# Providing Scalable Database Services on the Cloud

Chun Chen §1, Gang Chen §2, Dawei Jiang #3, Beng Chin Ooi #4, Hoang Tam Vo #5, Sai Wu #6, and Quanqing Xu #7

#National University of Singapore §Zhejiang University, China
3,4,5,6,7{jiangdw, ooibc, voht, wusai, xuqq}@comp.nus.edu.sg
1,2{chenc,cg}@cs.zju.edu.cn

**Abstract.** The Cloud is fast gaining popularity as a platform for deploying Software as a Service (SaaS) applications. In principle, the Cloud provides unlimited compute resources, enabling deployed services to scale seamlessly. Moreover, the pay-as-you-go model in the Cloud reduces the maintenance overhead of the applications. Given the advantages of the Cloud, it is attractive to migrate existing software to this new platform. However, challenges remain as most software applications need to be redesigned to embrace the Cloud.
In this paper, we present an overview of our current on-going work in developing epiC – an elastic and efficient power-aware data-intensive Cloud system. We discuss the design issues and the implementation of epiC's storage system and processing engine. The storage system and the processing engine are loosely coupled, and have been designed to handle two types of workload simultaneously, namely data-intensive analytical jobs and online transactions (commonly referred as OLAP and OLTP respectively). The processing of large-scale analytical jobs in epiC adopts a phase-based processing strategy, which provides a fine-grained fault tolerance, while the processing of queries adopts indexing and filter-and-refine strategies.

## 1 Introduction

Data has become an important commodity in modern business where data analysis facilitates better business decision making and strategizing. However, a substantial amount of hardware is required to ensure reasonable response time, in addition to data management and analytical software. As the company's business grows, its workload outgrows the hardware capacity and it needs to be upgraded to accommodate the increasing demand. This indeed presents many challenges both in terms of technical support and cost, and therefore the Cloud becomes a feasible solution that mitigates the pain.

The web applications, such as online shopping and social networking, are currently the majority of applications deployed in the Cloud. Excellent system scalability, low service response time and high service availability are required for such applications, as they are generating unprecedented massive amounts of data. Therefore, large-scale ad-hoc analytical processing of the data collected from those web services is becoming increasingly valuable to improving the quality and efficiency of existing services, and supporting new functional features. However, traditional online analytical processing (OLAP) solutions, such as parallel database systems and data warehouses, fall short of scaling dynamically with load and need.

Typically, OLTP (online transaction processing) and OLAP workloads are often handled separately by separate systems with different architectures – RDBMS for OLTP and data warehousing system for OLAP. To maintain the data freshness between these two systems, a data extraction process is periodically performed to migrate the data from the RDBMS into the data warehouse. This system-level separation, though provides flexibility and the required efficiency, introduces several limitations, such as the lack of up-to-date data freshness in OLAP, redundancy of data storage, as well as high startup and maintenance cost.

The need to dynamically provide for capacity both in terms of storage and computation, and to support online transactional processing (OLTP) and online analytical processing (OLAP) in the Cloud demands the re-examination of existing data servers and architecting possibly "new" elastic and efficient data servers for Cloud data management service. In this paper, we present epiC – a Cloud data management system which is being designed to support both functionalities (OLAP and OLTP) within the same storage and processing system. As discussed above, the approaches adopted by parallel databases cannot be directly applied to the Cloud data managements. The main issue is the elasticity. In the Cloud, thousands of compute nodes are deployed to process petabyte of data, and the demand for resources may vary drastically from time to time. To provide data management service in such environment, we need to consider the following issues.

1. Data are partitioned among multiple compute nodes. To facilitate different access patterns, various storage systems are developed for the Cloud. For example, GFS [20] and HDFS [1] are designed for efficient scan or batch access, while Dynamo [16], BigTable [12] and Cassandra [25] are optimized for key-based access. OLAP queries may involve scanning multiple tables, while OLTP queries only retrieve a small number of records. Since it is costly to maintain two storage systems, to support both workloads, a hybrid storage system is required by combining the features of existing solutions.
2. The efficiency and scalability of the Cloud is achieved via parallelism. The query engine must be tuned to exploit the parallelism among the compute nodes. For this purpose, we need to break down the relational operators into more general atomic operators. The atomic operators are tailored for the cluster settings, which are naturally parallelizable. This approach is adopted by MapReduce [15], Hive [35], Pig [30], SCOPE [11], HadoopDB [8] and our proposed epiC system [6]. The query engine should be able to transform a SQL query into the atomic operators and optimize the plans based on cost models.

3. Machine failure is not an exception in the Cloud. In a large cluster, a specific node may fail at any time. Fault tolerance is a basic requirement for all Cloud services, especially the database service. When a node fails, to continue the database service, we need to find the replicas to recover the lost data and schedule the unfinished jobs of the failed node to others. Indeed, the fault tolerance issues affect the designs of both the storage layer and the processing engine of the Cloud data management system.
4. Last but not the least, the Cloud data management system should provide tools for users to immigrate from their local databases. It should support a similar interface as the conventional database systems, which enables the users to run their web services, office softwares and ERP systems without modification.

Due to the distinct characteristics of OLAP and OLTP workload, the query processing engine of epiC is loosely coupled with the underlying storage system and adopts different strategies to process queries from the two different workloads. This enables the query process and the storage process to be deployed independently. One cluster machine can host one or more query processes or storage processes, providing more space for load balancing.

In epiC, OLAP queries are processed via parallel scans, while OLTP queries are handled by indexing and localized query optimization. The OLAP execution engine breaks down conventional database operations such as join into some primitives, and enables them to run in MapReduce-like or filter-and-refine phases. The motivation for this design is that, although the widely adopted MapReduce computation model has been designed with built-in parallelism and fault tolerance, it does not provide data schema support, declarative query language and cost-based query optimization. To avoid the access contention between the two workloads, we relax the data consistency of OLAP queries by providing snapshot-based results, which are generally sufficient for decision making.

The rest of paper is organized as follows. Section 2 reviews the efforts of previous work and discusses the challenges of implementing a database service in the Cloud. Section 3 presents the design and implementation of our proposed epiC system for Cloud data management service. We conclude the paper in Section 4.

## 2 Challenges of Building Data Management Applications in the Cloud

In this section, we review related work and discuss how they affect the design of epiC system.

## 2.1 Lessons Learned from Previous Work

**Parallel Database Systems** Database systems capable of performing data processing on shared-nothing architectures are called parallel database systems [1]. The systems mostly adopt relational data model and support SQL. To parallelize SQL query processing, the systems employ two key techniques pioneered by GRACE [19] and Gamma [18] projects: 1) horizontal partition of relational tables and 2) partitioned execution of SQL queries.

The key idea of horizontal partitioning is to distribute the tuples of relational tables among the nodes in the cluster based on certain rules or principles so that those tuples can be processed in parallel. A number of partitioning strategies have been proposed, including hash partitioning, range partitioning, and round-robin partitioning [17]. For example, to partition the tuples in a table $T$ among $n$ nodes under the hash-partitioning scheme, one must apply a universal hash function on one or more attributes of each tuple in $T$ in order to determine the node that it will be stored.

To process SQL queries over partitioned tables, the partition based execution strategy is utilized. Suppose we want to retrieve tuples in $T$ within a given date range (e.g., from `'2010-04-01'` to `'2010-05-01'`). The system first generates a query plan $P$ for the whole table $T$, then partitions $P$ into $n$ subquery plans $\{P_1, \ldots, P_n\}$ such that each subquery plan $P_i$ can be independently processed by node $n_i$. All the subquery plans apply the same principles by applying the filtering condition to the tuples stored on the local node. Finally, the intermediate results from each node are sent to a selected node where a merge operation is performed to produce the final results.

Parallel database systems are robust, high-performance data processing platforms. In the past two decades, many techniques have been developed to enhance the performance of the systems, including indexing, compression, materialized views, result caching and I/O sharing. These technologies are matured and well tested. While some earlier systems (e.g., Teradata [2]) have to be deployed on proprietary hardware, recent systems (e.g., Aster [3], Vertica [4] and Greenplum [5]), can be deployed on commodity machines. The ability of deploying on low-end machines makes parallel database systems Cloud-ready. To our knowledge, Vertica and Aster have in fact already released their Cloud editions.

However, despite the fact that some parallel database systems can be deployed on Cloud, these systems may not be able to take full advantage of the Cloud. Cloud allows users to elastically allocate resources from the Cloud and only pay for the resources that are actually utilized. This enables users to design their applications to scale their resource requirements up and down in a pay-as-you-go manner. For example, suppose we have to perform data analysis on two datasets with the size of 1TB and 100GB consecutively. Under the elastic scale-up scheme, we can allocate a 100 node cluster from Cloud to analyze the

[1] In database context, database systems employing shared-memory and shared-disk architectures are also called parallel database systems. In this paper, we only cover shared-nothing parallel database systems since Cloud as of now is mostly deployed on a large shared-nothing cluster.

1TB dataset and shrink down the cluster to 10 nodes for processing the 100GB dataset. Suppose the data processing system is linearly scaled-up, the two tasks will be completed in roughly the same time. Thus, the elastic scale-up capability along with the pay-as-you-go business model results in a high performance/price ratio.

The main drawback of parallel database systems is that they are not able to exploit the built-in elasticity feature (which is deemed to be conducive for startups, small and medium sized businesses) of the Cloud. Parallel database systems are mainly designed and optimized for a cluster with a fixed or fairly static number of nodes. Growing up and shrinking down the cluster requires a deliberate plan (often conducted by a DBA) to migrate data from existing configuration to the new one. This data migration process is quite expensive as it often causes the service to be unavailable during migration and thus is avoided in most production systems. The inflexibility for growing up and shrinking down clusters on the fly affect parallel database systems' elasticity and their suitability for pay-as-you-go business model. Another problem of parallel database systems is their degree of fault tolerance. Historically, it is assumed that node failure is more of an exception than a common case, and therefore only transaction level fault tolerance is often provided. When a node fails during the execution of a query, the entire query must be restarted. As argued in [8], the restarting query strategy may cause parallel database systems not being able to process long running queries on clusters with thousands of nodes, since in these clusters hardware failures are common rather than exceptional. Based on this analysis, we argue that parallel database systems are best suitable for applications whose resource requirements are relatively static rather than dynamic. However, many design principles of parallel database systems could form the foundation for the design and optimization of systems to be deployed in the Cloud. In fact, we have started to witness the introduction of declarative query support and cost based query processing into MapReduce-based systems.

**MapReduce-based Systems** MapReduce [15], developed in Google, is a programming model and associated implementation for processing datasets on shared-nothing clusters. This system was designed as a purely data processing system with no built-in facilities to store data. This "*pure processing engine*" design is in contrast to parallel database systems which are equipped with both processing engine and storage engine.

Although MapReduce has been designed to be independent of the underlying data storage system, the system makes at least one assumption about the underlying storage system, namely the data stored in the storage system are already (or can be) horizontally partitioned and the analytical program can be independently launched on each data split [2].

[2] In this perspective, MapReduce actually shares the same key techniques with parallel database systems for parallel data processing: 1) horizontal data partitioning and 2) partitioned execution.

The MapReduce programming model consists of two user specified functions: `map()` and `reduce()`. Without loss of generality, we assume that the MapReduce system is used to analyze data stored in a distributed file system such as GFS and HDFS. To launch a MapReduce job, the MapReduce runtime system first collects partitioning information of the input dataset by querying the metadata of the input files from the distributed file system. The runtime system subsequently creates $M$ map tasks for the input, one for each partition and assigned those map tasks to the available nodes. The node which is assigned a map task reads contents from the corresponding input partition and parses the contents into key/value pairs. Then, it applies the user-specified map function on each key/value pair and produces a list of key/value pairs. The intermediate key/value pairs are further partitioned into $R$ regions in terms of the key and are materialized as local files on the node. Next, the runtime system collects all intermediate results and merges them into $R$ files (intermediate results in the same region are merged together), and launches $R$ reduce tasks to process the files, one for each file. Each reduce task invokes user specified reduce function on each key/value list and produces the final answer.

Compared to the parallel database systems, MapReduce system has a few advantages: 1) MapReduce is a pure data processing engine and is independent of the underlying storage system. This storage independence design enables MapReduce and the storage system to be scaled up independently and thus goes well with the pay-as-you-go business model. The nice property of the Cloud is that it offers different pricing schemes for different grades and types of services. For example, the storage service (e.g., Amazon S3) is charged by per GB per month usage while the computing service (e.g., Amazon EC2) is charged by per node per hour usage. By enabling independent scaling of the storage and processing engine, MapReduce allows the user to minimize the cost on IT infrastructure by choosing the most economical pricing schemes. 2) Map tasks and reduce tasks are assigned to available nodes on demand and the number of tasks (map or reduce) is independent of the number of nodes in the cluster. This runtime scheduling strategy makes MapReduce to fully unleash the power of elasticity of the Cloud. Users can dynamically increase and decrease the size of the cluster by allocating nodes from or releasing nodes to the Cloud. The runtime system will manage and schedule the available nodes to perform map or reduce tasks without interrupting the running jobs. 3) Map tasks and reduce tasks are independently executed from each other. There are however communications or dependencies between map tasks or reduce tasks. This design makes MapReduce to be highly resilient to node failures. When a single node fails during the data processing, only map tasks and reduce tasks on the failed node need to be restarted; the whole job needs not to be restarted.

Even though MapReduce has many advantages, it has been noted that achieving those capabilities comes with a potentially large performance penalty. Benchmarking in [31] shows that Hadoop, an open source MapReduce implementation, is slower than two state of the art parallel database systems by a factor of 3.1 to 6.5 on a variety of analytical tasks and that the large performance gap between

MapReduce and parallel databases may offset all the benefits that MapReduce provides. However, we showed that by properly choosing the implementation strategies for various key operations, the performance of MapReduce can be improved by a factor of 2.5 to 3.5 for the same benchmark [23]. The results show that there is a large space for MapReduce to improve its performance. Based on our study, we conclude that MapReduce system is best suitable for large-scale deployment (thousands of nodes) and data analytical applications which demand dynamic resource allocation. The system has not been designed to support real time updates and search as in conventional database systems.

**Scalable Data Management Systems** Providing scalable data management has posed a grand challenge to database community for more than two decades. Distributed database systems were the first general solution that is able to deal with large datasets stored on distributed shared-nothing environment. However, these systems could not scale beyond a few machines as the performance degrades dramatically due to synchronization overhead and partial failures. Therefore it is not surprising that modern scalable Cloud storage systems, such as BigTable [12], Pnuts [13], Dynamo [16], and Cassandra [25], abandon most of the designs advocated by distributed database systems and adopt different solutions to achieve the desired scalability. The techniques widely adopted by these scalable storage systems are: 1) employ simple data model 2) separate meta data and application data 3) relax consistency requirements.

*Simple Data Model.* Different from distributed databases, most of current scalable Cloud storage systems adopt a much simpler data model in which each record is identified by a unique key and the atomicity guarantee is only supported at the single record level. Foreign key or other cross records relationship are not supported. Restricting data access to single records significantly enhance the scalability of system since all data manipulation will only occurred in a single machine, i.e., no distributed transaction overhead is introduced.

*Separation of Meta Data and Application Data.* A scalable Cloud data management system needs to maintain two types of information: meta data and application data. Meta data is the information that is required for system management. Examples of meta data are the mappings of a data partition to machine nodes in the cluster and to its replicas. Application data is the business data that users stored in the system. These systems make a separation between meta data and application data since each type of data has different consistency requirements. In order for the system to operate correctly, the meta data must always be consistent and up-to-date. However, the consistency requirement of application data is entirely dependent on the applications and varies from application to application. As a result, Cloud data management systems employ different solutions to manage meta data and application data in order to achieve scalability.

*Relaxed Consistency.* Cloud data management systems replicate application data for availability. This design, however, introduces non-negligible overhead of synchronization of all replicas during updating data. To reduce the synchronization overhead, relaxed consistency model like eventual consistency (e.g. in Dynamo [16]) and timeline consistency (e.g. in Pnuts [13]) are widely adopted. The detailed comparison between current scalable cloud data serving systems on their supported consistency models and other aspects such as partitioning and replication strategies can be found in [32].

## 2.2 How to Manage the Cloud - The Essential of Cloud

In this section, we describe the desired properties of a Cloud data management system and our design consideration.

**Scalability** There is a trend that the analytical data size is growing exponentially. As an example, Facebook reports that 15TB new data are inserted into their data warehouse every day, and a huge amount of the scientific data such as mechanobiological data and images is generated each day due to the advancement in X-ray technologies and data collection tools (as noted in the second keynote [28] at VLDB 2010). To process such huge amount of data within a reasonable time, a large number of compute nodes are required. Therefore, the data processing system must be able to deploy on very large clusters (hundreds or even thousands of nodes) without much problems.

**Elasticity** As we argued previously, elasticity is an invaluable feature provided by Cloud. The ability of scaling resource requirements up and down on demand results in a huge cost saving and is extremely attractive to any operations when the cost is a concern. To unleash the power of Cloud, the data processing system should be able to transparently manage and utilize the elastic computing resources. The system should allow users to add and remove compute nodes on the fly. Ideally, to speed up the data processing, one can simply add more nodes to the cluster and the newly added nodes can be utilized by the data processing system immediately (i.e., the startup cost is negligible). Furthermore, when the workload is light, one can release some nodes back to the Cloud and the cluster shrinking process will not affect other running jobs such as causing them to abort.

**Fault Tolerance** Cloud is often built on a large number of low-end, unreliable commodity machines. As a result, hardware failure is fairly common rather than exceptional. The Cloud data processing system should be able to highly resilient to node failures during data processing. Single or even a large number of node failures should not cause the data processing system to restart the running jobs.

**Performance** A common consensus in MapReduce community is that scalability can compensate for the performance. In principle, one can allocate more nodes from the Cloud to speed up the data processing. However this solution is not cost efficient in a pay-as-you-go environment and may potentially offset the benefit of elasticity. To maximize the cost saving, the data processing system indeed needs to be efficient.

**Flexibility** It is well accepted that the Cloud data processing system is required to support various kinds of data analytical tasks (e.g., relational queries, data mining, text processing). Therefore, the programming model of the Cloud data processing system must be flexible and yet expressive. It should enable users to easily express any kinds of data analytical logic. SQL is routinely criticized for its insufficient expressiveness and thus is not ideal for Cloud data processing systems. The MapReduce programming model is deemed much more flexible. The `map()` and `reduce()` functions can be used to express any kinds of logic. The problem of the programming model is that MapReduce has no built-in facilities to manage a MapReduce pipeline. Due to its simple programming model (only two functions are involved), all real world data analytical tasks must be expressed as a set of MapReduce jobs (called a MapReduce pipeline). Hence, the synchronization and management of jobs in the MapReduce pipeline poses a challenge to the user.

## 3 epiC, elastic power-aware data intensive Cloud system

A typical web data management system has to process real-time updates and queries by individual users, and as well as periodical large scale analytical jobs. While such operations take place in the same domain, the transactional and periodical analytical processing have been handled differently using different systems or even hardware. Such a system-level separation naturally leads to the problems of data freshness and data storage redundancy. To alleviate such problems, we have designed and implemented epiC, an elastic power-aware data-intensive Cloud platform for supporting both online analytical processing (OLAP) and online transaction processing (OLTP).

Figure 1 shows the architecture of epiC system, which is composed of three main modules, the **Query Interface**, the **Elastic Execution Engine** ($E^3$) and the **Elastic Storage System** ($ES^2$). The query interface provides a SQL-like language for up-level applications. It compiles the SQL query into a set of analytical jobs (for OLAP query) or a series of read and write operations (for OLTP query). $E^3$, a sub-system of epiC, is designed to efficiently perform large scale analytical jobs on the Cloud. $ES^2$ is the underlying storage system, which is designed to provide an always-on data service. In what follows, we shall briefly introduce the implementation of each module.

### 3.1 Query Interface and Optimization

Some systems, such as MapReduce [15] and its open source Hadoop [1], provide a flexible programming model by exposing some primitive functions (e.g. map and reduce). Users can implement their processing logic via customized functions. This design facilitates the development of applications, but does not provide an

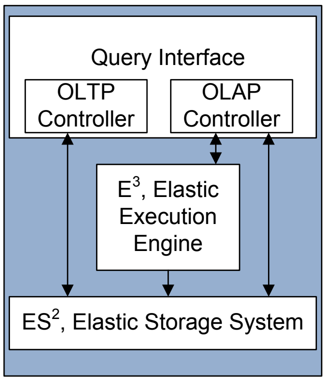

**Fig. 1.** Architecture of epiC

interface for the end-users, who are more familiar with SQL. In epiC, we provide a SQL-like language (in fact, a subset of SQL) as our query language. Currently, non-nested queries and nested queries in the where clause are supported. DBMS users can adapt to epiC without much difficulty.

Inside the query interface module, two controllers, namely OLAP controller and OLTP controller, are implemented to handle different types of queries and monitor the processing status. After a query is submitted to epiC, the query interface first checks whether the query is an analytical query or a simple select query. In the former case, the query is forwarded to the OLAP controller. It transforms the query into a set of $E^3$ jobs. The OLAP controller interacts with $E^3$ to process the jobs. Normally, the jobs are processed by $E^3$ one by one. A specific processing order of jobs is actually a unique query plan. The OLAP controller employs a cost based optimizer to generate a low-cost plan. Specifically, histograms are built and maintained in underlying $ES^2$ system by running some $E^3$ jobs periodically. The OLAP controller queries the metadata catalog of the $ES^2$ to retrieve the histograms, which can be used to estimate the cost of a specific $E^3$ job. We iteratively permute the processing order of the jobs and estimate the cost of each permutation. The one with lowest cost is then selected as the query plan. Based on the optimized query plan, the OLAP controller submits jobs to $E^3$. For each job, the OLAP controller defines the input and output (both are tables in $ES^2$). The processing functions of $E^3$ are auto-generated by the controller. After $E^3$ completes a job, the controller collects the result information and returns to the user if necessary.

If the query is a simple select query, the OLTP controller will take over the query. It first checks the metadata catalog of $ES^2$ to get histogram and index information. Based on the histograms, it can estimate the number of involved records. Then, the OLTP controller selects a proper access method (e.g. index lookup, random read or scan), which reduces the total number of network and disk I/Os. Finally, the OLTP controller calls the data access interface of $ES^2$

to perform the operations. For more complex queries, the OLTP controller will make use of histograms and other statistical information, available join strategies and system loads to generate an efficient query plan, and execute it as in OLAP query processing. Both OLAP and OLTP controller rely on the underlying ES$^2$ system to provide transactional support. Namely, we implement the transaction mechanism in the storage level. Detailed descriptions can be found in the Section 3.3.

Besides above optimization, we also consider multi-query optimizations in both OLAP and OLTP controller. When multiple queries involve the same table, instead of scanning it repeatedly, we group the queries together and process them in a batch. This strategy can significantly improve the performance. A similar approach on MapReduce is adopted in [29].

In addition, since MapReduce [15] has not been designed for generic data analytical workload, most cloud-based query processing systems, e.g. Hive [35], may translate a query into a long chain of MapReduce jobs without optimization, which incurs a significant overhead of startup latency and intermediate results I/Os. Further, this multi-stage process makes it more difficult to locate performance bottlenecks, limiting the potential use of self-tuning techniques. The OLAP controller can exploit data locality, as a result of offline low-cost data indexing, to efficiently perform complex relational operations such as n-way joins. The detailed implementation of these optimization techniques and experimental results are presented in our technical report [21].

## 3.2 E$^3$: Elastic Execution Engine

To perform the data analytical tasks (jobs), OLAP controller submits the jobs to the master node of E$^3$. The master node then distributes the jobs to all available nodes for parallel execution. Figure 2 describes the overall architecture of E$^3$.

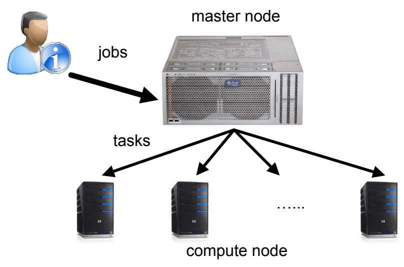

**Fig. 2.** Architecture of E3

Like MapReduce [15], E$^3$ is also a pure data processing system and is independent of the underlying storage systems it operates on. We only assume that the input data can be partitioned into even sized data chunks and the underlying storage system can provide necessary information for the data partitions. This design should enable users to take full advantage of the flexible pricing schemes offered by Cloud providers and thereby minimize their operational costs.

Users may choose to write data analytical tasks in Java [3]. The core abstractions of $E^3$ are implemented as Java classes and can be used in any Java programs. The goal of this design is twofolds: 1) By choosing a general programming language such as Java for writing data analytical tasks, $E^3$ enables users to express arbitrarily complex data processing logic; 2) As $E^3$ jobs are normal Java programs, users can utilize conventional IDE and profiling tools to debug and tune the jobs. Compared to the approach of embedding SQL to a general programming language such as Java for data processing (commonly adopted by major parallel database systems), this design should significantly increase the productivity of development as it is difficult to provide a powerful tool to debug and profile complex SQL. If queries are submitted via the epiC's query interface, the query interface will automatically generate the Java codes for each job based on predefined code templates.

$E^3$ provides three core abstractions for users to specify data processing logics. The first abstraction is `Table<K,V>` which represents the input dataset as a key/value data structure. By default, a table is unordered. There is another class `OrderedTable<K,V>` represents an ordered table. Tables can be loaded from any kinds of data sources (e.g., HDFS or databases). An example of loading an input dataset from HDFS looks like follows [4]:

```
Table<Integer,String> input = load("/root/data", splitter, reader)
```

In this example, the data are loaded from HDFS to the table object `input`. $E^3$ implements the `load()` function using deferred evaluation. When the code is executed, nothing happens. The system just initializes a `Table` object and populates the object with necessary metadata. The actual data retrieval only occurs when the data is being processed. The `splitter` and `reader` specify how to split and read data from a given data partition respectively.

The way to process the data in a `Table` is to call `Table.do()` function. The simplest version of this function requires two parameters: grouper and processor. The grouper specifies a grouping algorithm which groups records according to the key. The processor specifies the logic to process each group of records that the grouper produces. Typically, users only need to specify processor as $E^3$ provides many built-in groupers. However, if the built-in groupers are not sufficient, users can introduce new groupers by implementing the `Grouper` interface. The following code describes how to apply a filter operation on each record of the input table:

```
Table<Integer, String> result = input.do(SingleRecGrouper(), Filter())
```

[3] For standard SQL query, users are suggested to use epiC's query interface, unless they try to apply special optimization techniques for the query.

[4] For clarity, the sample code is an abstraction of the real code

In the above code, `SingleRecGrouper()` is a grouper which places each record of `input` in a separate group. `Filter()` denotes the filtering operation that will be applied to each group of records. In E$^3$, the grouper is the only primitive that will run in parallel. This design simplifies the implementation significantly. Furthermore, E$^3$ enforces groupers to be state-less. Therefore, parallelizing groupers is straightforward. The runtime system just launches several copies of the same grouper code on the slave nodes and runs them in parallel. By default, the number of processes for launching groupers is automatically determined by the runtime system based on the input data size. Users, however, can override the default behavior by providing a specific task allocation strategy. This is accomplished by passing a `TaskAlloc` object as the final argument of `Table.do()`.

The `Table.do()` function returns a new table object as the result. Users can invoke `Table.do()` on the resulting table for further processing if necessary. There is no limitation on the invocation chain. User can even call `Table.do()` in a loop. This is a preferred way to implement iterative data mining algorithms (e.g., k-means) in E$^3$.

The execution of `Table.do()` also follows deferred evaluation strategy. In each invocation, the runtime system just records necessary information in internal data structures. The whole job is submitted to the master node for execution when the user calls `E3Job.run()`. The runtime system merges all the `Table.do()` calls, analyzes and optimizes the call graphs and finally produces the minimum number of groupers to execute.

Compared to MapReduce [15], E$^3$ provides built-in support for specifying and optimizing data processing flows. Our experiences show that specifying complex data processing flows using E$^3$ is significantly easier than specifying the same logic using a MapReduce chain.

### 3.3 ES$^2$: Elastic Storage System

This section presents the design of ES$^2$, the Elastic Storage System of epiC. The architecture of ES$^2$ comprises of three major modules, as illustrated Figure 3, **Data Import Control**, **Data Access Control** and **Physical Storage**. Here we will provide a brief description of each module. For more implementation details of ES$^2$ and experimental results, please refer to our technical report [10].

In ES$^2$, as in conventional database systems, data can be fed into the system via the OLTP operations which insert or update specific data records or via the **data import control** module which supports efficient data bulk-loading from external data sources. The data could be loaded from various data sources such as databases stored in conventional DBMSs, plain or structured data files, and the intermediate data generated by other Cloud applications. The data import control module consists of two sub-components: *import manager* and *write cache*. The import manager implements different protocols to work with the various types of data sources. The write cache resides in memory and is used for buffering the imported data during the bulk-loading process. The data in the

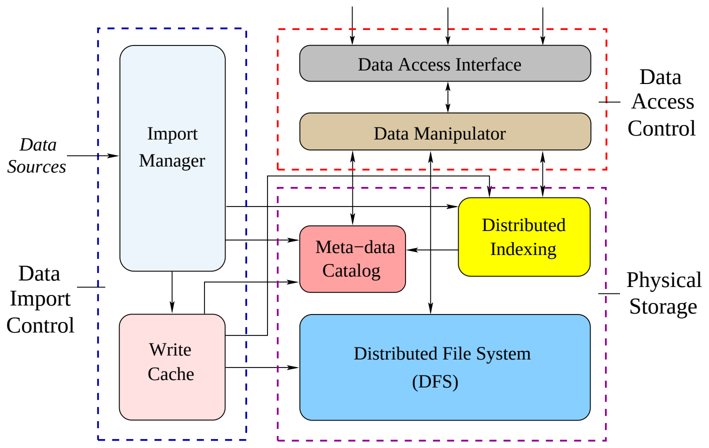

**Fig. 3.** Architecture of ES$^2$

buffer will be eventually flushed to the physical storage when the write cache is full.

The **physical storage module** contains three main components: *distributed file system (DFS)*, *meta-data catalog* and *distributed indexing*. The DFS is where the imported data are actually stored. The meta-data catalog maintains both meta information about the tables in the storage and various fine-grained statistics information required by the data access control module.

The **data access control module** is responsible for performing data access requests from the OLTP/OLAP controller and the E3 engine. It has two sub-components: *data access interface* and *data manipulator*. The data access interface parses the data access requests into the corresponding internal representations that the data manipulator operates on and chooses a near optimal data access plan such as parallel sequential scan or index scan or hybrid for locating and operating on the target data stored in the physical storage module.

Now, we briefly introduce the implementation of ES$^2$ in various aspects including data model, data partitioning scheme, load-adaptive replication, transaction management and secondary indexes.

**Data Model.** We adopt the widely accepted *relational* data model. Although this model has been said to be an overkill in Cloud data management and is replaced by the more flexible *key-value* data model for systems such as BigTable [12], Dynamo [16] and Cassandra [25], we observe that all these systems are transaction-oriented with heavy emphasis on the handling of OLTP queries. On the other hand, systems that focus on ad-hoc analysis of massive data sets (i.e., OLAP queries), including Hive [35], Pig [30] and SCOPE [11], are sticking to the relational data model or its variants. Since we aim to provide effective and efficient supports for both OLTP and OLAP queries and multi-tenancy in the future, we choose the relational model for our storage system.

**Data Partitioning.** ES$^2$ has been designed to operate on a large cluster of shared-nothing commodity machines. Consequently, ES$^2$ employs both vertical and horizontal data partitioning schemes. In this hybrid scheme, columns in a table schema that are frequently accessed together in the query workload are grouped into a column group and stored in a separate physical table. This vertical partitioning strategy facilitates the processing of OLAP queries which often access only a subset of columns within a logical table schema. In addition, for each physical table corresponding to a column group, a horizontal partitioning scheme is carefully designed based on the database workload so that transactions which span multiple partitions are only necessary in the worst case.

**Load-adaptive Replication.** Cloud services need always-on (24×7) data provision, which can be achieved via data replication. A straightforward replication approach is to replicate all data records in the system with the same replication level. However, if the replication level is set to too high, the system storage and the overhead to keep them consistent can be considerably high. Additionally, the data access pattern in web applications is often skewed and changes frequently. Therefore, we develop a two-tier load-adaptive replication strategy to provide both data availability and load balancing function for ES$^2$.

In this replication scheme, each data record is associated with two types of replicas - namely secondary and slave replicas - in addition to its primary copy. The first tier of replication consists of totally $K$ copies of data inclusive of the primary copy and its secondary replicas. The objective of this replication tier is to facilitate the data reliability requirement ($K$ is typically set to small values). At the second tier, frequently accessed records are associated with additional replicas, called slave replicas, as a way to facilitate load balancing for the “hot” queried data. When a primary copy or secondary replica faces a flash crowd query, it will create slave replicas (which become associated with it) to help resolve the sudden surge in the workload. The two-tier load-adaptive replication strategy incurs much less replication costs, which include the storage cost and consistency maintenance cost, than the approach replicating all data records at high replication level, and it can efficiently facilitate load balancing at the same time.

For Cloud data management service, we have to meet the service level agreement (SLA) on various aspects such as service availability and response time. Therefore, synchronous replica consistency maintenance is not suitable due to the high latency for write operations, especially when there are storage node failures or when storage nodes are located in distributed clouds [9]. Instead, we employ the asynchronous replication method in ES$^2$. In particular, the primary copy is always updated immediately, while the update propagation to secondary replicas can be deferred until the storage node has spare network bandwidth. Although this optimistic replication approach guarantees low end-user latency, it might entail other problems such as “lost updates” due to different types of machine failures. For instance, when the primary copy crashes suddenly, there is a possibility that the modification to this copy gets lost because the update has not been propagated to other secondary copies. In ES$^2$, we adopt the writeahead logging scheme and devise a recovery technique to handle the problem of "lost updates". In this manner, ES$^2$ guarantees that updates to the primary copy are durable and eventually propagated to the secondary copies.

**Transaction Management.** In the Cloud context, the management of transactions is a critical and broad research problem. MegaStore [7] and G-Store [14] have started to provide transactional semantics in Cloud storages which guarantee consistency for operations spanning multiple keys clustered in an entity group or a key group. In [26], it has been proposed that Cloud storages can be designed as a system comprising of loosely coupled transactional components and data components. The consistency rationing approach, which categorizes application data into three types and devises a different consistency treatment for each category, has been recently proposed in [24].

In ES$^2$, allowing OLAP and OLTP queries to operate within the same storage system further complicates the problem of transaction management. In our recent study [36], we examined transaction management in distributed environment where distributed data structures and replications are common. ES$^2$ uses replication mainly for load balancing and data reliability requirements, and multi-versioning transaction management technique to support both OLTP and OLAP workloads. Consequently, the OLTP operations access the latest version of the data, while the OLAP data analysis tasks execute on a recent consistent snapshot of the database.

**Secondary Indexes.** For OLTP queries and OLAP queries with high selectivities, it is not efficient to perform sequential or parallel scan on the whole table just to retrieve a few records. However, scanning the whole table is inevitable if query predicates do not contain attributes that have been used to horizontally partition the data. To handle this problem, we support various types of distributed secondary indexes over the data in ES$^2$. Recently, we have proposed two types of distributed secondary indexes for Cloud data: the distributed B$^+$-tree index which supports one-dimensional range queries [39] and the distributed multi-dimensional index which supports multi-dimensional range queries and nearest neighbor (NN) queries [38]. Both approaches share the common key idea of two-level indexing, whereby P2P routing overlays are used to index and guide the search based on the index information published by local storage nodes based on their local indexes.

Different P2P overlays are proposed to handle different types of queries. [37] gives a survey of existing P2P overlays. In reality, we cannot afford the cost of maintaining multiple overlays in the cluster for different types of distributed indexes. Consequently, we develop a *unified indexing framework*, which provides an abstract template overlay based on the Cayley graph model [27]. Based on this framework, the structure and behaviors of different overlays can be customized and mapped onto the template. In current implementation, we have integrated the structures of Chord [34], CAN [33] and BATON [22] in the framework. More P2P overlays will be included in the future.

## 4 Conclusion

Cloud computing is the next generation computation model. It hides the underlying implementation details and provides a resizable resource pool to users, which simplifies the deployment of large-scale applications and services. In this paper, we have reviewed some previous work on building scalable Cloud systems. We have briefly analyzed the advantages and disadvantages of each design. We have applied our background on parallel database systems and observations on other systems to an on-going project, epiC, which aims to provide a flexible framework for supporting various database applications. We have briefly introduced its design and implementations, and we will conduct extensive system benchmarking in the near future.

**Acknowledgement**: We would like to thank other team members from National University of Singapore and Zhejiang University for their valuable contributions, and we would also like to thank Divyakant Agrawal for his valuable comments and the numerous discussions during the course of the implementation of epiC. We would like to thank the conference chairs and PC chairs for their patience.

## References

1. http://hadoop.apache.org/.
2. http://www.teradata.com/.
3. http://www.asterdata.com/.
4. http://www.vertica.com/.
5. http://www.greenplum.com/.
6. *epiC project.* http://www.comp.nus.edu.sg/∼epic/.
7. *Google MegaStore's Presentation at SIGMOD 2008.* http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx.
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.*, 2(1):922–933, 2009.
9. S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: automated data placement for geo-distributed cloud services. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.
10. Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. A cloud data storage system for supporting both oltp and olap. *Technical Report, National University of Singapore, School of Computing. TRA8/10*, 2010.
11. R. Chaiken, B. Jenkins, P.-Å. 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.
12. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

13. 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.*, 1(2):1277–1288, 2008.
14. S. Das, D. Agrawal, and A. E. Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.
15. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
16. 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. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.
17. D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
18. D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.
19. S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 209–219, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
20. S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.
21. F. Guo, X. Li, B. C. Ooi, and K.-L. Tan. Guinea: An efficient data processing framework on large clusters. *Technical Report, National University of Singapore, School of Computing. TRA9/10*, 2010.
22. H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.
23. D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1):472–483, 2010.
24. 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.
25. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.
26. D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. *Proc. VLDB Endow.*, 2(1):265–276, 2009.
27. M. Lupu, B. C. Ooi, and Y. C. Tay. Paths to stardom: calibrating the potential of a peer-based data management system. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 265–278, New York, NY, USA, 2008. ACM.
28. P. Matsudaira. High-end biological imaging generates very large 3d+ and dynamic datasets. *Proc. VLDB Endow.*, 2010.
29. T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1):494–505, 2010.
30. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.
31. 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 '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

32. R. Ramakrishnan. Data management challenges in the cloud. In *Proceedings of ACM SIGOPS LADIS*, http://www.cs.cornell.edu/projects/ladis2009/talks/ramakrishnan-keynote-ladis2009.pdf, 2009.
33. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.
34. I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.
35. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Z. 0002, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.
36. H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *Proc. VLDB Endow.*, 3(1):506–517, 2010.
37. Q. Vu, M. Lupu, and B. C. Ooi. Peer-to-peer computing: Principles and applications. *Springer-Verlag*, 2009.
38. J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 591–602, New York, NY, USA, 2010. ACM.
39. S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient b-tree based indexing for cloud data processing. *Proc. VLDB Endow.*, 3(1):1207–1218, 2010.