Federation in Cloud Data Management: Challenges and Opportunities

Gang Chen, *Member, IEEE,* H. V. Jagadish, *Member, IEEE,* Dawei Jiang, David Maier, *Senior Member, IEEE,* Beng Chin Ooi, *Fellow, IEEE,* Kian-Lee Tan, *Member, IEEE,* Wang-Chiew Tan, *Member, IEEE*

Abstract—Companies are increasingly moving their data processing to the cloud, for reasons of cost, scalability, and convenience, among others. However, hosting multiple applications and storage systems on the same cloud introduces resource sharing and heterogeneous data processing challenges due to the variety of resource usage patterns employed, the variety of data types stored, and the variety of query interfaces presented by those systems. Furthermore, real clouds are never perfectly symmetric – there often are differences between individual processors in their capabilities and connectivity. In this paper, we introduce a federation framework to manage such heterogeneous clouds. We then use this framework to discuss several challenges and their potential solutions.

Index Terms—Cloud Computing, Database Systems, Database Applications

1 Introduction

There is increasing interest for companies to deploy their business systems on the cloud. The typical expectation is that the cloud computing service provider will provision computing resources (e.g., processors, storage, software, and possibly even applications) on-demand over the network. In the same way that the electricity distribution grid revolutionized business use of electricity by allowing companies to access electricity without investing in the infrastructure to produce it, cloud computing allows companies to access powerful computing resources without requiring large IT investments from the companies. Small and medium companies can deploy their business systems on the public cloud, hosted by service providers such as Google or Amazon, and thus eliminate the costs of ownership. Large corporations can build an on-premise private cloud to consolidate computing resources, and thus reduce the overall cost of operations.

A typical first step for deploying a business system in the cloud is to move back-end data management to the cloud. Thus, cloud infrastructures have to support data management, for multiple applications and business units, in an efficient and scalable manner. The standard way to address this requirement is through an elastic scaling-out approach, namely, on-demand provision of more server instances for speeding up data processing tasks or handling increasing workloads. Therefore, a data management system is suitable for cloud deployment only if it is architecturally designed to leverage the elastic scaling-out feature. Initially, only distributed file systems (e.g., GFS [1] and HDFS [2]) and massive parallel processing (MPP) analytical database systems [3] could readily serve as the basis for cloud deployment. With advances in database sharding and NoSQL storage systems (such as BigTable [4], HBase [5] and Cassandra [6]), additional options for cloud deployment are now available. Today, there continues to be a great deal of effort towards the study of the design, architecture, and algorithms for building scalable data management systems [3], [7], [8], [9], [10]. We expect that in the future, more back-end scale-out possibilities will be developed, and hence, even more business systems will migrate to the cloud environment.

Notwithstanding the advantages of deploying business systems in the cloud, hosting multiple data storage systems in the same cloud introduces issues of resource sharing and heterogeneous data processing [11]. The cloud has to manage the variety of storage resource usage patterns (e.g., large file system block size vs. small block size), the variety of data managed (e.g., structured data, unstructured data, or media data such as digital maps and images), and the variety of programming interfaces (e.g., SQL, MapReduce, key-value retrieval) presented by different systems.

- *Gang Chen is with Computer Science College, Zhejiang University, Hangzhou 310027, China.*
- *H. V. Jagadish is with Department of Electrical Engineering and Computer Science, University of Michigan, USA 48109-2121.*
- *Dawei Jiang, Beng Chin Ooi and Kian-Lee Tan are with School of Computing, National University of Singapore, Singapore 117417.*
- *David Maier is with Department of Computer Science, Portland State University, Portland 97201.*
- *Wang-Chiew Tan is with Computer Science Department, University of California, Santa Cruz, USA 95064.*

This article explores the challenges and opportunities in the design and implementation of data management systems on the cloud that can handle the variety in storage usage patterns, the variety of data managed, and the variety of programming interfaces. We first consider the shared-nothing architecture, an architecture currently followed by data management systems for cloud deployment. We then explore the challenges of hosting multiple architecturally similar data management systems on the cloud and suggest directions towards possible solutions. Finally, we introduce epiC, a system that supports transparent resource management and heterogeneous data processing for federated cloud data management systems through a framework design.

The rest of the paper is organized as follows. Section 2 presents the physical characteristics of the cloud environment and the shared-nothing architecture for data management systems hosted on the cloud. Section 3 presents the issues of deploying multiple data management systems on the cloud and desired features for such federated data management. We also discuss why such features are either missing or incompletely supported in current data management systems. Section 4 overviews the epiC system, followed by conclusion in Section 5.

2 Characteristics of Cloud Data Management Systems

In this section, we first present the physical characteristics of cloud computing environments and the architecture of a data management system that is suitable for cloud deployment. Afterwards, we discuss the desired features in providing federated data management service in the cloud.

2.1 Characteristics of Cloud Computing

Under the hood, the cloud computing service provider (e.g., Google, Amazon or an IT department of an organization) manages a large number of computers interconnected by the network for on-demand provisioning presented later. Those computers are hosted either inside a single data center or among multiple data centers connected by a wide area network (WAN). The service provider models a single data center as a *'big'* virtual computer and multiple data centers as a *networked* virtual computer [12]. The virtual computer (big or networked) is then virtualized as a set of virtual server instances of different types based on their computing power, such as CPU and storage capacity. For example, the Amazon EC2 cloud offering provides six different instance types. The most powerful instance type is equipped with 15 GB memory and 8 virtual cores [13].

The service provider serves its customers with virtual server instances based on requests for the instance type and instance number. Usually, the service provider offers a fixed number of instance types to choose from. But, there is no a priori limit on the maximum number of virtual server instances that customers can request. The actual limit (called a soft limit) depends on the capacity of the cloud and varies over time

2.2 The Architecture of Scalable Cloud Data Management Systems

There are two approaches to scale data management systems deployed on the cloud: scale up and scale out. In the scale-up approach, the company deploys the database system on a single virtual server instance. When the workload increases, the business upgrades the virtual server instance to a more powerful type. In the scale-out approach, the company deploys the database on a set of virtual server instances of the same instance type. Scaling involves requesting more virtual server instances for processing the increased workload. Since the service provider poses a fixed limit on the instance types that one can choose and merely a soft limit on the number of virtual server instances that one can request, the scale-out approach is usually preferred. We will focus on scale-out in this paper.

Parallelism has been studied extensively in the database community. A standard approach to parallelism is called *shared-nothing*, and this has turned out to be suitable for building scale-out data management systems [14]. The shared-nothing architecture decomposes the data management system into two components: master and worker. A typical shared-nothing data management system consists of a single master server and any number (at least theoretically) of worker servers. Worker servers are independent from each other. Each worker server independently serves read and write requests to its local data without requiring help from other worker servers. The master server coordinates the worker servers to give users the illusion of data distribution transparency. Figure 1 presents a typical shared-nothing architecture. The shared-nothing architecture has two main advantages. First, it can leverage the elastic scaling out capability provided by cloud services to handle increased workload by adding more worker servers to the system without adversely affecting the higher level applications. Second, since worker servers are independent of each other, the whole data management system (also called a cluster) is highly resilient to worker failures. When a worker fails, only data stored on that worker need to be recovered. Data stored on other workers are not affected. Furthermore, one can speed up the recovery process by involving several "healthy" workers in the recovery process, with each healthy worker recovering over a portion of data stored on the failed worker.

As an example, GFS [1] (and its corresponding open

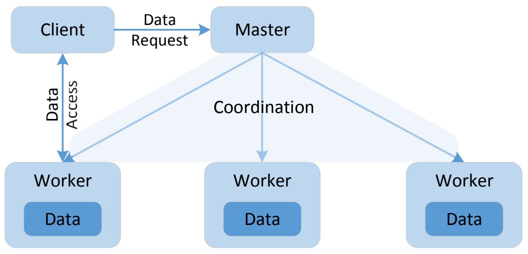


Fig. 1. The Shared-nothing Architecture

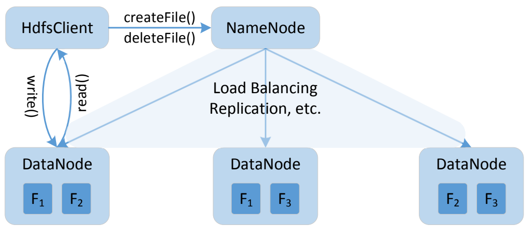


Fig. 2. The HDFS Architecture. F1, F2 and F3 are replicated files.

source implementation HDFS) is a typical shared-nothing storage system. The GFS system provides a service of reading and writing files of arbitrary size. The system consists of a single master and a number of slaves (i.e., workers). A file is stored as a number of fixed size blocks, which are stored on the slaves. Each slave is capable of reading and writing file blocks stored on that slave. The master is responsible for maintaining the meta-information (e.g., file name, file size, block locations) of each file. Figure 2 shows the architecture of HDFS system.

The shared-nothing architecture is not only useful for building scalable storage systems but also suitable for building a distributed data processing system. Similar to a storage system, a distributed data processing system also consists of a single master and any number of independent workers. Each worker is capable of performing a query or a data processing task independently. For a data processing task, the master coordinates the workers to complete the data processing task.

Shared-nothing architectures are commonly used in cloud data management, and we will focus on these for this paper. In other words, we assume that we have a number of independent workers serving local requests and a master that coordinates those workers.

3 Federation Challenges in Cloud Data Management

A cloud system comprises a set of perfectly reliable, identical independent workers, collectively dedicated to the completion of one task of interest. While this simple picture is useful in many contexts, it is inaccurate when it comes to actual deployment of a cloud system. Relaxing each of the simplifying assumptions is a challenge, as we describe in this section. In the next three subsections we respectively consider the challenges due to workers not being identical, due to their having to accomplish a heterogeneous collection of tasks, and due to their being less than perfectly reliable.

3.1 Asymmetric Hardware Capability

It is typical to assume, in scale-out systems, that all nodes are identical in capability. While this situation may be true for a short while following a new cloud installation, it is unlikely to remain true over time. It is often the case that a cloud owner will add services to the cloud over time, and in doing so, s/he will purchase new hardware. If even only a few months have gone by since the previous purchase, technology will likely have progressed, and the new nodes may have better clock speeds, more cache or memory, and so on. It is likely the cloud owner will desire to purchase newer models of hardware available at the time of purchase, rather than try to replicate a previous purchase of older hardware. At the same time, the cloud owner is unlikely to retire the older model nodes until their useful life is over. Hence, we should not expect that individual nodes in a cloud are identical in practice.

In the classic map-reduce paradigm, there is a synchronization point at the reduce phase, which requires that all mappers complete their tasks before the reduce phase begins. If the nodes implementing the map phase are heterogeneous in their capabilities, then this heterogeneity must be taken into account when distributing tasks across the nodes so as to optimize the overall completion time for the map and reduce phases.

3.2 Dynamic Resource Sharing

It is well-recognized that there are two major classes of data management applications: transactional data applications and analytical data applications. Companies develop transactional data applications for their businesses needs (e.g., airline ticket reservations). Transactional data applications tend to have a large number of database reads and writes. Even though the total number of database reads and writes may be large, each read tends to process only a small number of records (a thousand records per read is considered large in most companies) and each write typically only inserts a single record or a small number of records into the database [15]. Companies build analytical data applications to derive insights from the business data for decision making. Such applications favor periodic reads and writes performed in batches. Each query or loading operation can involve retrieval or insertion of a huge number of records (millions or billions).

To support the two kinds of data applications, two kinds of data management systems are built: transactional and analytical. To achieve good performance, both types of data management systems persist data to the underlying file systems or storage devices (e.g., hard disks) in different ways. Analytical data management systems favor sequential reads and writes of a large number of records and thus tend to employ a large file-system block size (tens or hundreds of MB typically) for excellent throughput. On the other hand, transactional data management systems favor random reads and writes of a small number of records and thus, they usually adopt a small file system block size (tens of KB at most) to reduce data transmission time and latency.

Traditionally, the needs of these two different classes of applications are deemed so disparate that they cannot be run efficiently on the same machine. For the same reasons, deploying both transactional and analytical data management systems on the same cloud introduces similar conflicts in storage resource usage. In the shared-nothing architecture, if one allows both kinds of data management systems to share the same set of workers by employing a small file system block size, then one can potentially achieve good resource utilization (space freed by one system can be reclaimed by another system). However, the performance of the analytical data management system will be suboptimal. This is because the large number of insertions, deletions and updates introduced by the transactional database will fragment data files stored on hard disks over time and therefore, prevent the analytical data management system from claiming large consecutive storage blocks. On the other hand, if one separates the two data management systems by placing them on different workers, better performance can be achieved as the data management systems are effectively isolated from each other. However, the overall resource utilization is likely to be low since the storage space released by one system cannot be reused by the other.

Given that different kinds of data management systems favor different resource usage patterns, companies must deploy different file systems on the cloud for different data-persistence requirements. Each file system is dedicated to a specific storage system. Companies must also employ a resource-management system for dynamic resource sharing between those file systems. The resource-management system should balance two conflicting goals: performance isolation – ensuring that each file system operates on a dedicated storage space without being affected by other file systems – and resource utilization – ensuring storage space released by one file system can be reclaimed by another file system.

There are a few systems that support dynamic resource sharing on cloud environment. Cluster management systems such as Mesos [16] guarantee performance isolation on computing devices (e.g., CPU and memory). However, its performance isolation capabilities on storage devices (e.g., hard disks, SSDs) is only in the experimental stages. The project most closely related to this topic is HDFS federation [17]. This system allows companies to deploy multiple HDFS file systems on the same cloud. It allows the same data node (i.e., a worker server of HDFS storing file blocks) to be shared among multiple HDFS file systems. However, the system does not provide performance isolation. A single system cannot gain exclusive access to the underlying storage device.

In addition to sharing a computer node between different storage systems, there is another kind of resource sharing in an enterprise cloud environment, which is sharing a computer node between a computation framework such as MapReduce and a storage system such as GFS. In the latter case, a compute node contributes a fixed number of computing units (called slots) to the computation framework (e.g., MapReduce) for running tasks and a set of storage devices (e.g., disks) to the storage system for storing and retrieving data.

It is well-known that data retrieval or scanning performance dominates the performance of large-scale data analytical jobs. Thus, the aforementioned resource sharing scheme introduces an issue of how to efficiently fetch data from the storage system to the computation framework for processing. The current state of the art approach for handling this problem is to increase data-locality during data processing, namely, launching as many tasks on compute nodes that have those tasks' input as possible. Since data is primarily stored on disks, data locality is usually realized at the disk level (called disk locality). However, due to the advances in networking technology, the disk-locality approach may not work well in an in-house business cloud environment. Recent studies have shown that, with improved commodity network, the read bandwidth from local disk is only slightly (about 8%) faster than the read bandwidth from a remote disk hosted in the same rack [18]. Furthermore, when a data center employs bisection topology [19], the inter-rack read bandwidth is equal to the intra-rack read bandwidth. As a result, the performance of reading remote disks will be comparable to the performance of reading local disks, meaning the disk-locality approach may not deliver better data retrieval performance.

However, the principle of data-locality to improve data retrieval performance still applies. Instead of focusing on disk-locality, we need to focus on memory-locality. That is, to read data from local memory instead of from remote memory and disks, since reading data from local memory is at least two orders of magnitude faster than reading data from disks (local or remote) or remote memory, and the performance gap is unlikely to be affected by advances in network technology [18], [20].

To achieve memory-locality, two problems need to be resolved. First, can data be effectively held in the memory of the cluster? Second, how can we schedule tasks to increase memory-locality? To solve the first problem, a recent proposal suggests a RAMCloud approach, namely using memory as the primary storage device to store all input data and only using disks for backup or archival purposes [21]. Unfortunately, given that companies produce data at a tremendous rate these days and the capacity of disks in today's enterprise cloud environment is several orders of magnitude larger than the size of the memory, it is not viable to entirely replace the disks with memory. For the second problem, a recent work attempts to adopt a dynamical data repartitioning technique to reduce the inter-memory data transmission for repeated jobs [22].

In this article, we argue that a better and more feasible approach for handling the memory-capacity problem is to use memory as a cache. Instead of holding all input data in memory at any time (suggested by RAMCloud), the caching scheme caches data (both input data and intermediate data produced by analytical jobs) in memory as much as possible, and employs an eviction strategy to make space for newly arrived or computed data. The cache is adaptive and designed to facilitate data scheduling. Based on the query workload, the cache management system actively redistributes the cached data to increase memory-locality by ensuring most tasks can read data from local memory instead of remote memory or disks. The caching scheme presents several challenges including scalable meta data management (i.e., how to effectively maintain location information for cached data as the data may be distributed) and an eviction strategy designed for achieving better job-completion performance instead of high hit ratio. These challenges are not well studied so far.

3.2.1 Diversity in Node Query Interfaces

Given that cloud storage systems employ a shared-nothing architecture, data access requests are actually performed by storage workers. Different storage workers present different query interfaces. For example, a distributed file system worker exposes a byte-oriented interface for users to read and write bytes from file blocks; and a NoSQL store worker exposes a key-value interface for accessing key-value pairs. To facilitate programming, application programmers often wrap these native low-level query interface into a high-level data processing interface. In this section, we describe the desired features that such a high-level interface should support if the local data is expected to be processed by a distributed data processing system.

There are two kinds of query interfaces proposed so far: record-oriented and set-oriented. A record-oriented query interface presents users a set of interfaces (i.e., functions) with well defined semantics. The input of each function is a single record. Users specify the data processing logic in the body of the function. The return value of the function could be a record, a scalar value, or a list of records, depending on the semantics of that function. A set-oriented query interface presents a collection of pre-defined operators to users. Each operator takes a single set or a fixed number of sets of records as input and produces a set of records as the output. Users specify their data processing tasks by composing those operators. Here, the definition of a set is similar to the definition of a set in mathematics, except that duplicate records are permitted in some systems.

The advantage of the record-oriented approach is that the query interface allows users to implement arbitrary data-processing logic. There is no hard limit. A disadvantage of the record-oriented approach is that the query interface is not operationally closed. A function is operationally closed if the inputs and outputs are of the same type. For example, the integer arithmetic operations such as plus and minus are operationally closed since both the inputs and outputs of those operators are all integers. If query interfaces are operationally closed, one can easily compose an arbitrary number of functions by piping the output of one function to another and thus write queries and data processing tasks concisely. Furthermore, with operational closure a query can be optimized by a query planner to automatically generate a good evaluation order. This capability further reduces the burden for users to optimize the query performance.

The advantage of the set-oriented approach is that the query interface is operationally closed. The inputs and output of each operator are all sets. However, a disadvantage of set-oriented approach is that the queries and data-processing tasks that users can express are limited by the operators available. For example, if one wants to partition a dataset for parallel processing and the partition operator is missing in the interface, then the user has no systematic way to perform that task and has to resort to an ad-hoc solution. Another disadvantage of the set-oriented approach is that the programming model does not provide a way for users to customize the behavior of pre-defined operators. For example, if the built-in filtering operator is deemed inappropriate for an application's filtering logic, one may want to implement his or her very own filtering logic to replace the default filtering operator. Such modification is typically not allowed or difficult to do in set-oriented querying systems.

Given that there is an unbounded number of potential useful operators desired, we argue that the ideal query interface for per-node query processing should be a mixture of the record-oriented and set-oriented styles with capabilities for expanding the built-in operator set. The query interface should provide two collections of APIs: a record-oriented API for customizing the behavior of the built-in operators, an extension API for introducing user-defined operators to the system and a set-oriented API for composition of both built-in and user-defined operators. To evaluate a query, the optimizer should be able to generate query plans for queries composed by a mixture of built-in operators and user-defined operators.

The desired hybrid query interface and query optimization technique are missing in the current systems. Existing systems only offer a single query interface, which is either record-oriented or set-oriented. The MapReduce system employs a record-oriented query interface. Each local query processing task is either performed in a map or reduce function. The signatures of the two functions are as follows [23].

$$\texttt{map} \quad (k_1, v_1) \rightarrow \texttt{list}(k_2, v_2)$$

$$\texttt{reduce} \quad (k_2, \texttt{list}(v_2)) \rightarrow \texttt{list}(v_2)$$

It is clear that the interface is not operationally closed. The input and output of map or reduce functions are of different types. Therefore, it is impossible to directly feed the output of map functions to reduce functions. Actually, the major service offered by the MapReduce runtime system (i.e., sorting, shuffling and merging) is to transform the output format of mappers to the input format of reducers. Therefore, the runtime system implicitly enforces an evaluation order between mappers and reducers. Complex queries are performed by chaining a number of MapReduce jobs. This evaluation strategy often results in sub-optimal performance [9], [24], [25], [26].

All shared-nothing parallel databases employ a set-oriented query interface for local data processing. Even though the idea of employing user-defined functions for parameterizing built-in operators such as selection and aggregations is well discussed in the database community, such a feature is still missing in most database systems [27]. In addition, the support for user-defined functions is incomplete in all commercial database systems we are familiar with. Additionally, not every operator is allowed to be parameterized. For example, to the best of our knowledge, no commercial database system supports a user-defined join function, even though it is allowed in theory [28]. Furthermore, no commercial database system supports user defined composable operators.

3.2.2 Diversity in the System-Level Query Interface

As discussed previously, in addition to structured data stored in relational data management systems, companies also produce unstructured data stored in distributed file systems and other business data stored in NoSQL storage systems. These storage systems present fundamentally different data access methods, ranging from simple byte reading (file systems), and key-value retrieval (NoSQL stores) to relational query (SQL databases). If one designs the data processing system to be general and thus only assumes the underlying storage system provides a simple byte-oriented or key-value retrieval interface, the resulting system may lose the chance of leveraging advanced query processing provided by SQL database systems. As a result, users may need to manually perform query processing at the application level. If one designs the data processing system to use an advanced query interface for data retrieval, it will be difficult for the system to retrieve data from file systems or NoSQL stores as many advanced query features are missing in those systems. As mentioned previously, to process or analyze data stored in the cloud, a shared-nothing distributed data processing system is used. To protect the investment of businesses in building analytical data applications, the distributed data processing system should be able to process data stored inside a single storage system (called intra-system data processing) and within a set of storage systems (called inter-system data processing). The ability to support both intra and inter-system query processing allows all business analytical data applications to be built under the same framework. Thus, the knowledge and experiences of tuning and programming the data processing systems can be preserved, captured, and shared among developers.

Since both the data processing and storage systems employ shared-nothing architectures, the query-processing or data-storage tasks are decomposed into independent workers. This decomposition poses a major problem in designing effective data retrieval interfaces between the data-processing workers and the data-storage workers. Since there is a variety of query interfaces that can be exposed by a data-storage worker, ranging from record-oriented to set-oriented, to enable data processing over those interfaces, the data processing system should avoid introducing any preference (record or set oriented) in its programming model. Instead, the data processing system should only offer the worker as a *"placeholder"* for hosting the user's data processing program and let the user choose the right interface to retrieve data from the storage worker. In summary, an ideal cloud data processing system should be independent of any specific query models and enable users to write query-aware data-processing workers by fully leveraging query facilities offered by the underlying storage systems.

Current data-processing systems, such as MapReduce and shared-nothing parallel databases, do not support the desired model-independent data-processing style. The MapReduce system enforces a record-oriented approach in programming the map task and reduce task. If the underlying storage system features a powerful SQL engine, one must develop a reader to bridge to the SQL interface from the record-oriented interface. The bridging can only be performed at the map side since, according to the programming model, only map tasks can take input from readers, while reduce tasks cannot. As a result, systems such as HadoopDB [8] must shift part of the relational query processing tasks to the hand-coded MapReduce programs even though the underlying storage worker is capable of performing those queries.

Shared-nothing parallel databases, on the other hand, assume that each storage worker exposes a SQL query interface and therefore focus on coordination of those independent SQL engines. As a result, it is extremely difficult to use the system for analyzing data stored in distributed file systems or NoSQL stores, as workers of those systems do not expose a SQL interface. To process data stored in system without SQL support, one must employ an ETL system to extract data out of the original systems and load the extracted data into a parallel database for analysis. In many cases, the ETL process is either laborious or costly [29]. Even though some recent work partly reduces the loading overhead by pushing certain data processing logic to the native data processing systems, such as MapReduce to perform, in general, such data movement between different data processing systems (e.g., databases, MapReduce) during data processing is unavoidable [30].

3.3 Reliability

Not all nodes will be perfectly reliable. This impacts both the synchronization model and possible data replication, as we consider in the following two subsections, respectively.

3.3.1 Parallelism and Recovery

There are three parallelism models that can be used to build a distributed data processing system on top of a shared-nothing architecture, namely fine-grained, coarse-grained and embarrassing parallelism. These three parallelism models differ in the programming model, communication pattern and recovery and thus are suitable for different applications. The embarrassing parallelism model assumes that there is no communication between the execution of subtasks and the synchronization between up-stream tasks and down-stream tasks is achieved by blocking, namely the down-stream tasks wait for the completion of all up-stream tasks to start. An example of such parallelism model is MapReduce. There is no inter-task communication between map tasks (i.e., up-stream tasks) or reduce tasks (i.e., down-stream tasks). But the reduce function must wait for the completion of all map tasks to start. The fine-grained parallelism model assumes that the subtasks have to communicate in order to complete the whole computation and synchronization is required after a majority, but not all, tasks are completed. An example of such parallelism model is Pregel. Coarse-grained parallel systems are in between the two extremes. The computational requirement of an application makes one parallelism model more suitable than the other. For example, for graph processing and iterative processing, fine-grained parallelism is more efficient, while for simple non-repetitive processing, embarrassing parallelism is more efficient. Most systems have been designed to adopt one parallelism model for efficiency and ease of design.

With an increasing number of compute nodes as the data processing systems scale out to improve parallelism, the likelihood of node failure also increases. Existing systems employ either a checkpoint based recovery method or a confined recovery method. The checkpoint based recovery method requires each compute node to periodically and synchronously write its data to the local hard disk as a checkpoint. It requires all the active compute nodes to rollback to the most recent checkpoint when there is a failure, and uses an unused active compute node to replace the failed node and let all the nodes synchronously re-execute all the major steps. The confined recovery method will not rollback all active compute nodes as in the checkpoint based recovery method; instead, it requires every compute node to cache all messages sent to the other nodes, and uses an unused active node to replace the failed node. In this case, only the new node will re-execute all major steps. Other active nodes will just resend the messages to the new node. Since checkpointing and recovery are expensive processes, there have been recent efforts in trying to reduce or do away with checkpointing whenever possible, such as decomposing a task into subtasks that have few dependencies between them, and in speeding up recovery by relying on designs such as leveraging a precomputed recovery index to distribute the recovery process.

3.3.2 Consistency in Data Replication

All cloud storage systems adopt data replication for high availability. Data replication also fits the shared-nothing architecture well, as workers can independently serve the local data and replicate the data with the coordination of the master server. However, the CAP theorem states that, for data replication systems, when the network is partitioned, one can only achieve the 'C' (consistency) or 'A' (availability) but not both [31]. Here, consistency means that reads from the primary copy of the data and any replicas always return the same result; availability means that reads and writes always succeed. Since availability is always desired, in practice, most storage systems go for availability by adopting weaker data consistency [6], [7], [32].

There are three reasons that we need to rethink the weak consistency solution. First, some business applications such as financial systems cannot work with weak consistency storage systems. In such systems, a strong data consistency guarantee is necessary. Second, the proof of the CAP theorem makes an assumption that reading and writing on replicas are permitted. The reason for allowing such permission is to improve performance. The CAP theorem was originally designed for storage systems deployed on a wide area network. In such a setting, allowing reading and writing on a replica closest to the user's computer can dramatically improve performance. However, for a cloud system hosted at a single data center, the service provider typically adopts a tree structured topology for connecting computer servers. The number of hops between each pair of servers is bounded and small (2 or 3 hops at most in most offerings). Therefore, the latency of reading data from its primary copy is not significantly greater than the latency of reading the closest replica. As a result, it is possible to always perform read and write requests on the primary copy and only use replicas for fail-over. This primary-copy-always solution breaks the condition that the CAP theorem assumes. Third, traditionally, the backend storage system is considered to be the only software system that needs to be designed for fault tolerance. Data applications do not need to handle failures. However, we have observed that in the cloud software stack, each tier is capable of handling machine failures. For example, the MapReduce system is designed to be highly resilient to worker failures and network partitioning. When such a fault is detected, the MapReduce system will re-execute the failed tasks on healthy worker nodes. Therefore, it is possible for the storage system to delegate part of the fail-over task to the application framework and thus entirely bypass the CAP problem. For example, when the storage system detects certain primary copies or replicas are lost due to network partition, it notifies the upper-level MapReduce system to re-execute tasks that access those lost copies.

In summary, in a federated setting, a framework design that allows the designer of each storage system to choose the right solution for tradeoff between consistency and availability is preferable to the weak consistency solution.

4 The epiC System

In this section, we provide a brief description of the epiC system [10], [33], which has been designed as an attempt to provide the missing features of resource management and heterogeneous data processing for cloud federated data management and to support various granularity of parallelism, from fine-grained to embarrassingly parallelism. Fig. 3 shows the architecture of the epiC system. The epiC system has five major novel components: E^3 concurrent programming framework, trusted data service, ES^2 storage manager, virtual block service, and performance monitoring module.

At the lowest level, the epiC system provides a resource management framework (called virtual block service) for managing storage devices. The virtual block service manages all storage devices (e.g., disks and memory) and visualizes those devices as a set of virtual blocks. Users can deploy multiple distributed file systems on the same cloud. For each distributed file system, users allocate virtual blocks through virtual block services for that file system to store data. Currently, epiC supports both HDFS and its key-value storage system, ES^2 [34], to store data on virtual blocks. When one virtual block is allocated to a specific distributed file system, other distributed file system cannot access it. Thus, in epiC, I/O performance isolation is achieved at the virtual block level. Furthermore, virtual blocks that are explicitly released by a distributed file system can be reclaimed by other distributed file systems.

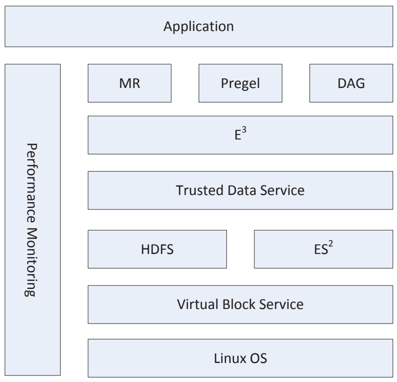


Fig. 3. The Architecture of epiC system

Virtual blocks are replicated for high availability. The epiC system provides a flexible framework design for handling data consistency issues when the network is partitioned. The upper-level distributed file system can choose to receive network partition notifications and handle the consistency issues itself, or ignore the network partition issue and request epiC to tackle data consistency by presenting certain conditions.

The epiC system also introduces a trusted data service for users to protect data privacy by encryption and a subsystem E^3 [35] to manage distributed data processing.

E^3 has been implemented based on the Actor concurrent programming model and designed for large clusters. To facilitate users access to their favorite data processing methods and data retrieval interface (record or set-oriented) for data processing, it provides three plug-ins (i.e., MapReduce, DAG and Pregel) for users to specify data processing logic and a hybrid set-oriented query interface for wrapping the native query interface. The query interface contains a set of built-in operators that are composable and customizable. It also allows users to specify their own user-defined operators.

Finally, the epiC system provides a performance monitoring module for users to profile the performance of their data processing jobs.

5 Conclusion

In this paper, we explored the challenges of building a real cloud system. Most existing cloud systems use an idealized representation of the cloud, which is of value in initial conceptualization and algorithm development. However, one has to go beyond this ideal environment to build and manage a real system. In this paper, we considered issues in deploying multiple data management systems on a cloud infrastructure. We highlighted the resource-sharing and data-processing issues introduced by hosting a federated data management system. We identified the desired features of a cloud-based federated data management system. These features are either missing or improperly supported in current systems. We hope we have convinced the readers of the need for fresh new perspectives on designing cloud data management systems to support these features.

Acknowledgment

This work was supported by the National Research Foundation, Prime Minister's Office, Singapore under Grant No. NRF-CRP8-2011-08. Dr. Dawei Jiang is the corresponding author.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP '03*, 2003, pp. 29–43.

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

[3] D. J. Abadi, "Data management in the cloud: Limitations and opportunities," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 3–12, 2009.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *OSDI*, pp. 205–218, 2006.

[5] http://hadoop.apache.org/hbase.

[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.

[8] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, Aug. 2009.

[9] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using MapReduce," *ACM Computing Survey*, vol. 46, no. 3, pp. 31:1–31:42, Jan. 2014.

[10] D. Jiang, G. Chen, B. C. Ooi, K.-L. Tan, and S. Wu, "epiC: an extensible and scalable system for processing big data," *PVLDB*, vol. 7, no. 7, pp. 541–552, 2014.

[11] B. Cui, H. Mei, and B. Ooi, "Big Data: The driver for innovation in databases," *National Science Review*, vol. 1, no. 1, pp. 27–30, 2014.

[12] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[13] http://aws.amazon.com/ec2/.

[14] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[15] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: A scalable log-structured database system in the cloud," *PVLDB*, vol. 10, pp. 1004–1015, 2012.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *NSDI'11*, 2011, pp. 22–22.

[17] http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop hdfs/Federation.html.

[18] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant," in *HotOS'13*. USENIX Association, 2011, pp. 12–12.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM '08*. ACM, 2008, pp. 63–74.

[20] H. Zhang, B. Tudor, G. Chen, and B. Ooi, "Efficient in-memory data management: An analysis," in *PVLDB*, 2014, pp. 833–836.

[21] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: scalable high-performance storage entirely in DRAM," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, 2010.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12*. USENIX Association, 2012, pp. 2–2.

[23] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI '04*, 2004, pp. 137–150.

[24] D. Jiang, A. K. H. Tung, and G. Chen, "Map-Join-Reduce: Toward scalable and efficient data analysis on large clusters," *TKDE*, vol. 23, no. 9, pp. 1299–1311, 2011.

[25] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *SIGMOD Conference*, 2013, pp. 13–24.

[26] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *ICDE '11*. IEEE Computer Society, 2011, pp. 1151–1162.

[27] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD*. ACM, 2009, pp. 165–178.

[28] E. F. Codd, *The Relational Model for Database Management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

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

[30] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling, "Split query processing in polybase," in *SIGMOD '13*, 2013, pp. 1255–1266.

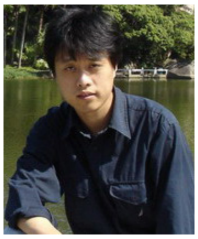
[31] N. Lynch and S. Gilbert, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP '07*. ACM, 2007, pp. 205–220.

[33] http://www.comp.nus.edu.sg/∼epic.

[34] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "ES^2: A cloud data storage system for supporting both OLTP and OLAP," in *ICDE*. IEEE Computer Society, 2011, pp. 291–302.

[35] G. Chen, K. Chen, D. Jiang, B. C. Ooi, L. Shi, H. T. Vo, and S. Wu, "E^3: an elastic execution engine for scalable data processing," *JIP*, vol. 20, no. 1, pp. 65–76, 2012.

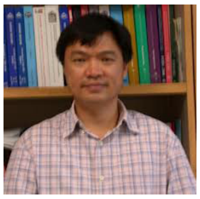


Gang Chen is currently a Professor at the College of Computer Science, Zhejiang University. He received his B.Sc., M.Sc. and Ph.D. in computer science and engineering from Zhejiang University in 1993, 1995 and 1998 respectively. His research interests include databases, information retrieval, information security and computer supported cooperative work. Prof. Gang is also the executive director of Zhejiang University – Netease Joint Lab on Internet Technology.

H. V. Jagadish is currently the Bernard A Galler Collegiate Professor of Electrical Engineering and Computer Science at the University of Michigan. He received his Ph.D. from Stanford University in 1985. His research interests include databases and Big Data.

Dawei Jiang is currently a Senior Research Fellow at the school of computing, National University of Singapore. He received both his B.Sc. and Ph.D. in computer science from the Southeast University in 2001 and 2008 respectively. His research interests include Cloud computing, database systems and large-scale distributed systems.

David Maier is currently the Maseeh Professor of Emerging Technologies at the Portland State University. David Maier received the BA degree in Mathematics and Computer Science from University of Oregon in 1974 and his Ph.D. in Electrical Engineering and Computer Science from Princeton University in 1978. His research interests include Data Streams, Superimposed Information Management, Scientific Information Management and Declarative Programming.



Beng Chin Ooi is currently a distinguished Professor of Computer Science at the National University of Singapore (NUS). He obtained his BSc (1st Class Honors) and PhD from Monash University, Australia, in 1985 and 1989 respectively. His research interests include database system architectures, performance issues, indexing techniques and query processing, in the context of multimedia, spatio-temporal, distributed, parallel, peer-to-peer, and Cloud database systems and applications.

Kian-Lee Tan is currently a Shaw Professor of Computer Science at the School of Computing, National University of Singapore (NUS). He received his B.S.(First-class Honors), M.S. and Ph.D. in computer science, from the National University of Singapore, in 1989, 1991 and 1994 respectively. His research interests include multimedia information retrieval, query processing and optimization in multiprocessor and distributed systems, database performance, and database security.

Wang-Chiew Tan is currently a Professor of Computer Science at the University of California, Santa Cruz. She received her B.Sc. (First Class) in Computer Science from the National University of Singapore in 1996 and her Ph.D. in Computer Science from the University of Pennsylvania in 2002. Her research interests include database systems, data provenance and information integration.