Abstract—The MapReduce (MR) paradigm represents a milestone in the field of large-scale data analysis, even though most of its fundamental aspects have been already studied thoroughly in the last 3 decades by the database community. In this report, we will highlight the design trade-offs made by MR to achieve better horizontal scalability, at the price of lower efficiency compared to a parallel database. We will then describe some recent large-scale systems for BigData processing – by addressing some of the characteristic MR trade-offs, they represent its natural evolution for specific use-cases (e.g. interactive analysis, incremental processing).

Index Terms—MapReduce, Percolator, Dremel, large-scale, data analysis

I. INTRODUCTION

The data deluge we are experiencing nowadays does not show any signs of slowing down. A recent report [4] indicates that 6.7 Exabytes of data are created or replicated every two days. With such daunting quantity in the picture, it does not come as a surprise that part of the data management researchers (and even developers) steered their attention towards scalability rather than additional functionalities. Most of the data, even if not processed with sophisticated techniques, can be used to extract valuable information [5].

Successful Web companies have been aware of this insight from the very beginning, and put considerable effort in handling what can be considered the largest artifact of mankind (at least in terms of amount of information): the World Wide Web. Internet giants juggle more data than any other IT organization.

II. BACKGROUND

This section is based on Pavlo et al. - “A Comparison of Approaches to Large-Scale Data Analysis” [8].

Though it may seem that MR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set of MapReduce jobs. Hence, it is important to understand the differences between the MapReduce approach to performing large-scale data analysis and the approach taken by parallel database systems.

1companies like Yahoo! and Cloudera invested several man-years on it
We will now describe the different choices made by the two classes of systems in several key areas, identifying the trade-offs that they entail.

A. Schema Support

Parallel DBMSs require data to fit into the relational paradigm of rows and columns. In contrast, the MR model does not require that data files adhere to a schema defined using the relational data model. That is, the MR programmer is free to structure their data in any manner or even to have no structure at all.

Both approaches require a similar amount of effort. While, for instance, SQL requires that the programmer must specify the “shape” of the data in a data definition facility, the MR programmer must often write a custom parser in order to derive the appropriate semantics for their input records.

MR is thus more flexible, and let the developer quickly process a dataset without prior knowledge of its structure. On the other hand, if the dataset becomes shared among more developers, it is likely that a certain amount of code will have to be shared. Though such requirement is not daunting nowadays due to factor schema definitions and integrity constraints out of application programs (i.e. DBMS catalogs) clearly represents a more elegant and reliable solution.

B. Indexing

While all modern DBMSs use hash or B-tree indexes to accelerate access to data, the initial model proposed for MR does not provide built-in indexes. The programmer is thus forced to implement its own indexing solution, or rely on specific characteristics of the dataset (that cannot be specified in any catalog, but just through the application logic in the codebase). As a result, when a query has high selectivity, MR represents a brute force solution which does not shine in terms of efficiency.

C. Programming Model and Flexibility

MR vs. declarative high-level languages (e.g. SQL) revamps a debate that the database research community engaged in during the 1970s. MR allows the vast majority of programmers (i.e. with an imperative language background) to perform record-level manipulation, without digging into a challenging language like SQL. Although SQL has passed the test of time (thanks to its expressiveness and succinctness) most of the complex data-manipulation pipelines based on DBMSs heavily rely on stored procedures. As a matter of fact, DBMS vendors have spent a sensible amount of resources to lower down the steep learning curve of stored procedures, implicitly admitting how cumbersome is their programming model.

MR, instead, is based on primitive concepts of functional programming, while proposing a simple procedural API that has been adopted even by DMBS vendors (e.g. Asterdata, Greenplum, Couchbase, etc.)

2considering the existence of build systems with complex dependency support

D. Data Distribution

The conventional wisdom for large-scale databases is to always send the computation to the data, rather than the other way around. In other words, one should send a small program over the network to a node, rather than importing a large amount of data from the node. Parallel DBMSs use knowledge of data distribution and location to their advantage: a parallel query optimizer strives to balance computational workloads while minimizing the amount data transmitted over the network connecting the nodes of the cluster.

While MR adopts the same philosophy, the restrictions imposed by the framework do not allow for operator reordering (i.e. a classic optimization performed by the query optimizer). The Map phase always scan the whole amount of data, while performing filtering. A DBMS, instead, can maintain a materialized view where each WHERE clause is pushed as up as possible in the operators tree, such that it will quickly narrow down the amount of processed data.

While the potential of materialized views is undeniable, the cost and complexity to maintain them cannot be ignored. As much as MR utilizes a brute-force approach, materialized views represent a constant resource burden for the DBMS, hindering its scalability and fair scheduling features.

E. Execution Strategy

In MR, each Map checkpoints the intermediate results on disk (i.e. one result file for each Reduce task). The intermediate results are then pulled by each Reduce task via a network file-transfer protocol. Aside from the network and checkpointing overhead, with 100s of Reduce instances running simultaneously, the “pull” phase induces large numbers of disk seeks due to concurrent accesses. This is why parallel database systems do not materialize their split files and instead use a push approach to transfer data instead of a pull.

F. Fault Tolerance

The MR frameworks provide a more sophisticated failure model than parallel DBMSs. While both classes of systems use some form of replication to deal with disk failures, MR is far more adept at handling node failures during the execution of a MR computation. In a MR system, if a unit of work (i.e. processing a block of data) fails, then the MR scheduler can automatically restart the task on an alternate node. Part of the flexibility is the result of the fact that the output files of the Map phase are materialized locally instead of being streamed to the nodes running the Reduce tasks.

This differs from parallel DBMSs, which have larger granules of work (i.e. transactions) that are restarted in the event of a failure. Part of the reason for this approach is that DBMSs avoid saving intermediate results to disk whenever possible. Thus, if a single node fails during a long running query in a DBMS, the entire query must be completely restarted.

Although in most scenarios the fault-tolerance provided by MR could be overkill, such feature is mandatory in large cluster deployments (i.e. thousands of nodes) where server failures happen daily (if not hourly).
Comparison benchmarks between MR and parallel databases displayed a significant performance advantage for the latter in executing a variety of data intensive analysis tasks. While it is clear that parallel databases can address a plethora of use-cases in a more efficient way, MR still represents the only viable solution for the few large-scale internet companies that have to deal with a constant data deluge. Not only the fault-tolerance features are mandatory in such deployments, but also the sensible advantage in data loading time\(^3\) makes MR a fundamental tool to explore data in a timely fashion [5].

What the DB community research [8] proved is that MR is still a relatively young technology compared to DBMSs – the minimalistic MR design choices make it efficient only in a few scenario. As we will show in the next section, large-scale systems are quickly recovering the gap with tremendous engineering effort, showing how specialization rather than broadness is the key factor to achieve stunning performances. As a consequence, these systems also show a substantial overlapping with parallel databases, blurring more and more the aforementioned differences.

III. SURVEY OF THE SELECTED PAPERS

This section is based on Melnik et al. - “Dremel: Interactive Analysis of Web-Scale Datasets” [6] and on Peng et al. - “Large-scale Incremental Processing Using Distributed Transactions and Notifications” [9].

A. Dremel

Dremel is a scalable, interactive ad-hoc query system for analysis of read-only nested data. By combining multi-level execution trees and columnar data layout, it is capable of running aggregation queries over trillion-row tables in seconds. The system scales to thousands of CPUs and petabytes of data, and has thousands of users at Google.

Unlike traditional databases, it is capable of operating on in situ nested data. In situ refers to the ability to access data “in place”, e.g. in a distributed file system (like the Hadoop File System) or another storage layer (e.g. Bigtable[1]).

Though it shares some fundamental aspects, Dremel does not replace but just complements MapReduce-based computing – it covers the interactivity niche because MapReduce, at least in Hadoop, is not tuned to return results in seconds.

Main contributions follow.

1) Column storage format: Dremel uses a column-striped storage representation, which enables it to read less data from secondary storage and reduce CPU cost due to cheaper compression. Contrary to column stores that have been adopted for analyzing relational data, Dremel also supports the nested data model. Figure 1 illustrates the main idea: all values of a nested field such as A.B.C are stored contiguously. Hence, A.B.C can be retrieved without reading A.E, A.B.D, etc. The data model is simply based on strongly-typed nested records: in this way, cross-language interoperability is achievable by means of a standard binary on-the-wire representation of records, in which field values are laid out sequentially as they occur in the record. Parsing overhead that afflicted MR is substantially mitigated in Dremel, effectively introducing schema support in the platform.

Physical layout of the nested columnar storage is, instead, more challenging. Record structure in columnar format must be represented in a lossless way, while achieving both fast encoding and efficient record assembly.

Lossless encoding is obtained by assigning to each record both a repetition and definition level. Intuitively, those fields represent the exact position of the record in the nested structure. Given that each column is stored as a set of blocks, each block contain the repetition and definition levels per each record, plus the compressed field values. Hence, NULLs are not stored explicitly as they are determined by the definition levels, achieving a remarkable space saving with very sparse data.

Thanks to this structured representation, splitting records into columns as reassembling them are both efficient operations. Missing fields almost do not require any processing, and I/O can be parallelized with a tree of field writers, whose structure matches the field hierarchy in the schema. Record reassembling, instead, can be performed with a finite state machine, where state transitions are labeled with repetition levels. If only a small subset of fields need to be retrieved, the record reassembling procedure will build a FSM which is simpler (i.e. less states and many collated transitions), hence cheaper to execute.

Figure 3 outlines the different performances between record-wise and columnar approaches, illustrating the time it takes

\[^3\]compared to a relational DBMS

---

**Fig. 1.** Record-wise vs. columnar representation of nested data

**Fig. 2.** Record-wise vs. columnar: performance breakdown
to read and uncompress the data, and assemble and parse the records, for a subset of the fields. The main takeaways of this experiment are the following: when few columns are read, the gains of columnar representation are of about an order of magnitude. Retrieval time for columnar nested data grows linearly with the number of fields. Record assembly and parsing are expensive, each potentially doubling the execution time. The crossover point of the two subgraph lies around few dozens of fields, making the record-wise approach palatable only when the bulk of the data needs to be retrieved.

2) Query Execution Tree: Dremel uses a multi-level serving tree to execute queries (see Figure 3). A root server receives incoming queries, reads metadata from the tables, and routes the queries to the next level in the serving tree. The leaf servers communicate with the storage layer or access the data on local disk.

For example, in an aggregation query, the root server determines all the horizontal partitions of the table, and dispatches a subquery to each one of the intermediate servers. With a similar rewriting technique, the query reaches the leaves of the system, obtaining the actual results. On the way up, intermediate servers perform a parallel aggregation of partial results.

Query dispatching (and hence scheduling) in a complex multi-user system has a fundamental role, not only for load balancing, but also to provide fault tolerance when one server becomes much slower than others (or just unreachable). Thus, the dispatcher has to keep track of each “horizontal partition” processing time, and redispach the query on another server when notices an outlier task. The query dispatcher also honors a parameter that specifies the minimum percentage of partitions that must be scanned before returning a result. Setting such parameter to a slightly lower value (e.g. 98% instead of 100%) speeds up execution significantly, just by ignoring the long tail of straggler tasks.

B. Percolator

Percolator is a system for incrementally processing updates to a large data set – it is currently deployed to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing, Google processes the same number of documents per day, while reducing the average age of documents in search results by 50% (as per Figure 4).

Incrementally updating an index of the web is an example of a class of data processing tasks that transform a large repository of data via small, independent mutations. While parallel databases do not meet the storage and throughput requirements needed for such large amount of data, MapReduce might be seen as a reasonable choice. Nevertheless, MR and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency. Thus, the need for a new platform that shares some key requirements with MR, but trade-offs efficiency for finer-granularity of processing and updating.

Although finer-granularity could have been achieved by simply utilizing Bigtable [1] as the backend storage system, incremental processing needs also two main abstractions: ACID transactions (to maintain invariants) and observers (to trigger the transactions). Percolator thus includes both requirements coming from distributed systems and DBMSs, making it lie in the performance space between MR and PDBMSs.

Main contributions follow.

1) Transactions: Percolator provides cross-row, cross-table transactions with ACID snapshot-isolation semantics. Snapshot isolation is a form of optimistic concurrency control that protects against write-write conflicts. At the same time, though, snapshot isolation does not provide serializability (i.e. subject to write skew). Percolator makes the assumption of a low contention access pattern, thus making read operations more efficient.

While it is possible to incrementally process data without the benefit of strong transactions, transactions make it more tractable for the user to reason about the state of the system and to avoid the introduction of errors into a long-lived repository. For example, in a transactional web-indexing system the programmer can make assumptions like: the hash of the contents of a document is always consistent with the table that indexes duplicates. Without transactions, an ill-timed crash could result in a permanent error: an entry in the document table that corresponds to no URL in the duplicates table. Transactions also make it easy to build index tables that are always up to date and consistent. Note that both of these
examples require transactions that span rows, rather than the single-row transactions that Bigtable already provides.

Because it is built as a client library accessing Bigtable, rather than controlling access to storage itself, Percolator faces a different set of challenges implementing distributed transactions than traditional PDBMSs. Other parallel databases integrate locking into the system component that manages access to the disk: since each node already mediates access to data on the disk it can grant locks on requests and deny accesses that violate locking requirements. By contrast, any node in Percolator can (and does) issue requests to directly modify state in Bigtable: there is no convenient place to intercept traffic and assign locks. As a result, Percolator explicitly maintains locks. Locks persist in the face of machine failure; if a lock could disappear between the two phases of commit, the system could mistakenly commit two transactions that should have conflicted. Similarly, the lock service provides high throughput; it can serve thousands of machines requesting locks simultaneously.

Figure 5 highlights how the Percolator design allows linear scalability in the transaction rate. Notably, such linear scalability comes at the cost of 30 times more CPU per transaction than the benchmark reference system (i.e. TPC-E flagship).

2) Notifications: In Percolator, user code runs as an observer which is triggered by changes to the observed table. Each observer registers a function and a set of columns with Percolator, and Percolator invokes the function after data is written to one of those columns in any row. Percolator applications are structured as a series of observers; each observer completes a task and creates more work for “downstream” observers by writing to the table.

Notifications are similar to database triggers or events in active databases, but unlike database triggers, they cannot be used to maintain database invariants. In particular, the triggered observer runs in a separate transaction from the triggering write, so the triggering write and the triggered observer’s writes are not atomic. Notifications are intended to help structure an incremental computation rather than to help maintain data integrity. This difference in semantics and intent makes observer behavior much easier to understand than the complex semantics of overlapping triggers.

Various optimizations (like message collapsing) allow Percolator to avoid computation by amortizing the cost of responding to many notifications, while being more responsive towards newly-generated triggers.

IV. CONCLUSION

Although MR authors already addressed on paper 3 many of the concerns risen by the data management community throughout the last years, this report shows how recent large-scale systems implicitly confirm that each design belong to its own niche. MR represents a novelty only for its massive scale capabilities, but it has nevertheless revolutionized the BigData community with the same strength of an avalanche. The outcome of such avalanche is a comfortable snowy slope, where researchers and developers realized (once again) that any effort to build the best tool for the job is a valuable effort, no matter its potential breakthrough-factor.

V. RESEARCH PROPOSAL

MapReduce, Dremel and Percolator cover a substantial portion of large-scale data processing usecases. Nevertheless, they are all designed with Web data as first-class citizen, even though the BigData scenario is quickly mutating. Web data (as per HTML pages, click logs, etc.) is not anymore the only large dataset that deserves to be thoroughly processed and analyzed. Mobile devices, for instance, can generate an accurate time-series of GPS coordinates of each user. Geographical deployments of hardware sensors keep track of environmental parameters (e.g. temperature, wind speed, pollution, etc.), generating massive amounts of time-series day after day. The simple concept of adding a timestamp to each piece of generated data, can give birth to a data inflation second to none. Intuitively, it would be like requiring a Web search engine to work with each crawled version of a Web page.

Software versioning and backup systems teach us a convincing lesson on the value of historical data. Recommendation systems incrementally build a user profile, analyzing his purchases and browsing activity. Ultimately, social-networking Websites study user behavior by analyzing the evolution of his social network and of “state mutations” (e.g. a Twitter message). Hence, there is a clear need for large-scale data analysis systems that do not consider time only as the 3rd dimension of the storage backend (e.g. OLAP cubes) but rather as a first-class citizen.

Being able to make such system scale, would mean being able to follow and collect all the trails a Web-user is leaving everyday in the cyberspace. What today exists only in a federated system (i.e. spread among a user’s Facebook, Twitter, Last.fm, etc. accounts) could be effectively collated and analyzed as a whole, giving birth to completely new scenarios (e.g. a reputation system based on factual algorithms rather than self-certification). Intuitively, the amount of data involved would be so daunting that new approaches in terms of computing-paradigm and compression will be needed.

A description of preliminary work on a time-series scalable platform follows.
A. TimeCloud

TimeCloud is a storage-and-computing platform for managing large-scale time series in the cloud. TimeCloud is built upon scalable, fault-tolerant distributed systems as Hadoop and HBase, while facilitating complex data analysis over massive time series by taking novel approaches:

**Partition-and-Cluster store:** to manage time series in a distributed environment, TimeCloud takes a novel data storage scheme, consisting of the following three steps: (i) It partitions a given time series into a set of tuple blocks where each block contains some consecutive tuples while preserving time order. (ii) For all tuple blocks collected from each time-series partitioning, it clusters similar tuple blocks together according to a given similarity or distance measure. (iii) We then store each set of clustered tuple blocks into a column family of HBase on the same node. TimeCloud offers four different data partitioning schemes, each of which embodies the partition-and-cluster store optimized for certain types of queries.

**Model-based cache:** Model-based views were introduced to achieve synergy between data processing using models and powerful data management in databases. TimeCloud adopts this approach for efficiently processing and transmitting data across nodes. Specifically, the front-end of TimeCloud represents a time series with an approximation model, then various types of queries (e.g. SELECT) are directly evaluated over the model-based views generated by the model. This enables fast query response for the front-end without fetching the correspond actual data from the back-end, as well as reduced workload for the back-end.

**Model-coding join:** We introduce a novel time-series compression scheme, model coding, that achieves very high compression of time series using bitmap encoding of predefined model parameters. Based on this model coding, we efficiently evaluate join queries for distributed time series that are stored in different nodes, by transmitting only bitmap-encoded data from one node to another.

A sketch of the system architecture is proposed in Figure 6. Notably, the frontend not only offers basic functionalities for Web-based time-series data management, including log-in access, data upload, group-based data share, and data visualization. It also embodies the model-based cache: specifically, model-generated data is used for data visualization by default. Only when necessary (e.g. user request from the UI), the front-end issues queries to retrieve full-precision data from the back-end, substantially reducing the back-end workload.

**REFERENCES**


