is used that splits the file in K almost equi-sized parts. In this way, each reduce task is assigned only with a part of the remainder file, at the expense of replicating documents to multiple reduce tasks. Compared to the algorithm in [102], the proposed algorithms perform worse in the map phase, but better in the reduce phase, and outperform [102] in total.

MRSimJoin [96,97] studies the problem of distance range join, which is probably the most common case of similarity join. Given two relations R and S, the aim is to retrieve pairs of objects $r \in R$ and $s \in S$, such that $d(r, s) \leq \epsilon$, where $d()$ is a metric distance function and ϵ is a user-specified similarity threshold. MRSimJoin extends the centralized algorithm QuickJoin [55] to become applicable in the context of MapReduce. From a technical viewpoint, MRSimJoin iteratively partitions the input data into smaller partitions (using a number of pivot objects), until each partition fits in the main memory of a single machine. To achieve this goal, multiple partitioning rounds (MapReduce jobs) are necessary, and each round partitions the data in a previously generated partition. The number of rounds decreases for increased number of pivots, however, this also increases the processing cost of partitioning. MRSimJoin is compared against [84] and shown to outperform it.

Parallel processing of top-k similarity joins is studied in [61], where the aim is to retrieve the k most similar (i.e., closest) pairs of objects from a database based on some distance function. Contrary to the other approaches for similarity joins, it is not necessary to define a similarity threshold, only k is user-specified. The basic technique used is partitioning of the pairs of objects in such a way that every pair appears in a single partition only, and then computing top-k closest pairs in each partition, followed by identifying the global top-k objects from all partitions. In addition, an improved strategy is proposed in which only a smaller sub-set of pairs (that includes the correct top-k pairs) is distributed to partitions. To identify this smaller sub-set, sampling is performed to identify a similarity threshold τ that bounds the distance of the kth closest pair.

In [4], fuzzy joins are studied. Various algorithms that rely on a single MapReduce job are proposed, which compute all pairs of records with similarity above a certain user-specified threshold, and produce the exact result. The algorithms focus on three distance functions: Hamming distance, Edit distance, and Jaccard distance. The authors provide a theoretical investigation of algorithms whose cost is quantified by means of three metrics; (1) the total map or pre-processing cost (M), (2) the total communication cost (C), and (3) the total computation cost of reduce tasks (R). The objective is to identify MapReduce procedures (corresponding to algorithms) that are not dominated when the cost space of (M, C, R) is considered. Based on a cost analysis, the algorithm that performs best for a sub-set of the aforementioned cost metrics can be selected.

4.8.4 *k-NN and top-k join*

The goal of k-NN join is to produce the k nearest neighbors of each point of a data set S from another data set R, and it has been recently studied in the MapReduce context in [77]. After demonstrating two baseline algorithms based on block nested loop join (*H-BNLJ*) and indexed block nested-loop join (*H-BRJ*), the authors propose an algorithm termed *H-zkNNJ* that relies on one-dimensional mapping (z-order) and works in three MapReduce phases. In the first phase, randomly shifted copies of the relations R and S are constructed and the partitions R_i and S_i are determined. In the second phase, R_i and S_i are partitioned in blocks, and also a candidate k nearest neighbor set for each object $r \in R_i$ is computed. The third phase simply derives the k nearest neighbors from the candidate set.

A top-k join (also known as rank join) retrieves the k join tuples from R and S with highest scores, where the score of a join tuple is determined by a user-specified function that operates on attributes of R and S. In general, the join between R and S in many-to-many.

RanKloud [24] computes top-k joins in MapReduce by employing a three-stage approach. In the first stage, the input data are repartitioned in a utility-aware balanced manner. This is achieved by sampling and estimating the join selectivity and threshold that serves as a lower bound for the score of the top-k join tuple. In the second stage, the actual join is performed between the new partitions. Finally, in the third stage, the intermediate results produced by the reduce tasks need to be combined to produce the final top-k result. RanKloud delivers the correct top-k result, however, the retrieval of k results cannot be guaranteed (i.e., fewer than k tuples may be produced), thus it is classified as an approximate algorithm. In [38] the aim is to provide an exact solution to the top-k join problem in MapReduce. A set of techniques that are particularly suited for top-k processing is proposed, including implementing sorted access to data, applying angle-based partitioning, and building synopses in the form of histograms for capturing the score distribution.

4.8.5 *Analysis*

In the following, an analysis of join algorithms in MapReduce is provided based on five important phases that improve the performance of query processing.

First, improved performance can be achieved by *preprocessing*. This includes techniques such as sampling, which can give a fast overview of the underlying data distribution or input statistics. Unfortunately, depending on the sta-

Table 3 Analysis of join processing in MapReduce

Join type	Approach	Pre-processing	Pre-filtering	Partitioning	Replication	Load balancing
Equi-join	*Map-Reduce-Merge* [29]	N/A	N/A	N/A	N/A	N/A
	Map-Join-Reduce [58]	N/A	N/A	N/A	N/A	N/A
	Afrati et al. [5,6]	No	No	Hash-based	“share”-based	No
	Repartition join [18]	Yes	No	Hash-based	No	No
	Broadcast join [18]	Yes	No	Broadcast	Broadcast R	No
	Semi-join [18]	Yes	No	Broadcast	Broadcast	No
	Per-split semi-join [18]	Yes	No	Broadcast	Broadcast	No
	Llama [74]	Yes	No	Vertical groups	No	No
Theta-join	*1-Bucket-Theta* [84]	No	No	Cover join matrix	Yes	Bounds
	M-Bucket [84]	Yes	No	Cover join matrix	Yes	Bounds
	Zhang et al. [110]	No	No	Hilbert space-filling curve	Minimize	Yes
Similarity join	Set-similarity join [102]	Global token ordering	No	Hash-based	Grouping	Frequency- based
	V-SMART-Join [81]	Compute frequencies	No	Hash-based	Yes	Cardinality-based
	SSJ-2R [16]	Similar to [102]	No	Remainder file	Yes	Yes
	Silva et al. [97]	No	No	Iterative	Yes	No
	Top-k similarity join [61]	Sampling	Essential pairs	Bucket-based	Yes	No
	Fuzzy join [4]	No	No		Yes	
k-NN join	*H-zkNNJ* [77]	No	No	Z-value based	Shifted copies of R, S	Quantile-based
Top-k join	*RanKloud* [24]	Sampling	No	Utility-aware	No	Yes
	Doulkeridis et al. [38]	Sorting	No	Angle-based	No	Yes

tistical information that needs to be generated, pre-processing may require one or multiple MapReduce jobs (e.g., for producing an equi-depth histogram), which entails extra costs and essentially prevents workflow-style processing on the results of a previous MapReduce job. Second, *pre-filtering* is employed in order to early discard input records that cannot contribute to the join result. It should be noted that pre-filtering is not applicable to every type of join, simply because there exist join types that need to examine all input tuples as they constitute potential join results. Third, *partitioning* is probably the most important phase, in which input tuples are assigned to partitions that are distributed to reduce tasks for join processing. Quite often, the partitioning causes *replication* (also called *duplication*) of input tuples to multiple reduce tasks, in order to produce the correct join result in a parallel fashion. This is identified as the fourth phase. Finally, *load balancing* of reduce tasks is extremely important, since the slowest reduce task determines the overall job completion time. In particular, when the data distribution is skewed, a load balancing mechanism is necessary to mitigate the effect of uneven work allocation.

A summary of this analysis is presented in Table 3. In the following, we comment on noteworthy individual techniques employed at the different phases.

Pre-processing. Llama [74] uses columnar storage of data and in particular uses vertical groups of columns that are stored sorted in HDFS. This is essentially a pre-processing step that greatly improves the performance of join operations, by enabling map-side join processing. In [102], the tokens are ordered based on frequencies in a pre-processing step. This information is exploited at different stages of the algorithm to improve performance. *V-SMART-Join* [81] scans the input data once to derive partial results and cardinalities of items. The work in [61] uses sampling to compute an upper bound on the distance of the kth closest pair.

Pre-filtering. The work in [61] applies pre-filtering by partitioning only those pairs necessary for producing the top-k most similar pairs. These are called *essential pairs*, and their computation is based on the upper bound of distance.

Partitioning. In the case of theta-joins [84], the partitioning method focuses on mapping the join matrix (i.e., representing the cross-product result space) to the available reduce tasks, which entails “covering” the populated cells by assignment to a reduce task. For multi-way theta joins [110], the join space is multi-dimensional and partitioning is performed using the Hilbert space-filling curve, which is shown to minimize the optimization objective of partitioning. The algorithm proposed for k-NN joins [77] uses z-ordering to map data to one-dimensional values and partition the one-dimensional space into ranges. In the case of top-k joins, both *RanKloud* [24] and the approach described in [38] propose partitioning schemes tailored to top-k processing. RanKloud employs a utility-aware partitioning that splits the space in non-equal ranges, such that the useful work produced by each partition is roughly the same. In [38], angle-based partitioning is proposed, which works better than traditional partitioning schemes for parallel skyline queries [103].

Replication. To reduce replication, the approach in [102] uses grouped tokens, i.e., maps multiple tokens to one synthetic key and partitions based on the synthetic key. This technique reduces the amount of record replication to reduce tasks. In the case of multi-way theta joins [110], a multi-dimensional space partitioning method is employed to assign partitions of the join space to reduce tasks. Interestingly, the objective of partitioning is to minimize the duplication of records.

Load balancing. In [102], the authors use the pre-processed token ordering to balance the workload more evenly to reduce tasks by assigning tokens to reduce tasks in a round-robin manner based on frequency. *V-SMART-Join* [81] also exploits cardinalities of items in multi-sets to address skew in data distribution. For theta-joins [84], the partitioning attempts to balance the workload to reduce tasks, such that the job completion time is minimized, i.e., no reduce task is overloaded and delays the completion of the job. Theoretical bounds on the number of times the "fair share" of any reduce task is exceeded are also provided. In the case of *H-zkNNJ* [77], load balancing is performed by partitioning the input relation in roughly equi-sized partitions, which is accomplished by using approximate quantiles.

5 Conclusions and outlook

MapReduce has brought new excitement in the parallel data processing landscape. This is due to its salient features that include scalability, fault-tolerance, simplicity, and flexibility. Still, several of its shortcomings hint that MapReduce is not perfect for every large-scale analytical task. It is therefore natural to ask ourselves about lessons learned from the MapReduce experience and to hypothesize about future frameworks and systems.

We believe that the next generation of parallel data processing systems for massive data sets should combine the merits of existing approaches. The strong features of MapReduce clearly need to be retained; however, they should be coupled with efficiency and query optimization techniques present in traditional data management systems. Hence, we expect that future systems will not extend MapReduce, but instead redesign it from scratch, in order to retain all desirable features but also introduce additional capabilities.

In the era of "Big Data", future systems for parallel processing should also support efficient exploratory query processing and analysis. It is vital to provide efficient support for processing on subsets of the available data only. At the same time, it is important to provide fast retrieval of results that are indicative, approximate, and guide the user to formulating the correct query.

Another important issue relates to support for declarative languages and query specification. Decades of research in data management systems have shown the benefits of using declarative query languages. We expect that future systems will be designed with declarative querying from the very beginning, rather than adding it as a layer later.

Finally, support for real-time applications and streaming data is expected to drive innovations in parallel large-scale data analysis. An approach for extending MapReduce for supporting real-time analysis is introduced in [20] by Facebook. We expect to see more frameworks targeting requirements for real-time processing and analysis in the near future.

Acknowledgments We would like to thank the editors and the anonymous reviewers for their very helpful comments that have significantly improved this paper. The research of C. Doulkeridis was supported under the Marie-Curie IEF grant number 274063 with partial support from the Norwegian Research Council.

6 Appendix

See Tables 4 and 5.

Table 4 Modifications induced by existing approaches to MapReduce

Approach	Modification in MapReduce/Hadoop
Hadoop++ [36]	No, based on using UDFs
HAIL [37]	Yes, changes the RecordReader and a few UDFs
CoHadoop [41]	Yes, extends HDFS and adds metadata to NameNode
Llama [74]	No, runs on top of Hadoop
Cheetah [28]	No, runs on top of Hadoop
RCFile [50]	No changes to Hadoop, implements certain interfaces
CIF [44]	No changes to Hadoop core, leverages extensibility features
Trojan layouts [59]	Yes, introduces Trojan HDFS (among others)
MRShare [83]	Yes, modifies map outputs with tags and writes to multiple output files on the reduce side
ReStore [40]	Yes, extends the JobControlCompiler of Pig
Sharing scans [11]	Independent of system
Silva et al. [95]	No, integrated into SCOPE
Incoop [17]	Yes, new file system, contraction phase, and memoization-aware scheduler
Li et al. [71,72]	Yes, modifies the internals of Hadoop by replacing key components
Grover et al. [47]	Yes, introduces dynamic job and Input Provider
EARL [67]	Yes, RecordReader and Reduce classes are modified, and simple extension to Hadoop to support dynamic input and efficient resampling
Top-k queries [38]	Yes, changes data placement and builds statistics
RanKloud [24]	Yes, integrates its execution engine into Hadoop and uses local B+Tree indexes
HaLoop [22,23]	Yes, use of caching and changes to the scheduler
MapReduce online [30]	Yes, communication between Map and Reduce, and to JobTracker and TaskTracker
NOVA [85]	No, runs on top of Pig and Hadoop
Twister [39]	Adopts an architecture with substantial differences
CBP [75,76]	Yes, substantial changes in various phases of MapReduce
Pregel [78]	Different system
PrIter [111]	No, built on top of Hadoop
PACMan [14]	No, coordinated caching independent of Hadoop
REX [82]	Different system
Differential dataflow [79]	Different system
HadoopDB [2]	Yes (substantially), installs a local DBMS on each DataNode and extends Hive
SAMs [101]	Yes, uses ZooKeeper for coordination between map tasks
Clydesdale [60]	No, runs on top of Hadoop
Starfish [51]	No, but uses dynamic instrumentation of the MapReduce framework
AQUA [105]	No, query optimizer embedded into Hive
YSmart [69]	No, runs on top of Hadoop
RoPE [8]	On top of different system (SCOPE [112]/Dryad [53])
SUDO [109]	No, integrated into SCOPE compiler
Manimal [56]	Yes, local B+Tree indexes and delta-compression
HadoopToSQL [54]	No, implemented on top of database system
Stubby [73]	No, on top of MapReduce
Hueske et al. [52]	Integrated into different system (Stratosphere)

Table 4 continued

Kolb et al. [62]	Yes, changes the distribution of map output to reduce tasks, but uses a separate MapReduce job for pre-processing
Ramakrishnan et al. [88]	Yes, sampler to produce the partition file that is subsequently used by the partitioner
Gufler et al. [48]	Yes, uses a monitoring component and changes the distribution of map output to reduce tasks
SkewReduce [64]	No, on top of MapReduce
SkewTune [65]	Yes, mostly on top of Hadoop, but some small changes to core classes in the map tasks
Sailfish [89]	Yes, extending distributed file system and batching data from map tasks
Themis [90]	Yes, significant changes to the way intermediate results are handled
Dremel [80]	Different system
Hyracks [19]	Different system
Tenzing [27]	Yes, sort-avoidance, streaming, memory chaining, and block shuffle
PowerDrill [49]	Different system
Shark [42,107]	Relies on a different system (*Spark* [108])
M3R [94]	Yes, in-memory caching and shuffling, re-use of structures, and minimize communication
BlinkDB [7,9]	No, built on top of Hive/Hadoop

Table 5 Overview of join processing in MapReduce

Join type	Approach	#Jobs in MapReduce	Exact/approximate		Self/binary/multiway		
			Exact	Approximate	Self	Binary	Multiway
Equi-join	*Map-Reduce-Merge* [29]	1	*			*	
	Map-Join-Reduce [58]	2	*			*	
	Afrati et al. [5,6]	1	*				*
	Repartition join [18]	1	*			*	
	Broadcast join [18]	1	*			*	
	Semi-join [18]	3	*			*	
	Per-split semi-join [18]	3	*			*	
	Llama [74]	1	*				*
Theta-join	*1-Bucket-Theta* [84]	1	*			*	
	M-Bucket [84]	3	*			*	
	Zhang et al. [110]	1 or Multiple	*				*
Similarity join	Set-similarity join [102]	≥ 3	*		*	*	
	V-SMART-Join (all-pair) [81]	4	*			*	
	SSJ-2R [16]	2	*		*		
	Silva et al. [97]	Multiple	*			*	
	Top-k similarity join [61]	2	*		*		
	Fuzzy join [4]	1	*		*		
k-NN join	*H-zkNNJ* [77]	3		*		*	
Top-k join	*RanKloud* [24]	3		*		*	
	Doulkeridis et al. [38]	2	*			*	

References

1. Abadi, D.J.: Data management in the cloud: limitations and opportunities. IEEE Data Eng. Bull. **32**(1), 3–12 (2009)
2. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. (PVLDB) **2**(1), 922–933 (2009)
3. Afrati, F.N., Borkar, V.R., Carey, M.J., Polyzotis, N., Ullman, J.D.: Map-reduce extensions and recursive queries. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 1–8 (2011)
4. Afrati, F.N., Sarma, A.D., Menestrina, D., Parameswaran, A.G., Ullman, J.D.: Fuzzy joins using MapReduce. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 498–509 (2012)
5. Afrati, F.N., Ullman, J.D.: Optimizing joins in a Map-Reduce environment. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 99–110 (2010)
6. Afrati, F.N., Ullman, J.D.: Optimizing multiway joins in a Map-Reduce environment. IEEE Trans. Knowl. Data Eng. (TKDE) **23**(9), 1282–1298 (2011)
7. Agarwal, S., Iyer, A.P., Panda, A., Madden, S., Mozafari, B., Stoica, I.: Blink and it's done: interactive queries on very large data. Proc. VLDB Endow. (PVLDB) **5**(12), 1902–1905 (2012)
8. Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., Zhou, J.: Re-optimizing data-parallel computing. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 21:1–21:14 (2012)
9. Agarwal, S., Panda, A., Mozafari, B., Milner, H., Madden, S., Stoica, I.: BlinkDB: queries with bounded errors and bounded response times on very large data. In: Proceedings of European Conference on Computer Systems (EuroSys) (2013)
10. Agrawal, D., Das, S., Abbadi, A.E.: Big data and cloud computing: current state and future opportunities. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 530–533 (2011)
11. Agrawal, P., Kifer, D., Olston, C.: Scheduling shared scans of large data files. Proc. VLDB Endow. (PVLDB) **1**(1), 958–969 (2008)
12. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: Proceedings of Very Large Databases (VLDB), pp. 169–180 (2001)
13. Aiyer, A.S., Bautin, M., Chen, G.J., Damania, P., Khemani, P., Muthukkaruppan, K., Ranganathan, K., Spiegelberg, N., Tang, L., Vaidya, M.: Storage infrastructure behind Facebook Messages: using HBase at scale. IEEE Data Eng. Bull. **35**(2), 4–13 (2012)
14. Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., Stoica, I.: PACMan: coordinated memory caching for parallel jobs. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 20:1–20:14 (2012)
15. Babu, S.: Towards automatic optimization of MapReduce programs. In: ACM Symposium on Cloud Computing (SoCC), pp. 137–142 (2010)
16. Baraglia, R., Morales, G.D.F., Lucchese, C.: Document similarity self-join with MapReduce. In: IEEE International Conference on Data Mining (ICDM), pp. 731–736 (2010)
17. Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U.A., Pasquin, R.: Incoop: MapReduce for incremental computations. In: ACM Symposium on Cloud Computing (SoCC), pp. 7:1–7:14 (2011)
18. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in MapReduce. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 975–986 (2010)
19. Borkar, V.R., Carey, M.J., Grover, R., Onose, N., Vernica, R.: Hyracks: a flexible and extensible foundation for data-intensive computing. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 1151–1162 (2011)
20. Borthakur, D., Gray, J., Sarma, J.S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., Aiyer, A.S.: Apache Hadoop goes realtime at Facebook. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1071–1080 (2011)
21. Bu, Y., Borkar, V.R., Carey, M.J., Rosen, J., Polyzotis, N., Condie, T., Weimer, M., Ramakrishnan, R.: Scaling datalog for machine learning on Big Data. The Computing Research Repository (CoRR), abs/1203.0160 (2012)
22. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: HaLoop: efficient iterative data processing on large clusters. Proc. VLDB Endow. (PVLDB) **3**(1), 285–296 (2010)
23. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDB J. **21**(2), 169–190 (2012)
24. Candan, K.S., Kim, J.W., Nagarkar, P., Nagendra, M., Yu, R.: RanKloud: scalable multimedia data processing in server clusters. IEEE Multimed. **18**(1), 64–77 (2011)
25. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec. **39**(4), 12–27 (2010)
26. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4:1–4:26 (2008)
27. Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., Wong, M.: Tenzing a SQL implementation on the MapReduce framework. Proc. VLDB Endow. (PVLDB) **4**(12), 1318–1327 (2011)
28. Chen, S.: Cheetah: a high performance, custom data warehouse on top of MapReduce. Proc. VLDB Endow. (PVLDB) **3**(2), 1459–1468 (2010)
29. Chih Yang, H., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1029–1040 (2007)
30. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce online. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 313–328 (2010)
31. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. (PVLDB) **1**(2), 1277–1288 (2008)
32. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2004)
33. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
34. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
35. Dittrich, J., Quiané-Ruiz, J.-A.: Efficient Big Data processing in Hadoop MapReduce. Proc. VLDB Endow. (PVLDB) **5**(12), 2014–2015 (2012)
36. Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). Proc. VLDB Endow. (PVLDB) **3**(1), 518–529 (2010)
37. Dittrich, J., Quiané-Ruiz, J.-A., Richter, S., Schuh, S., Jindal, A., Schad, J.: Only aggressive elephants are fast elephants. Proc. VLDB Endow. (PVLDB) **5**(11), 1591–1602 (2012)

38. Doulkeridis, C., Nørvåg, K.: On saying "enough already!" in MapReduce. In: Proceedings of International Workshop on Cloud Intelligence (Cloud-I), pp. 7:1–7:4 (2012)
39. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC), pp. 810–818 (2010)
40. Elghandour, I., Aboulnaga, A.: ReStore: reusing results of MapReduce jobs. Proc. VLDB Endow. (PVLDB) **5**(6), 586–597 (2012)
41. Eltabakh, M.Y., Tian, Y., Özcan, F., Gemulla, R., Krettek, A., McPherson, J.: CoHadoop: flexible data placement and its exploitation in Hadoop. Proc. VLDB Endow. (PVLDB) **4**(9), 575–585 (2011)
42. Engle, C., Lupher, A., Xin, R., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I.: Shark: fast data analysis using coarse-grained distributed memory. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 689–692 (2012)
43. Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V.: Spinning fast iterative data flows. Proc. VLDB Endow. (PVLDB) **5**(11), 1268–1279 (2012)
44. Floratou, A., Patel, J.M., Shekita, E.J., Tata, S.: Column-oriented storage techniques for MapReduce. Proc. VLDB Endow. (PVLDB) **4**(7), 419–429 (2011)
45. George, L.: HBase: The Definitive Guide: Random Access to Your Planet-Size Data. O'Reilly, Ireland (2011)
46. Goodhope, K., Koshy, J., Kreps, J., Narkhede, N., Park, R., Rao, J., Ye, V.Y.: Building LinkedIn's real-time activity data pipeline. IEEE Data Eng. Bull. **35**(2), 33–45 (2012)
47. Grover, R., Carey, M.J.: Extending map-reduce for efficient predicate-based sampling. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 486–497 (2012)
48. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Load balancing in MapReduce based on scalable cardinality estimates. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 522–533 (2012)
49. Hall, A., Bachmann, O., Büssow, R., Ganceanu, S., Nunkesser, M.: Processing a trillion cells per mouse click. Proc. VLDB Endow. (PVLDB) **5**(11), 1436–1446 (2012)
50. He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., Xu, Z.: RCFile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 1199–1208 (2011)
51. Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of MapReduce programs. Proc. VLDB Endow. (PVLDB) **4**(11), 1111–1122 (2011)
52. Hueske, F., Peters, M., Sax, M., Rheinländer, A., Bergmann, R., Krettek, A., Tzoumas, K.: Opening the black boxes in data flow optimization. Proc. VLDB End. (PVLDB) **5**(11), 1256–1267 (2012)
53. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of European Conference on Computer Systems (EuroSys), pp. 59–72 (2007)
54. Iu, M.-Y., Zwaenepoel, W.: HadoopToSQL: a MapReduce query optimizer. In: Proceedings of European Conference on Computer systems (EuroSys), pp. 251–264 (2010)
55. Jacox, E.H., Samet, H.: Metric space similarity joins. ACM Trans. Database Syst. (TODS) **33**(2), 7:1–7:38 (2008)
56. Jahani, E., Cafarella, M.J., Ré, C.: Automatic optimization for MapReduce programs. Proc. VLDB Endow. (PVLDB) **4**(6), 385–396 (2011)
57. Jiang, D., Ooi, B.C., Shi, L., Wu, S.: The performance of MapReduce: an in-depth study. Proc. VLDB Endow. (PVLDB) **3**(1), 472–483 (2010)
58. Jiang, D., Tung, A.K.H., Chen, G.: MAP-JOIN-REDUCE: toward scalable and efficient data analysis on large clusters. IEEE Trans. Knowl. Data Eng. (TKDE) **23**(9), 1299–1311 (2011)
59. Jindal, A., Quiané-Ruiz, J.-A., Dittrich, J.: Trojan data layouts: right shoes for a running elephant. In: ACM Symposium on Cloud Computing (SoCC), pp. 21:1–21:14 (2011)
60. Kaldewey, T., Shekita, E.J., Tata, S.: Clydesdale: structured data processing on MapReduce. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 15–25 (2012)
61. Kim, Y., Shim, K.: Parallel top-k similarity join algorithms using MapReduce. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 510–521 (2012)
62. Kolb, L., Thor, A., Rahm, E.: Load balancing for MapReduce-based entity resolution. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 618–629 (2012)
63. Kornacker, M., Erickson, J.: Cloudera Impala: real-time queries in Apache Hadoop, for real. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/
64. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.A.: Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In: ACM Symposium on Cloud Computing (SoCC), pp. 75–86 (2010)
65. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.A.: SkewTune: mitigating skew in MapReduce applications. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 25–36 (2012)
66. Lam, W., Liu, L., Prasad, S., Rajaraman, A., Vacheri, Z., Doan, A.: Muppet: MapReduce-style processing of fast data. Proc. VLDB Endow. (PVLDB) **5**(12), 1814–1825 (2012)
67. Laptev, N., Zeng, K., Zaniolo, C.: Early accurate results for advanced analytics on MapReduce. Proc. VLDB Endow. (PVLDB) **5**(10), 1028–1039 (2012)
68. Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. SIGMOD Rec. **40**(4), 11–20 (2011)
69. Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., Zhang, X.: YSmart: yet another SQL-to-MapReduce translator. In: Proceedings of International Conference on Distributed Computing Systems (ICDCS), pp. 25–36 (2011)
70. Leibiusky, J., Eisbruch, G., Simonassi, D.: Getting Started with Storm. O'Reilly, Ireland (2012)
71. Li, B., Mazur, E., Diao, Y., McGregor, A., Shenoy, P.J.: A platform for scalable one-pass analytics using MapReduce. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 985–996 (2011)
72. Li, B., Mazur, E., Diao, Y., McGregor, A., Shenoy, P.J.: SCALLA: a platform for scalable one-pass analytics using MapReduce. ACM Trans. Database Syst. (TODS) **37**(4), 27:1–27:38 (2012)
73. Lim, H., Herodotou, H., Babu, S.: Stubby: a transformation-based optimizer for MapReduce workflows. Proc. VLDB Endow. (PVLDB) **5**(11), 1196–1207 (2012)
74. Lin, Y., Agrawal, D., Chen, C., Ooi, B.C., Wu, S.: Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 961–972 (2011)
75. Logothetis, D., Olston, C., Reed, B., Webb, K.C., Yocum, K.: Stateful bulk processing for incremental analytics. In: ACM Symposium on Cloud Computing (SoCC), pp. 51–62 (2010)

76. Logothetis, D., Yocum, K.: Ad-hoc data processing in the cloud. Proc. VLDB Endow. (PVLDB) **1**(2), 1472–1475 (2008)
77. Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient processing of k nearest neighbor joins using MapReduce. Proc. VLDB Endow. (PVLDB) **5**(10), 1016–1027 (2012)
78. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 135–146 (2010)
79. McSherry, F., Murray, D.G., Isaacs, R., Isard, M.: Differential dataflow. In: Biennial Conference on Innovative Data Systems Research (CIDR) (2013)
80. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: interactive analysis of web-scale datasets. Proc. VLDB Endow. (PVLDB) **3**(1–2), 330–339 (2010)
81. Metwally, A., Faloutsos, C.: V-SMART-Join: a scalable MapReduce framework for all-pair similarity joins of multisets and vectors. Proc. VLDB Endow. (PVLDB) **5**(8), 704–715 (2012)
82. Mihaylov, S.R., Ives, Z.G., Guha, S.: REX: recursive, delta-based data-centric computation. Proc. VLDB Endow. (PVLDB) **5**(11), 1280–1291 (2012)
83. Nykiel, T., Potamias, M., Mishra, C., Kollios, G., Koudas, N.: MRShare: sharing across multiple queries in MapReduce. Proc. VLDB Endow. (PVLDB) **3**(1), 494–505 (2010)
84. Okcan, A., Riedewald, M.: Processing theta-joins using MapReduce. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 949–960 (2011)
85. Olston, C., Chiou, G., Chitnis, L., Liu, F., Han, Y., Larsson, M., Neumann, A., Rao, V.B.N., Sankarasubramanian, V., Seth, S., Tian, C., ZiCornell, T., Wang, X.: Nova: continuous Pig/Hadoop workflows. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1081–1090 (2011)
86. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1099–1110 (2008)
87. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 165–178 (2009)
88. Ramakrishnan, S.R., Swart, G., Urmanov, A.: Balancing reducer skew in MapReduce workloads using progressive sampling. In: ACM Symposium on Cloud Computing (SoCC), pp. 16:1–16:13 (2012)
89. Rao, S., Ramakrishnan, R., Silberstein, A., Ovsiannikov, M., Reeves, D.: Sailfish: a framework for large scale data processing. In: ACM Symposium on Cloud Computing (SoCC), pp. 4:1–4:13 (2012)
90. Rasmussen, A., Lam, V.T., Conley, M., Porter, G., Kapoor, R., Vahdat, A.: Themis: an I/O efficient MapReduce. In: ACM Symposium on Cloud Computing (SoCC), pp. 13:1–13:14 (2012)
91. Sakr, S., Liu, A., Batista, D.M., Alomari, M.: A survey of large scale data management approaches in cloud environments. IEEE Commun. Surv. Tutor. **13**(3), 311–336 (2011)
92. Schindler, J.: I/O characteristics of NoSQL databases. Proc. VLDB Endow. (PVLDB) **5**(12), 2020–2021 (2012)
93. Shim, K.: MapReduce algorithms for Big Data analysis. Proc. VLDB Endow. (PVLDB) **5**(12), 2016–2017 (2012)
94. Shinnar, A., Cunningham, D., Herta, B., Saraswat, V.A.: M3R: increased performance for in-memory Hadoop jobs. Proc. VLDB End. (PVLDB) **5**(12), 1736–1747 (2012)
95. Silva, Y.N., Larson, P.-A., Zhou, J.: Exploiting common subexpressions for cloud query processing. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 1337–1348 (2012)
96. Silva, Y.N., Reed, J.M.: Exploiting MapReduce-based similarity joins. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 693–696 (2012)
97. Silva, Y.N., Reed, J.M., Tsosie, L.M.: MapReduce-based similarity join for metric spaces. In: Proceedings of International Workshop on Cloud Intelligence (Cloud-I), pp. 3:1–3:8 (2012)
98. Stonebraker, M., Abadi, D.J., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1), 64–71 (2010)
99. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive—a warehousing solution over a Map-Reduce framework. Proc. VLDB Endow. (PVLDB) **2**(2), 1626–1629 (2009)
100. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive—a petabyte scale data warehouse using Hadoop. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 996–1005 (2010)
101. Vernica, R., Balmin, A., Beyer, K.S., Ercegovac, V.: Adaptive MapReduce using situation-aware mappers. In: Proceedings of International Conference on Extending Database Technology (EDBT), pp. 420–431 (2012)
102. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using MapReduce. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 495–506 (2010)
103. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based space partitioning for efficient parallel skyline computation. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 227–238 (2008)
104. White, T.: Hadoop—The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated). O'Reilly, Ireland (2012)
105. Wu, S., Li, F., Mehrotra, S., Ooi, B.C.: Query optimization for massively parallel data processing. In: ACM Symposium on Cloud Computing (SoCC), pp. 12:1–12:13 (2011)
106. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: International World Wide Web Conferences (WWW), pp. 131–140 (2008)
107. Xin, R., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I.: Shark: SQL and rich analytics at scale. The Computing Research Repository (CoRR), abs/1211.6176 (2012)
108. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I.: Fast and interactive analytics over Hadoop data with Spark. USENIX; login **37**(4), 45–51 (2012)
109. Zhang, J., Zhou, H., Chen, R., Fan, X., Guo, Z., Lin, H., Li, J.Y., Lin, W., Zhou, J., Zhou, L.: Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 22:1–22:14 (2012)
110. Zhang, X., Chen, L., Wang, M.: Efficient multiway theta-join processing using MapReduce. Proc. VLDB Endow. (PVLDB) **5**(11), 1184–1195 (2012)
111. Zhang, Y., Gao, Q., Gao, L., Wang, C.: PrIter: a distributed framework for prioritized iterative computations. In: ACM Symposium on Cloud Computing (SoCC), pp. 13:1–13:14 (2011)
112. Zhou, J., Bruno, N., Wu, M.-C., Larson, P.-Å., Chaiken, R., Shakib, D.: SCOPE: parallel databases meet MapReduce. VLDB J. **21**(5), 611–636 (2012)