ES^2: A Cloud Data Storage System for Supporting Both OLTP and OLAP

Yu Cao[†], Chun Chen[§], Fei Guo[†], Dawei Jiang[†], Yuting Lin[†],
Beng Chin Ooi[†], Hoang Tam Vo[†], Sai Wu[†], Quanqing Xu[†]

[†]*School of Computing, National University of Singapore, Singapore*
{caoyu, guofei, jiangdw, lin36, ooibc, voht, wusai, xuqq}@comp.nus.edu.sg

[§]*College of Computer Science, Zhejiang University, China*
chenc@cs.zju.edu.cn

***Abstract*—Cloud computing represents a paradigm shift driven by the increasing demand of Web based applications for elastic, scalable and efficient system architectures that can efficiently support their ever-growing data volume and large-scale data analysis. A typical data management system has to deal with real-time updates by individual users, and as well as periodical large scale analytical processing, indexing, and data extraction. While such operations may take place in the same domain, the design and development of the systems have somehow evolved independently for transactional and periodical analytical processing. Such a system-level separation has resulted in problems such as data freshness as well as serious data storage redundancy. Ideally, it would be more efficient to apply ad-hoc analytical processing on the same data directly. However, to the best of our knowledge, such an approach has not been adopted in real implementation.**

Intrigued by such an observation, we have designed and implemented epiC, an elastic power-aware data-itensive Cloud platform for supporting both data intensive analytical operations (ref. as OLAP) and online transactions (ref. as OLTP). In this paper, we present ES^2 – the elastic data storage system of epiC, which is designed to support both functionalities within the same storage. We present the system architecture and the functions of each system component, and experimental results which demonstrate the efficiency of the system.

I. Introduction

With the increasing popularity of Web 2.0 applications, massive amounts of different types of data are being generated at an unprecedented scale. Given this rate of continuous growth, coupled with advancement in broadband connectivity, virtualization, and other technologies, the cloud computing model, with its capability to dynamically provide for computation and storage, has emerged as an ideal choice for data-intensive and database-as-a-service computing infrastructures. The need to provide for capacity both in terms of storage and computation, and to support online transactional processing and online analytical processing in the cloud, has given rise to major challenges in architecting elastic and efficient data servers.

The web-service applications provided by Internet companies such as emailing, online shopping and social networking, are all based on online transactions that are essentially similar to those in traditional OLTP (online transaction processing) systems. However, in such web applications, system scalability, service response time and service availability demand higher priority than transactional data consistency, which is the foremost requirement of traditional OLTP systems. Several data management systems for hosting various web applications have been designed and built, including BigTable [1], PNUTS [2], Dynamo [3] and Cassandra [4].

To better support search and data sharing, large-scale ad-hoc analytical processing of data collected from those web services is becoming increasingly valuable to improving the quality and efficiency of existing services, and supporting new functional features. Due to the massive size of web data, traditional OLAP (online analytical processing) solutions (i.e., parallel database systems) fail to scale dynamically with needs. Therefore, both commercial companies and open-source communities have proposed new large-scale data processing systems, such as Hadoop [5], Hive [6], Pig [7] and Dryad [8].

Historically, OLTP and OLAP workloads are handled separately by two systems with different architectures – RDBMS for OLTP and data warehousing system for OLAP. Periodically, data in RDBMS are extracted, transformed and loaded (aka. ETL) into the data warehouse. The system-level separation was motivated by the facts that OLAP is computationally expensive and its execution on a separate system will not compete for resources with the response-critical OLTP operations, and snapshot-based results are generally sufficient for decision making. Although this system-level separation provides flexibility and efficiency, it also results in several inherent limitations, for example, *lack of data freshness in OLAP*, *redundancy of data storage*, as well as *high startup investment and high maintenance cost*. With the emergence of cloud infrastructures, it is therefore timely and desirable to have an integrated system with both high-performance OLTP and OLAP capabilities. Not surprisingly, a main-memory resident database system that handles both OLTP and OLAP has recently been proposed in [9].

Unlike the situation with OLTP and OLAP workloads, the divergence between Web 2.0 application hosting and web data analysis is mainly by design. The storage layer and processing layer are loosely coupled so that the processing layer can read data in any format in bulk and perform the necessary processing to produce the indexes or views required by the applications. The frequency at which an analytical or bulk-

processing task is invoked is a business decision, and its data freshness is therefore determined based on needs. However, such design causes applications to rely heavily on periodically generated meta data (e.g., indexes) due its lack of OLTP support and transaction management. Further, due to design by choice, these systems do not support indexing mechanisms that facilitate ad-hoc query processing, and therefore, are limited in use.

The goal of the epiC project [10] is to develop an **e**lastic **p**ower-aware data-**i**ntensive **C**loud computing platform for providing scalable database services on the cloud. In epiC, two typical types of workloads – data intensive analytical jobs and online transactions (hereafter simply referred as OLAP and OLTP respectively) – are supported to simultaneously and interactively run within the same storage and processing system. In this paper, we focus on the design and functionalities of ES^2 - the elastic storage system of epiC. We note that the processing of OLAP and OLTP queries will ride on the primitive data access interface provided by ES^2. Particularly, OLAP queries are processed via parallel sequential scans, while OLTP queries are handled by indexing and localized query optimization. E^3, the elastic execution engine of epiC, will break down the conventional database operations such as join into some primitives, and enables them to run in 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. The overall architecture of epiC system and how its components work together are described in [11].

ES^2 is essentially designed to operate on a large cluster of shared-nothing commodity machines. It 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.

We also address the problem with OLTP queries and OLAP queries of low record selectivities: It is not efficient to perform full table scans to retrieve only a few qualified records. However, scanning the whole table is inevitable if query predicates do not contain attributes that determine the horizontal data partitioning scheme in the system. To handle this problem, we maintain various types of distributed secondary indexes over the data in ES^2 to facilitate different kinds of queries. For examples, distributed hash indexes support single-dimensional exact-match queries, distributed B$^+$-tree-like indexes support single-dimensional range queries, and distributed multi-dimensional indexes support multi-dimensional range and KNN queries. P2P overlays provide good structures for supporting distributed searches [12]. However, we cannot afford to implement and maintain multiple overlays in the cluster for different types of distributed indexes. Consequently, we have developed a *generalized indexing framework*, which provides an abstract template overlay based on the Cayley graph model. Based on this framework, the structure and behaviors of different overlays can be customized and mapped onto the template, thereby overloading the overlay with multiple search indexes. In addition, we have also developed a distributed bitmap index scheme to support more indexed columns and a wider range of queries. Unlike existing distributed file systems, the ES^2 provides the basic indexes as in a conventional DBMS.

In summary, we design ES^2 – a cloud data storage system to support both OLTP and OLAP workloads. The system provides data access interfaces for upper layer applications. To facilitate efficient processing of ad-hoc queries, its distributed indexing component supports declaration of indexes over the distributed data and hence provides efficient data retrieval. To validate the efficiency of each component, we conduct extensive experiments on a commodity cluster. The results confirm the benefit of providing distributed indexes and accesses to the data for both OLTP and OLAP queries.

The rest of this paper is organized as follows. Section II introduces the previous work relevant to our storage system. Section III illustrates the overall system architecture and briefly discusses the system components, whose details are then elaborated in Section IV, V and VI. In Section VII, we present performance results obtained from the current system snapshot. We conclude in Section VIII.

II. Related Work

Distributed and Parallel Databases. A thorough review on the techniques used by various research and commercial parallel database systems is presented in [13]. epiC is designed to provide dynamic scalability while traditional parallel database systems fall short of scaling dynamically with load and need.

More recently, some commercial parallel database systems have started to integrate the MapReduce framework into their execution engines in order to implement user-defined functions which lack efficient support in conventional parallel database systems. Aster [14] and Greenplum [15] showed that the combination of MapReduce and other relational operators can improve the query processing performance of the system.

Cloud Data Serving Systems. Dynamo [3], a highly available key-value store in cluster environment developed by Amazon, chooses to provide "always writable" data availability with the trade-off eventual consistency. Cassandra [4] is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers. Cassandra provides highly available service with no single point of failure with the use of peer-to-peer model. We notice that the design of Dynamo and Cassandra are mainly focussed for web OLTP workloads. Bigtable [1] also provides record oriented access to very large tables which are distributed in a commodity

cluster. Note that in these systems, the support of secondary indexes has not been reported in the system design. PNUTS [2], the cloud data platform from Yahoo!, provides data storage organized as hashed or ordered tables. In addition, it supports record-level consistency guarantees.

Data Freshness. Data freshness is a very important attribute of data quality and has been extensively studied in various fields [16], [17], [18]. Intuitively, the concept of data freshness is a measure on how old are the data. In traditional OLAP systems (i.e., data warehouses), freshness is studied through the *currency factor* [16] in the context of view materialization, which introduces potential inconsistencies with the sources (i.e. OLTP systems) and thus makes the warehouse data become out-of-date. In ES^2, we measure the data freshness by adopting a *currency metric* [16] for the currency factor. More specifically, when an ad-hoc OLAP job is launched on a dataset that is stored in our storage system and is also being manipulated by OLTP queries, we measure how older is the visible version of the dataset to the analysis of the OLAP job, compared to the up-to-date version of the data applied with OLTP updates arriving during the execution of the OLAP job.

III. System Overview

In this section, we outline the overall architecture of ES^2, as well as the design principles and assumptions based on which our system is built. We briefly outline the functions of each system component and will elaborate the details in latter sections.

Data Model. We exploit the well-understood *relational* data model. Although this model has been criticized as an overkill in cloud data management and is replaced by the more flexible *key-value* data model for systems such as BigTable [1], PNUTS [2], Dynamo [3] and Cassandra [4], we observe that all these systems are transactional-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 [6], Pig Latin [19] and SCOPE [20], are sticking to the relational data model or its variants.

Since the objective of epiC and therefore ES^2 is to provide database-as-a-service and efficiently support both OLTP and OLAP workloads, we choose the relational model for our storage system. However, we also enhance the use of this model in our system according to the characteristics of transactional data via flexible data partitioning schemes as described below.

Data Partitioning. ES^2 employs both vertical and horizontal data partitioning schemes. First, we optionally divide columns of a table schema into several *column groups* based on the query workload. Each column group contains columns that are often accessed together and will be stored separately in a physical table.

Such vertical partitioning scheme benefits the OLAP queries, which often require only a subset of columns within a table. Additionally, transactional accesses to data records often update the values of some columns in a column group. Hence, the vertical partitioning technique improves the overall performance of the system significantly by reducing the I/O cost in most cases.

We then further horizontally partition the data of each column group when the system actually stores the data in the physical table corresponding to this column group. A horizontal partitioning scheme is carefully designed based on the database workload to reduce or even eliminate distributed transactions across storage nodes and thus simplify the transaction management in the system.

Note that since we have designed the vertical partitioning scheme based on the trace of query workload, tuple reconstruction is only necessary in the worst case. Moreover, since each column group still embeds the primary key of data records as one of its componential columns, to re-construct the tuple, ES^2 collects the data in all column groups using the primary key as the the selection predicate.

Transaction Management. In ES^2, with OLAP and OLTP queries operating within the same storage system, the problem will surely become more complicated than in pure OLTP systems. In our recent study [21], we work towards the ultimate objective of managing transactions on data that are accessed simultaneously by both OLTP and OLAP queries. This study investigated various aspects, including the distributed data structure together with the replication and transaction management, in a coherent system. ES^2 adopts similar approach in which the replication is mainly used for load balancing and data reliability requirements, while the multi-versioning transaction management technique supports both OLTP and OLAP workload. In particular, 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.

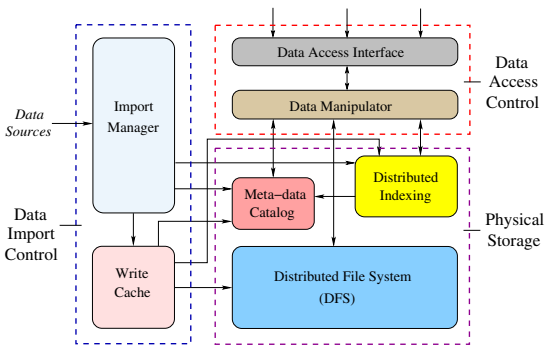


Fig. 1. The architecture of ES^2

Figure 1 illustrates the architecture of ES^2 storage system, which comprises of three major modules: **Data Import Control**, **Data Access Control** and **Physical Storage**.

In ES^2, data can be fed into the system via the transactional 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 come from databases stored in conventional DBMSs, standalone (plain or structured) data files, and intermediate data that are dynamically generated by cloud applications. We also support the write-back of results by E^3, the analytical execution engine

of epiC, which in the meantime retrieves its input data from our storage system. The data import control module consists of two sub-components, namely *import manager* and *write cache*. The import manager has separate protocols to communicate with different data sources. The write cache resides in memory and temporarily buffers the imported data that are eventually flushed to the physical storage when the write cache is full. The write cache serves multiple purposes and will be discussed in detail in Section IV-B. It is notable that the data import control module also supports reversely exporting data from ES^2 to other storage systems.

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. Essentially, we treat it as a raw byte device and rely on its built-in capacities of fault tolerance and load balancing to achieve high data scalability and availability. With minor modifications, this component is replaceable with most of the existing distributed file systems. In our current development, we make use of the Hadoop [5] DFS system (HDFS) for implementation and validation. The meta-data catalog maintains the meta information about the tables in the storage as well as various essential fine-grained statistics information for the data access control module to operate efficiently.

For OLTP queries and OLAP queries with low record selectivities, it is usually un-affordable to sequentially scan the whole dataset for just a few records, even with the help of parallel scanning. With hash-based (or range-based) horizontal data partitioning, the system can be considered as being equipped with a hash index (or range index) over the underlying storage layer. This built-in index can be used to facilitate fast data locating. However, the data partitioning scheme is typically based on a fixed set of table attributes. If the query does not use these attributes in the search conditions, a full table scan is still unavoidable. The solution to this problem is to build secondary indexes. To this end, we have the distributed indexing sub-component which maintains various distributed secondary indexes over the data stored in DFS.

The **data access control** module is responsible for performing data access requests from both OLAP jobs executed by E^3 and OLTP requests submitted by end users. It has two sub-components: *data access interface* and *data manipulator*. The data access interface parses the access requests into corresponding internal representations, with which the data manipulator then chooses the optimal data access plan (parallel sequential scan or index scan or hybrid) for locating and operating on the target data stored in the physical storage.

In Section IV, we will describe the data import control, in conjunction with the physical storage, except for the distributed indexing, which instead will be discussed in Section V. We will describe the data access control in Section VI.

IV. Data Import and Physical Storage

In this section, we discuss how the data import module works together with the physical storage module to import data into our storage system.

A. Import Manager

ES^2 supports three main types of external data sources: database tables in traditional DBMSs, stand-alone files which are either plain (e.g., txt and csv) or semi-structured (e.g., XML and HTML), and intermediate data that is dynamically generated by other applications, such as the results of an analytical task executed by E^3 or by the data stream from a remote server. For each data source, there is a tailored *data adaptor*, through which the import manager communicates with the data source.

The import manager works as follows. Given a data source whose data are to be imported, the manager creates and launches a pair of the corresponding data adaptor and an accompanying *data importer*. The data importer is a daemon process forked as the child of the import manager process, and takes over the rest of the data import task.

First of all, the importer acquires the target data schema either by directly contacting with the data source (in case of database tables) or via manual configuration. For efficiency and security reasons, users can define which part of the data source they wish to export to ES^2 for processing purposes. The importer will perform a schema mapping, when there is some inconsistency between the original data schema and the target schema. Moreover, it determines the data partitioning scheme and notifies the distributed indexing component about the set of secondary indexes to be built, which could be by default and/or user specified. The information of data partitioning and indexes, along with the target schema itself, will be stored in the meta-data catalog. Besides, relevant statistics are also initialized and will be updated as the data import progresses.

Subsequently, the data are imported from the data source and parsed into one or multiple physical tables. For each table, we allocate a write cache in memory to temporarily buffer the records. When the write cache is full, the records are flushed to the underlying DFS. In this way, we reduce the number of costly I/O requests to DFS. This cycle is repeated until the whole table has been written to the DFS.

Note that the import manager supports parallelism in two ways. First, multi users can concurrently perform the import function. Second, it can process each import request from a user in fine-grained manner: loading multi tables in parallel from the external data source into ES^2. However, the performance of the import operation is actually affected by other two bottlenecks: the read throughput from the data source and the write throughput of the underlying DFS.

B. Write Cache

The purpose of allocating a write cache in the importing process is manifold. First, in the write cache, the data records are organized into pages following a specific storage layout PAX [22]. A record will be completely included by one specific page. Second, the distributed indexing component works more efficient in a bulk-loading way, after the write cache becomes full. For each record, its key value and the id

of the page containing it are capsuled as an index entry and published into the cluster. The write cache allows the indexing component to package the inserted index entries to the same index node in a single message, which reduces the cost of routing. Third, it significantly reduces the number of function calls to the underlying DFS, thus leading to a big cost saving on the invocation overhead. Last but not the least, the relevant table-level and record-level statistics in the meta-data catalog are updated in a batch. Therefore, the overhead incurred by the locking mechanism adopted for the meta-data consistency is also reduced.

PAX Page Layout. As mentioned before, we treat the DFS as a raw byte device, which means that tables are stored in the DFS as binary byte streams. Therefore, we need to implement a page-based storage layout, to explicitly interpret the output byte streams from the DFS into records.

We employ the PAX [22] (Partition Attribute Across), which is essentially a DSM-like [23] organization within an NSM (N-ary Storage Model) page. In general, for a table with n columns, its records are stored into pages. PAX vertically partitions the records within each page into n *minipages*, each of which consecutively stores the values of a single column.

While the disk access pattern of PAX is the same as that of NSM and does not have an impact on the memory-disk bandwidth, it does improve on the cache-memory bandwidth and therefore the CPU efficiency. This is because of the fact that the column-based layout of PAX physically isolates values of different columns within the same page, and thus enables an operator to read only necessary columns, e.g. the aggregated columns that are referred in an aggregation OLAP query. The use of PAX layout also has great potential for data compression. For each column, we add a compression flag in the header of a PAX page to indicate which compression scheme is utilized.

Memory Management of Write Caches. Multiple tables can be imported simultaneously, leading to more than one write caches being allocated in memory. All these caches share a limited amount of memory. Thus, the memory allocation among them is dynamically adjusted, based on the factors such as the table sizes, the data arrival rates, etc. When necessary, some write caches could be suspended and their records are temporarily dumped to the local disk, in order to give way to other caches with higher priorities. Once more free memory is available (e.g., some write caches have finished their jobs and thus been deallocated), these suspended caches are resumed and their records are read back from the local disk.

C. Meta-data Catalog

The meta-data catalog provides (a) schema storage, update and retrieval, (b) statistics storage, update and inquiry, and (c) runtime statistics collected via daemon processes. The catalog will be accessed by the import manager, write cache, distributed indexing and data manipulator.

The information stored in the meta-data catalog includes: (1) table ownerships and definitions such as column names, data types and primary/foreign key(s); (2) dataset/partitioning information such as collocated tables, partitioning keys, clustering keys (i.e., sort orders); (3) table cardinalities, single or multiple dimensional histograms built on columns, available secondary indexes and table access statistics.

Since data access plans are composed based on the meta-data in the catalog, it is important that ES^2 maintains a consistent view for the meta-data. A naive solution is to put the whole catalog in a single node and adopt a simple locking mechanism with fine granularity to enable the data synchronization. The locking mechanism includes a sharable read lock and an exclusive write lock. However, in order to improve the scalability and availability of the catalog, in reality we choose to deploy the catalog to a set of distributed nodes and add the data replication mechanism with an appropriate data synchronization technique. In particular, we apply different consistent model for different type of meta-data, e.g. strict consistent for schema information and relaxed consistent for runtime statistics.

V. Distributed Indexing

In ES^2, the distributed indexing subsystem maintains secondary distributed indexes over the data stored in the underlying DFS. It works as a middleware between the data access control module and DFS, interacts with the DFS and provides data retrieval interfaces for the data manipulator.

The reasons for distributing the indexes are twofold. On one hand, the size of an index is proportional to the size of its indexed data. Since we are managing the cloud data which typically are huge in volume, a single server machine (referred to as an *index node*) is usually not capable of storing all the indexes. On the other hand, the indexes are employed to serve high load of concurrent online queries. As such, a single index node itself may become the performance bottleneck, when dealing with a large number of concurrent requests.

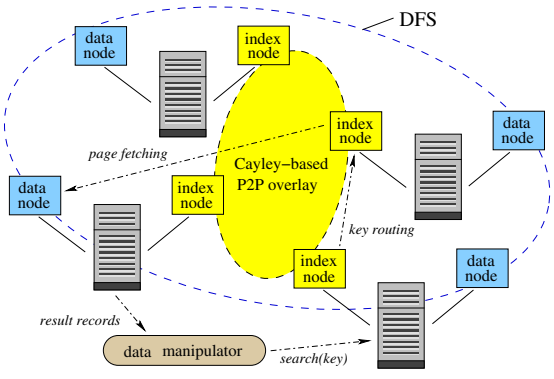


Fig. 2. The Distributed Secondary Index

In our system, both data nodes and index nodes share the same set of machines, as depicted in Figure 2. In particular, each machine in the cluster can assume two roles: a data node in DFS and an index node in the distributed indexes. Each index node maintains a portion of the distributed indexes, which can be applied to retrieve data from the underlying DFS.

An index page and its correspondingly indexed data could be hosted by two distinct machines.

When a specific record is accessed, the system first checks the meta-data catalog for available indexes. If an index can be exploited to facilitate the search, the system will connect to the corresponding index node to retrieve the index entry for the record. The index entry contains the index key and the page ID, which can be used to retrieve the physical PAX page in DFS. Then, we parse and assemble the record from the PAX page. Since a page of the table could be replicated by the DFS across several data nodes, and it is the duty of the DFS to decide from which data node the page should be fetched. It is notable that the indexes in ES^2 consist of secondary indexes, and thus index pages are stored independently from data pages. ES^2 also provides another option for data collocation by implementing materialized indexes or materialized views. This approach improves the query processing time by reducing the network I/Os and disk I/Os. The additional storage cost of this approach is acceptable in the case the sizes of data records are relatively small.

Recently, we have proposed two types of distributed secondary indexes for cloud data: the distributed B$^+$-tree-like index which supports one-dimensional range queries [24] and the distributed multi-dimensional index which supports multi-dimensional range queries and KNN queries [25]. Both approaches share the common idea of two-level index techniques which combine the P2P routing overlay with disk-resident local indexes in different index nodes. This idea can also be adopted when deriving new types of distributed indexes.

It is certainly desirable to include various types of distributed indexes in our system in order to facilitate different kinds of queries. However, we cannot afford to implement and maintain multiple P2P overlays in our cluster. As has been shown in [26], many P2P overlays are essentially instances of the Cayley graph, and consequently we develop a *generalized indexing framework*, which provides the implementation of an abstractive template overlay based on the Cayley graph model.

The framework provides several interfaces to customize the structure and behaviors of an overlay. An overlay manager is established to maintain the overlay instances. When we intend to create a new type of distributed index, we first check the overlay manager for existing overlay instances. If none of them is applicable for building the new index, a new overlay instance is initialized via the overlay template. By implementing the interfaces, we can define the routing protocols and key assignment policy, which are tailored for the new distributed index. In our system, we provide four basic types of indexes, distributed bitmap, hash, B$^+$-tree-like and kd-tree-like indexes. The detailed design and implementation of the generalized framework for indexing cloud data are presented in [27].

We utilize the TCP protocol for message passing in the P2P overlay. One message could be forwarded by multiple index nodes before finally reaching its destination. Since setting up a new TCP connection for two nodes is expensive, we prefer keeping the established connection as long as there are message flow between the nodes. However, each node can only maintain a limited number of TCP connections. Therefore, we develop a *connection manager* at each index node to monitor the real-time network status and decide how to dynamically create and maintain the set of TCP connections in the overlay. The connection manager records the message routing patterns and always try to keep the most valuable connections.

Our indexing system is designed to support a large number of concurrent distributed indexes. As such, each index node will have to maintain many local indexes belonging to different distributed indexes. To improve the search efficiency, pages of local indexes can be buffered in memory. Due to the limited memory size, it is impossible to accommodate all local indexes in the memory. We thereby develop a *buffer manager* for each index node, which adaptively adjusts the index pages to be kept in memory, according to the current query workload.

VI. Data Access Processing

The data access control module of ES^2 deals with the data access requests for processing OLAP queries (executed by the analytical execution engine E^3) and OLTP queries (submitted by application users). In this module, the *data access interface* is exposed to handle these requests, while the *data manipulator* is in charge of the manipulation of the data stored in the physical storage, according to the interpreted commands passed by the data access interface. Note that to prevent the data manipulator from being the bottleneck, ES^2 can replicate and deploy this component on multiple nodes to share the workload of incoming data access requests.

A. Data Access Interface

The data access control module defines two independent interfaces, namely OLTP interface and OLAP interface, for OLTP queries and OLAP queries respectively. This is beneficial as these two types of query have diverse data access patterns and thus present different requirements on the data access interface. We separately discuss these two interfaces as follows.

OLTP Interface. For OLTP workload, data is typically accessed via a *point* or *small-range* query. With this type of query, only one or several records within a single table will be located and manipulated. We define three major APIs for this OLTP workload:

- `get(table, key, columns)`
- `put(table, record)`
- `delete(table, key)`

The parameter `table` represents a logical table, whose internal storage consists of one or multiple physical tables through vertical partitioning (refer to Section III). The `key` is a set of conjunctive or disjunctive column selection conditions, and is used for locating the target records, which are then either retrieved by the `get` operation or eliminated by the `delete` operation. The `columns` is the set of projected columns out of the target records. The `record` is a new record to be inserted

into the table by the put operation. The operation of updating a record is realized as appending a new version of the record to the system, as will be discussed in Section VI-B.

OLAP Interface. For OLAP workload, data is usually accessed via batch processing operations, which means not only that a large number of records are read, but multiple tables are also accessed within a single query.

Compared to MapReduce-based systems, in which all participating tables are sequentially scanned in parallel and unqualified records are filtered out by Map tasks, ES^2 provides an additional index scan functionality, which is especially useful when the set of qualified records is merely a small portion of all records that would be touched by the sequential scan. Therefore, we allow the execution engine E^3 to push the record selection and projection conditions, along with the table names, through the OLAP interface and into the data manipulator. As a result, only required columns of qualified records will be returned to the execution engine. At the level of the data manipulator, we also consider further supporting some other relational operations (e.g., aggregation and sorting) that could enable additional optimizations (e.g., *early aggregation* [28] and *shared scan* [29]).

The OLAP interface is basically in the *iterator mode* [30], with three major APIs: open(), next() and close(). At the beginning, the open function is called to initialize the scan of one logical table, parameterized by the record selection and projection conditions. These parameters will be used by the data manipulator to determine data access plans (e.g., scanning a specific data partition or performing an index scan). open returns when the preparation of the underlying data manipulator is done. After that, the next function is repetitively invoked, with a group of records returned each time. After all the qualified records are retrieved, the close function does some house keeping tasks, such as closing up the physical tables and updating the relevant data access statistics in the meta-data catalog.

It is possible that the execution engine E^3 also deals with data formats other than the record representation (e.g., key-value pairs). Therefore, inside the next function we implement a *data format translator* which optionally converts the retrieved records into the desired formats before returning them to the execution engine.

B. Data Manipulator

1) OLAP and OLTP Isolation: To handle both OLAP and OLTP workload within the same storage, we adopt the multi-versioning strategy. Specifically, we keep multiple versions of the data in the system. Each version is assigned with a version number. When updating a record, we actually append a new version of the record to the system. We set a threshold up to which each record can maintain a set of versions. When the total number of versions maintained by a record exceeds the threshold, the dead (obsolete) versions will be discarded.

The multi-versioning strategy simplifies our design of query processing. In a system that support both OLAP and OLTP queries, we cannot adopt the locking mechanism, due to its high overheads (e.g. if an OLAP query scans the whole dataset and concurrent OLTP queries are being processed simultaneously, locking the whole dataset is not preferred). Instead, we adopt a timestamp-based approach. A loosely synchronized clock in the system can be implemented as follows. We discretize the time dimension into epochs. A node in the cluster plays as the timestamp authority (TA) and increases the epoch after every period of time (which is configurable by the user). The TA then messages the new epoch to all other nodes in the cluster, and the whole system will move to this new snapshot. Possible failures of the TA can be handled by a stand-by node. We mark each data version with a timestamp, indicating when it is created. When an OLAP query q is submitted, we generate a timestamp ts for it. In ES^2, we are trying to provide the snapshot consistency for OLAP queries. Therefore, for each record, we use the version, whose timestamp is just before ts, to answer the query q. If that version is being discarded due to too many updates, we choose to use the latest version. In that case, the snapshot consistency cannot be guaranteed and we will notify the users about it.

With the multi-versioning strategy described above, the put and delete operations in OLTP are actually isolated from the get operation in OLTP and the next operation in OLAP. In the following, we concentrate on the pure record reading issues for both OLTP queries and OLAP queries.

2) Data Access Optimizer: Essentially, the record retrieval commands passed from the OLTP and OLAP interfaces are the same and thus will be uniformly processed by the data manipulator. There are two data access methods, *parallel sequential scan* and *index scan*, for record retrieval.

Note that the OLTP and OLAP queries only work with the logical tables via the data access interface and are transparent with the physical organization of these tables. However, the columns of a logical table are organized as column groups (refer to Section III), each of which is stored in a separate physical table. Therefore, the data manipulator may need to assemble projected records for a logical table with corresponding records of some or all componential physical tables. In case of sequential scan, the data manipulator is able to read records belonging to different horizontal partitions of a physical table in a parallel manner (under this situation, an OLAP query will invoke multiple next functions simultaneously, one for each partition). This feature of parallel scanning is analogous to the way how a MapReduce job reads its input data.

For OLTP queries and OLAP queries with low selectivities, the various distributed secondary indexes maintained by the distributed indexing system enable the index scan alternative to the parallel sequential scan. However, index scan is not necessarily a better choice than parallel sequential scan. First of all, the appropriate indexes are not always available. In addition, the underlying DFS usually has a very high latency for random access [31]. As a result, even if the index traversal is sufficiently fast, the overall delay including reading records from DFS, would be large. This means that, in many cases parallel sequential scan would still be preferred. Therefore,

TABLE I
PARAMETERS FOR DATA ACCESS OPTIMIZER

Parameter	Definition
c_s	cost ratio of sequential read in DFS
c_r	cost ratio of random read in DFS
c'_r	cost ratio of random read with sequential offsets
s_d	size of a data chunk
$f(Q)$	number of I/Os for query Q
$g(T_i, Q)$	number of T_i's tuples that satisfy the predicates of Q
n	total number of nodes in the cluster

the data manipulator cannot always assume that index scan is more suitable for OLAP queries with low selectivities.

To address the above problem, we introduce a *data access optimizer* within the data manipulator component, which dynamically chooses the best data access scheme for a specific query, relying on the statistics stored in the meta-data catalog.

The core issue of data access optimization is finding the most efficient access method to read a specified set of records from each individual physical table involved by the query. The naive solution would be based on a threshold of the selectivity. In other words, for each physical table, if the query selectivity is below some predetermined threshold, we shall choose index scan; otherwise, we shall choose parallel sequential scan. Clearly, this solution lacks of adaptability. Even if we assume that the cluster of nodes is static (i.e. with a fixed configuration), the possibility of having a sub-optimal access method would be high, not to mention that the cluster configuration, as well as the query workload, changes frequently. Therefore, we instead adopt a cost-based approach.

For simplifying the discussion, we list the parameters used in our cost model in Table I. As ES^2 adopts the DFS as the underlying storage system, the data is typically partitioned into equal-size (s_d) data chunks. Given a query Q, we define function $f(Q)$ to denote the size of data involved in the processing. For parallel sequential scan, if table T_1,...,T_k are involved in query Q, $f(Q)$ is computed as $\sum_{i=1}^{k} |T_i|$. In index scan, $f(Q)$ is estimated as $\sum_{i=1}^{k} g(T_i, Q)$, where $g(T_i, Q)$ denotes the number of tuples in T_i that satisfies the predicates of Q based on our histograms.

In particular, the costs of different processing strategies are estimated as:

1) The latency of using parallel sequential scan to process Q can be computed as:

$$c_{pscan} = \lceil \frac{f(Q)}{s_d n} \rceil c_s s_d \quad (1)$$

Equation 1 is based on the assumption that data are uniformly distributed over the cluster and each node only needs to scan its local data chunks. In real systems, although the assumption is not always satisfied, Equation 1 provides a good enough estimation for evaluating the performance of parallel sequential scan.

2) In index scan, we group the requests to the same data chunk and perform the random access in sequential offsets. This strategy is based on the observation in [31] that the cost of random access via sequential offsets (c'_r) is far less than the cost of random access via random offsets (c_r). Suppose the retrieved tuples are uniformly distributed over the data chunks. We have

$$\frac{\sum_{i=1}^{k} |T_i|}{s_d}$$

data chunks. And the latency of index scan is estimated as:

$$c_{iscan} = \frac{\sum_{i=1}^{k} |T_i|}{s_d} c_r + \sum_{i=1}^{k} g(T_i, Q) c'_r \quad (2)$$

In above equation, we discard the cost of accessing index, as such cost is negligible compared to the data retrieval cost.

Given a query, the optimizer estimates the costs of different strategies. If $c_{pscan} > c_{iscan}$, the parallel sequential scan is used to process the query. Otherwise, the index scan is used. Periodically, we run a micro-benchmark to test the performance of raw random and sequential I/Os and update the values of c_s, c_r and c'_r, respectively.

VII. PERFORMANCE STUDY

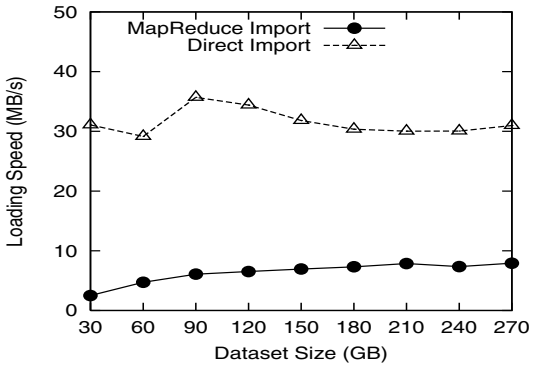
In this section, we evaluate the performance of our data storage system, which resides in an in-house commodity cluster `awan` constructed for the `epiC` project [10]. `awan` contains 72 cluster nodes, which are connected via three switches. Each node is equipped with Intel X3430 2.4 GHz processor, 8 GB of memory, 2x500GB SATA disks, gigabit ethernet, and operates CentOS 5.5. The cluster nodes are evenly divided into three racks and are used to accommodate our data storage system.

In what follows, we study the performance of ES^2 in three parts. The first part is the evaluation of the data import and storage components. The second part presents the experimental results of the distributed indexing component. The last part measures the data freshness that ES^2 can provide for OLAP queries. In this paper, we mainly describe the design and implementation of ES^2, the storage manager of `epiC` cloud system, and provide the performance evaluation of its main functionalities. The processing of OLAP and OLTP queries will ride on the functionalities provided by ES^2, and benchmarking of the whole `epiC` system is our future work.

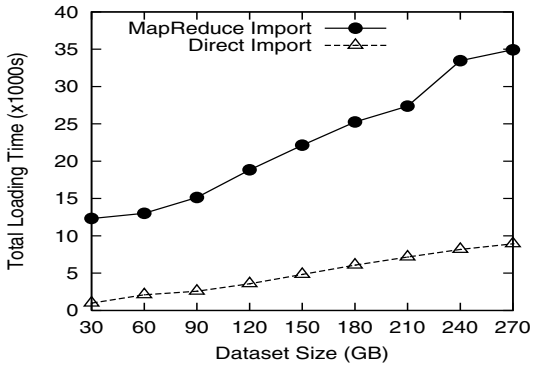
A. Data Import and Storage

We make use of the HDFS version 0.20.1 for the DFS component of ES^2. In our experiments, we fix the size of a data block to 64MB and replicate the block three times. We also fix the PAX page size to 8KB and utilize the TPC-H [32] dataset with scales ranging from 30GB to 270GB. Further, we also vary another set of experimental parameters which are discussed in the following.

1) Data Loading: We first evaluate the performance of loading TPC-H datasets from an external DBMS (MySQL 5.0.77 residing in a node of `awan`) into ES^2. We have two approaches for data loading: *direct import* which fetches data with the aid of MySQL's built-in data export tool `mysqldump`; *MapReduce import* which runs a MapReduce job reading records from MySQL via JDBC. The performance



(a) Data Loading Speed



(b) Total Data Loading Time

Fig. 3. Performance on Loading TPC-H Dataset of Different Sizes with 5 Data Nodes

scalability is studied in two aspects: varying the dataset sizes and varying the number of data nodes in the system. Here we fix the size of each write cache to 64MB.

Figure 3 shows the results of loading TPC-H datasets of different sizes into ES^2 with 5 data nodes. The total loading time includes fetching records from MySQL, assembling records into PAX pages in write caches and finally flushing these PAX pages into the underlying HDFS. It can be observed that MapReduce import is much slower than direct import, due to the very inefficient JDBC as well as the intermediate data that are generated by the MapReduce job and materialized in HDFS. Moreover, the data loading speeds of both approaches are not very high ($\leq$35MB/s), compared with the network speed (~110MB/s) and the write speed (~50MB/s) of HDFS with 5 data nodes and 3 replications. The key reason is that the MySQL becomes a performance bottleneck: both approaches can only sequentially retrieve records from MySQL at very low speeds.

We also tested the loading of a 270GB TPC-H dataset into ES^2 with various numbers of data nodes in the system (ranging from 5 to 35 nodes). The results (not shown) show that the increasing number of data nodes does not significantly improve the loading time in both approaches, direct import and the MapReduce import. It is because of the fact that the performance of the import function is actually affected by two bottlenecks: the read throughput from the external data source and the write throughput of the underlying DFS.

2) Effect of Write Cache Size: In this experiment, we investigate how the size of the write cache (refer to Section IV-B) will affect the overall data loading performance. We import 30GB TPC-H dataset from MySQL into ES^2 with 5 data nodes. In the meantime, we vary the size of each write cache. The minimum size is 8KB which is just enough for one PAX page, while the maximum size is 64MB which is also the data block size of the underlying HDFS. We expect that the total loading time will decrease as the cache size increases, which agrees with the experimental results shown in Figure 4.

Moreover, the total loading time drops readily till the write cache size increases to 1MB (128*8KB). After that, when the write cache size increases from 1MB to 64MB, the total loading time keeps growing downwards but at a extremely slow speed. Therefore, setting the write cache size to 1MB can maintain a good balance between the data loading performance and the total memory overhead incurred by the write caches.

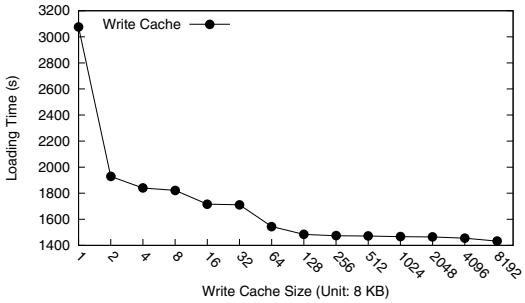
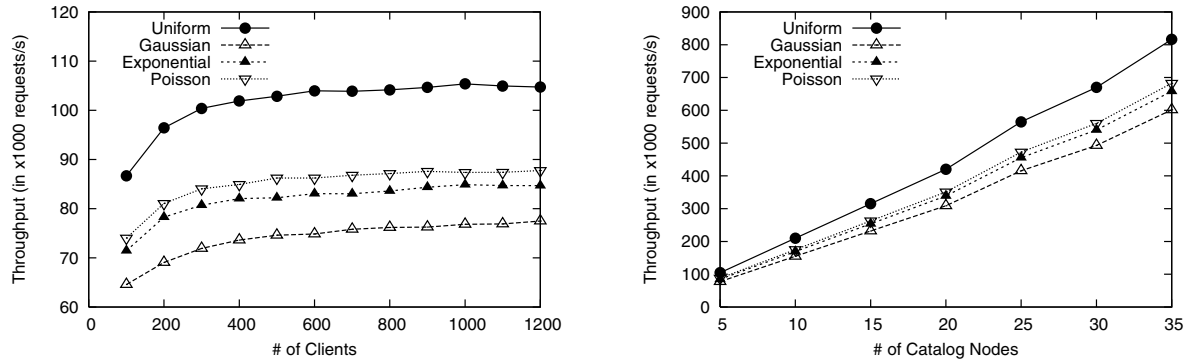


Fig. 4. Performance on Loading 30GB TPC-H Dataset into HDFS with 5 Data Nodes, Varying the Write Cache Size

3) Pressure Test on Meta-data Catalog: In this test, we study the scalability and availability of the catalog when multiple clients send massive read/writes requests to it. Each client is represented by a separate thread and submits read/write requests on entries within the catalog. All the clients start up simultaneously and send their requests to the catalog concurrently. Each client submits its own requests in a sequential order and waits till the previous request gets acknowledged before sending the next request.

We vary both the total number of clients and the number of nodes maintaining the catalog. The content of the meta-data catalog is fixed and evenly range partitioned over all catalog nodes. In our test, the type of each request sent by a client follows these distributions, *uniform*, *Gaussian*, *exponential* and *Poisson*. The client will submit a read request if the calculated probability is below 0.5, and a write request otherwise.

Figure 5 shows the throughputs, i.e. the total number of requests processed by the meta-data catalog per second, under different parameter settings. In Figure 5(a), it can be observed that the meta-data catalog is able to always maintain a stable and relatively high throughput, no matter how the client number and the total request number change. It is also obvious that the throughputs are at different levels under various probability distributions: *uniform* distribution has the highest throughputs, followed by *Poisson* and *exponential*, and *Gaussian* has the lowest throughput. The reason for such difference is the



(a) With 5 Catalog Nodes, Varying Client #

(b) With 1200 Clients, Varying Catalog Node #

Fig. 5. Performance on Concurrently Sending Read/Write Requests to Meta-data Catalog

percentage of write requests within all the requests sent by a client. Intuitively, with a fixed total number of requests, more write requests would lead to longer processing time, as write is expected to be more expensive than read due to locking. We realize that *Gaussian* distribution always results in the highest percentage of write requests, followed by *exponential* and *Poisson*, and *uniform* generates the lowest percentage. This percentage order is consistent with the observations from Figure 5(a).

In addition, Figure 5(b) illustrates that the total throughput increases almost linearly with the number of nodes maintaining the catalog. As the number of catalog nodes increases, each node host a smaller portion of the meta-data catalog content. The resulted benefit is two-fold. On one hand, the time spent on entry search is reduced. On the other hand, the average overhead of entry consistency control also drops.

B. Distributed Indexes

We now present the results of scalability tests on the distributed indexes in ES^2. We measure the index search latency and throughput when varying the size of the dataset and the number of index nodes in the system. The size of TPC-H dataset ranges from 30GB to 270GB. We also test the performance of the indexes with various system sizes, ranging from from 5 index nodes to 35 index nodes. The default value of the dataset size and the system size are 30GB and 5 nodes respectively. We populate 100 client threads at each node to continuously submit requests into the system. A completed requests will be immediately followed up by another request.

In this experiment, we employ distributed indexes to improve the performance of the processing of the following queries, whose predicates do not contain the attributes that are used to partition the base tables.

Q1: SELECT * FROM Part WHERE partname='x'

Q2: SELECT custkey, count(orderkey), sum(totalprice)
FROM Orders
WHERE totalprice $\geq y$ and totalprice $\leq y + 100$ and
orderdate $\geq z$ and orderdate $\leq z + 1$ month
GROUP BY (custkey)

In particular, we build a distributed hash index on the *partname* attribute of the *Part* table and a distributed kd-tree-like index on the *(totalprice, orderdate)* attributes of the *Orders* table, by instantiating the corresponding type of indexes from the generalized indexing framework developed in ES^2.

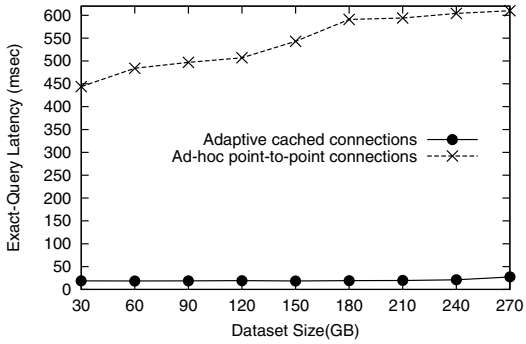
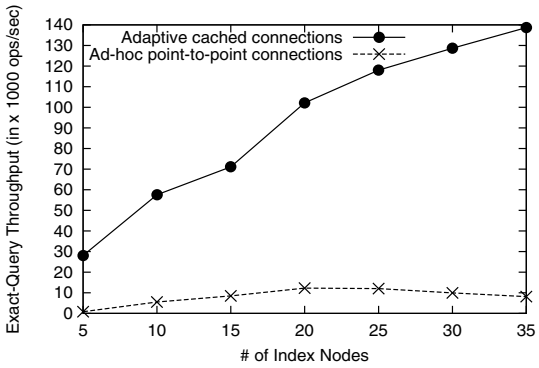


Fig. 6. Exact-match Query (Q1) Latency

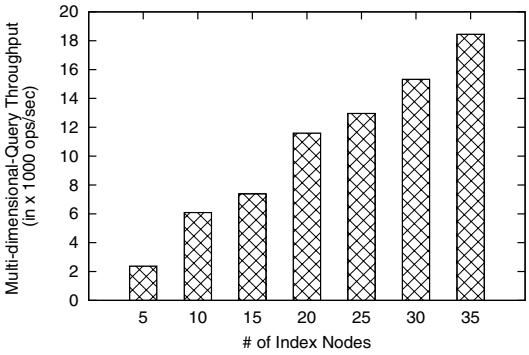
The scalability of the distributed hash index is well-demonstrated in Figure 6. The system has nearly constant index search latency when we increase the size of dataset, especially with the use of adaptive cached connections. Setting up ad-hoc point-to-point connections in order to send only a message between two index nodes is expensive since creating a new TCP connection incurs additional overheads such as handshaking time delay and memory buffer for the connection.

Instead, in our approach, referred to as adaptive cached connections, each index node in the cluster adaptively keeps a limited number of established connections to other frequently accessed nodes in the distributed index overlay. In this way, we do not need to pay the cost of creating new connections when sending a message between two index nodes. This is the reason why the system suffers from much higher index search latency in the case of using ad-hoc point-to-point connections compared to the adaptive cached connections approach.

In Figure 7(a), the advantage of the adaptive cached connections approach is again confirmed. Since the persistent connections are always available to use, the network bandwidth in the distributed index overlay is significantly improved. Therefore, the system with adaptive cached connections has much higher throughput compared to ad-hoc connections. In addition, with adaptive cached connections, the system throughput increases steadily with the number of index nodes. In contrast, the system throughput with the ad-hoc connections



(a) Exact-match Query (Q1) Throughput



(b) Multi-dimensional Query (Q2) Throughput

Fig. 7. Performance of Distributed Indexes

approach degrades when the number of index nodes increases to large values. The reason is that in the experiment setting the submitted workload is proportional to the system size; however, the approach to create ad-hoc point-to-point connections cannot handle this big workload. When the system reaches the threshold, the throughput begins to decrease.

Figure 7(b) shows the system throughput when using the distributed index to process multi-dimensional queries in various system sizes. We can observe that the system achieves almost linear throughput with respect to the increasing number of index nodes. In our experiment setting, the more index nodes in the systems, the more job requests will be populated. However, with the efficient support of the distributed kd-tree-like index, the system can comfortably handle these queries. The system is therefore able to scale well.

C. Data Freshness

In this experiment, we measure the *data freshness* that $\texttt{ES}^2$ can provide for OLAP queries. In other words, when an ad-hoc OLAP job is launched on a dataset that is being simultaneously manipulated by OLTP operations, we measure how older is the visible version of the dataset to the OLAP job, compared to the up-to-date version of the dataset.

In particular, we setup a 64-node cluster in `awan` and insert different sizes of data (32GB to 512GB). In our experiment, each record maintains a maximum of 8 versions. We employ 5 cluster nodes to submit updates to the system continuously at the rate of 100 operations/sec. The updates follow uniform distribution or normal distribution. In the diagrams, we use U and N to denote the uniform updates and normally distributed updates, respectively. Two metrics are used in the benchmark. In a sequential scan starting at t_0, when reading record r, we get the ith version, whose timestamp is t_1. As defined by $\texttt{ES}^2$, $t_1 \leq t_0$ and r does not have any other version between t_1 and t_0. After the scan operator completes, the latest version of r is j and the timestamp is t_2. The version difference regarding to r is $j-i$ and the time delay is t_2-t_1. For comparison purpose, we use another sequential scan approach, which always gets the latest version. Namely, when it reads r, it get r's current latest version. We use *es2* and *recent* to represent the two sequential scan approaches.

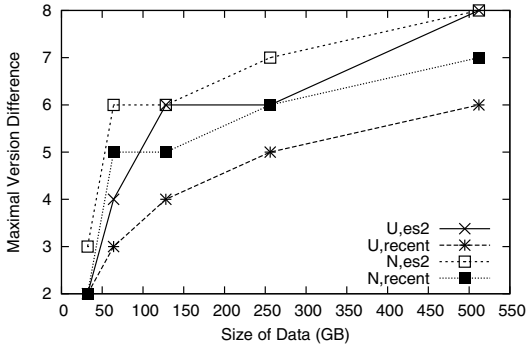


Fig. 8. The Maximal Version Difference

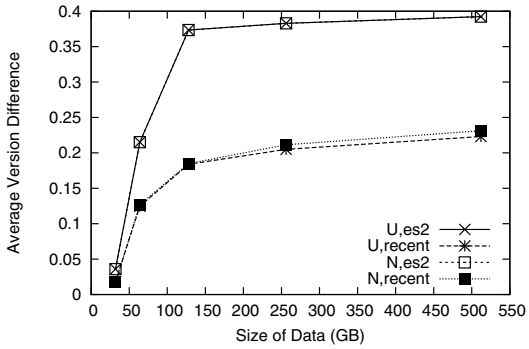


Fig. 9. The Average Version Difference

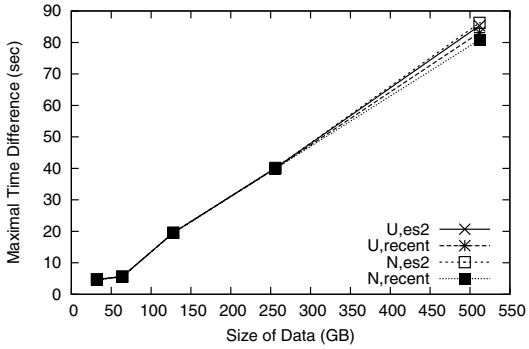


Fig. 10. The Maximal Time Delay

Figure 8 and Figure 9 show the maximal and average version difference among all records, respectively. When the data size increases, it incurs more overhead to scan the dataset. Hence, both *es2* and *recent* provide a stale version. However, scanning 512GB dataset just leads to maximal 8

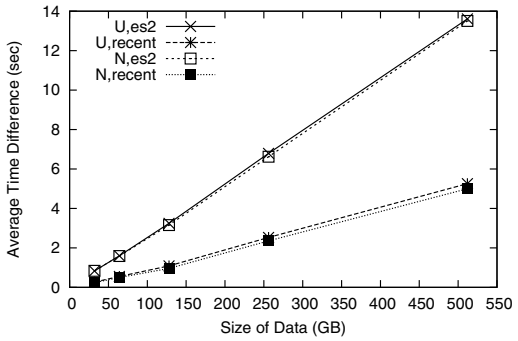


Fig. 11. The Average Time Delay

version difference. In the experiments, we get two unexpected observations. First, *recent* does not provide a more "fresh" result than *es2*, even it always read the latest version. This is because in ES^2, multiple cluster nodes start the scanning in parallel, which is quite efficient. Second, the update pattern does not affect the "freshness". Uniform updates generate a similar result as the normally distributed ones. For such a large dataset, a specific record will not get too much updates, even in a skewed distribution.

Figure 10 and Figure 11 show the maximal and average time delay. We get a similar result as in the version difference case. Scanning 512GB dataset only incurs a maximal delay of 90 seconds. In most cases, such delay is acceptable. Users do not mind to get a global statistics, which provides a view for the system of 90 seconds ago.

Based on experimental results above, ES^2 can provide for most OLAP queries a fresh and consistent snapshot of the data which are simultaneously manipulated by OLTP operations.

VIII. Conclusion

In this paper, we propose a new system architecture for supporting database operations on the cloud. We describe our storage system, ES^2 – an elastic cloud data storage system, which has been designed to support both OLTP and OLAP workloads efficiently within the same storage and processing system. The system provides efficient data loading from different sources, flexible data partitioning scheme, index and parallel sequential scan. We also present experimental results which demonstrate the efficiency of ES^2. The results confirm the benefit of providing distributed indexes and accesses to the data for both OLTP and OLAP queries. We are integrating ES^2 with other major components of the epiC cloud system [10], and we will benchmark the whole system in the near future.

Acknowledgment

The work in this paper was in part supported by the Singapore Ministry of Education Grant No. R-252-000-394-112 under the project name of Utab. We would also like to thank the anonymous reviewers for their insightful comments.

References

[1] 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*, 2006, pp. 205–218.

[2] 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," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[3] 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*, 2007, pp. 205–220.

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

[5] Apache Hadoop. http://wiki.apache.org/hadoop.

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

[7] Apache Pig. http://hadoop.apache.org/pig/.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.

[9] H. Plattner, "A common database approach for oltp and olap using an in-memory column database," in *SIGMOD*, 2009, pp. 1–2.

[10] The epiC project. http://www.comp.nus.edu.sg/∼epic/.

[11] C. Chen, G. Chen, D. Jiang, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "Providing scalable database services on the cloud," in *Wise*, 2010.

[12] Q. H. Vu, M. Lupu, and B. C. Ooi, *Peer-to-Peer Computing: Principles and Applications*. Springer Publishing Company, Incorporated, 2009.

[13] D. J. DeWitt and J. Gray, "Parallel database systems: The future of database processing or a passing fad?" *SIGMOD Rec.*, vol. 19, pp. 104–112, 1991.

[14] E. Friedman, P. Pawlowski, and J. Cieslewicz, "Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions," *PVLDB*, vol. 2, pp. 1402–1413, 2009.

[15] Greenplum MapReduce. http://www.greenplum.com/technology/mapreduce/.

[16] A. Segev and W. Fang, "Currency-based updates to distributed materialized views," in *ICDE*, 1990, pp. 512–520.

[17] R. Hull and G. Zhou, "A framework for supporting data integration using the materialized and virtual approaches," *SIGMOD Rec.*, vol. 25, no. 2, pp. 481–492, 1996.

[18] A. Labrinidis and N. Roussopoulos, "Exploring the tradeoff between performance and data freshness in database-driven web servers," *The VLDB Journal*, vol. 13, no. 3, pp. 240–255, 2004.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008, pp. 1099–1110.

[20] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," *PVLDB*, vol. 1, no. 2, pp. 1265–1276, 2008.

[21] H. T. Vo, C. Chen, and B. C. Ooi, "Towards elastic transactional cloud storage with range query support," *PVLDB*, vol. 3, no. 1, pp. 506–517, 2010.

[22] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *VLDB*, 2001, pp. 169–180.

[23] G. P. Copeland and S. Khoshafian, "A decomposition storage model," in *SIGMOD*, 1985, pp. 268–279.

[24] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient b$^+$-tree based indexing for cloud data processing," *PVLDB*, vol. 3, no. 1, pp. 1207–1218, 2010.

[25] J. Wang, S. Wu, H. Gao, J. Z. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *SIGMOD*, 2010, pp. 591–602.

[26] M. Lupu, B. C. Ooi, and Y. C. Tay, "Paths to stardom: calibrating the potential of a peer-based data management system," in *SIGMOD*, 2008, pp. 265–278.

[27] H. T. Vo, S. Wu, and B. C. Ooi, "Towards generalized framework for indexing cloud data," *Technical Report, National University of Singapore, School of Computing. TRA11/10*, 2010.

[28] P.-A. Larson, "Grouping and duplicate elimination: Benefits of early aggregation," in *Microsoft Technical Report*, 1997.

[29] Y. Cao, G. C. Das, C.-Y. Chan, and K.-L. Tan, "Optimizing complex queries with multiple relation instances," in *SIGMOD*, 2008, pp. 525–538.

[30] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, 1993.

[31] HDFS Community Forum - https://issues.apache.org/jira/browse/HDFS-236.

[32] TPC-H Benchmark. http://www.tpc.org/tpch/.