IMT - Institutions Markets Technologies

Institute for Advanced Studies
Lucca

Cloud computing for large scale data analysis

PhD Program in Computer Science and Engineering
XXIV Cycle

Gianmarco De Francisci Morales

gianmarco.dfmorales@imtlucca.it

Supervisor:
Dott. Claudio Lucchese

Co-supervisor:
Dott. Ranieri Baraglia

15 February 2010

Abstract

An incredible *"data deluge"* is currently drowning the world. Data sources are everywhere, from the Web 2.0 to large scientific experiments, from social networks to sensors. This huge amount of data is a valuable asset in our information society.

Data analysis is the process of inspecting data in order to extract useful information. Decision makers commonly use this information to drive their choices. The quality of the information extracted by this process greatly benefits from the availability of extensive data sets.

As we enter the *"Petabyte Age"*, traditional approaches for data analysis begin to show their limits. *"Cloud computing"* is an emerging alternative technology for large scale data analysis. Data oriented cloud systems combine both storage and computing in a distributed and virtualized manner. They are built to scale to thousands of computers, and focus on fault tolerance, cost effectiveness and ease of use.

This thesis aims to provide a coherent framework for research in the field of large scale data analysis on cloud computing systems. The goal of our research is threefold. I) To build a toolbox of algorithms and propose a way to evaluate and compare them. II) To design extensions to current cloud paradigms in order to support these algorithms. III) To develop methods for online analysis of large scale data.

In order to reach our goal, we plan to adopt principles from database research. We postulate that many results in this field are relevant also for cloud computing systems. With our work, we expect to provide a common ground on which the database and cloud communities will be able to communicate and thrive.

Contents

List of Figures **iii**

List of Tables **iv**

List of Acronyms **v**

1 Introduction **1**

1.1 The *"Big Data"* Problem 2

1.2 Methodology Evolution 5

2 State of the Art **10**

2.1 Technology Overview 10

2.2 Comparison with PDBMS 18

2.3 Case Studies 21

2.4 Research Directions 22

3 Research Plan **29**

3.1 Research Problem 29

3.2 Thesis Proposal 31

Bibliography **33**

List of Figures

1.1 The Petabyte Age 2
1.2 Data Information Knowledge Wisdom hierarchy 3

2.1 Data oriented cloud computing architecture 11
2.2 The MapReduce programming model 15

List of Tables

2.1 Major cloud software stacks 12
2.2 Relative advantages of PDBMS and Cloud Computing 20

List of Acronyms

ACID	Atomicity, Consistency, Isolation, Durability 19
BASE	Basically Available, Soft state, Eventually consistent 19
DAG	Direct Acyclic Graph 16
DBMS	Database Management System 5
DHT	Distributed Hash Table 14
DIKW	Data Information Knowledge Wisdom 3
GFS	Google File System 12
HDFS	Hadoop Distributed File System 12
IaaS	Infrastructure as a Service 8
MPI	Message Passing Interface 8
MR	MapReduce 14
OLAP	Online Analysis Processing 6
OLTP	Online Transaction Processing 6
PaaS	Platform as a Service 8
PDBMS	Parallel Database Management System 6
PDM	Parallel Disk Model 1
PVM	Parallel Virtual Machine 8
RAM	Random Access Machine 1
SaaS	Software as a Service 8
SOA	Service Oriented Architecture 8
SQL	Structured Query Language 6
UDF	User Defined Function 7

Chapter 1

Introduction

How would you sort 1GB of data? Today's computers have enough memory to hold this quantity of data, so the best way is to use an in-memory algorithm like quicksort. What if you had to sort 100 GB of data? Even if systems with more than 100 GB of memory exist, they are by no means common or cheap. So the best solution is to use a disk based sort like mergesort or greedsort. However, what if you had 10 TB of data to sort? Today's hard disks are usually 1 to 2 TB in size, which means that just to hold 10 TB of data we need multiple disks. In this case the bandwidth between memory and disk would probably become a bottleneck. So, in order to obtain acceptable completion times, we need to employ multiple computing nodes and a parallel algorithm like bitonic sort.

This example illustrates a very general point: the same problem at different scales needs radically different solutions. In many cases, the model we use to reason about the problem needs to be changed. In the example above, the Random Access Machine (RAM) model [59] should be used in the former case, while the Parallel Disk Model (PDM) [67] in the latter. The reason is that the simplifying assumptions made by the models do not hold at every scale. Citing George E. P. Box "Essentially, all models are wrong, but some are useful" [7], and arguably "Most of them do not scale".

For data analysis, scaling up of data sets is a *"double edged"* sword. On the one hand, it is an opportunity because *"no data is like more data"*. Deeper insights are possible when more data is available. Oh the other hand, it is a challenge. Current methodologies are often not suitable to handle very large data sets, so new solutions are needed.

1.1 The *"Big Data"* Problem

The issues raised by large data sets in the context of analytical applications are becoming ever more important as we enter the *"Petabyte Age"*. Figure 1.1 shows some of the data sizes for today's problems [5]. The data sets are orders of magnitude greater than what fits on a single hard drive, and their management poses a serious challenge. Web companies are currently facing this issue, striving to find efficient solutions. The ability to analyze or manage more data is a distinct competitive advantage. This problem has been labeled in various ways: petabyte scale, web scale or *"big data"*.

Figure 1.1: The Petabyte Age

Currently, an incredible *"data deluge"* is drowning the world with data. The quantity of data we need to sift through every day is enormous, just think of the results of a search engine query. They are so many that we are not able to examine all of them, and indeed the competition on web search now focuses on ranking the results. This is just an example of a more general trend.

Data sources are everywhere. Web users produce vast amounts of text, audio and video contents in the so called *Web 2.0*. Relationships and tags in social networks create large graphs spanning millions of vertexes and edges.

Scientific experiments are another huge data source. The Large Hadron Collider (LHC) at European Council for Nuclear Research (CERN) is expected to generate around 50 TB of raw data per day. The Hubble telescope

captured hundreds of thousands astronomical images, each hundreds of megabytes large. Computational biology experiments like high-throughput genome sequencing produce large quantities of data that require extensive post-processing.

In the future, sensors like Radio-Frequency Identification (RFID) tags and Global Positioning System (GPS) receivers will spread everywhere. These sensors will produce petabytes of data just as a result of their sheer numbers, thus starting *“the industrial revolution of data”* [33].

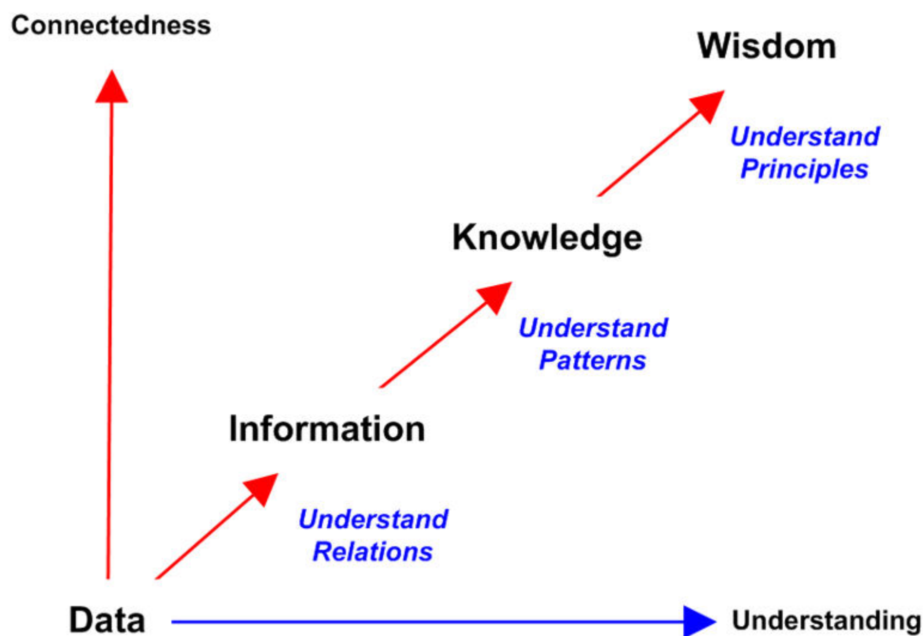

Figure 1.2: Data Information Knowledge Wisdom hierarchy

But why are we interested in data? It is common belief that data without a model is just noise. Models are used to describe salient features in the data, which can be extracted via data analysis. Figure 1.2 represents the popular Data Information Knowledge Wisdom (DIKW) hierarchy [56]. In this hierarchy data stands at the lowest level and bears the smallest level of *understanding*. Data needs to be processed and condensed into more connected forms in order to be useful for event comprehension and decision making. Information, knowledge and wisdom are these forms of understanding. Relations and patterns that allow to gain deeper awareness of the process that

generated the data, and principles that can guide future decisions.

The large availability of data potentially enables more powerful analysis and unexpected outcomes. For example, Google can detect regional flu outbreaks up to ten days faster than the Center for Disease Control and Prevention by monitoring increased search activity for flu-related topics [29]. Chapter 2 presents more examples of interesting large scale data analysis problems.

But how do we define *"big data"*? The definition is of course relative and evolves in time as technology progresses. Indeed, thirty years ago one terabyte would be considered enormous, while today we are commonly dealing with such quantity of data.

According to Jacobs [37], big data is "data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time". This means that we can call *"big"* an amount of data that forces us to use or create innovative research products. In this sense big data is a driver for research. The definition is interesting because it puts the focus on the subject who needs to manage the data rather than on the object to be managed. The emphasis is thus on user requirements such as throughput and latency.

Let us highlight some of the requirements for a system used to perform data intensive computing on large data sets. Given the effort to find a novel solution and the fact that data sizes are ever growing, this solution should be applicable for a long period of time. Thus the most important requirement a solution has to satisfy is scalability.

Scalability is defined as "the ability of a system to accept increased input volume without impacting the profits". This means that the gains from the input increment should be proportional to the increment itself. This is a broad definition used also in other fields like economy, but more specific ones for computer science are provided in the next section.

For a system to be fully scalable, its scale should not be a design parameter. Forcing the system designer to take into account all possible deployment sizes for a system, leads to a scalable architecture without fundamental bottlenecks. We define such a system *"scale-free"*.

However, apart from scalability, there are other requirements for a large scale data intensive computing system. Real world systems cost money to build and operate. Companies attempt to find the most cost effective way of building a large system because it usually requires a significant money

investment. Partial upgradability is an important money saving feature, and is more easily attained with a loosely coupled system. Operational costs like personnel salary (i.e. system administrators) account for a large share of the budget of IT departments. To be profitable, large scale systems must require as little human intervention as possible. Therefore autonomic systems are preferable, systems that are self-configuring, self-tuning and self-healing. In this respect, a key property is fault tolerance.

Fault tolerance is "the property of a system to operate properly in spite of the failure of some of its components". When dealing with a large system, the probability that a disk breaks or a server crashes raises dramatically: it is the norm rather than the exception. A performance degradation is acceptable as long as the systems does not halt completely. A denial of service of a system usually has a negative economic impact, especially for Web-based companies. The goal of fault tolerance techniques is to create a highly available system.

To summarize, a large scale data analysis system should be scalable, cost effective and fault tolerant.

1.2 Methodology Evolution

Providing data for analysis is a problem that has been extensively studied. Many different solutions exist. However, the usual approach is to employ a Database Management System (DBMS) to store and manage the data.

DBMSs were born in the '60s and used a navigational model. The so called CODASYL approach, born together with COBOL, used a hierarchical or network schema for data representation. The user had to explicitly traverse the data using links to find what he was interested in. Even simple queries like *"Find all people in India"* were prohibitively expensive as they required to scan all the data.

The CODASYL approach was partly due to technological limits: the most common mass storage technology at the time was tape. Magnetic tapes have very high storage density and throughput for linear access, but extremely slow random access performance due to large seek times. Notice that this description fits almost perfectly current disk technology, and provides the rationale for some of the new developments in data management systems [31].

During the '70s Codd [13] introduced the famous relational model that

is still in use today. The model introduces the familiar concepts of tabular data, relation, normalization, primary key, relational algebra and so on. The model had two main implementations: IBM's System R and Berkley's INGRES. All the modern databases can be in some way traced back to these two common ancestors in terms of design and code base [69]. For this reason DBMSs bear the burden of their age and are not always a good fit for today's applications.

The original purpose of DBMSs was to process transactions in business oriented processes, also known as Online Transaction Processing (OLTP). Queries were written in Structured Query Language (SQL) and run against data modeled in relational style. On the other hand, currently DBMSs are used in a wide range of different areas: besides OLTP, we have Online Analysis Processing (OLAP) applications like data warehousing and business intelligence, stream processing with continuous queries, text databases and much more [60]. Furthermore, stored procedures are used instead of plain SQL for performance reasons. Given the shift and diversification of application fields, it is not a surprise that most existing DBMSs fail to meet today's high performance requirements [61, 62].

High performance has always been a key issue in database research. There are usually two approaches to achieve it: vertical and horizontal. The former is the simplest, and consists in adding resources (cores, memory, disks, etc...) to an existing system. If the resulting system is capable of taking advantage of the new resources it is said to scale up. The inherent limitation of this approach is that the single most powerful system available on earth could be not enough. The latter approach is more complex, and consists in adding in parallel new separate systems to the existing one. If higher performance is achieved the systems is said to scale out. The multiple systems are treated as a single logical unit. The result is a parallel system with all the (hard) problems of concurrency.

Typical parallel systems are divided into three categories according to their architecture: shared memory, shared disk or shared nothing. In the first category we find Symmetric Multi-Processors (SMPs) and large parallel machines. In the second one we find rack based solutions like Storage Area Network (SAN) or Network Attached Storage (NAS). The last category includes large commodity clusters interconnected by a local network and is usually deemed the most scalable [63].

Parallel Database Management Systems (PDBMSs) [21] are the result of these considerations. They are an attempt to achieve high performance in a parallel fashion. Almost all the designs of a PDBMS use the same basic dataflow pattern for queries and horizontal partitioning of the tables on a cluster of shared nothing machines [23].

Unfortunately, PDBMS are very complex systems. They need fine tuning of many *"knobs"* and feature simplistic fault tolerance policies. In the end, they do not provide the user with adequate ease of installation and administration (the so called *"one button"* experience), and flexibility of use (poor support of User Defined Functions (UDFs)).

To date, despite numerous claims about their scalability, PDBMSs have proven to be profitable only up to the tens or hundreds of nodes. It is legitimate to question whether this is the result of a fundamental theoretical problem in the parallel approach.

Parallelism has some well known limitations. Amdahl [4] in his 1967 paper argued in favor of a single-processor approach to achieve high performance. Indeed, the famous "Amdahl's law" states that the parallel speedup of a program is inherently limited by the inverse of his serial fraction, the non parallelizable part of the program. His law also defines the concept of strong scalability, in which the *total* problem size is fixed. Equation 1.1 specifies Amdahl's law for N parallel processing units where r_s and r_p are the serial and parallel fraction of the program measured on a single processor ($r_s + r_p = 1$)

$$SpeedUp(N) = \frac{1}{r_s + \frac{r_p}{N}} \tag{1.1}$$

Nonetheless, parallel techniques are justified from the theoretical point of view. Gustafson [32] in 1988 re-evaluated Amdahl's law using different assumptions, i.e. the problem sizes increases with the computing units. In this case the problem size *per unit* is fixed. Under these assumptions the achievable speedup is almost linear, as stated in Equation 1.2. In this case r'_s and r'_p are the serial and parallel fraction measured on the parallel system instead of the serial one. The equation also defines the concept of scaled speedup or weak scalability.

$$SpeedUp(N) = r'_s + r'_p * N = N + (1 - N) * r'_s \tag{1.2}$$

Even though the two formulations are mathematically equivalent [58], they make drastically different assumptions. In our case the size of the problem is large and ever growing. Hence it seems appropriate to adopt Gustafson's point of view, which justifies the parallel approach.

There are also practical considerations that corroborate parallel methods. Physical limits for processors have been reached. Chip makers are no more raising clock frequencies because of heat dissipation and power consumption concerns. As a result, they are already introducing parallelism at the CPU level with manycore processors [35]. In addition, the most cost effective option to build a large system is to use many commodity inexpensive components instead of deploying high-end servers. These issues have led to a renewed popularity of parallel computing.

Parallel computing has a long history. It has traditionally focused on *"number crunching"*. Common applications were tightly coupled and CPU intensive (e.g. large simulations or finite element analysis). Control-parallel programming interfaces like Message Passing Interface (MPI) or Parallel Virtual Machine (PVM) are still the de-facto standard in this area. These systems are notoriously hard to program. Fault tolerance is difficult to achieve and scalability is an art. They require explicit control of parallelism and are sometimes referred to as *"the assembly language of parallel computing"*.

In stark contrast with this legacy, a new class of parallel systems has emerged: cloud computing. Cloud systems focus on being scale-free, fault tolerant, cost efficient and easy to use.

Cloud computing is the result of the convergence of three technologies: grid computing, virtualization and Service Oriented Architecture (SOA). The aim of cloud computing is thus to offer services on a virtualized parallel back-end system. These services are divided in categories according to the resource they offer: Infrastructure as a Service (IaaS) like Amazon's EC2 and S3, Platform as a Service (PaaS) like Google's App Engine and Microsoft's Azure Services Platform, Software as a Service (SaaS) like Salesforce, OnLive and virtually every Web application.

Lately cloud computing has received a substantial amount of attention from industry, academia and press. As a result, the term "cloud computing" has become a buzzword, overloaded with meanings. There is lack of consensus on what is and what is not *cloud*, as even simple client-server applications

on portable devices are sometimes included in the category [17]. As usual, the boundaries between similar technologies are fuzzy, so there is no clear distinction among grid, utility, cloud, and other kind of computing technologies. In spite of the many attempts to describe cloud computing [48], there is no widely accepted definition.

Even without a clear definition, there are some properties that we think a cloud computing system should have. Each of the three aforementioned technologies brings some feature to cloud computing. According to SOA principles, a cloud system should be a distributed system, with separate, loosely coupled entities collaborating among each other. Virtualization (not intended just as x86 virtualization but as a general abstraction of computing, storage and communication facilities) provides for location, replication and failure transparency. Finally, grid computing endorses scalability. All these properties are clearly well grounded in parallel and distributed systems research. Indeed, cloud computing can be thought of as the current step in this area.

Within cloud computing, there is a subset of technologies that is more geared towards data analysis. These systems are aimed mainly at I/O intensive tasks, are optimized for streaming data from disk drives and use a data-parallel approach. An interesting feature is they are *"dispersed"*: computing and storage facilities are distributed, abstracted and intermixed. These systems attempt to move computation as close to data as possible because moving large quantities of data is expensive. Finally, the burden of dealing with the issues caused by parallelism is removed from the programmer. This provides the programmer with a *scale-agnostic* programming model.

Data oriented cloud computing systems are a natural alternative to PDBMSs when dealing with large scale data. As such, a fierce debate is currently taking place, both in industry and academy, on which is the best tool.

The remainder of this document is organized as follows. Chapter 2 gives a technical and research overview of data oriented cloud systems. To set the frame for the thesis dissertation we show a comparison with PDBMS. Finally, Chapter 3 describes the research proposal in greater detail.

Chapter 2

State of the Art

Large scale data challenges have spurred a large number of projects on data oriented cloud computing systems. Most of these projects feature important industrial partners alongside academic institutions. Big internet companies are the most interested in finding novel methods to manage and use data. Hence, it is not surprising that Google, Yahoo!, Microsoft and few others are leading the trend in this area.

In the remainder of this document we will focus only on the data oriented part of cloud technologies, and will simply refer to them as cloud computing. In the next sections we will describe the technologies that have been developed to tackle large scale data intensive computing problems, and the research directions in this area.

2.1 Technology Overview

Even though existing cloud computing systems are very diverse, they share many common traits. For this reason we propose a general architecture of cloud computing systems, that captures the commonalities in the form of a multi-layered stack. Figure 2.1 shows the proposed software stack of a cloud system.

At the lowest level we find a coordination level that serves as a basic building block for the distributed services higher in the stack. This level deals with basic concurrency issues.

The distributed data layer builds on top of the coordination one. This layer deals with distributed data access, but unlike a traditional distributed file system it does not offer standard POSIX semantics for the sake of per-

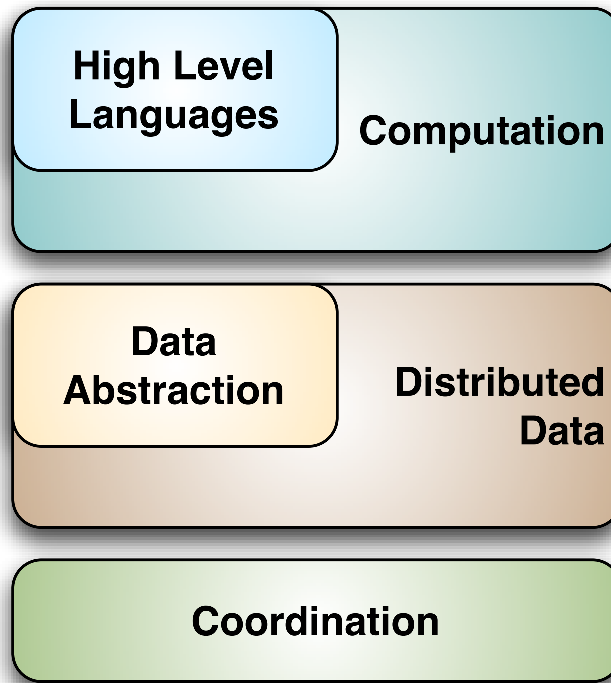

Figure 2.1: Data oriented cloud computing architecture

formance. The data abstraction layer is still part of the data layer and offers different, more sophisticated interfaces to data.

The computation layer is responsible for managing distributed processing. As with the data layer, generality is sacrificed for performance. Only embarrassingly data parallel problems are commonly solvable in this framework. The high level languages layer encompasses a number of languages, interfaces and systems that have been developed to simplify and enrich access to the computation layer.

Table 2.1 reports some of the most popular cloud computing softwares classified according to the architecture of Figure 2.1. Google has described his software stack in a series of published papers, but the software itself is not available outside the company. Some independent developers, later hired by Yahoo!, implemented the stack described by Google and released the project under an open-source license. Hadoop [6] is now an umbrella project of the Apache Software Foundation and is used and developed extensively by Yahoo!. Microsoft has released part of its stack in binary format, but the

	Google	**Yahoo!**	**Microsoft**	**Others**
High Level Languages	Sawzall	Pig Latin	DryadLINQ SCOPE	Hive Cascading
Computation	MapReduce	Hadoop	Dryad	
Data Abstraction	BigTable	HBase PNUTS		Cassandra Voldemort
Distributed Data	GFS	HDFS	Cosmos	Dynamo
Coordination	Chubby	Zookeeper		

Table 2.1: Major cloud software stacks

software is proprietary.

In the coordination layer we find two implementations of a consensus algorithm. Chubby [8] is a distributed implementation of Paxos [42] and Zookeeper is Hadoop's re-implementation in Java. They are centralized services for maintaining configuration information, naming, providing distributed synchronization and group services. All these kinds of services are used by distributed applications.

Chubby has been used to build a Domain Name System (DNS) alternative, to build a replicated distributed log and as a primary election mechanism. Zookeeper has also been used as a bootstrap service locator and a load balancer. The main characteristics of these services are very high availability and reliability, sacrificing high performance. Their interface is similar to a distributed file system with *whole-file* reads and writes (usually metadata), advisory locks and notification mechanisms. They have solid roots in distributed systems research.

On the next level, the distributed data layer presents different kinds of data storages. A common feature in this layer is to avoid full POSIX semantic in favor of simpler ones. Furthermore, consistency is somewhat relaxed for the sake of performance.

Google File System (GFS) [27], Hadoop Distributed File System (HDFS) and Cosmos are all distributed file systems geared towards data streaming. They are not general purpose file systems. For example, in HDFS files can only be appended but not modified. They use large blocks of 64 MB or

more, which are replicated for fault tolerance.

GFS uses a master-slave architecture where the master is responsible for metadata operations while the slaves (chunkservers) are used for data. It provides loose consistency guarantees, for example a record might be appended more than once (*at least once* semantics). GFS is implemented as a user space process and stores the data chunks as lazily allocated files on a local file system. HDFS basically implements the same architecture. In both systems the master (namenode in HDFS terminology) is a single point of failure.

Cosmos [9]is Microsoft's internal file system for cloud computing. There is not much public information available about this component. It is an append-only file system optimized for streaming of petabyte scale data. Data is replicated for fault tolerance and compressed for efficiency. From this scarce information it looks like a re-implementation of Google's GFS.

Dynamo [20] is a storage service used at Amazon. It is a key-value store used for low latency access to data. It has a peer-to-peer (P2P) architecture that uses consistent hashing to spread the load and a gossip protocol to guarantee eventual consistency [68]. Amazon's shopping cart and other services (e.g. S3) are built on top of Dynamo.

The systems described above are, with the exception of Dynamo, mainly append-only and stream oriented file systems. However, it is sometimes convenient to access data in a different fashion. For example using a more elaborate data model or enabling read/write operations. For these reasons data abstractions are usually built on top of the aforementioned systems.

BigTable [10] and HBase [6] are non-relational databases. They are actually multidimensional, sparse, sorted maps, useful for semi-structured or non structured data. As in many other similar systems, access to data is provided via primary key only, but each key can have more "columns". The data are stored column-wise, which allows for a more efficient representation ("null" values are stripped out) and for better compression (data is homogeneous). The data is also partitioned in tablets and distributed over a large number of tablet servers. These systems provide efficient read/write access built on top of their respective file system (GFS and HDFS) with support for versioning, timestamps and single row transactions. Google Earth, Google Analytics and many other user-facing services use this kind of storage.

The fundamental data structure of BigTable and HBase is a SSTable. It

is the underlying file format used to store data. SSTables are designed so that a data access requires a single disk access. An SSTable is never changed. If new data is added a new SSTable is created and they are eventually *"compacted"*. The immutability of SSTable is at the core of data checkpointing and recovery routines, and resembles the design of a log structured file system.

PNUTS [16] is a similar storage service developed by Yahoo! outside the Hadoop project. PNUTS allows key-value based lookup where values may be structured as typed columns or "blobs". PNUTS users may control the consistency of the data they read using "hints" to require the latest version or a potentially stale one. Data is split into range or hash tablets by a tablet controller. This layout is soft-cached in message routers that are used for queries. The primary use of PNUTS is in web applications like Flickr.

PNUTS uses a *"per-record timeline"* consistency model which guarantees that every replica will apply operations to a single record in the same order. This is stronger than eventual consistency and relies on a guaranteed, totally-ordered-delivery message brokering service. Each record is geographically replicated around several data centers. The message broker does not guarantee in order delivery between different data centers. Thus each record has a single master copy which is updated by the broker, committed and then published to replicas. This makes sure that every update gets replayed in a canonical order at each replica. In order to reduce latency the mastering of the replica is switched according to changes in access location.

Cassandra [25] is a Facebook project (now open-sourced into Apache incubator) that aims to build a system with a BigTable-like interface on a Dynamo-style infrastructure. Voldemort [44] is an open-source non relational database built by LinkedIn. It is basically a large persistent Distributed Hash Table (DHT). This area has been very fruitful and there are many other products, but all of them share in some way the features sketched in these paragraphs: distributed architecture, non relational model, no ACID guarantees, limited relational operations (no join).

In the computation layer we find paradigms for large scale data intensive computing. They are mainly dataflow paradigms with support for automated parallelization. We can recognize the same pattern found in previous layers also here: trade off generality for performance.

MapReduce (MR) [19] is a distributed computing engine inspired by

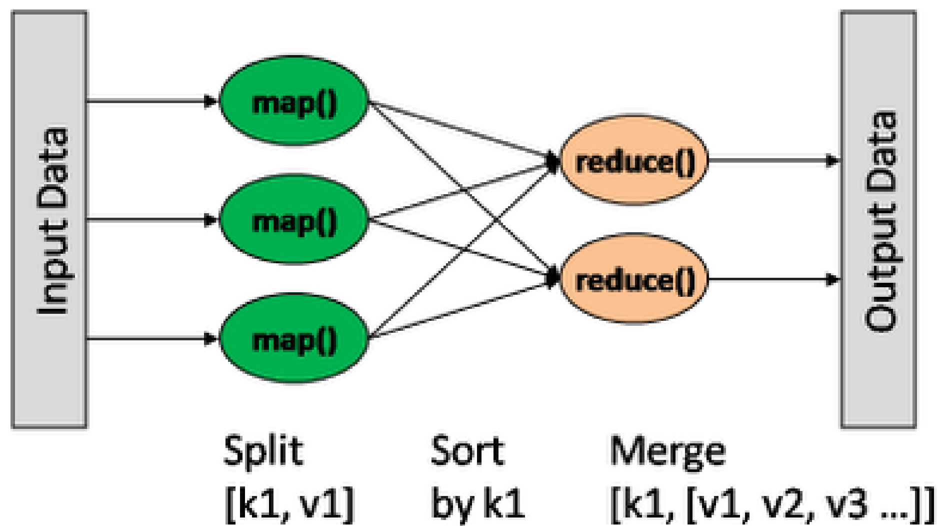

Figure 2.2: The MapReduce programming model

concepts of functional languages. More specifically, MR is based on two higher order functions: Map and Reduce. The Map function reads the input as a list of key-value pairs and applies a UDF to each pair. The result is a second list of intermediate key-value pairs. This list is sorted and grouped by key and used as input to the Reduce function. The Reduce function applies a second UDF to every intermediate key with all its associated values to produce the final result. The two phases are non overlapping as detailed in Figure 2.2.

The Map and Reduce function are purely functional and thus without side effects. This is the reason why they are easily parallelizable. Furthermore, fault tolerance is easily achieved by just re-executing the failed function. The programming interface is easy to use and does not allow any explicit control of parallelism. Even though the paradigm is not general purpose, many interesting algorithms can be implemented on it. The most paradigmatic application is building the inverted index for Google's search engine. Simplistically, the crawled and filtered web documents are read from GFS, and for every word the couple $\langle word, doc_id \rangle$ is emitted in the Map phase. The Reduce phase needs just to sort all the document identifiers associated with the same word $\langle word, [doc_id1, doc_id2, ...] \rangle$ to create the corresponding posting list.

Hadoop [6] is a Java implementation of MapReduce, which is roughly equivalent to Google's version, even though there are many reports about the performance superiority of the latter [30]. Like the original MR, it uses a master-slave architecture.

The Job Tracker is the master to which client applications submit MR jobs. It pushes tasks (job splits) out to available slave nodes on which a Task Tracker runs. The slave nodes are also HDFS chunkservers, and the Job Tracker knows which node contains the data. The Job Tracker thus strives to keep the jobs as close to the data as possible. With a rack-aware filesystem, if the work cannot be hosted on the actual node where the data resides, priority is given to nodes in the same rack. This reduces network traffic on the main backbone network for the Map phase.

Mappers write intermediate values locally on disk. Each reducer in turn pulls the data from various remote disks via HTTP. The partitions are already sorted by key by the mappers, so the reducer just merge-sorts the different partitions to bring the same keys together. These two phases are called shuffle and sort and are also the most expensive in terms of I/O operations. In the last phase the reducer can finally apply the Reduce function and write the output to HDFS.

Dryad [36] is Microsoft's alternative to MapReduce. Dryad is a distributed execution engine inspired by macro-dataflow techniques and massively data-parallel shader programming languages. Program specification is done by building a Direct Acyclic Graph (DAG) whose vertexes are operations and whose edges are data channels. The programmer specifies the DAG structure and the functions (standard or user defined) to be executed in each vertex. The system is responsible for scheduling vertex execution on a shared nothing cluster, keeping track of dependencies, deciding which channels to use (shared memory queues, TCP pipes or files), deciding the parallelism degree of operations and which vertexes to aggregate in a single process. An interesting characteristic is that the programming paradigm of Dryad is more general than MapReduce. Indeed MR workflows can be expressed in Dryad but not vice-versa.

At the last level we find high level interfaces to these computing systems. These interfaces are meant to simplify writing programs for cloud systems. Even though this task is easier than writing custom MPI code, MapReduce, Hadoop and Dryad still offer fairly low level programming interfaces, which

require the knowledge of a full programming language (C++ or Java). The interfaces in this level usually allow even non programmers to take advantage of the cloud infrastructure.

Sawzall [54] is a high level, parallel data processing scripting language built on top of MR. Its intended target are filter and aggregation scripts of record sets, akin to the AWK scripting language. The user can employ a set of predefined aggregators like counting, sampling and histograms. The script needs only to declare the desired aggregators, implement the filtering logic, which is applied record by record, and emit partial results to the aggregators. The resulting language is at the same time expressive and simple. It forces the programmer to think one record at a time, and allows the system to massively parallelize the computation without any programming effort.

Pig Latin [51] is a more sophisticated language for general data manipulation. It is an interpreted imperative language which is able to perform filtering, aggregation, joining and other complex transformations. Pig Latin statements produce Hadoop jobs that are executed in parallel. Pig Latin has a rich nested data model, fully supports UDFs and aims for the "sweet spot" between SQL and MR.

The authors claim and report witnesses that an imperative language (i.e. Pig Latin) is simpler to understand and program than a declarative one (i.e. SQL). At the same time Pig Latin uses relational-algebra-style primitives that can exploit the same kind of optimizations found in databases. Nevertheless the users retains more control over the way their programs are executed. Somewhat disappointingly, Pig Latin has no loop and conditional statements, and thus suffers from the same problem of "awkward embedding" of SQL. The authors are working on a Pig-aware version of scripting languages like Perl and Python to overcome this problem.

DryadLINQ [71] is a set of language extensions and a corresponding compiler. The compiler trasforms Language Integrated Query (LINQ) expressions into Dryad plans. LINQ is a query language integrated in Microsoft's .NET framework. It is based on lambda-expressions and thus it is easily parallelized. It is strongly typed and has both an imperative (object-oriented) and a declarative (SQL-like) version.

LINQ expressions are compiled to Dryad execution plans lazily. The expressions are decomposed into subexpression each to be run into a separate Dryad vertex. The optimizer removes redundant operations, pushes

aggregation upstream and merges multiple operators in pipelines where possible. The DAG runs in parallel and once successfully competed the data is available to the client through an iterator, akin to a normal SQL recordset.

SCOPE [9] is a declarative scripting language for Dryad based on SQL. It has the same tabular data model and implements most of its commands (select, group by, join, etc..). It also provides some more complex operations like preprocessing, reduction and combination of rows. Programmers can write UDFs and data adapters using C#. The code can be directly embedded into the scripts and the scripts themselves can be imported and parameterized. A scope script compiles to a DAG and exectutes on Dryad.

Hive [65] is a Facebook project to build a data warehousing system on top of Hadoop. It employs a tabular data model where the tables can be partitioned on columns and bucketed to different files. The files reside on HDFS, each table in a different directory. The metadata (schemas, location, statistics and auxiliary information) is kept separately in a relational DBMS called Hive-Metastore. This speeds up metadata-only operations but complicates consistency. Hive employs a declarative query language called HiveQL derived from SQL. However custom MR scripts can be easily plugged in. Queries are compiled to a DAG of MR jobs. Hive is still a young project and still lacks many features like a cost-based optimizer and columnar storage.

Cascading [14] is a Java API for creating complex and fault tolerant data processing workflows on top of Hadoop. Cascading uses "tuples and streams" as data containers and a "pipe and filters" model for defining data processes. It supports splits, joins, grouping, and sorting but also custom MR operations.

2.2 Comparison with PDBMS

How do cloud technologies compare with their direct competitors, PDBMSs? To put this comparison in the right perspective we would like to recall the *"CAP theorem"* [26, 28] stated by Fox and Brewer. The theorem says that in every system we can get at most two out of these three properties: Consistency, Availability, Partition tolerance. We can build systems with different features depending on which properties we choose. If we choose Consistency and Availability we can build a single database or a cluster database using two phase commit (which is not partition tolerant). To have Consistency

and Partition tolerance we can employ distributed locking and consensus algorithms to build a distributed database. Availability is reduced because of blocking and unavailability of minority partitions. Finally, if we are ready to give up Consistency to have a highly available and partition tolerant system, we can build systems like the DNS. In this case we need to employ techniques that deal with inconsistency like leases and conflict resolution.

Partition tolerance is required to build large scale distributed systems, where network failure is guaranteed. Cloud systems usually prefer having high availability for economic reasons, especially if they are used to power user-facing Web applications. On the other hand PDBMSs traditionally prefer consistency, as they are the evolution of traditional databases.

The choice is then between Basically Available, Soft state, Eventually consistent (BASE) [55] and Atomicity, Consistency, Isolation, Durability (ACID), two complementary design philosophies. With BASE the application requirements need to be deeply understood in order to address consistency issues that are not automatically solved by the system. The payback is increased scalability [45]. With ACID we need not worry about state or consistency issues, but the performance is constrained by total serializability.

Cloud computing and most PDBMS also target different markets. Cloud technology is used mainly for OLAP tasks, PDBMS for OLTP, but there are also niche databases targeted towards OLAP workloads.

PDBMSs employ the traditional relational model for data. Cloud systems are more flexible in this sense, providing more sophisticated data models (i.e. nested). The method to express queries is also quite different: parallel databases use SQL and select-project-join queries, cloud systems are more focused on UDFs and custom transformations. This is a result of their respective goals, which are slightly different: parallel databases *manage* large scale data, cloud systems *analyze* large scale data.

Also the interaction with the system differs. A cloud system usually does not require to define a data schema upfront. The user can avoid the burden of loading the data into rigid tables and start right away with his analysis. If the result is good the analysis can be repeated and improved with time. This approach is more agile and responds better to changes compared to the traditional database approach.

Not everyone agrees on the merits of these new cloud systems [22]. There is an ongoing fierce debate on which approach is the best [53]. Most of

Parallel Databases	Cloud Computing
multipurpose (analysis and data update, batch and interactive)	designed for scaling on large commodity clusters
high data integrity via ACID transactions	very high availability and fault tolerance
many compatible tools (loading, management, reporting, mining, data visualization)	flexible data model, no need to convert data to a normalized schema at load time
standard declarative high-level query language: SQL	full integration with familiar programming languages
automatic query optimization	full control over performance

Table 2.2: Relative advantages of PDBMS and Cloud Computing

the critiques are towards the inefficiency of the brute force approach when compared to database technology [64]. More generally, cloud computing ignores years of database research and results [21, 23].

Advocates of the cloud approach answer that not every problem involving data has to be solved with a DBMS [18]. Moreover, both SQL and the relational model are not always the best answer. Actually, this belief is held also inside the database community [62].

For these reasons many non relational databases are flourishing on the Web. The so called *"NoSQL"* movement synthesizes this trend. Sacrifice the ability to perform joins and ACID properties for the sake of performance and scalability, by going back to a simpler key-value design. Common practices in everyday database use also justify this trend. The application logic is often too complex to be expressed in relational terms (the *impedance mismatch* problem). Programmers thus commonly use *"binary blobs"* stored inside tables and access them by primary key only.

Declarative languages like SQL are as good as the optimizer beneath. Many times the programmer has to give "hints" to the optimizer in order to achieve the best query plan. This problem gets more evident as the

size of the data grows [37]. In these cases, giving the control back to the programmer is usually the best option.

Table 2.2 summarizes the relative advantages of each system. Each entry in the table can be seen as an advantage that the system has over its competitor, and thus a flaw in the latter.

On a final note, advocates for a hybrid system see value in both approaches and call for a new DBMS designed specifically for cloud systems [1].

2.3 Case Studies

Data analysis researchers have seen in cloud computing the perfect tool to run their computations on huge data sets. In the literature, there are many examples of successful applications of the cloud paradigm in different areas of data analysis. In this section, we review a subset of the most significant case studies.

Information retrieval has been the classical area of application for cloud technologies. Indexing is a representative application of this field, as the original purpose of MR was to build Google's index for web search. In order to take advantage of increased hardware and input sizes, novel adaptations of single-pass indexing have also been proposed [46].

Pairwise document similarity is a common tool for a variety of problems such as clustering and cross-document coreference resolution. When the corpus is large, MR is convenient because it allows to efficiently decompose the computation. The inner products involved in computing similarity are split into separate multiplication and summation stages, which are well matched to disk access patterns across several machines [24].

Frequent itemset mining is commonly used for query recomendation. When the data set size is huge, both the memory use and the computational cost can be prohibitively expensive. Parallel algorithms designed so that each machine executes an independent group of mining tasks are the ideal fit for cloud systems [43].

Search logs contain rich and up-to-date information about users' preferences. Many data-driven applications highly rely on online mining of search logs. A cloud based OLAP system for search logs has been devised in order to support a variety of data-driven applications [72].

Machine learning has been a fertile ground for cloud computing applica-

tions. For instance, MR has been employed for the prediction of user rating of movies, based on accumulated rating data [11]. In general, many widely used machine learning algorithms can be expressed in terms of MR [12].

Graph analysis is a common and useful task. Graphs are ubiquitous and can be used to represent a number of real world structures, e.g. networks, roads and relationships. The size of the graphs of interest has been rapidly increasing in recent years. Estimating the graph diameter for graphs with billions of nodes (e.g. the Web) is a challenging task, which can benefit from cloud computing [38].

Similarly, frequently computed metrics such as the clustering coefficient and the transitivity ratio involve the execution of a triangle counting algorithm on the graph. Approximated triangle counting algorithms that run on cloud systems can achieve impressive speedups [66].

Co-clustering is a data mining technique which allows simultaneous clustering of the rows and columns of a matrix. Co-clustering searches for submatrices of rows and columns that are interrelated. Even though powerful, co-clustering is not practical to apply on large matrices with several millions of rows and columns. A distributed co-clustering solution that uses Hadoop has been proposed to address this issue [52].

A number of different graph mining tasks can be unified via a generalization of matrix-vector multiplication. These tasks can benefit from a single highly optimized implementation of the multiplication. Starting from this observation, a library built on top of Hadoop has been proposed as a way to perform easily and efficiently large scale graph mining operations [39].

Cloud technologies have also been applied to other fields, such as scientific simulations of earthquakes [11], collaborative web filtering [49] and bioinformatics [57].

2.4 Research Directions

The systems described in the previous sections are fully functional and commonly used in production environment. Nonetheless, many research efforts have been made in order to improve and evolve them. Most of the efforts have focused on MapReduce, also because of the availability and success of Hadoop. The research in this area can be divided in three categories: computational models, programming paradigm enrichments and online analytics.

2.4.1 Computational Models

Various models for MapReduce have been proposed. Afrati and Ullman [3] describe a general I/O cost model that captures the essential features of MapReduce systems. The key assumptions of the model are:

- Files are replicated recordsets stored on a GFS-like file system with a very large block size b and can be red/written in parallel by processes;
- Processes are the conventional unit of computation but have limits on I/O: a lower limit of b (the block size) and an upper limit of s, a quantity that can represent the available main memory of processors;
- Processors, with a CPU, main memory and secondary storage, are available in infinite supply.

The authors present various algorithms for multiway join and sorting, and analyze the communication and processing costs for these examples. Differently from standard MR, an algorithm in this model is a Direct Acyclic Graph of processes, in a way similar to Dryad or Cascading. Additionally, the model assumes that keys are not delivered in sorted order to the Reduce. Because of these departures from the actual MR paradigm, the model is not appropriate to compare real-world algorithms on Hadoop. However, an implementation of the augmented model would be a good starting point for future research. At the same time, it would be interesting to examine the best algorithms for various common tasks on this model.

Karloff et al. [40] propose a novel theoretical model of computation for MapReduce. The authors formally define the Map and Reduce functions and the steps of a MR algorithm. Then they define a new algorithmic class: $\mathcal{MRC}^i$. An algorithm in this class is a finite sequence of Map and Reduce rounds with the following limitations, given an input of size n:

- each Map or Reduce is implemented by a RAM that uses sub-linear space and polynomial time w.r.t. n;
- the total space used by the output of each Map is less than quadratic in n;
- the number of rounds is $O(\log^i n)$

The model makes a number of assumptions on the underlying infrastructure to justify the definition. The number of available processors is sub-linear. This restriction guarantees that algorithms in $\mathcal{MRC}$ are practical. Each processor has a sub-linear amount of memory. Given that the Reduce phase can not begin until all the Maps are done, the intermediate results must be stored temporarily in memory. This explains the space limit on the Map output which is given by the total memory available across all the machines. The authors give examples of algorithms for graph and string problems. The result of their analysis is an algorithmic design technique for $\mathcal{MRC}$.

Lammel [41] developes an interesting functional model of MapReduce based on Haskell. The author reverse-engineered the original papers on MapReduce and Sawzall to derive an executable specification. Their analysis shows various interesting points. First, they show what are the relationships between Google's MR and the functional operators of `map` and `reduce`. The user defined mapper is the argument of a `map` combinator. The reducer is as well the argument of a `map` combinator over the intermediate data. The reducer is *typically* also an application of a `reduce` combinator, but it can perform also other operations like sorting.

Second, the authors disambiguate the type definition of MR which is very vague in the original paper. They decompose MR in three phases highlighting the shuffling and grouping operation that is usually performed silently by the system. The signatures are as follows:

$$\begin{aligned} map: &\quad Map(k_1 : v_1) &\rightarrow [(k_2, v_2)] \\ groupBy: &\quad [(k_2, v_2)] &\rightarrow Map(k2 : [v2]) \\ reduce: &\quad Map(k2 : [v2]) &\rightarrow Map(k2 : v3) \end{aligned}$$

The $Map(key : value)$ data type is commonly called associative array in PHP or dictionary in Python, basically a set of keys with associated values. The square brackets [ ] and parenthesis ( ) indicate respectively a list and a tuple. These definition are more rigorous in the way they use the terms "list" and "set".

Third they analyze the `combiner` function of MR and find that it is actually useless. If it defines a function that is logically different from the reducer there is no guarantee of correctness. Otherwise there is no need to

define two separate functions for the same functionality. However we argue that there might be practical reasons to separate them, like performing output or sorting only in the last phase.

Finally they analyze Sawzall's aggregators. They find that the aggregators identify the *ingredients* of a *reduce* (in the functional sense). They contend that the essence of a Sawzall program is to define a list homomorphism: a function to be mapped over the list elements as well as a "monoid" to be used for reduction. A monoid is a simple algebraic structure composed of a set, an associative operation and its unit. They conclude that the distribution model of Sawzall is both simpler and more general thanks to the use of monoids, and that it cannot be based directly on MR because of its multi-level aggregation strategy (machine-rack-global).

2.4.2 Programming Paradigm Enrichments

A number of extensions to the base MR system have been developed. These works differentiate from the models above because their result is a working system. Most of these works focus on extending MR towards the database area.

Yang et al. [70] propose an extension to MR in order to simplify the implementation of relational operators. Namely, the implementation of join, complex, multi-table select and set operations. The normal MR workflow is extended with a third final Merge phase. This function takes in input two different key-value pair lists and outputs a third key-value pair list. The model assumes that the Reduce produces key-value pairs that are then fed to the Merge. The signature of the functions are as follows:

$$
\begin{aligned}
map: &\quad (k_1, v_1)_\alpha &\rightarrow [(k_2, v_2)]_\alpha \\
reduce: &\quad (k_2, [v_2])_\alpha &\rightarrow (k_2, [v_3])_\alpha \\
merge: &\quad ((k_2, [v_3])_\alpha, (k_3, [v_4])_\beta) &\rightarrow [(k_4, v_5)]_\gamma
\end{aligned}
$$

where α, β, γ represent data lineages, k is a key and v is a value. The lineage is used to distinguish the source of the data, a necessary feature for joins. The implementation of the Merge phase is quite complex, so we will present only some key features. A Merge is a UDF like Map and Reduce. To

determine the data sources for the Merge, the user needs to define a partition selector module. After being selected, the two input lists are accessed via two logical iterators. The user has to describe the iteration pattern implementing a move function that is called after every merge. Other optional functions may also be defined for preprocessing purposes. Finally, the authors describe how to implement common join algorithms on their framework and provide some insight on the communication costs.

The framework described in the paper, even though efficient, is quite complex for the user. To implement a single join algorithm the programmer needs to write up to five different functions for the Merge phase only. Moreover, the system exposes many internal details that pollute the clean functional interface of MR. To be truly usable, the framework should provide a wrapper library with common algorithms already implemented. Even better, this framework could be integrated into high level interfaces like Pig or Hive for a seamless and efficient experience.

Another attempt to integrate database feature into MR is proposed by Abouzeid et al. [2]. Their HadoopDB is an architectural hybrid between Hadoop and a DBMS (PostgreSQL). The goal is to achieve the flexibility and fault tolerance of Hadoop and the performance and query interface of a DBMS. The architecture of HadoopDB comprises single instances of PostgreSQL as the data storage (instead of HDFS), Hadoop as the communication layer, a separate catalog for metadata and a query rewriter (SMS Planner) that is a modified version of Hive. The SMS Planner converts HiveQL queries into MR jobs that read data from the PostgreSQL instances. The core idea is to leverage the I/O efficiency given by indexes in the database to reduce the amount of data that needs to be read from disk.

The system is then tested with the same benchmarks used by Pavlo et al. [53]. The results show a performance improvement over Hadoop in most tasks. Furthermore HadoopDB is relatively more fault tolerant than a common PDBMS, in the sense that crashes do not trigger expensive query restarts. HadoopDB is an interesting attempt to join distributed systems and databases. Nonetheless, we argue that reusing principles of database research rather than its artifacts should be the preferred method.

2.4.3 Online Analytics

A different research direction aims to enable online analytics on large scale data. This gives substantial competitive advantages in adapting to changes, and the ability to process stream data. Systems like MR are essentially batch systems, while BigTable provides low latency but is just a lookup table. To date there is no system that provides low latency general querying.

Olston et al. [50] describe an approach to interactive analysis of web-scale data. The scenario entails interactive processing of a single negotiated query over static data. Their idea is to split the analysis phase in two: an offline and an online phase. In the former, the user submits a template to the system. This template is basically a parameterized query plan. The system examines the template, optimizes it and computes all the necessary auxiliary data structures to speed up query evaluation. In the process it may negotiate restrictions on the template. In the online phase the user instantiates the template by binding the parameters into the template. The system computes the final answer using the auxiliary structures in "real-time".

The offline phase works in a batch environment with large computing resources (i.e. a MR-style system). The online phase may alternatively be run on a single workstation, if enough data reduction happens in the offline phase. The authors focus on parameterized filters as examples and show how to optimize the query plan for the 2-phase split. They try to push all the parameter-free operations in the offline phase, and create indexes or materialized views at phase junctures. Various types of indexing approaches are then evaluated, together with other background optimization techniques. This work looks promising and there are still questions to be answered like what kind of templates are amenable to interactivity, how to help the user building the query template and how to introduce approximate query processing techniques (e.g. online aggregation [34]).

On this last issue we find a work by Condie et al. [15]. They modify Hadoop in order to pipeline data between operators, support online aggregation and continuous queries. They name their system Hadoop Online Prototype (HOP). In HOP a downstream dataflow element can begin consuming data before a producer element has finished execution. Hence they can generate and refine an approximate answer using online aggregation [34]. Pipelining also enables to push data as it comes inside a running job, which

in turn enables stream processing and continuous queries.

To implement pipelining the simple pull interface between Reduce and Map has to be modified into a hybrid push/pull interface. The mappers push data to reducers in order to speed up the shuffle and sort phase. The pushed data is treated ad tentative to retain fault tolerance: if one of the entities involved in the transfer fails the data is simply discarded. For this reason mappers also write the data to disk as in normal MR, and the pull interface is retained.

Online aggregation is supported by simply applying the reduce function to all the pipelined data received so far. This aggregation can be triggered on specific events (e.g. 50% of the mappers completed). The result is a snapshot of the computation and can be accessed via HDFS. Continuous MR jobs can be run using both pipelining and aggregation on stream data. The reduce function is invoked at intervals defined by time or number of inputs. To retain fault tolerance the system retains a rolling suffix of the history of Map output.

HOP can also pipeline and do online aggregation between jobs, but this is defined "problematic" by the authors. For pipelining, they say that "the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped". This is actually not true because the Map function does not need the full input before starting, but operates record-wise. For online aggregation the problem is that the output of Reduce on 50% of the input is not directly related to the final output. This means that every snapshot must be recomputed from scratch, using considerable computing resources. This problem can be alleviated for Reduce functions that are declared to be distributive, associative or algebraic aggregate. This line of research is extremely promising and some of the shortcomings of this work are directly addressable. For example we can enable overlapping the computation for a Reduce-to-Map pipelining. Moreover, we could use semantic clues to specify properties about the functions or the input. This could easily be implemented using Java annotations.

Chapter 3

Research Plan

In the previous Chapters we have shown the relevance of data intensive cloud computing and we have highlighted some of the currently pursued research directions. This research field is new and rapidly evolving. As a result, the initial efforts are not organic.

This thesis aims to fill this gap by providing a coherent research agenda in this field. We will address the emerging issues in employing cloud computing for large scale data analysis. We will build our research upon a large base of literature in data analysis, database and parallel systems research. As a result we expect to significantly improve the state of the art in the field, and possibly evolve the current computational paradigm.

3.1 Research Problem

Cloud computing is an emerging technology in the data analysis field, used to capitalize on very large data sets. "There is no data like more data" is a famous motto that epitomizes the opportunity to extract significant information by exploiting huge volumes of data.

Information represents a competitive advantage for companies operating in the information society, an advantage that is all the greater the sooner it is achieved. In the limit, online or real-time analytics will become an invaluable support for decision making.

To date, cloud computing technologies have been successfully employed for batch processing. Adapting these systems to online analytics is a challenging problem, mainly because of the fundamental design choices that favor throughput over latency [47].

Another driver for change is the shift towards higher level languages. These languages often incorporate traditional relational operators or build upon SQL. This means that efficient implementation of relational operators is a key problem. Nevertheless, operators like join are currently an *"Achille's heel"* of cloud systems [53].

However, we argue that this limit in the current unstructured and brute-force approach is not fundamental. This approach is just the easiest to start with, but not necessarily the best one. As research progresses more sophisticated approaches will be available. High level languages will hide their complexity and allow their seamless integration.

Many data analysis algorithms spanning different application areas have been proposed for cloud systems. So far, speedup and scalability results have been encouraging.

Still, it is not clear in the research community which problems are a good match for cloud systems. More importantly, the ingredients and recipes for building a successful algorithm are still hidden. A library of *"algorithmic design patterns"* would enable faster and easier adoption of cloud systems.

Even though there have been attempts to address these issues, current research efforts in this area have been mostly "one shot". The results are promising but their scope and generality is often limited to the specific problem at hand. These efforts also lack a coherent vision and a systematic approach. For this reason they are also difficult to compare and integrate with each other.

This "high entropy" level is typical of new research areas whose agenda is not yet clear. One of the main shortcomings is that the available models for cloud systems are at an embryonal stage of development. This hampers the comparison of algorithms and the evaluation of their properties from a theoretical point of view.

Furthermore, up to now industry has been leading the research in cloud computing. Academia has now to follow and "catch-up". Nonetheless this fact also has positive implications. The research results are easily adapted to industrial environments because the gap between production and research systems is small.

The problems described up to this point are a result of the early stage of development in which this area of research currently is. For this reason we claim that further effort is needed to investigate this area.

3.2 Thesis Proposal

The goal of this thesis will be to provide a coherent framework for research in the field of large scale data analysis on cloud computing systems. More specifically we aim to provide a framework that tries to answer these questions:

- Which are the best algorithms for large scale data analysis?
- How to support these algorithms on cloud computing systems?
- Is it possible to carry out online data analysis on such systems?

We will approach these problems according to the following methodology.

We will study the most common data analysis algorithms. A wide spectrum of analytical algorithms are being applied to large amounts of data. These algorithms range from typical text processing tasks, such as indexing, to pattern extraction, machine learning and graph mining. Such algorithms encompass different computational patterns, which might not properly match the popular paradigms available today. Therefore, the first step will be to define an analytical workload that is representative of their new unsatisfied requirements.

This analysis will allow us to identify the weaknesses of existing systems, and to design a roadmap of contributions to the state of the art. To this end, we propose to selectively and coherently integrate into cloud systems *principles* from database research, as opposed to its *artifacts*. For this reason we will develop a comprehensive approach to integration. We will pick relevant principles from the database literature, which we can apply to satisfy the requirements. We will re-elaborate these principles in new forms and adapt them to the new context. If unable to find suitable results in the literature, we will formulate novel original approaches.

We plan to evaluate the quality of our proposed contributions both from an experimental and theoretical point of view. We will propose computational models for cloud computing that take into account the features of the system and its possible extensions. We will use these models to analyze our proposed solutions, and also to provide a common reference ground for the comparison of other solutions, algorithms and paradigms.

In addressing interactive and online analytics, we will evaluate different approaches to query answering, like sampling and approximate answers. In

this respect, providing confidence intervals for approximate answers is an important feature, missing from current systems, that we plan to explore. We will also try to address the issue of skewed distribution of data, which often leads to *straggler* tasks that slow down the entire job. Furthermore, we will try to leverage properties of the input or of the computation in order to perform optimizations and reduce the job turnaround time.

We will generally use MapReduce as the target paradigm, because of its widespread adoption by the scientific community and of the open-source availability of its implementations (e.g. Hadoop). Nevertheless, we will strive to propose principles and approaches that are as general as possible. We hope that other cloud systems and the next generation of massively parallel systems will also be able to benefit from our work.

We expect to produce extensions to MapReduce and to the MapReduce paradigm in order to support online analytics and other common computation patterns. This will conceivably lead to a new paradigm, an evolution beyond the original MapReduce. In our activity we will give attention to the integration with high level languages, as this shields the user from the evolution of the underlying infrastructure.

As a byproduct of our research, we will develop a toolbox of algorithms for large scale data analysis. This toolbox will exploit our proposed extensions of the MapReduce paradigm, thus validating the value and soundness of our research efforts.

To conclude, the expected contributions of this thesis are:

- To build and evaluate a toolbox of algorithms for large scale data analysis on cloud computing systems
- To design extensions to existing programming paradigms in order to support these algorithms
- To develop methods to speed up these algorithms in order to support online processing

Our research is an effort to bring together the distributed systems and database communities. Cross-contamination of research areas is a great stimulus for scientific advancement. Our research is a step toward the creation of a common ground on which these communities will be able to communicate and thrive together.

Bibliography

[1] D. J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *IEEE Data Engineering Bulletin*, vol. 32, no. 1, pp. 3–12, March 2009.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," in *Proceedings of the VLDB Endowment*, vol. 2, no. 1, August 2009, pp. 922–933.

[3] F. Afrati and J. Ullman, "A New Computation Model for Rack-Based Computing," 2009, submitted to PODS 2010: Symposium on Principles of Database Systems.

[4] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference.* ACM, April 1967, pp. 483–485.

[5] C. Anderson, "The Petabyte Age: Because more isn't just more—more is different," *Wired*, vol. 16, no. 07, July 2008.

[6] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. (2005) Hadoop: A framework for running applications on large clusters built of commodity hardware. [Online]. Available: http://hadoop.apache.org

[7] G. E. P. Box and N. R. Draper, *Empirical model-building and response surface.* John Wiley & Sons, Inc., 1986, p. 424.

[8] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, November 2006, pp. 335–350.

[9] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and efficient parallel processing of massive data sets," in *Proceedings of the VLDB Endowment*, vol. 1, no. 2. VLDB Endowment, August 2008, pp. 1265–1276.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "BigTable: A distributed storage system for structured data," in *OSDI '06: Proceedings of the 7th Symposium on Operating systems Design and Implementation*, November 2006, pp. 205–218.

[11] S. Chen and S. Schlosser, "Map-Reduce Meets Wider Varieties of Applications," Intel Research, Pittsburgh, Tech. Rep. IRP-TR-08-05, May 2008. [Online]. Available: http://www.pittsburgh.intel-research.net/~chensm/papers/IRP-TR-08-05.pdf

[12] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," in *Advances in Neural Information Processing Systems.* MIT Press, 2007, vol. 19, pp. 281–288.

[13] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[14] Concurrent. (2008, January) Cascading: An API for executing workflows on Hadoop. [Online]. Available: http://www.cascading.org

[15] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," University of California, Berkeley, Tech. Rep. UCB/EECS-2009-136, October 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-136.html

[16] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," in *Proceedings of the VLDB Endowment*, vol. 1, no. 2. VLDB Endowment, August 2008, pp. 1277–1288.

[17] M. Creeger, "Cloud Computing: An Overview," *Queue*, vol. 7, no. 5, pp. 3–4, June 2009.

[18] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, January 2010.

[19] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI '04: Proceedings of the 6th Symposium on Opearting Systems Design and Implementation.* USENIX Association, December 2004, pp. 137–150.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP '07: Proceedings of 21st ACM SIGOPS symposium on Operating systems principles.* ACM, October 2007, pp. 205–220.

[21] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, June 1992.

[22] D. DeWitt and M. Stonebraker. (2008, January) MapReduce: A major step backwards. [Online]. Available: http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards http://databasecolumn.vertica.com/database-innovation/mapreduce-ii

[23] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The Gamma database machine project," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 44–62, March 1990.

[24] T. Elsayed, J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with MapReduce," in *HLT '08: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies.* ACL, June 2008, pp. 265–268.

[25] Facebook. (2008, August) Cassandra: A highly scalable, eventually consistent, distributed, structured key-value store. [Online]. Available: http://incubator.apache.org/cassandra

[26] A. Fox and E. A. Brewer, "Harvest, Yield, and Scalable Tolerant Systems," in *HOTOS '99: Proceedings of the the 7th Workshop on Hot*

Topics in Operating Systems. IEEE Computer Society, March 1999, pp. 174–178.

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in *SOSP '03: Proceedings of the 19th ACM symposium on Operating systems principles.* ACM, October 2003, pp. 29–43.

[28] S. Gilbert and N. Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, June 2002.

[29] J. Ginsberg, M. Mohebbi, R. Patel, L. Brammer, M. Smolinski, and L. Brilliant, “Detecting influenza epidemics using search engine query data,” *Nature*, vol. 457, no. 7232, pp. 1012–1014, 2008.

[30] Google Blog. (2008, November) Sorting 1PB with MapReduce. [Online]. Available: http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html

[31] J. Gray and D. Patterson, “A Conversation with Jim Gray,” *Queue*, vol. 1, no. 4, pp. 8–17, July 2003.

[32] J. L. Gustafson, “Reevaluating Amdahl's law,” *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[33] J. M. Hellerstein, “Programming a Parallel Future,” University of California, Berkeley, Tech. Rep. UCB/EECS-2008-144, November 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-144.html

[34] J. M. Hellerstein, P. J. Haas, and H. J. Wang, “Online aggregation,” in *SIGMOD '97: Proceedings of the 23rd ACM SIGMOD international conference on Management of data.* ACM, May 1997, pp. 171–182.

[35] M. D. Hill and M. R. Marty, “Amdahl's Law in the Multicore Era,” *IEEE Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[36] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems.* ACM, March 2007, pp. 59–72.

[37] A. Jacobs, "The pathologies of Big Data," *Communications of the ACM*, vol. 52, no. 8, pp. 36–44, August 2009.

[38] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec, "HADI: Fast diameter estimation and mining in massive graphs with Hadoop," Carnegie Mellon University, Pittsburgh, Tech. Rep. CMU-ML-08-117, December 2008. [Online]. Available: http://reports-archive.adm.cs.cmu.edu/anon/anon/home/ftp/ml2008/CMU-ML-08-117.pdf

[39] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *ICDM '09: Proceedings of the 9th IEEE International Conference on Data Mining*. IEEE Computer Society, December 2009, pp. 229–238.

[40] H. Karloff, S. Suri, and S. Vassilvitskii, "A Model of Computation for MapReduce," in *SODA '10: Symposium on Discrete Algorithms*. ACM, January 2010.

[41] R. Lammel, "Google's MapReduce programming model – Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, January 2008.

[42] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[43] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "PFP: Parallel FP-Growth for Query Recommendation," in *RecSys '08: Proceedings of the 2nd ACM conference on Recommender Systems*. ACM, October 2008, pp. 107–114.

[44] LinkedIn. (2009, June) Project Voldemort: A distributed database. [Online]. Available: http://project-voldemort.com/

[45] F. Marinescu and C. Humble. (2008, March) Trading Consistency for Scalability in Distributed Architectures. [Online]. Available: http://www.infoq.com/news/2008/03/ebaybase

[46] R. M. C. McCreadie, C. Macdonald, and I. Ounis, "On single-pass indexing with MapReduce," in *SIGIR '09: Proceedings of the 32nd*

international ACM SIGIR conference on Research and development in Information Retrieval. ACM, July 2009, pp. 742–743.

[47] M. K. McKusick and S. Quinlan, "GFS: Evolution on Fast-forward," *Queue*, vol. 7, no. 7, pp. 10–20, August 2009.

[48] P. Mell and T. Grance. (2009, October) Definition of Cloud Computing. National Institute of Standards and Technology (NIST). [Online]. Available: http://csrc.nist.gov/groups/SNS/cloud-computing

[49] M. G. Noll and C. Meinel, "Building a Scalable Collaborative Web Filter with Free and Open Source Software," in *SITIS '08: Proceedings of the 4th IEEE International Conference on Signal Image Technology and Internet Based Systems.* IEEE Computer Society, November 2008, pp. 563–571.

[50] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed, "Interactive Analysis of Web-Scale Data," in *CIDR '09: Proceedings of the 4th Conference on Innovative Data Systems Research*, January 2009.

[51] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *SIGMOD '08: Proceedings of the 34th ACM SIGMOD international conference on Management of data.* ACM, June 2008, pp. 1099–1110.

[52] S. Papadimitriou and J. Sun, "DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining," in *ICDM '08: Proceedings of the 8th IEEE International Conference on Data Mining.* IEEE Computer Society, December 2008, pp. 512–521.

[53] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker, "A Comparison of Approaches to Large-Scale Data Analysis," *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pp. 165–178, June 2009.

[54] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, October 2005.

[55] D. Pritchett, "BASE: An ACID Alternative," *Queue*, vol. 6, no. 3, pp. 48–55, May 2008.

[56] J. Rowley, "The wisdom hierarchy: representations of the DIKW hierarchy," *Journal of Information Science*, vol. 33, no. 2, pp. 163–180, April 2007.

[57] M. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, p. 1363, June 2009.

[58] Y. Shi, "Reevaluating Amdahl's Law and Gustafson's Law," October 1996, Computer Sciences Department, Temple University. [Online]. Available: http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html

[59] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. Springer, 2008, p. 31.

[60] M. Stonebraker and U. Cetintemel, "One Size Fits All: An Idea Whose Time Has Come and Gone," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering.* IEEE Computer Society, April 2005, pp. 2–11.

[61] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik, "One size fits all? Part 2: Benchmarking results," in *CIDR '07: Proceedings of the 3rd Conference on Innovative Data Systems Research*, January 2007.

[62] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The End of an Architectural Era (It's Time for a Complete Rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on Very Large Data Bases.* ACM, September 2007, pp. 1150–1160.

[63] M. Stonebraker, "The Case for Shared Nothing," *IEEE Database Engineering Bulletin*, vol. 9, no. 1, pp. 4–9, March 1986.

[64] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and Parallel DBMSs: Friends

or Foes?" *Communications of the ACM*, vol. 53, no. 1, pp. 64–71, January 2010.

[65] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," in *Proceedings of the VLDB Endowment*, vol. 2, no. 2. VLDB Endowment, August 2009, pp. 1626–1629.

[66] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "DOULION: Counting Triangles in Massive Graphs with a Coin," in *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge Discovery and Data mining.* ACM, April 2009, pp. 837–846.

[67] J. Vitter and E. Shriver, "Algorithms for parallel memory, I: Two-level memories," *Algorithmica*, vol. 12, no. 2, pp. 110–147, September 1994.

[68] W. Vogels, "Eventually Consistent," *Queue*, vol. 6, no. 6, pp. 14–19, December 2008.

[69] Wikipedia. (2010, January) Database Management System. [Online]. Available: http://en.wikipedia.org/wiki/Database_management_system

[70] H. Yang, A. Dasdan, R. Hsiao, and D. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *SIGMOD '07: Proceedings of the 33rd ACM SIGMOD international conference on Management of data.* ACM, June 2007, pp. 1029–1040.

[71] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI '08: Proceedings of the 8th Symposium on Operating System Design and Implementation*, December 2008.

[72] B. Zhou, D. Jiang, J. Pei, and H. Li, "OLAP on search logs: an infrastructure supporting data-driven applications in search engines," in *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge Discovery and Data mining.* ACM, June 2009, pp. 1395–1404.