A Survey of Large Scale Data Management Approaches in Cloud Environments

Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari

***Abstract*—In the last two decades, the continuous increase of computational power has produced an overwhelming flow of data. Moreover, the recent advances in Web technology has made it easy for any user to provide and consume content of any form. This has called for a paradigm shift in the computing architecture and large scale data processing mechanisms. Cloud computing is associated with a new paradigm for the provision of computing infrastructure. This paradigm shifts the location of this infrastructure to the network to reduce the costs associated with the management of hardware and software resources.**

This paper gives a comprehensive survey of numerous approaches and mechanisms of deploying data-intensive applications in the cloud which are gaining a lot of momentum in both research and industrial communities. We analyze the various design decisions of each approach and its suitability to support certain classes of applications and end-users. A discussion of some open issues and future challenges pertaining to scalability, consistency, economical processing of large scale data on the cloud is provided. We highlight the characteristics of the best candidate classes of applications that can be deployed in the cloud.

***Index Terms*—Cloud Computing, Cloud Data Storage, MapReduce, NoSQL.**

I. Introduction

IN THE LAST two decades, the continuous increase of computational power has produced an overwhelming flow of data which called for a paradigm shift in the computing architecture and large scale data processing mechanisms. In a speech given just a few weeks before he was lost at sea off the California coast in January 2007, Jim Gray, a database software pioneer and a Microsoft researcher, called the shift a "*fourth paradigm*" [1]. The first three paradigms were experimental, theoretical and, more recently, computational science. Gray argued that the only way to cope with this paradigm is to develop a new generation of computing tools to manage, visualize and analyze the data flood. In general, the current computer architectures are increasingly imbalanced where the latency gap between multi-core CPUs and mechanical hard disks is growing every year which makes the challenges of data-intensive computing harder to overcome [2]. However, this gap is not the single problem to be addressed. Recent applications that manipulate TeraBytes and PetaBytes of distributed data [3] need to access networked environments with guaranteed Quality of Service (QoS). If the network mechanisms are neglected, the applications will just have access to a best effort service, which is not enough to their requirements [4]. Hence, there is a crucial need for a systematic and generic approach to tackle these problems with an architecture that can also scale into the foreseeable future. In response, Gray argued that the new trend should focus on supporting cheaper clusters of computers to manage and process all this data instead of focusing on having the biggest and fastest single computer. Unlike to previous clusters confined in local and dedicated networks, in the current clusters connected to the Internet, the interconnection technologies play an important role, since these clusters need to work in parallel, independent of their distances, to manipulate the data sets required by the applications [5].

Currently, data set sizes for applications are growing at incredible pace. In fact, the advances in sensor technology, the increases in available bandwidth and the popularity of handheld devices that can be connected to the Internet have created an environment where even small scale applications need to store large data sets. A terabyte of data, once an unheard-of amount of information, is now commonplace. For example, modern high-energy physics experiments, such as *DZero*[1], typically generate more than one TeraByte of data per day. With datasets growing beyond a few hundreds of terabytes, we have no off-the-shelf solutions that they can readily use to manage and analyze these data [1]. Thus, significant human and material resources were allocated to support these data-intensive operations which lead to high storage and management costs.

Additionally, the recent advances in Web technology have made it easy for any user to provide and consume content of any form. For example, building a personal Web page (e.g. Google Sites[2]), starting a blog (e.g. WordPress[3], Blogger[4], LiveJournal[5]) and making both searchable for the public have now become a commodity. Therefore, one of the main goals of the next wave is to facilitate the job of implementing every application as a *distributed*, *scalable* and *widely-accessible* service on the Web. For example, it has been recently reported that the famous social network Website, Facebook[6], serves 570 billion page views per month, stores 3 billion new photos every month, manages 25 billion pieces of content (e.g. status

Manuscript received 28 June 2010; revised 2 December 2010.

S. Sakr is with University of New South Wales and National ICT Australia (NICTA), Sydney, Australia (e-mail: sherif.sakr@nicta.com.au).

A. Liu is with University of New South Wales and National ICT Australia (NICTA), Sydney, Australia (e-mail: anna.liu@nicta.com.au).

D. M. Batista is with the Institute of Computing, State University of Campinas, Campinas, Brazil (e-mail: batista@ic.unicamp.br).

M. Alomari is with School of Information Technologies, University of Sydney, Sydney, Australia (e-mail: miomari@it.usyd.edu.au).

Digital Object Identifier 10.1109/SURV.2011.032211.00087

[1]http://www-d0.fnal.gov/
[2]http://sites.google.com/
[3]http://wordpress.org/
[4]http://www.blogger.com/
[5]http://www.livejournal.com/
[6]http://www.facebook.com/

updates, comments) every month and runs its services over 30K servers. Although services such as Facebook, Flickr[7], YouTube[8], and Linkedin[9] are currently leading this approach, it becomes an ultimate goal to make it easy for everybody to achieve these goals with the minimum amount of effort in terms of software, CPU and network.

Recently, there has been a great deal of hype about cloud computing. Cloud computing is on the top of Gartner's list of the ten most disruptive technologies of the next years [6]. Cloud computing is associated with a new paradigm for the provision of computing infrastructure. This paradigm shifts the location of this infrastructure to the network to reduce the costs associated with the management of hardware and software resources. Hence, businesses and users become able to access application services from anywhere in the world on demand. Therefore, it represents the long-held dream of envisioning computing as a utility [7] where the economy of scale principles help to drive the cost of computing infrastructure effectively down. Big players such as Amazon, Google, IBM, Microsoft and Sun Microsystems have begun to establish new data centers for hosting Cloud computing applications in various locations around the world to provide redundancy and ensure reliability in case of site failures.

In fact, there has been much discussion in industry and academia about what cloud computing actually means [8], [9], [10]. The US National Institute of Standards and Technology (NIST) has developed a working definition that covers the commonly agreed aspects of cloud computing. The NIST working definition [11] is widely referenced in US government documents and projects. It summarizes cloud computing as: "*a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*". According to this definition, cloud computing has the following essential characteristics:

1) *On-demand self-service*. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.
2) *Broad network access*. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs).
3) *Resource pooling*. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, virtual networks and virtual machines.
4) *Rapid elasticity*. Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time.
5) *Measured Service*. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service.

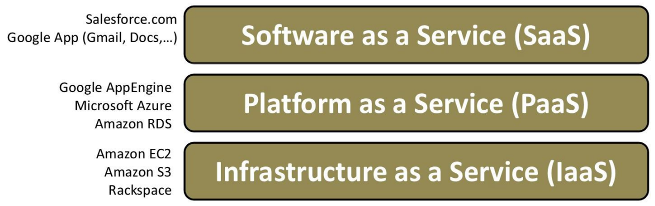

Salesforce.com
Google App (Gmail, Docs,...)
Software as a Service (SaaS)
Google AppEngine
Microsoft Azure
Amazon RDS
Platform as a Service (PaaS)
Amazon EC2
Amazon S3
Rackspace
Infrastructure as a Service (IaaS)

Fig. 1. Models of Cloud Services.

The definition also specifies the following levels of abstraction (Figure 1) which are referred to as *service models*:

1) *Infrastructure as a Service (IaaS)*: Provision resources such as servers (often in the form of virtual machines), network bandwidth, storage, and related tools necessary to build an application environment from scratch (e.g. Amazon EC2[10]).
2) *Platform as a Service (PaaS)*: Provides a higher-level environment where developers can write customized applications (e.g. Microsoft Azure[11], Google AppEngine[12]). The maintenance, load-balancing and scale-out of the platform are done by the service provider and the developer can concentrate on the main functionalities of his application.
3) *Software as a Service (SaaS)*: Refers to special-purpose software made available through the Internet (e.g. SalesForce[13]). Therefore, it does not require each end-user to manually download, install, configure, run or use the software applications on their own computing environments.

The definition also describes the following types of *cloud deployment model* depending on who owns and uses them:

1) *Private cloud*. A cloud that is used exclusively by one organization. It may be managed by the organization or a third party and may exist on premise or off premise. A private cloud offers the highest degree of control over performance, reliability and security. However, they are often criticized for being similar to traditional proprietary server farms and do not provide benefits such as no up-front capital costs.
2) *Community cloud*. The cloud infrastructure is shared by

[7]http://www.flickr.com/
[8]http://www.youtube.com/
[9]http://www.linkedin.com/

[10]http://aws.amazon.com/ec2/
[11]http://www.microsoft.com/windowsazure/
[12]http://code.google.com/appengine/
[13]http://www.salesforce.com/

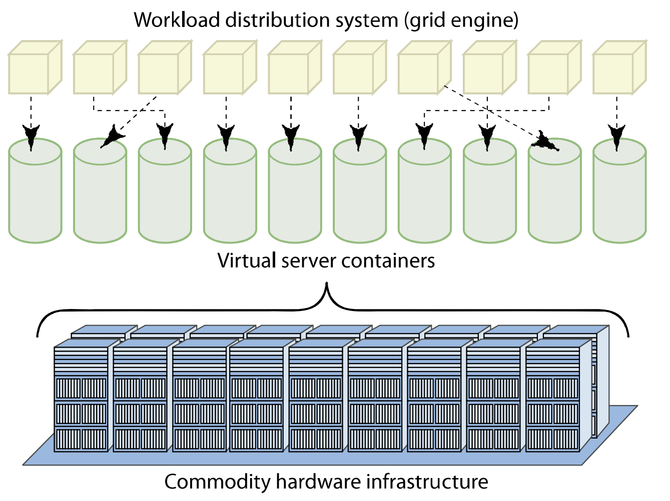

Workload distribution system (grid engine)
Virtual server containers
Commodity hardware infrastructure

Fig. 2. Exploitation of Virtualization Technology in The Architecture of Cloud Computing [10].

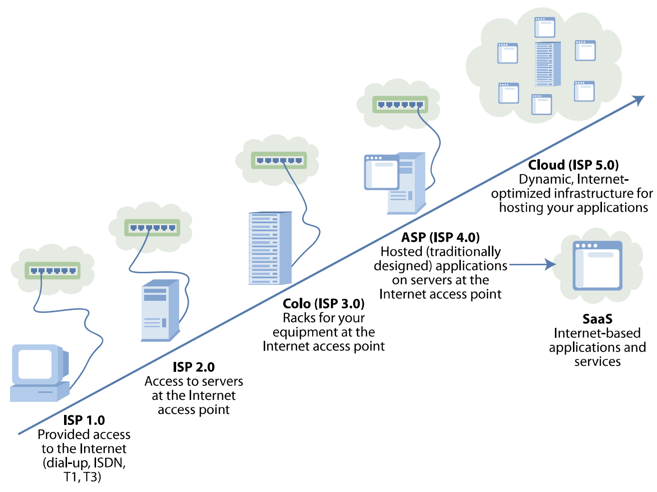

ISP 1.0
Provided access to the Internet (dial-up, ISDN, T1, T3)
ISP 2.0
Access to servers at the Internet access point
Colo (ISP 3.0)
Racks for your equipment at the Internet access point
ASP (ISP 4.0)
Hosted (traditionally designed) applications on servers at the Internet access point
Cloud (ISP 5.0)
Dynamic, Internet-optimized infrastructure for hosting your applications
SaaS
Internet-based applications and services

Fig. 3. The Evolution Towards Cloud Computing in Hosting Software Applications [10].

several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations).

3) *Public cloud*. The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services (e.g. Amazon, Google, Microsoft). Since customer requirements of cloud services are varying, service providers have to ensure that they can be flexible in their service delivery. Therefore, the quality of the provided services is specified using Service Level Agreement (SLA) which represents a contract between a provider and a consumer that specifies consumer requirements and the provider's commitment to them. Typically an SLA includes items such as uptime, privacy, security and backup procedures. In practice, Public clouds offer several key benefits to service consumers such as: including no initial capital investment on infrastructure and shifting of risks to infrastructure providers. However, public clouds lack fine-grained control over data, network and security settings, which may hamper their effectiveness in many business scenarios.

4) *Hybrid cloud*. The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds). In particular, cloud bursting is a technique used by hybrid clouds to provide additional resources to private clouds on an as-needed basis. If the private cloud has the processing power to handle its workloads, the hybrid cloud is not used. When workloads exceed the private cloud's capacity, the hybrid cloud automatically allocates additional resources to the private cloud. Therefore, Hybrid clouds offer more flexibility than both public and private clouds. Specifically, they provide tighter control and security over application data compared to public clouds, while still facilitating on-demand service expansion and contraction. On the down side, designing a hybrid cloud requires carefully determining the best split between public and private cloud components.

In principle, the concept of renting computing power goes back decades to the days when companies would share space on a single mainframe with big spinning tape drives and it has been envisioned that computing facilities will be provided to the general public like a utility [12]. Recently, the technology industry has matured to the point where there is now an emerging mass market for this rental model. Hence, cloud computing is not a revolutionary new development. However, it is an evolution that has taken place over several decades. Figure 3 illustrates the evolution towards cloud computing in hosting software applications. In fact, cloud computing is often compared to the following technologies, each of which shares certain aspects with cloud computing [13]:

- *Virtualization*: is a technology that abstracts away the details of physical hardware and provides virtualized resources for high-level applications [14], [15], [16]. A virtualized server is commonly called a virtual machine (VM). VMs allow both the isolation of applications from the underlying hardware and other VMs. It also allows the customization of the platform to suit the needs of the end-user. Providers can expose applications running within VMs, or provide access to VMs themselves as a service thereby allowing consumers to install their own applications. Therefore, virtualization forms the foundation of cloud computing, as it provides the capability of pooling computing resources from clusters of servers and dynamically assigning or reassigning virtual resources to applications on-demand. While convenient, the use of VMs gives rise to further challenges such as the intelligent allocation of physical resources for managing competing resource demands of the users. Figure 2 illustrates a sample exploitation of virtualization technology in the cloud computing environments. VMs are the basis to create specific virtual networks, which can be necessary for certain applications executed on the clouds. In this case, the VMs are virtual network routers instantiated over real machines of the cloud and virtual links are created between the routers (one virtual link can use resources of several real links and routers of the cloud).

- *Grid Computing*: is a distributed computing paradigm that coordinates networked resources to achieve a common computational objective. The development of Grid computing was originally driven by scientific applications which are usually computation-intensive, but applications requiring transfer and manipulation of a massive quantity of data also took advantage of the grids [17], [18], [19]. Cloud computing is similar to Grid computing in that it also employs distributed resources to achieve application-level objectives. However, cloud computing takes one step further by leveraging virtualization technologies at multiple levels (hardware and application platform) to realize resource sharing and dynamic resource provisioning. Foster et al. [20] have detailed a comparison between the grid computing and cloud computing technologies.
- *Utility Computing*: represents the model of provisioning computing resources as a metered services similar to those provided by traditional public utility company. In particular, it allows providing resources on-demand and charging customers based on usage rather than a flat rate. Cloud computing can be perceived as a realization of utility computing. It adopts a utility-based pricing scheme entirely for economic reasons. With on-demand resource provisioning and utility-based pricing, service providers can maximize resource utilization and minimize their operating costs. Since users must be able to access the services anywhere, the network is essential to the effective implementation of utility computing, as well as in cloud computing.
- *Autonomic Computing*: aims at building computing systems capable of self-management, i.e. reacting to internal and external observations without human intervention. The goal of autonomic computing is to overcome the management complexity of today's computer systems (network management was the main motivation to autonomic computing). Although cloud computing exhibits certain autonomic features such as automatic resource provisioning, its objective is to lower the resource cost rather than to reduce system complexity.

In principle, one of the main reasons for the success of cloud computing is the role it has played in eliminating the size of an enterprise as a critical factor in its economic success. An excellent example of this change is the notion of *data centers* which provide clients with the physical infrastructure needed to host their computer systems, including redundant power supplies, high bandwidth communication capabilities, environment monitoring, and security services. These data centers eliminate the need for small companies to make a large capital expenditure in building an infrastructure to create a global customer base [24]. The data center model has been effective since it allows an enterprise of any size to manage growth with the popularity of its product or service while at the same time also allows the enterprise to cut its losses if the launched product or service does not succeed.

In practice, the main idea behind data centers is to leverage the virtualization technology to maximize the utilization of computing resources. Therefore, it provides the basic ingredients such as storage, CPUs, and network bandwidth as a commodity by specialized service providers at low unit cost. Hence, users of cloud services should not need to worry about scalability because the storage provided is virtually infinite and the network links are virtually capable to quickly transfer any quantity of data between the available servers. Therefore, *from the data management point of view*, cloud computing provides full availability where: 1) Users can read and write data at any time without ever being blocked. 2) The response times are (virtually) constant and do not depend on the number of concurrent users, the size of the database or any other system parameter. Furthermore, users do not need to worry about backups. If components fail, it is the responsibility of the service provider to replace them and make the data available using replicas in the meantime. Another important reason to build new services based on cloud computing is that no investments are needed upfront and the cost grows linearly and predictably with the increase of usage (*pay-as-you-go*). For example, if a company is launching a new product that requires a large database, it does not need to go out and buy one. However, it can use one hosted by a cloud provider such as Amazon or Google. Hence, all of these advantages allow companies to focus on their business innovation rather than focusing on building larger data centers to achieve scalability goals.

This paper gives a comprehensive survey of numerous approaches and mechanisms of deploying data-intensive applications in the cloud which are gaining a lot of momentum in both research and industrial communities. We discuss the advantages and the disadvantages of each approach and its suitability to support certain classes of applications and end-users. Such survey is very important to the networking community since by knowing the characteristics of the applications and how they manage the data, it is possible to modify the network mechanisms in order to optimize the utilization of the cloud, just as happened to the grid computing [21], [22], [23]. The remainder of this paper is organized as follows. Section II gives an overview of the main goals and challenges of deploying data-intensive applications in cloud environments. Section III surveys the state-of-the-art of cloud data management systems while a discussion of the different programming models for cloud applications are presented in Section IV. Several real-world case studies are discussed in Section V before we conclude the paper in Section VI.

II. Cloud Data Management: Goals and Challenges

In this section, we give an overview of the main goals and challenges for deploying data-intensive computing application in cloud environments.

A. Goals

In general, successful cloud data management systems are designed to satisfy as much as possible from the following *wish list* [25], [26]:

- *Availability*: They must be always accessible even on the occasions where there is a network failure or a whole datacenter has gone offline. Towards this goal, the concept of Communication as a Service (CaaS) emerged to support such requirements, as well as network security, dynamic provisioning of virtual overlays for traffic isolation or dedicated bandwidth, guaranteed message delay, communication encryption, and network monitoring [27]. For example, [28], [29] present various architectural design decisions, protocols and solutions to provide QoS communications as a service. In practice, VoIP telephone systems, audio, video conferencing and instant messaging are candidate cloud applications that can be composed of CaaS and can in turn provide composable cloud solutions to other common applications.

- *Scalability*: They must be able to support very large databases with very high request rates at very low latency. They should be able to take on new tenants or handle growing tenants without much effort beyond that of adding more hardware. In particular, the system must be able to automatically redistribute data to take advantage of the new hardware.
- *Elasticity*: They must be able to satisfy changing application requirements in both directions (scaling up or scaling down). Moreover, the system must be able to gracefully respond to these changing requirements and quickly recover to its steady state.
- *Performance*: On public cloud computing platforms, pricing is structured in a way such that one pays only for what one uses, so the vendor price increases linearly with the requisite storage, network bandwidth, and compute power. Hence, the system performance has a direct effect on its costs. Thus, efficient system performance is a crucial requirement to save money.
- *Multitenancy*: They must be able to support many applications (tenants) on the same hardware and software infrastructure. However, the performance of these tenant must be isolated from each another. Adding a new tenant should require little or no effort beyond that of ensuring that enough system capacity has been provisioned for the new load.
- *Load and Tenant Balancing*: They must be able to automatically move load between servers so that most of the hardware resources are effectively utilized and to avoid any resource overloading situations.
- *Fault Tolerance*: For transactional workloads, a fault tolerant cloud data management system needs to be able to recover from a failure without losing any data or updates from recently committed transactions. Moreover, it needs to successfully commit transactions and make progress on a workload even in the face of worker node failures. For analytical workloads, a fault tolerant cloud data management system should not need to restart a query if one of the nodes involved in query processing fails.
- *Ability to run in a heterogeneous environment*: On cloud computing platforms, there is a strong trend towards increasing the number of nodes that participate in query execution. It is nearly impossible to get homogeneous performance across hundreds or thousands of compute nodes. Part failures that do not cause complete node failure, but result in degraded hardware performance become more common at scale. A cloud data management system should be designed to run in a heterogeneous environment and must take appropriate measures to prevent performance degrading due to parallel processing on distributed nodes.
- *Flexible query interface*: They should support both SQL and non-SQL interface languages (e.g MapReduce). Moreover, they should provide mechanism for allowing the user to write user defined functions (UDFs) and queries that utilize these UDFs should be automatically parallelized during their processing.

It is possible to observe that even some more application-specific goals can influence the networking mechanisms of the cloud. For example, to make possible the *load and tenant balancing*, it is necessary to guarantee that the migration of the applications will take a time which will not affect the quality of the service provided to the user of these migrated applications. This time is directly related to network metrics like available bandwidth and delay. A *flexible query interface* can also influence the network configuration of the cloud since it can require the parallelism of some tasks, which needs the efficient transfer of data from one source to several destinations.

B. Challenges

Deploying data-intensive applications on cloud environment is not a trivial or straightforward task. Armbrust et al. [7] and Abadi [30] argued a list of obstacles to the growth of cloud computing applications as follows.

- *Availability of a Service*: In principle, a distributed system is a system that operates robustly over a wide network. A particular feature of network computing is that the network links can potentially disappear. Organizations worry about whether cloud computing services will have adequate availability. High availability is one of the most challenging goals because even the slightest outage can have significant financial consequences and impacts customer trust.
- *Data Confidentiality*: In general, moving data off premises increases the number of potential security risks and appropriate precautions must be made. Transactional databases typically contain the complete set of operational data needed to power mission-critical business processes. This data includes detail at the lowest granularity, and often includes sensitive information such as customer data or credit card numbers. Therefore, unless such sensitive data is encrypted using a key that is not located at the host, the data may be accessed by a third party without the customer's knowledge.
- *Data Lock-In*: APIs for cloud computing have not been, yet, subject of active standardization. Thus, customers cannot easily extract their data and programs from one site to run on another. The concerns about the difficulties of extracting data from the cloud is preventing some organizations from adopting cloud computing. Customer lock-in may be attractive to cloud computing providers but cloud computing users are vulnerable to price increases, to reliability problems, or even to providers going out of business.

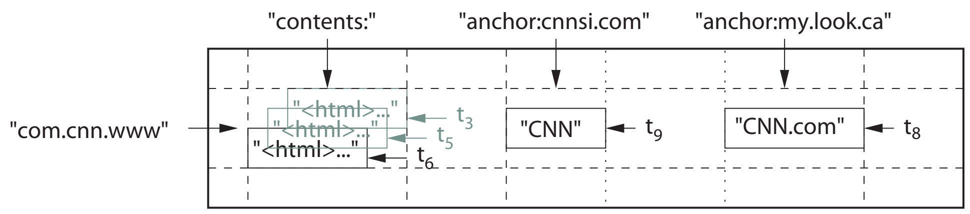


Fig. 4. Sample BigTable Structure [32]

- *Data Transfer Bottlenecks*: Cloud users and cloud providers have to think about the implications of placement and traffic at every level of the system if they want to minimize costs.
- *Application Parallelization*: Computing power is elastic but only if workload is parallelizable. Getting additional computational resources is not as simple as just upgrading to a bigger and more powerful machine on the fly. However, the additional resources are typically obtained by allocating additional server instances to a task.
- *Shared-Nothing Architecture*: Data management applications designed to run on top of cloud environment should follow a shared-nothing architecture [31] where each node is independent and self-sufficient and there is no single point of contention across the system. Most of transactional data management systems do not typically use a shared-nothing architecture.
- *Performance Unpredictability*: Many HPC applications need to ensure that all the threads of a program are running simultaneously. However, today's virtual machines and operating systems do not provide this service.
- *Application Debugging in Large-Scale Distributed Systems*: A challenging aspect in cloud computing programming is the removal of errors in these very large scale distributed systems. A common occurrence is that these bugs cannot be reproduced in smaller configurations, so the debugging must occur at the same scale as that in the production datacenters.

III. Cloud Data Management Systems: State-of-the-art

The task of storing data persistently has been traditionally achieved through filesystems or relational databases. In recent years, this task is increasingly achieved through simpler storage systems that are easier to build and maintain at large scale while achieving reliability and availability as primary goals. This section provides a survey of the state-of-the-art of large scale data management systems in the cloud environments organized by their source service provider (e.g. Google, Yahoo!, Microsoft, open source projects).

A. Google: Bigtable / AppEngine Datastore

1) Bigtable: Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size (petabytes of data) across thousands of commodity servers [32]. It has been used by more than sixty Google products and projects such as: Google search engine[14], Google Finance[15], Orkut[16], Google Docs[17] and Google Earth[18]. These products use Bigtable for a variety of demanding workloads which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

Bigtable does not support a full relational data model. However, it provides clients with a simple data model that supports dynamic control over data layout and format. In particular, a Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes. Thus, clients usually need to serialize various forms of structured and semi-structured data into these strings. A concrete example that reflects some of the main design decisions of Bigtable is the scenario of storing a copy of a large collection of web pages into a single table. Figure 4 illustrates an example of this table where $URLs$ are used as row keys and various aspects of web pages as column names. The contents of the web pages are stored in a single column which stores multiple versions of the page under the timestamps when they were fetched.

The row keys in a table are arbitrary strings where every read or write of data under a single row key is atomic. Bigtable maintains the data in lexicographic order by row key where the row range for a table is dynamically partitioned. Each row range is called a *tablet* which represents the unit of distribution and load balancing. Thus, reads of short row ranges are efficient and typically require communication with only a small number of machines. Bigtables can have an unbounded number of columns which are grouped into sets called *column families*. These column families represent the basic unit of access control. Each cell in a Bigtable can contain multiple versions of the same data which are indexed by their timestamps. Each client can flexibly decide the number of n versions of a cell that need to be kept. These versions are stored in decreasing timestamp order so that the most recent versions can be always read first. The fact that reads of short row ranges require low communication can affect the development of queries, so that they are suitable to the available network resources.

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as

[14]http://www.google.com/

[15]http://www.google.com/finance

[16]http://www.orkut.com/

[17]http://docs.google.com/

[18]http://earth.google.com/

```
SELECT [* | __key__] FROM <kind>
[WHERE <condition> [AND <condition> ...]]
[ORDER BY <property> [ASC | DESC] [, <property> [ASC | DESC] ...]]
[LIMIT [<offset>,]<count>]
[OFFSET <offset>]

<condition> := <property> {< | <= | > | >= | = | != } <value>
<condition> := <property> IN <list>
<condition> := ANCESTOR IS <entity or key>
```

Fig. 5. Basic GQL Syntax

access control rights. Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. On the transaction level, Bigtable supports only *single-row* transactions which can be used to perform atomic read-modify-write sequences on data stored under a single row key (i.e. no general transactions across row keys).

On the physical level, Bigtable uses the distributed Google File System (GFS) [33] to store log and data files. The Google *SSTable* file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Bigtable relies on a distributed lock service called *Chubby* [34] which consists of five active replicas, one of which is elected to be the *master* and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Bigtable uses Chubby for a variety of tasks such as: 1) Ensuring that there is at most one active master at any time. 2) Storing the bootstrap location of Bigtable data. 3) Storing Bigtable schema information and the access control lists. The main limitation of this design is that if Chubby becomes unavailable for an extended period of time, the whole Bigtable becomes unavailable.

On the runtime, each Bigtable is allocated to one master server and many tablet servers which can be dynamically added (or removed) from a cluster based on the changes in workloads. The master server is responsible for assigning tablets to tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations. Each tablet server manages a set of tablets. The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

2) AppEngine Datastore: While the Bigtable system is used internally in Google Inc, Google has released the Google AppEngine datastore[19] which provides a scalable schemaless object data storage for web application. It performs queries over data objects, known as *entities*. An entity has one or more *properties* where one property can be a reference to another entity. Datastore entities are schemaless where two entities of the same kind are not obligated to have the same properties, or use the same value types for the same properties. Each entity also has a key that uniquely identifies the entity. The simplest key has a kind and a unique numeric ID provided by the datastore. An application can fetch an entity from the datastore by using its key or by performing a query that matches the entity's properties. A query can return zero or more entities and can return the results sorted by property values. A query can also limit the number of results returned by the datastore to conserve memory and execution time.

With the AppEngine datastore, every attempt to create, update or delete an entity happens in a transaction. A transaction ensures that every change made to the entity is saved to the datastore. However, in the case of failure, none of the changes are made. This ensures the consistency of data within an entity. The datastore uses optimistic concurrency to manage transactions. The datastore replicates all data to multiple storage locations, so if one storage location fails, the datastore can switch to another and still be able to access the data. To ensure that the view of the data stays consistent as it is being updated, an application uses one location as its primary location and changes to the data on the primary are replicated to the other locations in parallel. An application switches to an alternate location only for large failures. For small failures in primary storage, such as a single machine becoming unavailable temporarily, the datastore waits for primary storage to become available again to complete an interrupted operation. This is necessary to give the application a reasonably consistent view of the data, since alternate locations may not yet have all of the changes made to the primary. In general, an application can choose between two read policies: 1) A read policy of *strong consistency* which always reads from the primary storage location and always return the last updated value. 2) A policy of *eventual consistency* [35] which will read from an alternate location when the primary location is unavailable. In particular, the eventual consistency policy guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme. This is an example of how the network metrics can influence the performance of the system. Similarly, if cloud applications based on AppEngine need to access the last updated value with a high probability, it is important that the cloud provides network paths between the datastores with minimum delay, low probability of drops, and minimum available bandwidth.

The AppEngine datastore provides a Python interface which includes a rich data modeling API and a SQL-like query language called *GQL*[20]. Figure 5 depicts the basic syntax of

[19]http://code.google.com/appengine/docs/python/datastore/

[20]http://code.google.com/appengine/docs/python/datastore/gqlreference.html

GQL. A GQL query returns zero or more entities or keys of the requested kind. In principle, a GQL query cannot perform a SQL-like join query. Every GQL query always begins with either *SELECT * FROM* or *SELECT (key) FROM* followed by the name of the kind. The optional *WHERE* clause filters the result set to those entities that meet one or more conditions. Each condition compares a property of the entity with a value using a comparison operator. GQL does not have an *OR* operator. However, it does have an *IN* operator which provides a limited form of *OR*. The optional *ORDER BY* clause indicates that results should be returned sorted by the given properties in either ascending (ASC) or descending (DESC) order. An optional *LIMIT* clause causes the query to stop returning results after the first count entities. The *LIMIT* can also include an offset to skip that many results to find the first result to return. An optional *OFFSET* clause can specify an offset if no *LIMIT* clause is present. Managers of clouds providing access to AppEngine datastores should consider the effects of the network metrics on the results of GQL queries with the *LIMIT* clause. As the cloud promises a platform with an infinite capacity, it should be capable of return fast results to the user independently of the *LIMIT*. Also, users paying more should not be forced to restrict their GQL queries with *LIMIT* clauses because of an unexpected drop on the performance of the cloud network.

B. Yahoo!: PNUTS/Sherpa

The *PNUTS* system (renamed later to Sherpa) is a massive-scale hosted database system which is designed to support Yahoo!'s web applications [36]. The main focus of the system is on data serving for web applications, rather than complex queries. It relies on a simple relational model where data is organized into tables of records with attributes. In addition to typical data types, *blob* is a main valid data type which allows storing arbitrary structures inside a record, but not necessarily large binary objects like images or audio. The PNUTS system does not enforce constraints such as referential integrity on the underlying data. Therefore, the schema of these tables are flexible where new attributes can be added at any time without halting any query or update activity. In addition, it is not required that each record have values for all attributes.

Figure 6 illustrates the system architecture of PNUTS. The system is divided into regions where each region contains a full complement of system components and a complete copy of each table. Regions are typically, but not necessarily, geographically distributed. Therefore, on the physical level, data tables are horizontally partitioned into groups of records called *tablets*. These tablets are scattered across many servers where each server might have hundreds or thousands of tablets. The assignment of tablets to servers is flexible in a way that allows balancing the workloads by moving a few tablets from an overloaded server to an underloaded server.

The query language of PNUTS supports selection and projection from a single table. Operations for updating or deleting existing record must specify the primary key. The system is designed primarily for online serving workloads that consist mostly of queries that read and write single records or small groups of records. Thus, it provides a *multiget* operation which supports retrieving multiple records in parallel by specifying a set of primary keys and an optional predicate. The *router* component (Figure 6) is responsible for determining which storage unit need to be accessed for a given record to be read or written by the client (Despite the name, there is no relation to a network router, although the implementation of traditional network routing algorithms, in this router, can influence the performance of the PNUTS system). Therefore, the primary-key space of a table is divided into intervals where each interval corresponds to one tablet. The router stores an interval mapping which defines the boundaries of each tablet and maps each tablet to a storage unit. The query model of PNUTS does not support join operations which are too expensive in such massive scale systems.

The PNUTS system does not have a traditional database log or archive data. However, it relies on a pub/sub mechanism that acts as a redo log for replaying updates that are lost before being applied to disk due to failure. In particular, PNUTS provides a consistency model that is between the two extremes of general serializability and eventual consistency [35]. The design of this model is derived from the observation that web applications typically manipulate one record at a time while different records may have activity with different geographic locality. Thus, it provides *per-record timeline* consistency where all replicas of a given record apply all updates to the record in the same order. In particular, for each record, one of the replicas (independently) is designated as the master where all updates to that record are forwarded to the master. The master replica for a record is adaptively changed to suit the workload where the replica receiving the majority of write requests for a particular record is selected to be the master for that record. Relying on the per-record timeline consistency model, the PNUTS system supports the following range of API calls with varying levels of consistency guarantees:

- *Read-any*: This call has the lower latency as it returns a possibly stale version of the record.
- *Read-critical(required version)*: It returns a version of the record that is strictly newer than, or the same as the *required version*.
- *Read-latest*: This call returns the latest copy of the record that reflects all writes that have succeeded. It is expected that the *read-critical* and *read-latest* can have a higher latency than read-any if the local copy is too stale and the system needs to locate a newer version at a remote replica. To guarantee the required QoS for users, mainly to those paying more, can be necessary to plan the capacity of the cloud resources. For example, records to these users could be stored in a pool of servers interconnected by dedicated network links with high capacity.
- *Write*: This call gives the same ACID guarantees as a transaction with a single write operation in it (e.g. blind writes).
- *Test-and-set-write(required version)*: This call performs the requested write to the record if and only if the present version of the record is the same as the required version. This call can be used to implement transactions that first read a record, and then do a write to the record based on the read, e.g. incrementing the value of a counter.

Since the system is designed to scale to cover several

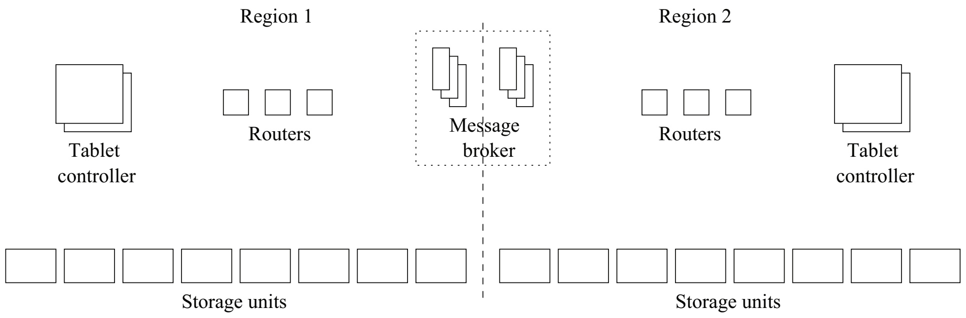


Fig. 6. PNUTS System Architecture [36]

worldwide replicas, automated failover and load balancing is the only way to manage the operations load. Therefore, for any failed server, the system automatically recover by copying data from a replica to other live servers.

C. Amazon: Dynamo / S3 / SimpleDB / RDS

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. In this environment, there are strict operational requirements in terms of performance, reliability and efficiency, and to be able to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. To meet these needs, Amazon has developed a number of storage technologies such as: Dynamo System [37], Simple Storage Service (S3)[21], SimpleDB[22] and Relational Database Service (RDS)[23].

1) Dynamo: The Dynamo system [37] is a highly available and scalable distributed key/value based datastore built for supporting *internal* Amazon's applications. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. There are many services on Amazon's platform that only need primary-key access to a data store. The common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Thus, Dynamo provides a simple primary-key only interface to meet the requirements of these applications. The query model of the Dynamo system relies on simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (blobs) identified by unique keys. No operations span multiple data items.

Dynamo's partitioning scheme relies on a variant of consistent hashing mechanism [38] to distribute the load across multiple storage hosts. In this mechanism, the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that the departure or arrival of a node only affects its immediate neighbors while other nodes remain unaffected. So, failures on network links which make some nodes disconnected will not affect all the data sets.

[21]https://s3.amazonaws.com/

[22]http://aws.amazon.com/simpledb/

[23]http://aws.amazon.com/rds/

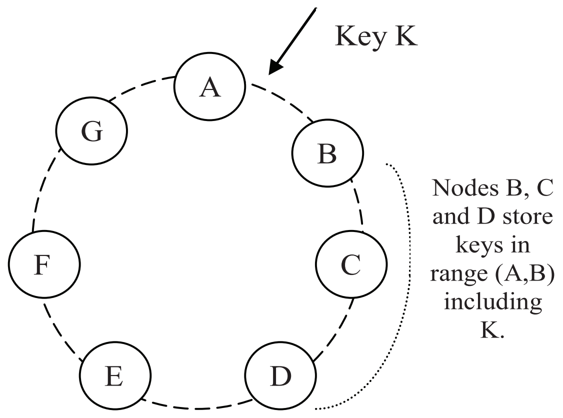


Fig. 7. Partitioning and replication of keys in Dynamo ring. [37]

In the Dynamo system, each data item is replicated at N hosts where N is a parameter configured "per-instance". Each key k is assigned to a coordinator node. The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the $N-1$ clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its N^{th} predecessor. As illustrated in Figure 7, node B replicates the key k at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges $(A, B]$, $(B, C]$, and $(C, D]$. The list of nodes that is responsible for storing a particular key is called the preference list. The system is designed so that every node in the system can determine which nodes should be in this list for any particular key.

2) S3 / SimpleDB / RDS: Amazon Simple Storage Service (S3) is an online public storage web service offered by Amazon Web Services. Conceptually, S3 is an infinite store for objects of variable sizes. An object is simply a byte container which is identified by a URI. Clients can read and update S3 objects remotely using a simple web services (SOAP or REST-based) interface. For example, $get(uri)$ returns an object and $put(uri, bytestream)$ writes a new version of the object.

In S3, stored data is organized over a two-level namespace: *buckets* and *objects*. Buckets are similar to folders or containers which can store an unlimited number of data objects. Objects are composed from two parts: an opaque blob (of up to 5GB in size) and metadata, which includes user-specified key/value pairs for each object (up to 2KB) and a small number of predefined HTTP metadata entries (e.g., Last-Modified). In S3, search function is limited to a single bucket and is based on the object name only. Metadata or content-based search capabilities are not provided. Thus, S3 can be considered as an online backup solution or for storing large objects which are not frequently updated.

Brantner et al. [39] have presented initial efforts of building Web-based database applications on top of S3. They described various protocols in order to store, read, and update objects and indexes using S3. For example, the *record manager* component is designed to manage records where each record is composed of a key and payload data. Both key and payload are bytestreams of arbitrary length where the only constraint is that the size of the whole record must be smaller than the page size. Physically, each record is stored in exactly one page which in turn is stored as a single object in S3. Logically, each record is part of a *collection* (e.g. a table). The record manager provides functions to create new objects, read objects, update objects and scan collections. The *page manager* component implements a buffer pool for S3 pages. It supports the reading of pages from S3, pinning the pages in the buffer pool, updating the pages in the buffer pool and marking the pages as updated. All these functionalities are implemented in a straightforward way just as in any standard database system. Furthermore, the page manager implements the commit and abort methods where it is assumed that the write set of a transaction (i.e. the set of updated and newly created pages) fits into the client's main memory or secondary storage (flash or disk). If an application commits, all the updates are propagated to S3 and all the affected pages are marked as unmodified in the client's buffer pool. Moreover, they implemented standard B-tree indexes on top of the page manager and basic redo log records. On the other side, there are many database-specific issues that has not be addressed, yet, by this work. For example, DB-style strict consistency and transactions mechanisms. Furthermore, query processing techniques (e.g. join algorithms and query optimization techniques) and traditional database functionalities such as: bulkload a database, create indexes and drop a whole collection need to be devised.

Similar to S3, Amazon has not published the details of its other two products: SimpleDB and RDS. Generally, SimpleDB is a designed for running queries on structured data. In SimpleDB, data in is organized into *domains* (e.g tables) within which we can put data, get data or run queries. Each domains consist of *items* (e.g records) which are described by *attribute* name/value pairs. It is not necessary to pre-define all of the schema information as new attributes can be added to the stored dataset when needed. Thus, the approach is similar to that of a spreadsheet and does not follow the traditional relational model. SimpleDB provides a small group of API calls that enables the core functionality to build client applications such as: $CreateDomain$, $DeleteDomain$, $PutAttributes$, $DeleteAttributes$, $GetAttributes$ and $Select$. The main focus of SimpleDB is fast reading. Therefore, query operations are designed to run on a single domain. SimpleDB keeps multiple copies of each domain where a successful write operation guarantees that all copies of the domain will durably persist. In particular, SimpleDB supports two read consistency options: eventually consistent read [35] and consistent read.

Amazon Relational Database Service (RDS) is another recent service which gives access to the full capabilities of a familiar MySQL database. Hence, the code, applications, and tools which are already designed on existing MySQL databases can work seamlessly with Amazon RDS. Once the database instance is running, Amazon RDS can automate common administrative tasks such as performing backups or patching the database software. Amazon RDS can also manage synchronous data replication and automatic failover management.

D. Microsoft: DRYAD / SQL Azure

Dryad is a general-purpose distributed execution engine introduced by Microsoft for coarse-grain data-parallel applications [40]. A Dryad application combines computational *vertices* with communication *channels* to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs. The Dryad system allows the developer fine control over the communication graph as well as the subroutines that live at its vertices. A Dryad application developer can specify an arbitrary directed acyclic graph to describe the application's communication patterns, and express the data transport mechanisms (files, TCP pipes, and shared-memory FIFOs) between the computation vertices. This direct specification of the graph gives the developer greater flexibility to easily compose basic common operations, leading to a distributed analogue of *piping* together traditional Unix utilities such as grep, sort and head.

Dryad is notable for allowing graph vertices (and computations in general) to use an arbitrary number of inputs and outputs while MapReduce restricts all computations to take a single input set and generate a single output set. The overall structure of a Dryad job is determined by its communication flow. A job is a directed acyclic graph where each vertex is a program and edges represent data channels. It is a logical computation graph that is automatically mapped onto physical resources by the runtime. At run time each channel is used to transport a finite sequence of structured items. A Dryad job is coordinated by a process called the *job manager* that runs either within the cluster or on a user's workstation with network access to the cluster. The job manager contains the application-specific code to construct the job's communication graph along with library code to schedule the work across the available resources. All data is sent directly between vertices and thus the job manager is only responsible for control decisions and is not a bottleneck for any data transfers. Therefore, much of the simplicity of the Dryad scheduler and fault-tolerance model come from the assumption that vertices are deterministic.

Dryad has its own high-level language called DryadLINQ [41]. It generalizes execution environments such as SQL and MapReduce in two ways: 1) Adopting an expressive data model of strongly typed .NET objects. 2) Supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language. DryadLINQ[24] exploits LINQ (Language INtegrated Query[25], a set of .NET constructs for programming with datasets) to provide a powerful hybrid of declarative and imperative programming. The system is designed to provide flexible and efficient distributed computation in any LINQ-enabled programming language including C#, VB, and F#[26]. Objects in DryadLINQ datasets can be of any .NET type, making it easy to compute with data such as image patches, vectors, and matrices. In practice, a DryadLINQ program is a sequential program composed of LINQ expressions that perform arbitrary side-effect-free transformations on datasets and can be written and debugged using standard .NET development tools. The DryadLINQ system automatically translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution platform.

While the Dryad system is used only internally in Microsoft and has not been released for public use, Microsoft has recently released the SQL Azure Database system[27]. Not much details has been published on the implementation of this project. However, it is announced as a cloud-based relational database service which has been built on Microsoft SQL Server technologies. SQL Azure is designed to provide a highly available, scalable, multi-tenant database service hosted in the cloud. As such, applications can create, access and manipulate tables, views, indexes, roles, stored procedures, triggers and functions. It can also execute complex queries and joins across multiple tables. It currently supports Transact-SQL (T-SQL), native ODBC and ADO.NET data access[28].

E. Open Source Projects

In practice, most of the cloud data management systems provided by the big players (e.g. BigTable, Dynamo, PNUTS) are designed for their internal use and are thus not available for public use. Therefore, many open source projects have been built to implement the concepts of these systems and are made available for public users. Some of these systems have started to gain a lot of interest from the research community. There are not much details that have been published about the implementation most of these systems yet. Therefore, here, we give a brief introduction about some of these projects. However, for the full list, we refer the reader to the NoSQL database website[29].

Cassandra[30] is presented as highly scalable, eventually consistent, distributed, structured key-value store [42]. It has been open sourced by Facebook in 2008. It is designed by Avinash Lakshman (one of the authors of Amazon's Dynamo) and Prashant Malik (Facebook Engineer). Cassandra brings together the distributed systems technologies from Dynamo and the data model from Google's Bigtable. Like Dynamo, Cassandra is eventually consistent. Like Bigtable, Cassandra provides a ColumnFamily-based data model richer than typical key/value systems. In Cassandra's data model, *column* is the lowest/smallest increment of data. It's a tuple (triplet) that contains a name, a value and a timestamp. A *column family* is a container for columns, analogous to the table in a relational system. It contains multiple columns, each of which has a name, value, and a timestamp, and are referenced by row keys. A *keyspace* is the first dimension of the Cassandra hash, and is the container for column families. Keyspaces are of roughly the same granularity as a schema or database (i.e. a logical collection of tables) in RDBMS. They can be seen as a namespace for ColumnFamilies and is typically allocated as one per application. *SuperColumns* represent columns that themselves have subcolumns (e.g. Maps).

The HyperTable[31] project is designed to achieve a high performance, scalable, distributed storage and processing system for structured and unstructured data. It is designed to manage the storage and processing of information on a large cluster of commodity servers, providing resilience to machine and component failures. In HyperTable, data is represented in the system as a multi-dimensional table of information. The HyperTable systems provides a low-level API and Hypertable Query Language (HQL) that allows the user to create, modify, and query the underlying tables. The data in a table can be transformed and organized at high speed by performing computations in parallel and pushing them to where the data is physically stored.

CouchDB[32] is a document-oriented database that can be queried and indexed in a MapReduce fashion using JavaScript. In CouchDB, documents are the primary unit of data. A CouchDB document is an object that consists of named fields. Field values may be strings, numbers, dates, or even ordered lists and associative maps. Hence, a CouchDB database is a flat collection of documents where each document is identified by a unique ID. CouchDB provides a RESTful HTTP API for reading and updating (add, edit, delete) database documents. The CouchDB document update model is lockless and optimistic. Document edits are made by client applications. If another client was editing the same document at the same time, the client gets an edit conflict error on save. To resolve the update conflict, the latest document version can be opened, the edits reapplied and the update tried again. Document

[24]http://research.microsoft.com/en-us/projects/dryadlinq/
[25]http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
[26]http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/
[27]http://www.microsoft.com/windowsazure/sqlazure/
[28]http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx
[29]http://nosql-database.org/
[30]http://cassandra.apache.org/
[31]http://hypertable.org/
[32]http://couchdb.apache.org/

TABLE I
DESIGN DECISIONS OF CLOUD STORAGE SYSTEMS.

System	Data Model	Query Interface	Consistency	CAP Options	License
BigTable	Column Families	Low-Level API	Eventually Consistent	CP	Internal at Google
Google AppEng	Objects Store	Python API - GQL	Strictly Consistent	CP	Commercial
PNUTS	Key-Value Store	Low-Level API	Timeline Consistent	AP	Internal at Yahoo
Dynamo	Key-Value Store	Low-Level API	Eventually Consistent	AP	Internal at Amazon
S3	Large Objects Store	Low-Level API	Eventually Consistent	AP	Commercial
SimpleDB	Key-Value Store	Low-Level API	Eventually Consistent	AP	Commercial
RDS	Relational Store	SQL	Strictly Consistent	CA	Commercial
SQL Azure	Relational Store	SQL	Strictly Consistent	CA	Commercial
Cassandra	Column Families	Low-Level API	Eventually Consistent	AP	Open source - Apache
Hypertable	Multi-dimensional Table	Low-Level API, HQL	Eventually Consistent	AP	Open source - GNU
CouchDB	Document-Oriented Store	Low-Level API	Optimistically Consistent	AP	Open source - Apache

updates are all or nothing, either succeeding entirely or failing completely. The database never contains partially saved or edited documents.

Many other variant projects have recently started to follow the NoSQL (Not Only SQL) movement and support different types of data stores such as: key-value stores (e.g. Voldemort[33], Dynomite[34]), document stores (e.g. MongoDB[35], Riak[36]) and graph stores (e.g. Neo4j[37], DEX[38])

F. Cloud Data Management: Trade-offs

In general, a cloud provider typically offers the following services for web applications [43]:

- *Elastic compute cluster*. The cluster includes an elastic number of virtual instances that run the applications and process the incoming requests.
- *Persistent storage*. The storage service stores application data in a similar fashion as traditional databases or file systems.
- *Intra-cloud network*. The network inside a cloud that connects the virtual instances of an application, and connects them to cloud-provided services such as the persistent storage service.
- *Wide-area delivery network*. The wide-area delivery network of a cloud delivers an application's contents to the end hosts from multiple geographically distributed data centers of the cloud.

In practice, all providers promise high bandwidth network paths (typically from hundreds of Mbps to Gbps) approximating a private data center network. The wide-area delivery network is used to serve the end user's request by directing them to the instance which is close to the geographic location of the user in order to minimize the wide-area network latency. Therefore, different clouds have data centers at different locations. For example, both of Amazon and Azure currently have data centers in US, Europe, and Asia. In fact, this is a particularly important aspect as today's web applications face a much broader and geographically more diverse user base than before. This aspect brings some questions to the location of the cloud resources if the electricity costs are considered. *Is it better to keep the majority of the hosts in places around the world with low energy costs? But will this compensate the costs with high capacity network links to connect the hosts?* As pointed in [44], electricity costs can vary each hour, motivating the implementation of an algorithm to decide how, in a certain time of the day, the load of the cloud should be geographically distributed. Besides the energy costs, it is important that the cloud providers include in the algorithm the costs necessary to keep the QoS when changing the utilized resources, because there is no guarantee that the networking providers interconnecting the resources will charge the same bill to the same service in different locations. It is important to observe that the network equipments (switches, routers, etc...) also affect the energy bill the same way as the computational power, with the difference that this impact must be considered in both ends of the network connections.

An important issue in designing large scale data management applications is to avoid the mistake of trying to be "*everything for everyone*". As with many types of computer systems, no one system can be best for all workloads and different systems make different tradeoffs in order to optimize for different applications. Therefore, the most challenging aspects in these application is to identify the most important features of the target application domain and to decide about the various design trade-offs which immediately lead to performance trade-offs. To tackle this problem, Jim Gray came up with the heuristic rule of "*20 queries*" [1]. The main idea of this heuristic is that on each project, we need to identify the 20 most important questions the user wanted the data system to answer. He said that five questions are not enough to see a broader pattern and a hundred questions would result in a shortage of focus. Table I summarizes the design decisions of our surveyed systems.

In general, it is hard to maintain ACID guarantees in the face of data replication over large geographic distances. The CAP theorem [45], [46] shows that a shared-data system can only choose at most two out of three properties: *Consistency* (all records are the same in all replicas), *Availability* (all replicas can accept updates or inserts), and *tolerance to Partitions* (the system still functions when distributed replicas cannot talk to each other). In practice, it is highly important for cloud-

[33]http://project-voldemort.com/
[34]http://wiki.github.com/cliffmoon/dynomite/dynomite-framework
[35]http://www.mongodb.org/
[36]http://wiki.basho.com/display/RIAK/Riak
[37]http://neo4j.org/
[38]http://www.dama.upc.edu/technology-transfer/dex

based applications to be always available and accept update requests of data and at the same time cannot block the updates even while they read the same data for scalability reasons. To deal with partial failures of the system, cloud-based applications do not work in synchronous communications which halt the overall system because of the lifetime synchronization among the transaction participants. Hence, while it is needed to decompose data or transactions for scalability, we should deal with the consistency between decomposed data or sub-transactions by using synchronous communications. However since we cannot practically use synchronous communications in view of partial failures, we need to weaken the consistency conditions of the applications. Therefore, When data is replicated over a wide area, this essentially leaves just consistency and availability for a system to choose between. Thus, the '*C*' (consistency) part of ACID is typically compromised to yield reasonable system availability [30]. Hence, most of the cloud data management overcome the difficulties of distributed replication by relaxing the ACID guarantees of the system. In particular, they implement various forms of weaker consistency models (e.g. eventual consistency, timeline consistency, session consistency [47]) so that all replicas do not have to agree on the same value of a data item at every moment of time. Therefore, transactional data management applications (e.g. banking, stock trading, supply chain management) which rely on the ACID guarantees that databases provide, tend to be fairly write-intensive or require microsecond precision are less obvious candidates for the cloud environment until the cost and latency of wide-area data transfer decrease significantly. Cooper et al. [48] discussed the tradeoffs facing cloud data management systems as follows:

- *Read performance versus write performance*: Log-structured systems that only store update deltas can be very inefficient for reads if the data is modified over time. On the other side, writing the complete record to the log on each update avoids the cost of reconstruction at read time but there is a correspondingly higher cost on update. Unless all data fits in memory, random I/O to the disk is needed to serve reads (e.g. as opposed to scans). However, for write operations, much higher throughput can be achieved by appending all updates to a sequential disk-based log. This tradeoff can influence decisions related with the network resources. It is possible to the cloud provider to acquire network links with an asynchronous capacity of upload and download to match the decision about data updates. If the system writes only the update deltas, it could be enough to prioritize the download bandwidth instead the upload bandwidth. This decision will save money from the provider and can be reflected on a higher profit.
- *Latency versus durability*: Writes may be synched to disk before the system returns a success result to the user or they may be stored in memory at write time and synched later. The advantages of the latter approach are that avoiding disk operations greatly improves write latency, and potentially improves throughput The disadvantage is the risk of data loss if a server crashes and loses unsynched updates. Clearly, this decision is influenced by the network performance. If the network delay is high, it would be better to prioritize the later disk synchronization. By doing this, it is possible to keep a low latency observed by the user.
- *Synchronous versus asynchronous replication*: Synchronous replication ensures all copies are up to date but potentially incurs high latency on updates. Furthermore, availability may be impacted if synchronously replicated updates cannot complete while some replicas are offline. Asynchronous replication avoids high write latency but allows replicas to be stale. Furthermore, data loss may occur if an update is lost due to failure before it can be replicated. The influence of the network on this tradeoff is similar to the influence on the *Latency versus durability* tradeoff. The asynchronous replication should be preferable if the network delay is high.
- *Data partitioning*: Systems may be strictly row-based or allow for column storage. Row-based storage supports efficient access to an entire record and is ideal if we typically access a few records in their entirety. Column-based storage is more efficient for accessing a subset of the columns, particularly when multiple records are accessed. This decision can influence the selection of hosts to guarantee that the rows/columns are not stored in very remote (in terms of network metrics) nodes.

Kraska et al. [49] have argued that finding the right balance between cost, consistency and availability is not a trivial task. High consistency implies high cost per transaction and, in some situations, reduced availability but avoids penalty costs. Low consistency leads to lower costs per operation but might result in higher penalty costs. Hence, they presented a mechanism that not only allows designers to define the consistency guarantees on the data instead at the transaction level but also allows the ability to automatically switch consistency guarantees at runtime. They described a dynamic consistency strategy, called *Consistency Rationing*, to reduce the consistency requirements when possible (i.e., the penalty cost is low) and raise them when it matters (i.e., the penalty costs would be too high). The adaptation is driven by a cost model and different strategies that dictate how the system should behave. In particular, they divide the data items into three categories (A, B, C) and treat each category differently depending on the consistency level provided. The A category represents data items for which we need to ensure strong consistency guarantees as any consistency violation would result in large penalty costs, the C category represents data items that can be treated using session consistency as temporary inconsistency is acceptable while the B category comprises all the data items where the consistency requirements vary over time depending on the actual availability of an item. Therefore, the data of this category is handled with either strong or session consistency depending on a statistical-based policy for decision making. Keeton et al. [50] have proposed a similar approach in a system called *LazyBase* that allows users to trade off query performance and result freshness. LazyBase breaks up metadata processing into a pipeline of ingestion, transformation, and query stages which can be parallelized to improve performance and efficiency. By breaking up the processing, LazyBase can independently determine how to schedule each stage for a given set of metadata, thus providing more flexibility than existing monolithic solutions. LazyBase uses models of transformation and query performance to determine how to schedule transformation operations to meet users' freshness and performance goals and to utilize resources efficiently.

Florescu and Kossmann [51] argued that in cloud environments, the main metric that needs to be optimized is the cost as measured in dollars. Therefore, the big challenge of data management applications is no longer on how fast a database workload can be executed or whether a particular throughput can be achieved; instead, the challenge is on how many machines are necessary to meet the performance requirements of a particular workload. This argument fits well with the rule of thumb calculation which has been proposed by Jim Gray regarding the opportunity costs of distributed computing in the Internet as opposed to local computations [52]. According to Gray's rule, $1 USD equals 1 GB sent over WAN or alternatively eight hours CPU processing time. Therefore, Gray reasons that except for highly processing-intensive applications outsourcing computing tasks into a distributed environment does not pay off because network traffic fees outnumber the savings in processing power. In principle, calculating the tradeoff between basic computing services can be useful to get a general idea of the economies involved. This method can easily be applied to the pricing schemes of cloud computing providers (e.g Amazon, Google). Li et al. [43] proposed a framework to estimate and compare the performance and costs of deploying an application on different cloud providers. This framework characterizes the common set of services provided by cloud providers such as: computation, storage and networking services. The initial results of this framework indicate that cloud providers differ significantly in both performance and costs of the services they provide and there is no provider that ace all services. Therefore, a systematic assessment for different providers in hosting applications based on their requirements is highly required.

Florescu and Kossmann [51] have also argued that in the new large scale web applications, the requirement to provide 100 percent read and write availability for all users has overshadowed the importance of the ACID paradigm as the gold standard for data consistency. In these applications, no user is ever allowed to be blocked. Hence, while having strong consistency mechanisms has been considered as a hard and expensive constraint in traditional database management systems, it has turned to be an optimization goal (that can be relaxed) in modern data management systems in order to minimize the cost of resolving inconsistencies. Therefore, it might be better to design a system that it can efficiently deal with resolving inconsistencies rather than having a system that prevents inconsistencies under all circumstances.

Kossmann et al. [53] conducted an end-to-end experimental evaluation for the performance and cost of running enterprise web applications with OLTP workloads on alternative cloud services (e.g. RDS, SimpleDB, S3, Google AppEngine, Azure). The results of the experiments showed that the alternative services varied greatly both in cost and performance. Most services had significant scalability issues. They confirmed the observation that public clouds lack the ability to support the upload of large data volumes. It was difficult for them to upload 1 TB or more of raw data through the APIs provided by the providers. With regard to cost, they concluded that Google seems to be more interested in small applications with light workloads whereas Azure is currently the most affordable service for medium to large services.

With the goal of facilitating performance comparisons of the trade-offs cloud data management systems, Cooper et al. [48] have presented the Yahoo! Cloud Serving Benchmark (YCSB) framework and a core set of benchmarks. The benchmark tool has been made available via open source[39] in order to allow extensible development of additional cloud benchmark suites that represent different classes of applications and to allow evaluating different cloud data management systems.

While NoSQL systems have got a lot of attractions, enterprises are still very cautious to rely on these systems because there are many limitations still need to be addressed such as:

- *Programming Model*: NoSQL systems offer limited support for ad-hoc querying and analysis operations. Therefore, significant programming expertise are usually required even for a simple query. In addition, missing the support of declaratively expressing the important join operation has been always considered one of the main limitations of these systems.
- *Transaction Support*: Transaction management is one of the powerful features of traditional relational database systems. The current limited support (if any) of the transaction notion from NoSQL database systems is considered as a big obstacle towards their acceptance in implementing mission critical systems.
- *Maturity*: Traditional relational database systems are well-know with their high stability and rich functionalities. In contrast, most NoSQL systems are open source projects or in pre-production stages where many key features are either not stable enough or still under development. Therefore, enterprises are still very cautious to deal with this new wave of database systems.
- *Support*: Enterprises look for the assurance that if the system fails, they will be able to get timely and competent support. All RDBMS vendors have great experience in providing high level of enterprise support. In contrast, most NoSQL systems are open source projects. Although there are few firms offering support for NoSQL database systems, these companies are still small start-ups without the global reach, support resources or credibility of an Oracle, Microsoft or IBM.
- *Expertise*: There are millions of developers around the world who are familiar with traditional concepts of relational database systems and programming models in every business domain. On the contrary, almost every NoSQL developer is still in a learning mode. It is natural that this limitation will be addressed over time. However, currently, it is far easier to find experienced RDBMS programmers or administrators than a NoSQL expert.

IV. Cloud Applications: Programming Models

This section provides a survey of the state-of-the-art of the alternative programming models which have been proposed

[39]http://wiki.github.com/brianfrankcooper/YCSB/

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
        result += ParseInt(v);
Emit(AsString(result));
```

Fig. 8. An Example MapReduce Program. [54]

to implement the application logics over cloud data storage systems.

A. MapReduce

As the amount of data that need to processed grows, database administrators face a difficult decision on how they should scale their databases. In practice, there are two main options: 1) *Scaling up*: aims at getting a bigger machine. 2) *Scaling out*: aims at partitioning data across more machines. The scaling up option has the main drawback that large machines are often very expensive and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. For the type of large web application that most people aspire to build, it is either impossible or not cost-effective to run off of one machine. Alternatively, it is both extensible and economical to scale out by adding storage space or increasing the performance through buying another commodity server and add it to the running cluster.

MapReduce is a simple and powerful programming model that enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. [54], [55]. The model is mainly designed to achieve high performance on large clusters of commodity PCs. It isolates the application from the details of running a distributed program, such as issues on data distribution, scheduling, and fault tolerance. In this model, the computation takes a set of input key/value pairs and produces a set of output key/value pairs. The MapReduce abstraction is inspired by the *Map* and *Reduce* functions which are commonly used in the functional languages such as *Lisp*. The Map function takes an input pair and produces a set of intermediate key/value pairs. The MapReduce framework groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function receives an intermediate key I with its set of values and merges them together. Typically just zero or one output value is produced per Reduce invocation. The main advantage of this models is that it allows large computations to be easily parallelized and re-execution to be used as the primary mechanism for fault tolerance. Figure 8 illustrates an example MapReduce program expressed in pseudo-code for counting the number of occurrences of each word in a collection of documents. In this example, the map function emits each word plus an associated mark of occurrences while the reduce function sums together all marks emitted for a particular word.

In principle, the design of the MapReduce framework has considered the following main principles [56]:

- *Low-Cost Unreliable Commodity Hardware*: Instead of using expensive, high-performance, reliable symmetric multiprocessing (SMP) or massively parallel processing (MPP) machines equipped with high-end network and storage subsystems, the MapReduce framework is designed to run on large clusters of commodity hardware. This hardware is managed and powered by open-source operating systems and utilities so that the cost is low.
- *Extremely Scalable RAIN Cluster*: Instead of using centralized RAID-based SAN or NAS storage systems, every MapReduce node has its own local off-the-shelf hard drives. These nodes are loosely coupled in rackable systems connected with generic LAN switches. These nodes can be taken out of service with almost no impact to still-running MapReduce jobs. These clusters are called Redundant Array of Independent (and Inexpensive) Nodes (RAIN).
- *Fault-Tolerant yet Easy to Administer*: MapReduce jobs can run on clusters with thousands of nodes or even more. These nodes are not very reliable as at any point in time, a certain percentage of these commodity nodes or hard drives will be out of order. Hence, the MapReduce framework applies straightforward mechanisms to replicate data and launch backup tasks so as to keep still-running processes going. To handle crashed nodes, system administrators simply take crashed hardware off-line. New nodes can be plugged in at any time without much administrative hassle. There is no complicated backup, restore and recovery configurations like the ones that can be seen in many DBMS.
- *Highly Parallel yet Abstracted*: The most important contribution of the MapReduce framework is its ability to automatically support the parallelization of task executions. Hence, it allows developers to focus mainly on the problem at hand rather than worrying about the low level implementation details such as memory management, file allocation, parallel, multi-threaded or network programming. Moreover, MapReduce's shared-nothing architecture [31] makes it much more scalable and ready for parallelization.

Hadoop[40] is an open source Java software that supports data-intensive distributed applications by realizing the implementation of the MapReduce framework. On the implementation level, the Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g. hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user. Figure 9 illustrates an example of the overall flow of a MapReduce operation which goes through the following sequence of actions:

1) The input files of the MapReduce program is split into M pieces and starts up many copies of the program on a cluster of machines.
2) One of the copies of the program is elected to be the

[40]http://hadoop.apache.org/

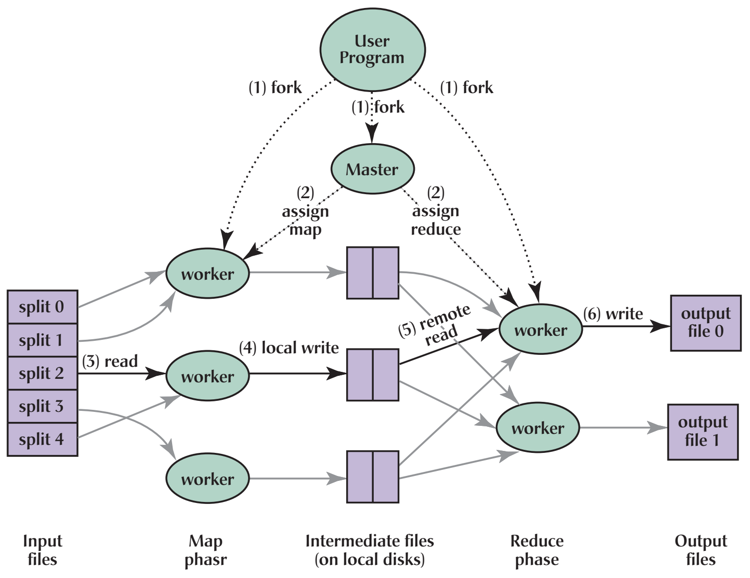


Fig. 9. An Overview of the Flow of Execution a MapReduce Operation. [54]

master copy while the rest are considered as *workers* that are assigned their work by the master copy. In particular, there are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3) A worker who is assigned a map task reads the contents of the corresponding input split and parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4) Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5) When a reduce worker is notified by the master about these locations, it reads the buffered data from the local disks of the map workers which is then sorted by the intermediate keys so that all occurrences of the same key are grouped together. The sorting operation is needed because typically many different keys map to the same reduce task.
6) The reduce worker passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7) When all map tasks and reduce tasks have been completed, the master program wakes up the user program. At this point, the MapReduce invocation in the user program returns back to the user code.

During the execution process, the master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed, independent of if the failure is in the network or locally in the worker. Any map tasks marked completed or in progress by the worker are reset back to their initial idle state and therefore become eligible for scheduling on other workers. Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system. Recognizing the fact that network bandwidth is the most precious resource in distributed data management systems, MapReduce tries to achieve the *data locality* feature by locating the data with the compute node so that data access operation become fast.

From their discussion with some users of the MapReduce framework, Dean and Ghemawat [57] have indicated that MapReduce is very useful for handling data processing and data loading in a heterogenous system with many different storage systems. Moreover, it provides a flexible framework for the execution of more complicated functions than that can be directly supported in SQL. On the other side, some useful lessons has be drawn which need to be addressed such as: 1) MapReduce users should take advantage of natural indices (such as timestamps in log file names) whenever possible. 2) Most MapReduce output should be left unmerged since there is no benefit of merging them if the next consumer is just another MapReduce program. 3) MapReduce users should avoid using inefficient textual formats.

One main limitation of the MapReduce framework is that it does not support the joining of multiple datasets in one task. However, this can still be achieved with additional MapReduce steps. For example, users can map and reduce one dataset and read data from other datasets on the fly. Blanas et al. [58] have reported about a study that followed this approach and evaluated the performance of different distributed join algorithms (e.g. Repartition Join, Broadcast Join) using the MapReduce framework. To tackle the limitation of the join phase in the MapReduce framework, Yang et al. [56] have proposed the Map-Reduce-Merge model that enables processing of multiple datasets. Figure 10 illustrates the framework of this model where the map phase transforms an input key/value pair $(k1, v1)$ into a list of intermediate

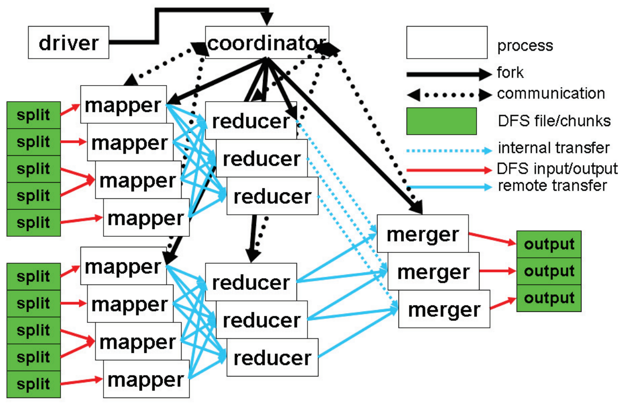
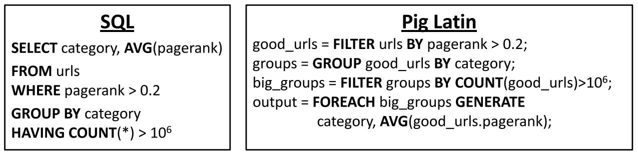


Fig. 10. An Overview of The Map-Reduce-Merge Framework. [56]

key/value pairs $[(k2, v2)]$. The reduce function aggregates the list of values $[v2]$ associated with $k2$ and produces a list of values $[v3]$ which is also associated with k2. Note that inputs and outputs of both functions belong to the same lineage (α). Another pair of map and reduce functions produce the intermediate output $(k3, [v4])$ from another lineage (β). Based on keys $k2$ and $k3$, the merge function combines the two reduced outputs from different lineages into a list of key/value outputs $[(k4, v5)]$. This final output becomes a new lineage (γ). If $\alpha = \beta$ then this merge function does a self-merge which is similar to self-join in relational algebra. The main differences between the processing model of this framework and the original MapReduce is the production of a key/value list from the reduce function instead of just that of values. This change is introduced because the merge function needs input datasets organized (partitioned, then either sorted or hashed) by keys and these keys have to be passed into the function to be merged. In the original framework, the reduced output is final. Hence, users pack whatever needed in $[v3]$ while passing $k2$ for the next stage is not required. Yang et al. [59] have also proposed improving the Map-Reduce-Merge framework by adding a new primitive called *Traverse*. This primitive can process index file entries recursively, select data partitions based on query conditions, and feed only selected partitions to other primitives.

The basic architecture of the MapReduce framework requires that the entire output of each map and reduce task to be materialized into a local file before it can be consumed by the next stage. This materialization step allows for the implementation of a simple and elegant checkpoint/restart fault tolerance mechanism. Condie et al. [60] proposed a modified architecture in which intermediate data is pipelined between operators which widens the domain of problems to which the MapReduce framework can be applied. For example, it can be then used to support *continuous queries* where MapReduce jobs can run continuously, accept new data as it arrives and analyze it immediately. Hence, it allows MapReduce to be used for applications such as event monitoring and stream processing.

Gu and Grossman [61] have reported the following important lessons which they have learned from their experiments with MapReduce framework:



SQL

```
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
```

Pig Latin

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
            category, AVG(good_urls.pagerank);
```

Fig. 11. An Example SQL Query and Its Equivalent Pig Latin Program. [66]

- *The importance of data locality*. Locality is a key factor especially with relying on inexpensive commodity hardware.
- *Load balancing and the importance of identifying hot spots*. With poor load balancing, the entire system can be waiting for a single node. It is important to eliminate any "hot spots" which can be caused by data access (accessing data from a single node) or network I/O (transferring data into or out of a single node).
- *Fault tolerance comes with a price*. In some cases, fault tolerance introduces extra overhead in order to replicate the intermediate results. For example, in the cases of running on small to medium sized clusters, it might be reasonable to favor performance and re-run any failed intermediate task when necessary.
- *Streams are important*. Streaming is quite important in order to reduce the total running time of MapReduce jobs.

Recently, several research efforts have reported about applying the MapReduce framework for solving challenging data processing problem on large scale datasets in different domains. For example, Wang et al. [62] have presented the *MapDupReducer* system for detecting near duplicates over massive datasets. *Surfer* [63] and *Pregel* [64] systems have been designed to achieve efficient distributed processing of large scale graphs. *Ricardo* [65] is a scalable platform for applying sophisticated statistical methods over huge data repositories.

B. SQL-Like

For programmers, a key appealing feature in the MapReduce framework is that there are only two high-level declarative primitives (*map* and *reduce*) that can be written in any programming language of choice and without worrying about the details of their parallel execution. On the other side, the MapReduce programming model has its own limitations such as:

- Its one-input and two-stage data flow is extremely rigid. As we previously discussed, to perform tasks having a different data flow (e.g. joins or n stages), inelegant workarounds have to be devised.
- Custom code has to be written for even the most common operations (e.g. projection and filtering) which leads to the fact that the code is usually difficult to reuse and maintain.
- The opaque nature of the map and reduce functions impedes the ability of the system to perform optimizations.

Moreover, many programmers could be unfamiliar with the MapReduce framework and they would prefer to use

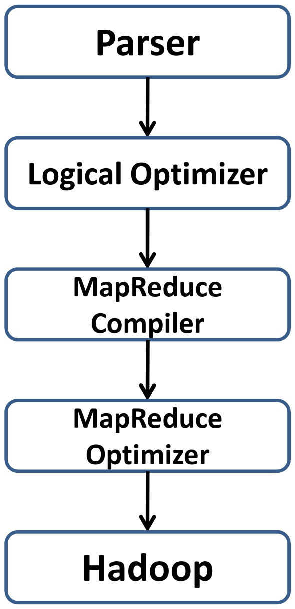


Fig. 12. Pig compilation and execution steps. [67]

SQL (because they are more proficient in) as a high level declarative language to express their task while leaving all of the execution optimization details to the backend engine. In the following subsection we discuss research efforts that have been proposed to tackle these problems and add the SQL flavor on top of the MapReduce framework.

1) Pig Latin: Olston et al. [66] have presented a language called *Pig Latin* that takes a *middle* position between expressing task using high-level declarative querying model in the spirit of SQL and low-level/procedural programming using MapReduce. Pig Latin is implemented in the scope of *Pig* project[41] and is used by programmers at Yahoo! for developing data analysis tasks.

Writing a Pig Latin program is similar to specifying a query execution plan (e.g. a data flow graph). To experienced programmers, this method is more appealing than encoding their task as an SQL query and then coercing the system to choose the desired plan through optimizer hints. In general, automatic query optimization has its limits especially with uncataloged data, prevalent user-defined functions, and parallel execution, which are all features of the data analysis tasks targeted by MapReduce framework. Figure 11 shows an example SQL query and its equivalent Pig Latin program. Given a URL table with the structure $(url, category, pagerank)$, the task of the SQL query is to find each large category and its average pagerank of high-pagerank urls (> 0.2). A Pig Latin program is described as a sequence of steps where each step represents a single data transformation. This characteristic is appealing to many programmers. At the same time, the transformation steps are described using high-level primitives (e.g. filtering, grouping, aggregation) much like in SQL.

Pig Latin has several other features that are important for casual ad-hoc data analysis tasks. These features include support for a flexible, fully nested data model, extensive support for user-defined functions, and the ability to operate over plain input files without any schema information. In particular, Pig Latin has a simple data model consisting of the following four types:

1) *Atom*: An atom contains a simple atomic value such as a string or a number, e.g. “alice”.
2) *Tuple*: A tuple is a sequence of fields, each of which can be any of the data types, e.g. (“alice”, “lakers”).
3) *Bag*: A bag is a collection of tuples with possible duplicates. The schema of the constituent tuples is flexible where not all tuples in a bag need to have the same number and type of fields
   $$\text{e.g.} \left\{ \begin{array}{l} (\text{“alice”, “lakers”}) \\ (\text{“alice”, (“iPod”, “apple”)}) \end{array} \right\}$$
4) *Map*: A map is a collection of data items, where each item has an associated key through which it can be looked up. As with bags, the schema of the constituent data items is flexible However, the keys are required to be data atoms,
   $$\text{e.g.} \left\{ \begin{array}{l} \text{“k1”} \rightarrow (\text{“alice”, “lakers”}) \\ \text{“k2”} \rightarrow \text{“20”} \end{array} \right\}$$

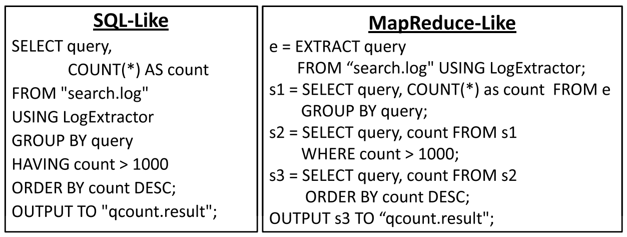
To accommodate specialized data processing tasks, Pig Latin has extensive support for user-defined functions (UDFs). The input and output of UDFs in Pig Latin follow its fully nested data model. Pig Latin is architected such that the parsing of the Pig Latin program and the logical plan construction is independent of the execution platform. Only the compilation of the logical plan into a physical plan depends on the specific execution platform chosen. Currently, Pig Latin programs are compiled into sequences of MapReduce jobs which are executed using the Hadoop MapReduce environment. In particular, a Pig Latin program goes through a series of transformation steps [67] before being executed as depicted in Figure 12. The parsing steps verifies that the program is syntactically correct and that all referenced variables are defined. The output of the parser is a canonical logical plan with a one-to-one correspondence between Pig Latin statements and logical operators which are arranged in a directed acyclic graph (DAG). The logical plan generated by the parser is passed through a logical optimizer. In this stage, logical optimizations such as projection pushdown are carried out. The optimized logical plan is then compiled into a series of MapReduce jobs which are then passed through another optimization phase. The DAG of optimized MapReduce jobs is then topologically sorted and jobs are submitted to Hadoop for execution.

```
    count: table sum of int;
    total: table sum of float;
    sumOfSquares: table sum of float;
    x: float = input;
    emit count $<$- 1;
    emit total $<$- x;
    emit sumOfSquares $<$- x * x;
```

Fig. 13. An Example Sawzall Program. [68]

2) Sawzall: Sawzall [68] is a scripting language used at Google on top of MapReduce. A Sawzall program defines the operations to be performed on a single record of the data. There is nothing in the language to enable examining multiple

[41]http://incubator.apache.org/pig



SQL-Like

```
SELECT query,
        COUNT(*) AS count
FROM "search.log"
USING LogExtractor
GROUP BY query
HAVING count > 1000
ORDER BY count DESC;
OUTPUT TO "qcount.result";
```

MapReduce-Like

```
e = EXTRACT query
    FROM "search.log" USING LogExtractor;
s1 = SELECT query, COUNT(*) as count  FROM e
     GROUP BY query;
s2 = SELECT query, count FROM s1
     WHERE count > 1000;
s3 = SELECT query, count FROM s2
     ORDER BY count DESC;
OUTPUT s3 TO "qcount.result";
```

Fig. 14. Two Equivalent SCOPE Scripts in SQL-Like Style and MapReduce-Like Style. [69]

input records simultaneously, or even to have the contents of one input record influence the processing of another. The only output primitive in the language is the *emit* statement, which sends data to an external aggregator (e.g. Sum, Average, Maximum, Minimum) that gathers the results from each record and correlates and processes the result. The authors argue that aggregation is done outside the language for a couple of reasons: 1) A more traditional language can use the language to correlate results but some of the aggregation algorithms are sophisticated and are best implemented in a native language and packaged in some form. 2) Drawing an explicit line between filtering and aggregation enables a high degree of parallelism and hides the parallelism from the language itself.

Figure 13 depicts an example Sawzall program where the first three lines declare the aggregators *count*, *total* and *sum of squares*. The keyword *table* introduces an aggregator type which are called tables in Sawzall even though they may be singletons. These particular tables are *sum* tables which add up the values emitted to them, *ints* or *floats* as appropriate. The Sawzall language is implemented as a conventional compiler, written in C++, whose target language is an interpreted instruction set, or byte-code. The compiler and the byte-code interpreter are part of the same binary, so the user presents source code to Sawzall and the system runs it directly. It is structured as a library with an external interface that accepts source code and compiles and runs it, along with bindings to connect to externally-provided aggregators. The datasets of Sawzall programs are often stored in Google File System (GFS) [33]. The business of scheduling a job to run on a cluster of machines is handled by software called *Workqueue* which creates a large-scale time sharing system out of an array of computers and their disks. It schedules jobs, allocates resources, reports status, and collects the results.

3) SCOPE: *SCOPE* (Structured Computations Optimized for Parallel Execution) is a scripting language which is targeted for large-scale data analysis and is used daily for a variety of data analysis and data mining applications inside Microsoft [69]. SCOPE is a declarative language. It allows users to focus on the data transformations required to solve the problem at hand and hides the complexity of the underlying platform and implementation details. The SCOPE compiler and optimizer are responsible for generating an efficient execution plan and the runtime for executing the plan with minimal overhead.

Like SQL, data is modeled as sets of rows composed of typed columns. SCOPE is highly extensible. Users can easily define their own functions and implement their own versions of operators: extractors (parsing and constructing rows from a file), processors (row-wise processing), reducers (group-wise processing) and combiners (combining rows from two inputs). This flexibility greatly extends the scope of the language and allows users to solve problems that cannot be easily expressed in traditional SQL. SCOPE provides a functionality which is similar to SQL views. This feature enhances modularity and code reusability. It is also used to restrict access to sensitive data. SCOPE supports writing a program using traditional SQL expressions or as a series of simple data transformations. Figure 14 illustrates two equivalent scripts in the two different styles (SQL-Like and MapReduce-Like) to find from the search log the popular queries that have been requested at least 1000 times. In the MapReduce-Like style, the *EXTRACT* command extracts all query string from the log file. The first *SELECT* command counts the number of occurrences of each query string. The second *SELECT* command retains only rows with a count greater than 1000. The third *SELECT* command sorts the rows on count. Finally, the *OUTPUT* command writes the result to the file "*qcount.result*".

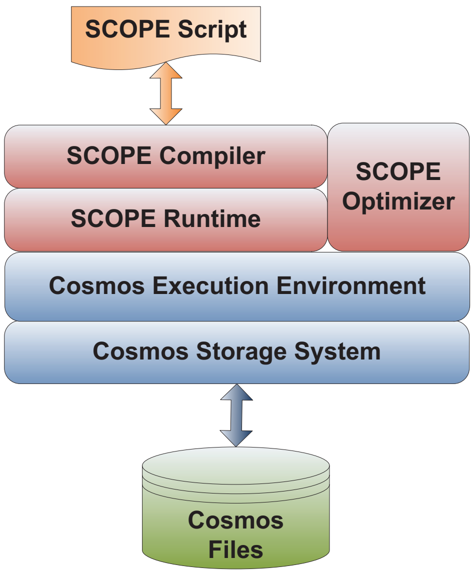


Fig. 15. SCOPE/Cosmos Execution Platform. [69]

Microsoft has developed a distributed computing platform, called *Cosmos*, for storing and analyzing massive data sets. Cosmos is designed to run on large clusters consisting of thousands of commodity servers. Figure 15 shows the main components of the Cosmos platform which is described as follows:

- *Cosmos Storage*: A distributed storage subsystem designed to reliably and efficiently store extremely large sequential files.
- *Cosmos Execution Environment*: An environment for deploy-ing, executing, and debugging distributed applications.
- *SCOPE*: A high-level scripting language for writing data analysis jobs. The SCOPE compiler and optimizer translate scripts to efficient parallel execution plans.

The Cosmos Storage System is an append-only file system

```
FROM (
    MAP doctext USING 'python wc_mapper.py' AS (word, cnt)
    FROM docs
    CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py';
```

Fig. 16. An Example HiveQl Query [73]

that reliably stores petabytes of data. The system is optimized for large sequential I/O. All writes are append-only and concurrent writers are serialized by the system. Data is distributed and replicated for fault tolerance and compressed to save storage and increase I/O throughput. In Cosmos, an application is modeled as a dataflow graph: a directed acyclic graph (DAG) with vertices representing processes and edges representing data flows. The runtime component of the execution engine is called the Job Manager which represents the central and coordinating process for all processing vertices within an application.

The SCOPE scripting language resembles SQL but with C# expressions. Thus, it reduces the learning curve for users and eases the porting of existing SQL scripts into SCOPE. Moreover, SCOPE expressions can use C# libraries where custom C# classes can compute functions of scalar values, or manipulate whole rowsets. A SCOPE script consists of a sequence of commands which are data transformation operators that take one or more rowsets as input, perform some operation on the data, and output a rowset. Every rowset has a well-defined schema to which all its rows must adhere. The SCOPE compiler parses the script, checks the syntax and resolves names. The result of the compilation is an internal parse tree which is then translated to a physical execution plan. A physical execution plan is a specification of Cosmos job which describes a data flow DAG where each vertex is a program and each edge represents a data channel. The translation into an execution plan is performed by traversing the parse tree in a bottom-up manner. For each operator, SCOPE has an associated default implementation rules. Many of the traditional optimization rules from database systems are clearly applicable also in this new context, for example, removing unnecessary columns, pushing down selection predicates and pre-aggregating when possible. However, the highly distributed execution environment offers new opportunities and challenges, making it necessary to explicitly consider the effects of large-scale parallelism during optimization. For example, choosing the right partition scheme and deciding when to partition are crucial for finding an optimal plan. It is also important to correctly reason about partitioning, grouping and sorting properties, and their interaction, to avoid unnecessary computations [70].

As similar to the SCOPE approach, Murray and Hand [71] have presented *Skywriting* as a purely-functional script language with its execution engine for performing distributed and parallel computations. A SkyWriting script can create new tasks asynchronously, evaluate data dependencies and perform unbounded (while-loop) iteration. This enables Skywriting to describe a more general class of distributed computations.

```
SELECT ...
FROM functionname(
    ON table-or-query
    [PARTITION BY expr, ...]
    [ORDER BY expr, ...]
    [clausename(arg, ...) ...]
    )
```

Fig. 17. Basic Syntax of SQL/MR Query Function. [72]

4) SQL/MapReduce: In general, user-defined function (UDF) is a powerful database feature that allows users to customize database functionality. Friedman et al. [72] introduced the SQL/MapReduce (SQL/MR) UDF framework which is designed to facilitate parallel computation of procedural functions across hundreds of servers working together as a single relational database. The framework is implemented as part of the *Aster Data Systems*[42] nCluster shared-nothing relational database.

The framework leverage ideas from the MapReduce programming paradigm to provide users with a straightforward API through which they can implement a UDF in the language of their choice. Moreover, it allows maximum flexibility as the output schema of the UDF is specified by the function itself at query plan-time. This means that a SQL/MR function is polymorphic as it can process arbitrary input because its behavior as well as output schema are dynamically determined by information available at query plan-time. This also increases reusability as the same SQL/MR function can be used on inputs with many different schemas or with different user-specified parameters. In particular, SQL/MR allows the user to write custom-defined functions in any programming language and insert them into queries that otherwise leverage traditional SQL functionality. A SQL/MR function is defined in a manner similar to MapReduce's map and reduce functions.

The syntax for using a SQL/MR function is depicted in Figure 17 where the SQL/MR function invocation appears in the SQL *FROM* clause and consists of the function name followed by a parenthetically enclosed set of clauses. The *ON* clause specifies the input to the invocation of the SQL/MR function. It is important to note that the input schema to the SQL/MR function is specified implicitly at query plan-time in the form of the output schema for the query used in the ON clause.

[42]http://www.asterdata.com/

In practice, a SQL/MR function can be either a mapper (*Row* function) or a reducer (*Partition* function). The definitions of row and partition functions ensure that they can be executed in parallel in a scalable manner. In the *Row Function*, each row from the input table or query will be operated on by exactly one instance of the SQL/MR function. Semantically, each row is processed independently, allowing the execution engine to control parallelism. For each input row, the row function may emit zero or more rows. In the *Partition Function*, each group of rows as defined by the *PARTITION BY* clause will be operated on by exactly one instance of the SQL/MR function. If the *ORDER BY* clause is provided, the rows within each partition are provided to the function instance in the specified sort order. Semantically, each partition is processed independently, allowing parallelization by the execution engine at the level of a partition. For each input partition, the SQL/MR partition function may output zero or more rows.

C. Hybrid Systems

1) Hive: The *Hive* project[43] is an open-source data warehousing solution which has been built by the Facebook Data Infrastructure Team on top of the Hadoop environment [73]. The main goal of this project is to bring the familiar relational database concepts (e.g. tables, columns, partitions) and a subset of SQL to the unstructured world of Hadoop while still maintaining the extensibility and flexibility that Hadoop enjoyed. Thus, it supports all the major primitive types (e.g. integers, floats, doubles, strings) as well as complex types (e.g. maps, lists, structs).

Hive supports queries expressed in a SQL-like declarative language, *HiveQL*[44], and therefore can be easily understood by anyone who is familiar with SQL. These queries are compiled into mapreduce jobs that are executed using Hadoop. In addition, HiveQL enables users to plug in custom MapReduce scripts into queries. For example, the canonical MapReduce word count example on a table of documents (Figure 8) can be expressed in HiveQL as depicted in Figure 16 where the *MAP* clause indicates how the input columns ($doctext$) can be transformed using a user program ('python wc_mapper.py') into output columns ($word$ and cnt). The *REDUCE* clause specifies the user program to invoke ('python wc_reduce.py') on the output columns of the subquery.

HiveQL supports data definition (DDL) statements which can be used to create, drop and alter tables in a database [74]. It allows users to load data from external sources and insert query results into Hive tables via the load and insert data manipulation (DML) statements respectively. However, HiveQL currently does not support the update and deletion of rows in existing tables (in particular, INSERT INTO, UPDATE and DELETE statements) which allows the use of very simple mechanisms to deal with concurrent read and write operations without implementing complex locking protocols.

The metastore component is the Hive's system catalog which stores metadata about the underlying table. This metadata is specified during table creation and reused every time the table is referenced in HiveQL. The metastore distinguishes Hive as a traditional warehousing solution when compared with similar data processing systems that are built on top of MapReduce-like architectures like Pig Latin [67].

2) HadoopDB: Parallel database systems have been commercially available for nearly two decades and there are now about a dozen of different implementations in the marketplace (e.g. Teradata[45], Aster Data[46], Netezza[47], Vertica[48], ParAccel[49], Greenplum[50]). The main aim of these systems is to improve performance through the parallelization of various operations such as loading data, building indexes and evaluating queries. These systems are usually designed to run on top of a shared-nothing architecture [31] where data may be stored in a distributed fashion and input/output speeds are improved by using multiple CPUs, disks in parallel and network links with high available bandwidth.

Pavlo et al. [75] have conducted a large scale comparison between the Hadoop implementation of MapReduce framework and parallel SQL database management systems in terms of performance and development complexity. The results of this comparison have shown that parallel database systems displayed a significant performance advantage over MapReduce in executing a variety of data intensive analysis tasks. On the other side, the Hadoop implementation was very much easier and more straightforward to set up and use in comparison to that of the parallel database systems. MapReduce have also shown to have superior performance in minimizing the amount of work that is lost when a hardware failure occurs. In addition, MapReduce (with its open source implementations) represents a very cheap solution in comparison to the very financially expensive parallel DBMS solutions (the price of an installation of a parallel DBMS cluster usually consists of 7 figures of U.S. Dollars) [76].

The *HadoopDB* project[51] is a hybrid system that tries to combine the scalability advantages of MapReduce with the performance and efficiency advantages of parallel databases [25]. The basic idea behind HadoopDB is to connect multiple single node database systems (PostgreSQL) using Hadoop as the task coordinator and network communication layer. Queries are expressed in SQL but their execution are parallelized across nodes using the MapReduce framework, however, as much of the single node query work as possible is pushed inside of the corresponding node databases. Thus, HadoopDB tries to achieve fault tolerance and the ability to operate in heterogeneous environments by inheriting the scheduling and job tracking implementation from Hadoop. Parallely, it tries to achieve the performance of parallel databases by doing most of the query processing inside the database engine.

Figure 18 illustrates the architecture of HadoopDB which consists of two layers: 1) A data storage layer or the Hadoop

[43]http://hadoop.apache.org/hive/
[44]http://wiki.apache.org/hadoop/Hive/LanguageManual
[45]http://www.teradata.com/
[46]http://www.asterdata.com/
[47]http://www.netezza.com/
[48]http://www.vertica.com/
[49]http://www.paraccel.com/
[50]http://www.greenplum.com/
[51]http://db.cs.yale.edu/hadoopdb/hadoopdb.html

TABLE II
DESIGN DECISIONS OF CLOUD PROGRAMMING ENVIRONMENTS.

System	Data Model	Programming Model	Provider
MapReduce	Key/Value	Procedural Programming	Open Source - Apache Hadoop
Pig Latin	Atom, Tuple, Bag, Map	Procedural, SQL-Like	Open Source - Apache Pig
Sawzall	Tuple	Procedural	Google-Specific
Scope	Tuple	SQL-Like and MapReduce-Like	Microsoft-Specific
SQL/MR	Tuple	SQL-Like and UDF	Aster Data Systems
Hive	Tuple	SQL subset (HiveQL)	Open Source - Apache Hive
HadoopDB	Tuple	SQL and MapReduce	OpenSource - HadoopDB
Teradata	Tuple	SQL (via JDBC) and MapReduce	Teradata DB System

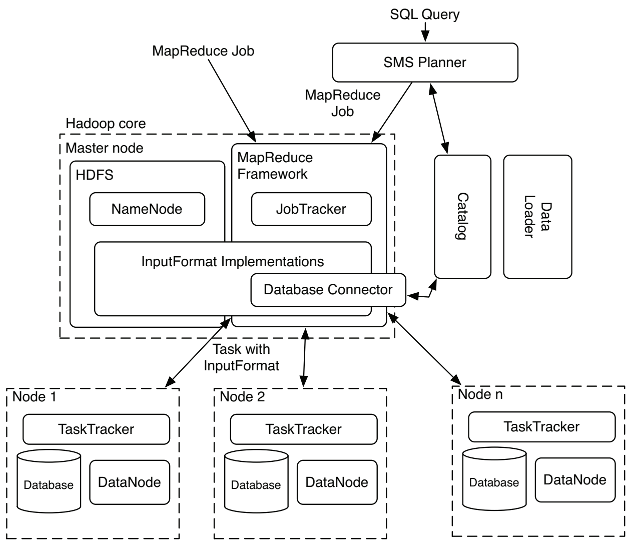


Fig. 18. The Architecture of HadoopDB. [25]

Distributed File System[52] (HDFS). 2) A data processing layer or the MapReduce Framework. In this architecture, HDFS is a block-structured file system managed by a central *NameNode*. Individual files are broken into blocks of a fixed size and distributed across multiple *DataNodes* in the cluster. The NameNode maintains metadata about the size and location of blocks and their replicas. The MapReduce Framework follows a simple master-slave architecture. The master is a single *Job-Tracker* and the slaves or worker nodes are *TaskTrackers*. The JobTracker handles the runtime scheduling of MapReduce jobs and maintains information on each TaskTracker's load and available resources. The *Database Connector* is the interface between independent database systems residing on nodes in the cluster and TaskTrackers. The Connector connects to the database, executes the SQL query and returns results as key-value pairs. The *Catalog* component maintains metadata about the databases, their location, replica locations and data partitioning properties. The *Data Loader* component is responsible for globally repartitioning data on a given partition key upon loading and breaking apart single node data into multiple smaller partitions or chunks. The *SMS planner* extends the HiveQL translator [73] (Section IV-C1) and transforms SQL into MapReduce jobs that connect to tables stored as files in HDFS. Abouzeid et al. [77] have demonstrated HadoopDB in action running two different application types: 1) A semantic web application that provides biological data analysis of protein sequences. 2) A classical business data warehouse.

Teradata [78] has recently started to follow the same approach of integrating Hadoop and parallel databases. It provides a fully parallel load utility called to load Hadoop data to its datawarehouse store. Moreover, it provides a database connector for Hadoop which allows MapReduce programs to directly access Teradata datawarehouses data via JDBC drivers without the need of any external steps of exporting (from DBMS) and loading data to Hadoop. It also provides a *Table* user-defined function (UDF) which can be called from any standard SQL query to retrieve Hadoop data directly from Hadoop nodes in parallel. That means, any relational tables can be joined with the Hadoop data that are retrieved by the Table UDF and any complex business intelligence capability provided by Teradata's SQL engine can be applied to both Hadoop data and relational data. Hence, no extra steps of exporting/importing Hadoop data to/from Teradata datawarehouse are required. Table II summarizes the characteristics of our surveyed cloud programming environments.

Avetisyan et al. [79] have presented the cloud computing testbed for the research community, *Open Cirrus*[53], that federates heterogeneous distributed data centers. Open Cirrus offers a cloud stack consisting of physical and virtual machines, and global services, such as sign-on, monitoring, storage, and job submission. It is currently composed of six sites in North America, Europe, and Asia. Each site consists of a cluster with at least 1000 cores and associated storage. Authorized users can access any Open Cirrus site using the same login credential.

Grossman et al. [80] have presented another large scale testbed and benchmarks called the *Open Cloud Testbed* (OCT)[54]. OCT currently consists of 120 nodes which are distributed over 4 data centers: Baltimore, Chicago (2 locations), and San Diego. OCT is not based on commodity Internet. However, it uses a wide area testbed and the four data centers are connected with a high performance 10Gb/s network, based on a foundation of dedicated lightpaths. Therefore, it can address the requirements of benchmarking the tasks over extremely large data streams. Other open testbeds are also available such as: *Emulab*[55] and *PlanetLab*[56].

[52]http://hadoop.apache.org/hdfs/

[53]https://opencirrus.org/

[54]http://opencloudconsortium.org/testbed/

[55]http://www.emulab.net/

[56]http://www.planet-lab.org/

V. Real-World Case Studies

In principle, many Cloud providers (e.g. Amazon, Microsoft, Google, Rackspace[57], GoGrid[58]) are currently looking for convincing companies to give up building and managing their own data centers and to use their computer capacity instead. Recently, many businesses are started to rethink on how they manage their computing resources.

Given the falling costs of transferring data over the Internet and companies' realization that managing complicated hardware and software building blocks is often a losing proposition, many are willing to outsource some of the job. Therefore, one of the things that Amazon concluded was that cloud computing can allow having access to a workforce that is based around the world and is able to do things that computer algorithms are not really good for. Therefore, Amazon has launched the Mechanical Turk (MTurk) system[59] as a crowd-sourcing Internet marketplace where computer programmers (Requesters) are able to pose tasks known as HITs (Human Intelligence Tasks) such as choosing the best among several photographs of a store-front, writing product descriptions, or identifying performers on music CDs. Workers (Providers) can then browse among existing tasks and complete them for a monetary payment set by the Requester. Requesters can ask that workers fulfill the required qualifications before engaging a task and they can set up a test in order to verify these qualifications. They can also accept or reject the result sent by the Worker which reflects on the Worker's reputation. Recently, Amazon has announced that the MTurk system has over 200K workers in 100 different countries.

In practice, many companies have used the cloud services for different purposes such as: *application hosting* (e.g. 99designs[60], the Guardian News and Media[61], ftopia[62]), *data backup and storage* (e.g. ElephantDrive[63], Jungle Disk[64]), *media hosting* (e.g. fotopedia[65], SmugMug[66]) and *Web hosting* (e.g. Digitaria[67], ShareThis[68]). Moreover, more than 80 companies and organizations (e.g. AOL, LinkedIn, Twitter, Adobe) are listed as users of Hadoop for processing their large scale data sources[69]. On one side, most cloud services have largely been aimed at start-ups, like the legion of Facebook and iPhone applications developers who found that they could rent a first-class computing infrastructure on the fly. In addition, some venture capital firms have made it almost a precondition of investing in startups that they use Amazon's cloud software. On the other side, the U.S federal government has announced the moving of Recovery.gov[70], that specializes in tracking economic recovery, to Amazon's EC2 platform. It is considered as the first federal government production system to run on Amazon EC2. However, it seems that many other federal agencies are planning to be shifting aggressively into full-scale adoption of the cloud services model[71].

Webmail.us[72] uses Amazon S3 to host more than 350K paid mailboxes with 10GB of mail per user. Netflix[73], a company that offers online services for a flat rate fee on DVD rental-by-mail and video streaming in the United States, has decided to move most of its Web technology (e.g. customer movie queues, search tools) to Amazon. Netflix has recently announced that it has more than 10m subscribers, over 100K DVD titles, 50 distribution centers and attracts over 12K instant titles. VISA[74] has also announced that they are using the Hadoop framework for analyzing its massive volumes of data and for applying analytic models to individual clients and not only for client segments.

Last.fm[75] is an Internet radio and music community website that offers many services to its users such as: free music streams and downloads, music and event recommendation and personalized charts. Currently, there are 25 million users generating huge amount of data that need to be processed. This data is processed to make many decisions about users' musical states and compatibility, and artist and track similarity. In addition, it produces many different types of charts such as weekly charts for track per country or per user.

In 2007, New York Times launched a project named *Times-Machine*[76]. The aim of this project was to build a service that provides access to any New York Times issue since 1851. Hence, the bulk of 11 million articles had to be served in the form of PDF files. To tackle the challenge of converting 4 Terabyte of source data into PDF, the project members decided to make use of Amazon's Web Services Elastic Compute Cloud (EC2) and Simple Storage Service (S3). They uploaded the source data to S3 and started a Hadoop cluster of customized EC2 Amazon Machine Images (AMIs). With 100 EC2 AMIs running in parallel, it was possible to complete the task of reading the source data from S3, converting it to PDF and storing it back to S3 within 36 hours.

VI. Conclusions

We live in the data age. In the last two decades, the continuous increase of computational power has produced an overwhelming flow of data. According to IDC, the size of the *digital universe* was about 0.18 zettabyte in 2006 and it is forecasting a tenfold growth by the end of 2011 to 1.8 zettabyte (a zattabyte is one billion terabytes). The result of this is the appearance of a clear gap between the amount of data that is being produced and the capacity of traditional systems to store, analyze and make the best use of this data. Cloud computing has gained much momentum in recent years due to its economic advantages. In particular, cloud computing has promised a number of advantages for its hosting to the deployments of data-intensive applications such as:

[57]http://www.rackspace.com/
[58]http://www.gogrid.com/
[59]https://www.mturk.com/mturk/
[60]http://99designs.com/
[61]http://www.guardian.co.uk/iphone
[62]http://www.ftopia.com/
[63]http://www.elephantdrive.com/
[64]https://www.jungledisk.com/
[65]http://www.fotopedia.com/
[66]http://www.smugmug.com/
[67]http://www.digitaria.com/
[68]http://sharethis.com/
[69]http://wiki.apache.org/hadoop/PoweredBy
[70]http://www.recovery.gov/
[71]http://www.informationweek.com/blog/main/archives/2010/06/federal_agencie.html
[72]http://webmail.us/
[73]http://www.netflix.com/
[74]http://www.slideshare.net/cloudera/hw09-large-scale-transaction-analysis
[75]http://www.last.fm/
[76]http://timesmachine.nytimes.com/

- Reduced time-to-market by removing or simplifying the time-consuming hardware provisioning, purchasing, and deployment processes.
- Reduced cost by following a pay-as-you-go business model.
- Reduced operational cost and pain by automating IT tasks such as security patches and fail-over.
- Unlimited (virtually) throughput by adding servers if the workload increases.

In this paper, we highlighted the main goals and basic challenges of deploying data intensive applications in cloud environments. We provided a survey on numerous approaches and mechanisms of tackling these challenges and achieving the required goals. We analyzed the various design decisions of each approach and its suitability to support certain class of applications and end-users. It was possible to note that several aspects of the approaches and mechanisms affect the management of network resources. Besides, the network performance can affect the performance of the data management mechanisms. A discussion of open issues pertaining to finding the right balance between scalability, consistency, economical aspects of the trade-off design decisions is provided. Finally, we reported about some real-world applications and case studies that started to realize the momentum of cloud technology.

In general, there are several important classes of existing applications that seems to be more compelling with cloud environments and contribute further to its momentum in the near future [7] such as:

1) *Mobile interactive applications*: Such applications will be attracted to the cloud not only because they must be highly available but also because these services generally rely on large datasets which is difficult to be stored on small devices with limited computing resources. Hence, these large datasets are most conveniently to be hosted in large datacenters and accessed through the cloud on their demand.
2) *Parallel batch processing*: Cloud computing presents a unique opportunity for batch-processing and analytics jobs that analyze terabytes of data and may take hours to finish. If there is enough data parallelism in the application, users can take advantage of the cloud's reduced cost model to use hundreds of computers for a short time costs the same as using a few computers for a long time.
3) *The rise of analytical applications*: While the large database industry was originally dominated by transaction processing, this fact is currently changing. A growing share of companies' resources is now directed to large scale data analysis applications such as: understanding customers behavior, efficient supply chains management and recognizing buying habits. Hence, decision support systems are growing rapidly, shifting the resource balance in database processing from online transaction processing (OLTP) systems to business analytics.
4) *Backend-support for compute-intensive desktop applications*: In general, CPU-intensive applications (e.g. multimedia applications) are among the best candidates for successful deployment on cloud environments. The latest versions of the mathematics software packages Matlab and Mathematica are capable of using cloud computing to perform expensive evaluations. Hence, an interesting alternative model might be to keep the data in the cloud and rely on having sufficient bandwidth to enable suitable visualization and a responsive GUI back to the human user (e.g. offline image rendering or 3D animation applications).

Recently, Amazon's chief executive has predicted that its cloud computing division will one day generate as much revenue as its retail business does now. However, Amazon and other cloud providers need to advance their technology to the level that they can convince big companies to rely on using cloud services and consequently they can achieve this goal.

References

[1] T. Hey, S. Tansly, and K. Tolle, editors. The Fourth Paradigm: Data-Intensive Scientific Discovery. Microsoft Research, October 2009.

[2] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *IEEE Computer*, 39(1):110–112, 2006.

[3] Massimo Lamanna. High-Energy Physics Applications on the Grid. In Lizhe Wang and Wei Jie and Jinjun Chen, editor, *Grid Computing: Infrastructure, Service, and Applications*, pages 433–458. CRC Press, 2009.

[4] Yehia El khatib and Christopher Edwards. A Survey-Based Study of Grid Traffic. In *GridNets '07*, pages 4:1–4:8, 2007.

[5] Daniel M. Batista, Luciano J. Chaves, Nelson L. S. da Fonseca, and A. Ziviani. Performance Analysis of Available Bandwidth Estimation Tools for Grid Networks. *Journal of Supercomputing*, 53:103–121, July 2010.

[6] Gartner. Gartner top ten disruptive technologies for 2008 to 2012. Emerging trends and technologies roadshow, 2008.

[7] M. Armbrust, A. Fox, G. Rean, A. Joseph, R. Katz, A. Konwinski, L. Gunho, P. David, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley View of Cloud Computing, Feb 2009.

[8] L. Gonzalez, L. Merino, J. Caceres, and M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *Computer Communication Review*, 39(1), 2009.

[9] D. Plummer, T. Bittman, T. Austin, D. Cearley, and D. Smith. Cloud computing: Defining and describing an emerging phenomenon. Technical report, Gartner, June 2008.

[10] J. Staten, S. Yates, F. Gillett, W. Saleh, and R. Dines. Is cloud computing ready for the enterprise? Technical report, Forrester Research, March 2008.

[11] P. Mell and T. Grance. Definition of cloud computing. Technical report, National Institute of Standard and Technology (NIST), July 2009.

[12] D. Parkhill. The challenge of the computer utility. Addison-Wesley, 1966.

[13] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-theart and research challenges. *J. Internet Services and Applications*, 1(1), 2010.

[14] Kernal based virtual machine. www.linux-kvm.org.

[15] Vmware esx server. www.vmware.com.

[16] Xensource inc. www.xensource.com.

[17] Ibrahim W. Habib, Qiang Song, Zhaoming Li, and Nageswara S. V. Rao. Deployment of the GMPLS control plane for grid applications in experimental high-performance networks. *IEEE Commun. Mag.*, 44(3):65–73, 2006.

[18] Thomas Lehman, Jerry Sobieski, and Bijan Jabbari. DRAGON: a framework for service provisioning in heterogeneous grid networks. *IEEE Commun. Mag.*, 44(3):84–90, 2006.

[19] Wei Guo, Weiqiang Sun, Yaohui Jin, Weisheng Hu, and Chunming Qiao. Demonstration of Joint Resource Scheduling in an Optical Network Integrated Computing Environment. *IEE Communications Magazine*, 48(5):76–83, 2010.

[20] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-degree Compared. *CoRR*, abs/0901.0131, 2009.

[21] C. Cardenas, M. Gagnaire, V. Lopez, and J. Aracil. Performance Evaluation of the Flow-Aware Networking (FAN) Architecture under Grid Environment. In *IEEE/IFIP NOMS 2008*, pages 481–487, Apr 2008.

[22] Daniel M. Batista, Nelson L. S. da Fonseca, Flavio K. Miyazawa, and Fabrizio Granelli. Self-Adjustment of Resource Allocation for Grid Applications. *Computer Networks*, 52:1762–1781, June 2008.

[23] Daniel M. Batista and Nelson L. S. da Fonseca. A Survey of Self-Adaptive Grids. *IEEE Commun. Mag.*, 48:94–100, Jul 2010.

[24] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database Management as a Service: Challenges and Opportunities. In *ICDE*, 1709–1716, 2009.

[25] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[26] B. Cooper, E. Baldeschwieler, R. Fonseca, J. Kistler, P. Narayan, C. Neerdaels, T. Negrin, R. Ramakrishnan, A. Silberstein, U. Srivastava, and R. Stata. Building a Cloud for Yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[27] L. Youseff, M. Butrico, and D. Da Silva. Towards a unified Ontology of Cloud Computing. In *GCE*, 2008.

[28] J. Hofstader. Communications as a service. http://msdn.microsoft.com/en-us/library/bb896003.aspx.

[29] Connected services framework. http://www.microsoft.com/serviceproviders/solutions/connectedservicesframework.mspx

[30] D. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12, 2009.

[31] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[32] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[33] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[34] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSD*, pages 335–350, 2006.

[35] W. Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, 2009.

[36] 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. *PVLDB*, 1(2):1277–1288, 2008.

[37] 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*, pages 205–220, 2007.

[38] D. Karger, E. Lehman, F. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[39] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. In *SIGMOD*, pages 251–264, 2008.

[40] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U . Erlingsson, P. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed dataparallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[42] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *PODC*, page 5, 2009.

[43] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Shopping for a Cloud Made Easy. In *HotCloud, USENIX Workshop*, 2010.

[44] Asfandyar Qureshi, Rick Weber, Hari Balakrishnan, John Guttag, and Bruce Maggs. Cutting the Electric Bill for Internet-Scale Systems. *SIGCOMM Comput. Commun. Rev.*, 39:123–134, August 2009.

[45] E. Brewer. Towards robust distributed systems (abstract). In *PODC*, page 7, 2000.

[46] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[47] A. Tanenbaum and M. Steen, editors. Distributed Systems: Principles and Paradigms. Prentice Hall, 2002.

[48] E. Tam R. Ramakrishnan B. Cooper, A. Silberstein and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, 2010.

[49] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[50] K. Keeton, C. Morrey, N. Soules, and A. Veitch. LazyBase: freshness vs. performance in information management. *SIGOPS Oper. Syst. Rev.*, 44(1):15–19, 2010.

[51] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[52] J. Gray. Distributed computing economics. Microsoft Research Technical Report MSRTR-2003-24, Microsoft Research, 2003.

[53] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD*, 2010.

[54] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[55] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[56] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.

[57] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[58] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010.

[59] H. Yang and D. Parker. Traverse: Simplified indexing on large MapReduce-Merge clusters. In *DASFAA*, pages 308–322, 2009.

[60] P. Alvaro J. Hellerstein K. Elmeleegy T. Condie, N. Conway and R. Sears. Mapreduce Online. In *NSDI*, 2010.

[61] Y. Gu and R. Grossman. Lessons learned from a year's worth of benchmarks of large data clouds. In *SC-MTAGS*, 2009.

[62] C. Wang, J. Wang, X. Lin, W. Wang, H. Wang, H. Li, W. Tian, J. Xu, and R. Li. MapDupReducer: detecting near duplicates over massive datasets. In *SIGMOD*, pages 1119–1122, 2010.

[63] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *SIGMOD*, pages 1123–1126, 2010.

[64] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[65] S. Das, Y. Sismanis, K. Beyer, R. Gemulla, P. Haas, and J. McPherson. Ricardo: integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[66] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[67] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[68] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[69] 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. *PVLDB*, 1(2):1265–1276, 2008.

[70] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, pages 1060–1071, 2010.

[71] D. Murray and S. Hand. Scripting the cloud with Skywriting. In *HotCloud, USENIX Workshop*, 2010.

[72] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[73] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a mapreduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[74] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[75] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[76] M. Stonebraker, D. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

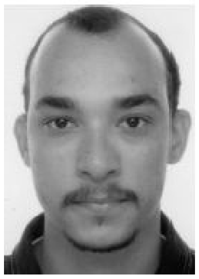
[77] A. Abouzeid, K. Bajda-Pawlikowski, J. Huang, D. Abadi, and A. Silberschatz. HadoopDB in action: Building real world applications. In *SIGMOD*, 2010.

[78] Y. Xu, P. Kostamaa, and L. Gao. Integrating Hadoop and Parallel DBMS. In *SIGMOD*, pages 969–974, 2010.

[79] A. Avetisyan, R. Campbell, I. Gupta, M. Heath, S. Ko, G. Ganger, M. Kozuch, D. O'Hallaron, M. Kunze, T. Kwan, K. Lai, M. Lyons, D. Milojicic, H. Yan Lee, Y. Soh, N. Ming, J. Luke, and H. Namgoong. Open Cirrus: A Global Cloud Computing Testbed. *IEEE Computer*, 43(4):35–43, 2010.

[80] R. Grossman, Y. Gu, M. Sabala, C. Bennet, J. Seidman, and J. Mambretti. The Open Cloud Testbed: A Wide Area Testbed for Cloud Computing Utilizing High Performance Network Services. *CoRR*, abs/0907.4810, 2009.

Sherif Sakr is a Research Scientist in the Managing Complexity Group at National ICT Australia (NICTA). He is also a Conjoint Lecturer in the School of Computer Science and Engineering (CSE) at the University of New South Wales (UNSW), Australia. He held a Visiting Scientist position at Microsoft Research, Redmond. Dr. Sakr received his Ph.D. degree in Computer Science from Konstanz University, Germany in 2007. His research interests is data and information management in general, particularly in areas of indexing techniques, query processing and optimization techniques, graph data management and the large scale data management in cloud platforms.



Daniel M. Batista received his Ph.D. in Computer Science from the State University of Campinas (UNICAMP), Brazil in 2010. He is now a professor at the University of So Paulo. His research interests include network virtualization, traffic engineering, grid networks and cloud computing.

Anna Liu is a Principal Researcher at National ICT Australia (NICTA). She also holds an Associate Professorship at the University of New South Wales, Australia. She previously held Group Management and Principal Architect positions at Microsoft, working on various EAI, SOA and Web 2.0 projects. Anna holds a PhD (Computer Science UNSW) and a BE (Computer Engineering UNSW). She has also held visiting scientist position at the Software Engineering Institute, Carnegie Mellon University, and serves on the Science Advisory board of a number of cloud startups.

Mohammad Alomari is a visiting research fellow at the School of Computer Science and Engineering at the University of Sydney, Australia. He received his Ph.D. degree in May 2009 from the School of Information Technologies at the University of Sydney, Australia. Dr. Alomari's research focuses on developing novel techniques in area of database (e.g. Transaction Management, Data consistency and replication, and Cloud Computing).