Phoenix
A Relational Storage Component for the Cloud

Davi E. M. Arnaut, Rebeca Schroeder and Carmem S. Hara
Universidade Federal do Paraná, Curitiba, PR, Brazil
{davi,rebecas,carmem}@inf.ufpr.br

***Abstract*—This paper describes the design and architecture of a cloud-based relational database system. The system's core component is a storage engine, which is responsible for mapping the logical schema, based on relations, to a physical storage, based on a distributed key-value datastore. The proposed stratified architecture provides physical data independence, by allowing different approaches for data mapping and partitioning, while the distributed datastore is responsible for providing scalability, availability, data replication and ACID properties. A prototype of the system, named Phoenix, has been developed based on the proposed architecture using a transactional key-value store. Experimental studies on a cluster of commodity servers show that Phoenix preserves the desired properties of key-value stores, while providing relational database functionality at a very low overhead.**

***Index Terms*—cloud computing, dht, distributed databases, peer-to-peer**

I. Introduction

The ever-increasing volume and diversity of data coupled with the dissemination and maturation of various concepts of cloud computing are transforming the economic aspects of computing, mainly due to the introduction of data based services with an affordable cost, which are on-demand and in real time [1], [2]. One aspect of this paradigm shift is due to new ways of handling and delivering data across service-based distributed architectures, so that data can be easily accessible and ubiquitous [3]. This new computing model enables scalability of services and enhances opportunities for collaboration, integration and analysis on a common and shared platform.

There are several systems developed recently to explore storage services in the cloud such as Amazon SimpleDB [4], [5], Google App Engine [6], [7], Windows Azure platform [8], Cassandra [9], PNUTS [10], and G-Store [11]. They all provide a similar service: a simple interface to store and retrieve key-value pairs individually. In addition to similar service-based interfaces, scalability and availability are commonly achieved through a synthesis of peer-to-peer (P2P) techniques, such as self-organization, decentralization and load balancing.

These cloud storage systems provide a scalable framework for accessing and managing data of large-scale web applications [7], [9]. However, disregarding several of the underpinnings of relational database management systems (RDBMS) can be considered a huge step backwards as it hinders important factors such as data independence, reliable transactions, and other cornerstone characteristics often required by applications that are fundamental to the database industry. In particular, the majority of today's cloud storage systems are suitable for OLAP applications, but not for OLTP [3], largely because it is hard to maintain strong consistency over replicas that may be spread over large geographic distances.

In a recent paper [12], it has been shown that a Paxos-based replication protocol, which ensures key range partitioning, and a transactional service is a competitive alternative to weaker notions of consistency adopted by the majority of cloud datastores. The question we tackle in this paper is to determine whether such a datastore is suitable to serve as the physical layer component of a database management system. More specifically, instead of reading and writing records directly to disks like traditional RDBMS, we would like to rely on a Paxos-based transactional key-value datastore to handle storage and distribution of data.

To this end, we propose a new architecture for providing a relational interface on top of a transactional key-value datastore, thus leveraging traditional RDBMS features with the availability and scalability of a cloud storage system. The goal is to provide a database system with low startup and maintenance costs, which is simple to manage. That is, a system that does not require highly skilled specialized personnel for tuning and managing, which is often the case for traditional distributed database systems. The envisioned scenario is a datacenter composed of a cluster of commodity servers, where the system can be deployed to provide a database service for the cloud. It means that data are stored on the cloud, but may be accessed using the same interface as standard RDBMSs, and with the same functionality expected from them.

The system has been designed upon one main principle: *data independence*. Logical data independence makes it possible to change the logical data structure with minimal impact on application programs. Physical data independence enables modifications to the physical storage structure without affecting the logical view of the data. Similar to traditional RDBMSs, we achieve data independence by adopting a stratified architecture, in which the physical level consists of a distributed key-value datastore, and the logical level consists of a traditional RDBMS.

One of the challenges for developing the proposed architecture is the definition of mappings between the relational and key-value models, and also transforming operations defined within the logical level of the relational database to corre-

sponding operations for the cloud datastore. This is equivalent to the physical design of traditional RDBMSs, where the database administrator takes into consideration both properties of the physical storage device and the query workload when defining a data storage schema that maximizes the system's performance. In a distributed system, this also involves determining different levels of granularity for partitioning data.

Contribution. We make the following contributions:

- an architecture for a relational cloud datastore;
- a model for bridging the gap between the relational and key-value pair-based models. The proposed model allows different forms of data partitioning, and also exploits the key-range data distribution provided by some datastores;
- development of a system, named Phoenix, based on the proposed architecture;
- an experimental study that determines the feasibility of our approach. It shows that Phoenix provides relational database functionality on top of a distributed datastore without sacrificing scalability and consistency. Moreover, the overhead associated with the use of Phoenix is minimum, compared to a datastore with no relational layer.

Organization. The remainder of this paper is organized as follows. Section II presents the design, architecture and system components. The implementation of Phoenix is described in section III and the experimental study is the subject of section IV. Section V presents related work, and we conclude on section VI by outlining some future work.

II. System Design

In this section we present an architecture for a relational cloud datastore. We follow the traditional three layers approach supported by relational database systems, consisted of physical, logical, and external layers. As depicted in Figure 1, the logical and external layers rely on traditional database concepts, but physical storage is provided by a cloud datastore.

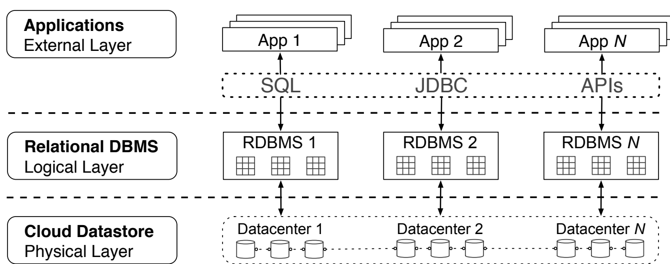


Fig. 1. Relational cloud datastore.

There may be multiple RDBMSs, and they may all be manipulating data stored on datacenters distributed over the network. Moreover, since the query language and application interface of the RDBMS remain unchanged, existing software have no conversion costs.

A. Cloud Datastore

The cloud datastore is responsible for ensuring the storage of data in the system, in the form of a distributed, and scalable service. In general, key-value stores are DHT (*Distributed Hash Table*) systems that provide a generic interface with three operations: `put(key,value)`, to store the `value` associated with a given `key` on a P2P network node; `get(key)` and `delete(key)` are operations to retrieve and remove the `value` associated with a `key`, respectively.

Although these systems provide a number of desirable functionalities to serve as a distributed storage component of a traditional RDBMS, such as decentralization, redundancy, adaptability, self-organization and low operating cost, there are two additional features that we believe they should provide in order to be used as the basis for our architecture: support for transactions with strong consistency, and content locality.

In traditional database systems, the transaction manager works with the storage engine for supporting atomicity, consistency, isolation and durability (ACID) properties. High-level queries are mapped to a sequence of read/write operations on disks, which are in turn packed into an atomic unit by the transaction manager. That is, transactions are defined as sequences of basic operations, that are executed by the storage engine. In our proposed architecture, the storage engine does not operate directly on disks, but do so through a datastore service. In order to provide a transactional functionality similar to a traditional database system, datastores should allow sequences of put-get operations to be packed into an atomic unit. That is, the datastore should provide a transactional service. Traditionally, datastores do not have support for strong notions of consistency. However, in a recent paper, it has been shown that with very low overhead a datastore based on Paxos protocol can achieve stronger notions of consistency [12].

The ability to support strong consistency is closely related to the ability to maintain data locality. The majority of cloud datastores favors availability over consistency based on the fact that it is hard to maintain strong consistency over data that may be spread over possibly distant geographic sites. Usually, DHT-based applications do not have any control over the placement of data, since the P2P node that stores a `(key, value)` pair is determined by the result of a hash function applied on `key`. Load balancing is the main motivation for randomly distribute data among peers that compose a network. For database systems, however, physical design expects some control over data locality in order to allow data clustering. Indeed, clustered indexing has long been recognized as a major technique for improving query performance. Similar to single-server database systems, in which clustering may minimize the number of disk accesses, data locality in a distributed system can minimize the volume and cost of data communication over the network. Moreover, the ability to control content locality on a distributed datastore provides a number of advantages for data retrieval, including improved availability, performance, manageability, and security [13].

Currently, datastores provide content locality in two different ways: either by providing key-range partitioning, as in PNUTS [10], App Engine's BigTable [7], and Spinnaker [12], or using a DHT-based system that maintains keys in lexicographical order, as in Scalaris [14]. One important observation

is that in both approaches, values associated with similar keys are kept either in the same or close servers. That is, these systems support content locality provided that appropriate keys have been defined on the datastore. In the next section, we propose a model for mapping relations to key-value pairs that exploits the idea of data clustering based on keys.

B. Relational and Cloud Interface

One of the main issues to be considered in our architecture is the interface between the relational and cloud datastore, which involves transformation between data models. Thus, data and operations defined on one model have to be mapped to equivalent ones on the other. In order to define such mappings, we propose an intermediate data model, called VOEM. Relations can be easily mapped to VOEM objects, which can in turn be mapped to different forms of key-value pairs, expressing multiple ways of fragmenting the original relation. Moreover, in VOEM, keys can be defined to exploit the range-order partitioning functionality provided by the underlying datastore. In short, VOEM is an intermediate representation of data that provides both fragmentation and data clustering in the proposed architecture.

1) VOEM: VOEM is an extension of the Object Exchange Model (OEM) [15], and stands for *Value-based OEM*. OEM is a self-describing object model which provides a substrate for representing a variety of other data structures. It represents complex data structures using concepts such as object identity and nesting. That is, in OEM every object has a unique identifier (*oid*) and relationships between objects are represented by using *oids* as subobjects or attribute values. In contrast, in the relational model, relationships between data are represented using the concept of key and foreign keys. That is, while OEM relationships are *oid*-based, in the relational model they are value-based. To bridge this gap, we propose VOEM, which extends OEM with the notion of a *key*, that is, a set of subobjects that uniquely identify an object through values.

A VOEM object is a quintuple $\langle$*oid*, *label*, *type*, *value*, *key*$\rangle$. The *oid* is a unique identifier for the object, and *label* describes what the object represents. An object's *type* can either be atomic or complex. An atomic object *value* is an instance of one of the basic atomic (scalar) data types, while the *value* of a complex object is a set of object references (*oid*s). The *key* of a complex object is a pair *(contextOid, subobjs)*, where *contextOid* is an object identifier that defines the context in which the key is defined, and *subobjs* is a subset of the object's *value* that can uniquely identify the object within the context. Examples of VOEM objects are given in Figure 2b.

The nested structure of VOEM objects can also be represented as a directed labeled graph, as depicted in Figure 2c. In this representation, nodes are labeled with *oid*s, and edges are labeled with object *label*s. A useful application of the graph representation is the concept of path expression. A path can be used to navigate through the object hierarchy. In a VOEM graph, a path expression is a sequence of edge labels, as in $/books/book/identifier$. Here, the leftmost "/" denotes the root node, while the others denote edge traversal.

Given the notion of VOEM graphs, we can now give a definition of VOEM keys, which is similar to the notion of XML keys [16]. First, we need to introduce some notation. Given objects o_1 and o_2 with *oids* i_1 and i_2, respectively, we denote by $path(i_1, i_2)$ the path defined by the edge labels traversed to reach o_2 from o_1. As an example, $path(book_R, t_1) = book/title$. We also define functions $\texttt{valueOf}(i_1)$ and $labelOf(i_1)$ to extract the *value* and *label* associated with an object. To represent the set of objects reachable by a path expression, we introduce the following notation: $o[\![p]\!]$ is the set of VOEM objects obtained by following path p starting from the node identified by o. If o is not specified than p's traversal starts from the root of the graph. For example, $book_R[\![book/title]\!] = \{ \langle t_1, title, string, \text{"Voss"}, k_{t1}\rangle, \ldots, \langle t_n, title, string, \text{"Title}_n\text{"}, k_{tn}\rangle \}$

We are now ready to define VOEM keys. Let $o_1 = \langle i_1$, *label*, *type*, *value*, $k\rangle$ be an object, where $k = (i_2, \{sub_1, \ldots sub_n\})$. The key specifies that the value of the set of pairs $\{(\texttt{labelOf}(sub_1), \texttt{valueOf}(sub_1)), \ldots, (\texttt{labelOf}(sub_n), \texttt{valueOf}(sub_n))\}$ uniquely identifies o_1 among the objects in $i_2[\![path(i_2, i_1)]\!]$. Informally, the values of the subobjects uniquely identify o_1 among those reached by following the same path as o_1 in the subgraph rooted at the object with oid i_2. As an example, the key in the object $book_1 = \langle b_1, book, set, \{i_1, t_1, a_1\}, (book_R, \{i_1\})\rangle$ determines that $\{(\text{identifier}, 1)\}$ uniquely identifies object $book_1$ among the set of `book` objects in the subgraph rooted at $book_R$. That is, there are no two `book` objects that agree on their `identifier` attribute.

The adoption of VOEM as the interface between the relational and key-value pairs models requires that both the database and the datastore be associated with mappings for converting its underlying data to VOEM objects and back to its original format.

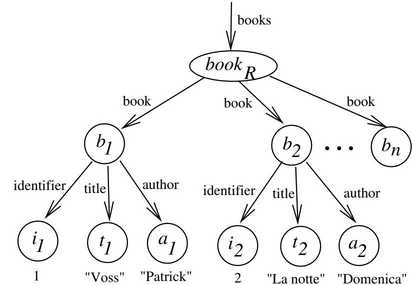
2) Relational to VOEM: The transformation of relational data to VOEM objects is a straightforward process. The four basic concepts of the relational model are: relation, tuple, attribute, and key. The first three concepts are related by a compositional hierarchy, in which a relation is a set of tuples, which in turn is a set of attributes with atomic values. This hierarchy can be directly represented by VOEM objects, with keys defined for tuple objects, in the context of relation objects. An example of this mapping is given in Figure 2. Here, a new object, with *oid* $book_R$ is created for representing the entire $books$ relation. It consists of a label `books`, of type `set` and has a set of *oid*s as its value, each of them associated with a tuple in the relation. Since in the relational model every relation has a primary key, the VOEM key defined for each tuple object is defined in the context of the relation object, and having all attributes that compose the primary key as VOEM key components. In the example of Figure 2b the key for the first `book` object is defined as $k_{b1} = (book_R, \{i_1\})$. That is, in the context of relation object $book_R$, a book tuple is identified by its subobject i_1, that corresponds to the primary key attribute `identifier` of the original relation.

Relation *books*		
identifier	***title***	***author***
1	Voss	Patrick
2	La notte	Domenica
3	Vendaval	Pilar
$\ldots$	$\ldots$	$\ldots$
n	$Title_n$	$Author_n$

(a) A relational table

$\langle book_R$, books, set, $\{b_1, b_2, \ldots, b_n\}$, $k_R\rangle$
$\langle b_1$, book, set, $\{i_1, t_1, a_1\}$, $k_{b1}\rangle$
$\langle i_1$, identifier, integer , 1, $k_{i1}\rangle$
$\langle t_1$, title, string , ''Voss'', $k_{t1}\rangle$
$\langle a_1$, author, string, '' Patrick '', $k_{a1}\rangle$
$\langle b_2$, book, set, $\{i_2, t_2, a_2\}$, $k_{b2}\rangle$
...
$\langle b_n$, book, set, $\{i_n, t_n, a_n\}$, $k_{bn}\rangle$
$\langle i_n$, identifier, integer , n, $k_{in}\rangle$
...

(b) The object structure of *books*



(c) The VOEM data graph

Fig. 2. Mapping relational data to a VOEM object structure.

3) VOEM to key-value pairs: A mapping from VOEM to key-value pairs is defined to satisfy one main condition: each pair must have a unique distinct *key* in order to avoid key-space collision. To distinguish the notion of a key in key-value pairs from VOEM keys, we use *storage key* and *storage value* to denote the components of a datastore pair. Given that VOEM objects have a distinct *oid*, the simplest way for mapping VOEM objects to key-value pairs, is by assigning the *oid* to the storage key, and the remaining fields to the storage value. Since *oids* do not carry any semantics, this approach does not allow data clustering. In order to explore key range-partitioning of the underlying datastore, storage keys should be defined based on VOEM keys. Based on our running example, an expression for extracting book authors as key-value pairs can be given as follows.

$$m = \{/book/author/ \circ \texttt{valueOf}(\$k), \texttt{valueOf}(\$a))|$$
$$\langle\ \$b,\ \texttt{book},\ _,\ _,\ (book_R,\ \{\$k\})\ \rangle \text{ in } [\![/books/book]\!],$$
$$\langle\ \$a,\ \texttt{author},\ _,\ _,\ _)\ \rangle \text{ in } \texttt{valueOf}(\$b)\}$$

Here, we use identifiers preceded with \$ to denote variables, "$\circ$" as an operator for string concatenation and "_" as placeholders for data that are not significant for the mapping. This mapping specifies an iteration over `book` objects in $[\![/books/book]\!]$. From each object in this set, the *oid* is extracted and assigned to variable $\$b$, while its key subobject is assigned to variable $\$k$. The value of b, given by its set of subobjects is then considered to obtain the one with label `author`. The *oid* of this object is then assigned to variable $\$a$. The storage key is constructed by concatenating path $/book/author/$ with the value of the $\$k$ object (the `identifier` attribute), and the storage value consists of the value of the `author` object identified by $\$a$. The result of this mapping on the VOEM objects of Figure 2b is the set of pairs $\{(/book/author/1, \text{“}Patrick''), \ldots (/book/author/n, \text{“}Author''_n)\}$. Similar mappings can be defined for generating pairs for `title` and `identifier` attributes. Observe that this mapping strategy results in a complete fragmentation of the relation, in which each "cell" of the table is individually stored in a key-value pair.

Note that if the datastore places data in order, according to storage key values, `author` key-value pairs generated according to mapping m given above, are likely to be stored in the same or close storage servers. This is because they all agree on their storage key prefix. On the other hand, pairs representing attributes that compose a tuple are likely *not* to be stored together. This is because storage keys like $/book/author/1$ and $/book/title/1$ are not closely located in a lexicographic order. Thus, this mapping defines a "column-based" (or vertical) distribution of data. In order to define a "tuple-based" distribution on the same "cell"-based fragmentation, the storage key should be built by concatenating path $/book/$ with the value of the `identifier` attribute, followed by path $/author$. In this case, keys $/book/1/author$ and $/book/1/title$ are in lexicographic order. Thus, these storage keys define a "tuple-based" (or horizontal) distribution of data. This shows that mappings through VOEM allow us to define not only different levels of granularity for data fragmentation, but also different ways of clustering fragments on storage servers when the underlying storage keeps data in order.

III. Phoenix: A Relational Cloud Datastore

We have developed a system, called Phoenix, based on the proposed architecture. In Phoenix, the logical layer consists of MySQL RDBMS [17], while Scalaris [14] acts as the cloud storage layer. Scalaris [14] is a transactional distributed key-value store. It runs on a virtual network, composed of a structured overlay network with logarithmic routing performance that is used to store and retrieve key/value pairs distributed among a set of nodes. The decision to use Scalaris is due to its features that closely match those defined in Section II-A. Scalaris ensures *content locality* by placing data in lexicographic order, based on storage keys. Transactions with strong consistency are provided by the Paxos consensus protocol. The consensus protocol is also used to implement transactions over multiple keys and to ensure that all replicas of a key are updated consistently. Besides, availability is achieved by symmetric replication [14].

For the logical layer, MySQL has been chosen because of its pluggable storage engine architecture. The storage engine is the underlying component of a RDBMS, responsible for defining how data structured as relations are physically stored, and also for exporting a relational view of stored data. MySQL architecture defines services that the storage engine must

provide such that new engines can be developed and integrated with other components of the system in a modular fashion. Thus, the development of Phoenix involves building a storage engine for MySQL based on Scalaris.

Conceptually, a storage engine should provide four basic functions: create, read, update and delete one tuple at a time. Thus, the development of Phoenix's storage engine involves both data mapping and operations transformations. Data mapping is provided by transforming data to an intermediate representation (VOEM objects), as proposed in Section II-B. Operations have been implemented entirely on top of the standard interface of a datastore. Thus, it can theoretically run on top of any datastore that exports the standard put-get programming interface (API). The primitive access path methods implemented by the storage engine are key (primary and unique) lookups, ordered storage key index scans (optionally with a range predicate) and full table scans. A key lookup is the most efficient method. Due to the way data is distributed, it is possible to determined exactly where the data is located. The remaining methods rely on the order preserving placement of values provided by Scalaris.

IV. Experimental Study

We have conducted an experimental study for determining the feasibility of Phoenix. The first experiment analyzes the overhead of layering MySQL on top of Scalaris, and also the scalability of the system. That is, using a synthetic workload, we measure the cost incurred by the relational layer on the datastore, and also the impact of adding new servers to the system. In order to validate the system with a workload that emulates an online transaction processing (OLTP) application, experiments with *SysBench* benchmark [18] have also been conducted. First, we consider a workload containing both read and write operations. Then, a read-only workload has been used for determining the effect of content locality on the system's performance.

These experiments were executed on a set of two to six servers interconnected through a departmental LAN. Each server has one dual core processor at 2.40GHz, with 2 GB of RAM and a 100Mbps Ethernet card. Servers have no hard disks, so data items are stored in memory. Not only data, but process instances of MySQL, Scalaris and other tools are also distributed among the servers. The communication between the storage engine and Scalaris is based on remote procedure calls (RPC) encoded in JSON (similar to XML-RPC) over HTTP, while communication among Scalaris nodes uses a native message passing layer. Scalaris is configured with a replication factor of two; that is, the system keeps two replicas of each data item stored.

A. Synthetic Workload

The goal of this experiment is to determine the overhead of accessing the datastore through a RDBMS. That is, we determine the additional cost incurred by retrieving data stored on a datastore using a high-level query language, as proposed by Phoenix. To this end, we have defined a simple transaction for incrementing a numeric attribute on a single tuple: `UPDATE table SET counter = counter + 1 WHERE id = key`. Here, `table` is a relation defined with attributes, `id` and `counter`, where `id` is a primary key. Two types of clients that execute 3000 iterations of this transaction have been developed. The first, denoted as *Scalaris client*, accesses data directly through a sequence of put-get operations packed in a Scalaris transaction. The other, called *Phoenix client*, invokes Phoenix to execute the SQL statement. When retrieving data directly from Scalaris, the (key, value) pair consists of the values of attributes `(id, counter)`.

The experiment was executed with two servers, each of them running a set of client applications, a MySQL server, and hosting a Scalaris node, as depicted in Figure 3. The number of clients deployed ranges from 16 to 128 in increments of 16. They are evenly distributed between the servers. That is, with 80 concurrent clients, there are 40 clients running on each server. Phoenix clients communicate with the MySQL server running on the same server, while Scalaris clients communicate directly with the DHT. In order to shield the experiment results from the number of possible conflicts among transactions, we have populated the relation (and the datastore) as follows. Each client is assigned a randomly generated unique identifier (GUID), and for each client a tuple (or key-value pair) is created using the GUID as its key. Then, each client executes a transaction for incrementing the counter associated with its corresponding tuple, thus avoiding any conflicts. Although in a real application conflicts among transactions are likely to occur, this setting allows us to precisely determine the effect of layering the RDBMS on the datastore.

In each run of the experiment either Scalaris clients or Phoenix clients were deployed on each server. Two metrics were collected: rate of transactions per second (TPS), and transaction response time (TRT) in seconds. By varying the number of clients, TPS provides a measurement on the workload the system is able to process, while TRT determines the impact of the workload on the system response time.

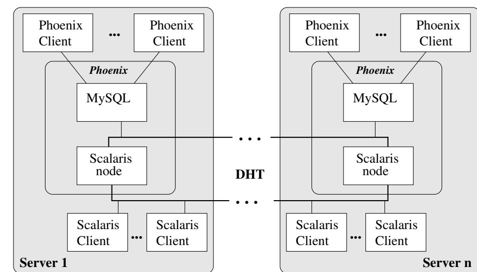


Fig. 3. Experiment with synthetic workload.

The results of this experiment are presented in Table I. The first column contains the number of clients being executed concurrently, while the following columns present the average TPS and TRT of Phoenix and Scalaris clients. For the TRT, it can be observed that for any number of clients, the difference

between the two settings is almost constant. That is, there is an overhead of around 3×10^{-3} seconds to run an update through Phoenix than executing it directly on Scalaris. We can conclude that the TRT is determined by the datastore, and the cost of Phoenix is minimum, given that the difference is practically negligible. However, Phoenix may cause a decrease on TPS of almost 18% in the worst case, and 9% in average, as shown in the last column. The impact of increasing the number of clients on TPS is around 50% when it grows from 16 to 32 and then to 48, as can observed from the three first lines of the table. From this point forward the degradation reaches lower levels.

TABLE I
EXPERIMENT WITH TWO SERVERS.

Clients	Phoenix		Scalaris		Dif. TPS %
	TPS	TRT	TPS	TRT	
16	16184	2.479×10^{-4}	17725	2.271×10^{-4}	-8.69
32	7608	5.258×10^{-4}	8357	4.804×10^{-4}	-8.97
48	4900	8.171×10^{-4}	4951	8.097×10^{-4}	-1.02
64	3340	1.198×10^{-3}	4069	9.872×10^{-4}	-17.90
80	2608	1.534×10^{-3}	2876	1.392×10^{-3}	-9.31
96	2134	1.877×10^{-3}	2202	1.820×10^{-3}	-3.08
112	1750	2.287×10^{-3}	1916	2.090×10^{-3}	-8.63
128	1460	2.743×10^{-3}	1461	2.738×10^{-3}	-0.06

In order to determine the impact of increasing the number of servers in the system, and thus horizontal scalability, we have run the experiment in the same setting, but with four servers. Table II shows the results. It can be observed that the average response time remains practically unchanged. For the TPS, on the other hand, the addition of 2 servers almost quadrupled the performance of the system. Since keys are evenly distributed among the nodes, the addition of nodes causes an expansion in the overall system capacity, but does not change the scenario for processing each individual transaction. This shows that the system is horizontally scalable.

TABLE II
EXPERIMENT WITH FOUR SERVERS.

Clients	Phoenix		Scalaris		Dif. TPS %
	TPS	TRT	TPS	TRT	
16	76029	2.167×10^{-4}	78290	2.108×10^{-4}	-2.89
32	37904	4.237×10^{-4}	36659	4.417×10^{-4}	3.40
48	21836	7.397×10^{-4}	22808	7.019×10^{-4}	-4.26
64	15114	1.061×10^{-3}	16358	9.823×10^{-4}	-7.60
80	12158	1.319×10^{-3}	12785	1.253×10^{-3}	-4.90
96	10180	1.577×10^{-3}	10771	1.489×10^{-3}	-5.48
112	8194	1.957×10^{-3}	8764	1.835×10^{-3}	-6.51
128	6860	2.339×10^{-3}	7192	2.227×10^{-3}	-4.61

B. SysBench benchmark

SysBench is a benchmark for assessing the performance of OLTP workloads. Clients are implemented using *threads* and communicate with a MySQL server using *sockets*. In the advanced transactional mode of SysBench, used in this experiment, each transaction request is composed of 21 operations, with the last one being a *commit*. Among transactions, 66% are read operations, while 23% write or remove tuples. All operations are on a relation with 100 thousand tuples and four attributes. The complete list of transactions, as well as the table schema can be found in [18].

We have run the experiment on four servers, but the setting is slightly different from the one shown in Figure 3. One Scalaris node has been deployed in each server, but only one of them has a MySQL server and Phoenix clients running Sysbench. Tests were executed varying the number of clients that access the database, from 1 to 128, with increments of 16. Table III shows the values collected when running the experiment for 300 seconds. In order to obtain stable results, the first transaction execution for each round was discarded. Despite the short period for each measurement, samplings on intervals up to 20 minutes showed no significant differences.

The first and second columns of the table contains the number of concurrent Phoenix clients, and the number of transactions per second (TPS), respectively. The remaining columns are measurements of transaction response times (TRT): the minimum, maximum, average, and 95th percentile of execution times in milliseconds.

TABLE III
SYSBENCH ON OLTP ADVANCED TRANSACTIONAL MODE.

Clients	TPS	Min.	Avg.	Max.	95th
1	213.06	3.22	4.69	52.57	9.96
2	380.50	3.85	5.25	51.14	6.13
4	537.56	4.87	7.43	71.31	9.34
8	600.25	3.75	13.32	1817.00	17.88
16	595.47	10.42	26.86	306.18	37.97
32	560.34	20.68	57.09	1830.21	79.54
64	506.88	4.15	126.24	1960.06	159.19
128	76.09	446.70	1681.76	11633.69	1935.99

The results show that for up to 4 concurrent clients there is an increase on TPS, and TRTs remain practically stable. For 8 clients, TPS is still higher, but compared to the previous round, with 4 clients, the maximum TRT is around 25 times higher and the 95th percentile doubles. From this point forward, by adding 16 clients, we can observe that there is a slight decrease on TPS, and both the average and 95th percentile response times almost double. However, with 128 clients there is a significant degradation on the system performance: transactions take 1.9 seconds on average to be executed, and can reach up to 11 seconds. We believe that this is due to the computational power of the servers. With two processing cores, the server reaches its limit at 64 clients, and with additional ones, competition for resources may lead to CPU overhead and transaction conflicts, which impacts the system performance. This experiment shows that on this setting, Phoenix can be effective for supporting OLTP applications with up to 64 simultaneous clients. However, for applications in which the response time is very restrictive, the number of clients may not exceed 8.

C. SysBench benchmark with Fragmentation

Analyzing SysBench, we observed that the number of range queries in the workload is greater than that of equality queries. Thus, a strategy of horizontally fragmenting the relation, such that range queries only need to access a few nodes may have an impact on the system's performance. In order to test this conjecture we have set an experiment in which range queries access 1 to 3 fragments stored on different Scalaris nodes, and determine the impact of this distribution on the number of transactions per second the system can execute.

The experiment setting is similar to the one described in Section IV-B but with 6 Scalaris nodes and only one of them executing a MySQL server and Sysbench clients. We have used a 3000 tuples relation such that every range query returns 200 tuples as its result. By maintaining a constant result size, the cost of the communication in the system is mainly based on the network latency, given that the volume of data transmitted among nodes remains almost unchanged. The relation is horizontally split in different number of fragments, such that in each run the system needs to access 1 to 3 different fragments to retrieve the 200 tuples in the result. We considered three scenarios. In the first, the relation is partitioned in 12 fragments of 250 lines, with each Scalaris node storing 2 fragments. In the second scenario, each node host 3 fragments of 167 lines each. The third consists of 24 fragments of 125 lines, with a distribution of 4 fragments in each node.

Table IV shows the results of this experiment, with the number of Phoenix clients running Sysbench on read-only mode varying from 2 to 10. The second, third, and fifth columns show the number of transactions per second (TPS) in each round, while the fourth and sixth columns show the rate decrease on TPS, compared to the round with 12 fragments.

TABLE IV
FRAGMENTATION EXPERIMENT.

Clients	TPS				
	Fragments:12	Fragments:16	%	Fragments:24	%
2	421.37	415.86	-1.30	402.10	-4.57
4	606.89	589.91	-2.80	579.55	-4.51
6	699.98	666.12	-4.84	657.40	-6.08
8	715.86	669.63	-6.46	658.29	-8.04
10	696.39	652.72	-6.27	643.62	-7.58

Corroborating the results in the previous experience, the peak TPS of the system is with 8 concurrent clients in all three scenarios, which also shows the maximum impact of data distribution on the system performance. This impact ranges from 1.3% to 6.46% when a second fragment is accessed and from 4.57% to 8.04% when three fragments are needed. Given that the volume transmitted data in all three scenarios is almost the same, these results show that the cost of the number of messages exchanged among the nodes is not negligible. This highlights an important trade-off. Ideally, we would like to minimize both the volume of data transmitted and the amount of messages exchanged. By increasing the size of a fragment, the number of messages may decrease, but the volume of (unwanted) data may increase. Some datastores provide a functionality for "packing" a set of requests for the same node into a single message for minimizing the cost of message passing. This functionality evidences the importance of maintaining data locality, as proposed by Phoenix. If the set of data requested by a query is maintained in the same or close nodes, this may reduce the number of messages, which may increase the system performance, as shown by this experiment.

V. Related Work

Architectures based on P2P principles have been widely applied to enhance the underlying mechanisms of storage platforms in the cloud, producing a lot of research in distributed systems with special focus on highly scalable and fault tolerant data storage. These systems have been categorized in three classes, according to their data model [19]: *Key-value stores*, which support a model based on values and an index for data retrieval (as Scalaris [14]); *Document stores*, with a model that defines a key on a set of attribute-value pairs (as Dynamo [5], the storage system of SimpleDB [4]); and *Extensible record stores*, which follow a column-oriented model (as Cassandra [9] and BigTable [7], the underlying datastore of the App Engine [6]).

A characteristic among these systems is an inclination towards self-descriptive (semi-structured) data models. However, most of the systems have a custom query language with limited expressive power. In both SimpleDB and App Engine, query operators such as joins and aggregation have to be implemented at the application level. A similar example is the low-level API of the Amazon S3 service [20], which only allows *put* and *get* operations on a key-value datastore. Our stratified architecture allows the use of a high-level SQL language provided by standard RDBMSs at the logical level. Thus, Phoenix provides low startup and maintenance costs through a standard user interface.

Designed for web applications, with emphasis on the performance of read operations, App Engine and Amazon datastores provide limited support for general OLTP applications. This is because most of them do not support strong consistency. Consistency in these systems is closely related to the data physical partition and distribution model, contradicting traditional notions of logical and physical independence. One example of this close interconnection is that the properties of consistency, scalability and replication are only ensured in the context of node *groups* on App Engine.

Alternative architectures for OLTP systems are evaluated in [21]. This study shows that systems, like SimpleDB, App Engine, MySQL Cluster [22] and MS Azure [8], achieve scalability through architectures based on replication, partitioning, and caching. However, in all these systems there are scalability limitations given that they all rely on a centralized database server control which maintains master copies. Similar to S3 architecture, the distributed control of Phoenix allows high scalability because the storage system and database servers are loosely coupled. Thus, servers can access concurrently

and autonomously the shared data stored on the key-value storage system. However, Phoenix differs from S3 by adopting a RDBMS at the logical level while in S3 high-level operations must be implemented on the application level.

Bridging the gap between cloud datastores and database systems has also been the focus of other works [23], [24]. The integration of MySQL and Cassandra [9] datastore is described in [24]. Although the system architecture is similar to Phoenix's, some issues for integrating datastores with database technology such as data mapping is not considered. The system described in [23] has similar goals, but has been developed with a different architecture which consists of a cluster of commodity servers running traditional RDBMSs with a distributed transaction mechanism on top.

The idea of developing an interface on top of existing datastores has also been adopted by AppScale [25] and *Yahoo!* [26]. However, the goal of both tools differs from Phoenix since they provide frameworks for comparing cloud datastores. To the best of our knowledge, there are no previous work that explore mappings between relational data and key-value pairs.

VI. CONCLUSION

Unlike traditional distributed RDBMS, our preeminent motivation for developing a new cloud based relational storage engine is to provide a scale-out database system without sacrificing key features found in traditional relational database systems, such as transactions and data independence. We have defined a general framework for data and operations mapping between the logical and physical layers of the proposed architecture, based on VOEM. Besides providing a general means for data mapping, VOEM can be used to support other data models in the logical level, such as XML.

We have argued that support for content locality is a key functionality that should be provided by a datastore to support the proposed architecture, as well as support for a strong notion of consistency. We describe the development of Phoenix, based on MySQL RDBMS and Scalaris datastore. Our experimental study shows that Phoenix provides scalability and relational functionality on top of a datastore with very low overhead. We use SysBench, an OLTP benchmark, for showing the feasibility of our approach, and also for measuring the impact the number of messages exchanged among datastore nodes may have on the system performance. Our results show that the communication of one additional node for retrieving data may decrease around 6% the number of transactions per second (TRT) the system can execute. This evidences the importance of data locality on Phoenix.

There are a number of issues we intend to investigate in the future. One of them is to explore the database storage engine interface with the query optimizer and index structures. Thus, plan refinements can allow the storage engine to (potentially) deploy parallel processes for data filtering and query evaluation, eliminating the need to transmit irrelevant tuples. Other issues that deserve further investigation include: comparison with other systems; storage of metadata in the cloud datastore, including access permissions, data fragmentation, key generation and other features related to multi-tenancy; efficient support for constraints, such as foreign keys; mechanisms for database recovery, such as periodic checkpoints; workload analysis for automatic generation of a data fragmentation schema; and definition of a high-level SQL-like language to allow user-defined fragmentation schemes.

REFERENCES

[1] L. Tang, J. Dong, Y. Zhao, and L.-J. Zhang, "Enterprise Cloud Service Architecture," in *IEEE Cloud10: International Conference on Cloud Computing*, 2010, pp. 27–34.

[2] Armbrust, Michael and et al, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

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

[4] Amazon, "Amazon SimpleDB," http://aws.amazon.com/simpledb/, 2007.

[5] G. DeCandia and et al, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[6] Google, "App Engine datastore," 2008. [Online]. Available: http://code.google.com/appengine/docs/datastore/

[7] F. Chang and et al, "Bigtable: a distributed storage system for structured data," in *Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, p. 15.

[8] Microsoft, "Windows Azure platform," 2010. [Online]. Available: http://www.microsoft.com/windowsazure/

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

[10] B. F. Cooper and et al, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[11] S. Das, D. Agrawal, and A. E. Abbadi, "G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud," in *ACM SOCC*, 2010, pp. 163–174.

[12] J. Rao, E. J. Shekita, and S. Tata, "Using Paxos to build a scalable, consistent, and highly available datastore," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 243–254, Jan. 2011.

[13] N. J. Harvey and et al, "Skipnet: A Scalable Overlay Network with Practical Locality Properties," in *USENIX Symposium on Internet Technologies and Systems*, 2003.

[14] T. Schütt and et al, "Scalaris: Reliable Transactional P2P Key/Value Store," in *ACM SIGPLAN workshop on ERLANG*, 2008, pp. 41–48.

[15] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object Exchange Across Heterogeneous Information Sources," in *IEEE International Conference on Data Engineering*, 1995, pp. 251–260.

[16] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan, "Keys for XML," *Computer Networks*, vol. 39, no. 5, pp. 473–487, 2002.

[17] Oracle, "MySQL Server," 2010. [Online]. Available: http://www.mysql.com/

[18] MySQL AB, "SysBench manual," http://sysbench.sourceforge.net/docs/#database_mode, 2010.

[19] R. Cattell, "High Performance Scalable Data Stores," available at http://cattell.net/datastores/Datastores.pdf/, Apr. 2010.

[20] M. Brantner and et al, "Building a database on S3," in *ACM SIGMOD international conference on Management of data*, 2008, pp. 251–264.

[21] D. Kossmann and et al, "An evaluation of alternative architectures for transaction processing in the cloud," in *ACM International conference on Management of data*, 2010, pp. 579–590.

[22] M. Ronström and L. Thalmann, "MySQL Cluster Architecture Overview - High Availability Features of MySQL Cluster," MySQL Technical White Paper, Tech. Rep., Apr. 2004.

[23] C. Curino and et al, "Relational Cloud: A Database-as-a-Service for the Cloud," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, 2011, pp. 235–240.

[24] D. Egger, "SQL in the Cloud," Master's thesis, Swiss Federal Institute of Technology Zurich (ETH), Sep. 2009.

[25] C. Bunch and et al, "An Evaluation of Distributed Datastores Using the AppScale Cloud Platform," in *IEEE Cloud10: International Conference on Cloud Computing*, Jul. 2010, pp. 305–312.

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.