# A Framework for Supporting DBMS-like Indexes in the Cloud

Gang Chen
*Zhejiang University*
cg@cs.zju.edu.cn

Hoang Tam Vo, Sai Wu, Beng Chin Ooi
*National University of Singapore*
{voht, wusai, ooibc}@comp.nus.edu.sg

M. Tamer Özsu
*University of Waterloo*
tozsu@cs.uwaterloo.ca

## ABSTRACT

To support "Database as a service" (DaaS) in the cloud, the database system is expected to provide similar functionalities as in centralized DBMS such as efficient processing of ad hoc queries. The system must therefore support DBMS-like indexes, possibly a few indexes for each table to provide fast location of data distributed over the network. In such a distributed environment, the indexes have to be distributed over the network to achieve scalability and reliability. Each cluster node maintains a subset of the index data. As in conventional DBMS, indexes incur maintenance overhead and the problem is more complex in the distributed environment since the data are typically partitioned and distributed based on a subset of attributes. Further, the distribution of indexes is not straight forward, and there is therefore always the question of scalability, in terms of data volume, network size, and number of indexes.

In this paper, we examine the problem of providing DBMS-like indexing mechanisms in cloud DaaS, and propose an extensible, but simple and efficient indexing framework that enables users to define their own indexes without knowing the structure of the underlying network. It is also designed to ensure the efficiency of hopping between cluster nodes during index traversal, and reduce the maintenance cost of indexes. We implement three common indexes, namely distributed hash indexes, distributed $B^+$-tree-like indexes and distributed multi-dimensional indexes, to demonstrate the usability and effectiveness of the framework. We conduct experiments on Amazon EC2 and an in-house cluster to verify the efficiency and scalability of the framework.

## 1. INTRODUCTION

The cloud simplifies the deployment of large-scale applications by shielding users from the underlying infrastructure and implementation details. There is substantial interest in cloud deployment of data-centric applications. One good example application is the Customer Relationship Management (CRM), which is used to monitor sales activities, and improve sales and customer relationships. While there are daily account maintenance and sales activities, there are certain periods when sales quota must be met, forecasting and analysis are required, etc., and these activities require more resources at peak periods, and the cloud is able to meet such inconsistent resource requirements.

To support data-centric applications, the cloud must provide an efficient and elastic database service with database functionality, which is commonly referred to as the "database as a service" (DaaS). This work is part of our cloud-based data management system, named epiC (elastic power-aware data-intensive Cloud)[1], which is designed to support both analytical and OLTP workloads. In this paper we focus on the provision of indexing functionality in the context of DaaS. One obvious requirement for this functionality is to locate some specific records among millions of distributed candidates in real time, preferably within a few milliseconds. A second requirement is to support multiple indexes over the data – a common service in any DBMS. Another important requirement is extensibility by which users can define new indexes without knowing the structure of the underlying network or having to tune the system performance by themselves.

Currently no DaaS satisfies these requirements. Most popular cloud storage systems are key-value based, which, given a key, can efficiently locate the value assigned to the key. Examples of these are Dynamo [5] and Cassandra [8]. These systems build a hash index over the underlying storage layer, partitioning the data by keys (e.g., primary index). For supporting primary range index, a distributed B-tree has been proposed [2]. While these proposals are efficient in retrieving data based on primary index, they are not useful when a query does not use the key as the search condition. In these cases, sequential (or even parallel) scanning of an entire (large) table is required to retrieve only a few records, and this is obviously inefficient.

Recently, Cassandra [8] has started to support native distributed indexes on non-key attributes. However, these secondary indexes are restricted to hash indexes. A second line of research has been view materialization in the cloud to support ad hoc queries. In [1], a view selection strategy was proposed to achieve a balance between query performance and view maintenance cost. Secondary indexes can be implemented as a specific type of materialized views. Maintaining materialized views is orthogonal to our research and the comparison with this approach is our on going research.

Two secondary indexes have been proposed for cloud systems: a distributed $B^+$-tree-like index to support single-dimensional range queries [22], and a distributed R-tree-like index to support multi-dimensional range and kNN (k Nearest Neighbor) queries [21]. The main idea of both indexes is to use P2P routing overlays as global indexes and combine with local disk-resident indexes at each index node. The overlays are used to build a logical network for partitioning data, routing queries and distributing the system workload. In contrast, IR based strategies in integrating distributed independent

[1] http://www.comp.nus.edu.sg/∼epiC


*Proceedings of the VLDB Endowment,* Vol. 4, No. 11

databases over unstructured network, without any global index, are proposed in [13].

Like Dynamo [5] and Cassandra [8], our system adopts the peer-to-peer (P2P) model to provide highly available service with no single point of failure. Moreover, using DHT overlays as infrastructure also allows building an elastic indexing service where index nodes can be added or reduced based on load characteristics. In addition, while the proposals in [21, 22] illustrate the feasibility of supporting an index using the underlying overlay network, it is not feasible to support multiple indexes using these approaches since it is expensive to maintain multiple overlays – each overlay for a specific index – over the distributed cluster nodes. In order to provide an indexing functionality that is able to support a variety of basic indexes efficiently, we propose an extensible indexing framework that supports multiple indexes over a single generic overlay.

Our framework is motivated by the observation that many P2P overlays are instances of the Cayley graph [12, 14]. Consequently, we abstract the overlay construction by a set of Cayley graph interfaces. By defining the customized Cayley graph instances, a user can create different types of overlays and support various types of secondary indexes. This approach avoids the overhead of maintaining multiple overlays, as discussed above, while maintaining flexibility, and it achieves the much needed scalability and efficiency for supporting multiple indexes of different types in cloud database systems.

The main challenge in developing this framework is how to map different types of indexes to the Cayley graph instances. To address this problem, we define two mapping functions, a data mapping function and an overlay mapping function. The data mapping function maps various types of values into a single Cayley graph key space. We propose two data mapping functions: a uniform mapping function, which assumes the data are uniformly distributed and maps data to different keys with the same probability, and a sampling-based mapping function, which maps data based on the distribution of samples. The overlay mapping function is composed of a set of operators that represent the routing algorithms of different Cayley graph instances. The user can define new types of overlays by implementing new operators, which simplifies the inclusion and deployment of new types of indexes. Additionally, the use of data mapping and overlay mapping reduces the cost of index creation and maintenance.

Performance tuning in a cloud system is not trivial, and therefore, our indexing framework is designed to be self tunable. Independent of the index type, our self tuning strategies optimize the performance by adaptively creating network connections, effectively buffering local indexes and aggressively reducing the random I/Os.

In summary, the contributions of the paper are as follows.

- We propose an extensible framework for implementing DBMS-like indexes in the cloud. We exploit the characteristics of Cayley graphs to provide the much needed scalability for supporting multiple distributed indexes of different types.
- We define a methodology to map various types of data and P2P overlays to a generalized Cayley graph structure.
- We propose self-tuning strategies to optimize the performance of the indexes defined over the generic Cayley overlay, thereby achieving high performance and scalability.
- We implement the proposed framework and conduct extensive experimental study on Amazon EC2 and an in-house cluster. The results confirm the robustness, efficiency and scalability of our proposed framework.

The paper proceeds as follows. In Section 2, we give an overview of the indexing framework. We present the Cayley graph based indexing scheme in Section 3. We evaluate the performance of the indexes in Section 4 and conclude the paper in Section 5.

## 2. OVERVIEW OF THE FRAMEWORK

The proposed framework consists of an indexing service that is loosely connected with other cloud services, such as storage service and processing service. This is a commonly accepted design principle in the cloud, which enables different services to scale up and down independently. The indexing service runs as a set of distributed processes on the virtual machines. The indexing service interacts with the underlying storage service where the base tables are stored, e.g. distributed file systems (DFS), and provides data retrieval interface for the upper layer applications.

We refer to the index process that runs on a virtual machine as an index node. The index nodes are organized into a generic Cayley graph based overlay. Figure 1 shows the architecture of the indexing service. The multiple types of indexes supported by the

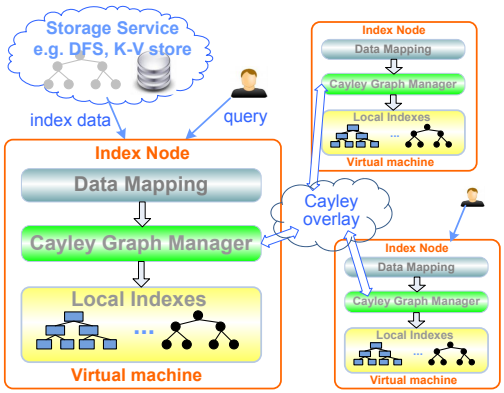


**Figure 1: Indexing service in the cloud**

framework are mapped to Cayley graph instance managed by the Cayley Graph Manager. One Cayley graph instance is required for each type of index. For example, a hash index on a string or numeric attribute is mapped to a Chord [19] instance while a kd-tree or R-tree index on multiple attributes can be supported by a CAN [16] instance. Cayley graph is described further in Section 3.1. We define a generalized key space $\mathcal{S}$ for the Cayley graph. A client application simply needs to define a data mapping function $\mathcal{F}$ that maps an index column $c$ to a value in $\mathcal{S}$. Based on the type of indexes, different $\mathcal{F}$s can be defined. If the index is built to support range index, $\mathcal{F}$ must preserve the locality of data. Otherwise, $\mathcal{F}$ can be defined as a universal hash function to balance the system load.

After being mapped with function $\mathcal{F}$, the value of the index keys are normalized to the Cayley graph key space. The detailed description of the data mapping technique is presented in Section 3.2. The indexing process in our framework is similar to the publication process in P2P overlays. Specifically, an index entry is composed of a key-value pair $(k, v)$, where $k$ denotes the value of the index column and $v$ is the offset of the tuple in the underlying DFS or the value of the tuple itself (or possibly portions of the tuple) if the user opts to use *covering indexes*. Under the covering index scheme, the indexes include additional, non-key columns in the index entries so that they can service the queries without having to refer to the base records, i.e., the indexes "cover" the queries. To index this key-value pair $(k, v)$, we apply a P2P overlay routing protocol to publish $(k, v)$ based on $k'$, the mapped value of $k$ with the map-

ping function $\mathcal{F}$. Upon receiving a key-value pair $(k, v)$ and the mapped key $k'$ from the data mapper, the Cayley graph manager retrieves the corresponding instance of the index column and applies the routing protocols of the instance to index the data. Based on the routing protocol of the Cayley graph instance, the index node in the cluster system is responsible for a key set and needs to keep a portion of index data (i.e., the published data).

A typical database application such as Human Resource Management or Customer Relationships Management has many tables, and each table has a few indexes. Given the size of the current datasets, the index data will be too large to be maintained in memory. To address this problem, each index node is expected to create one local disk-resident index such as the hash table or B$^+$-tree for maintaining the index data of a distributed index. Thus, the index lookup is performed in two steps. In the first step, we follow the routing protocol defined by the Cayley graph operators to locate the node responsible for the index data. In the second step, we search the node's local index to get the index data. To further improve the search performance, we employ a buffer manager locally to reduce the I/O cost for traversing local indexes.

It is important to note that we aim to design a simple yet efficient and scalable solution for indexing data in the cloud. Our indexing system is different from current proposals [21, 22] in two main aspects. First, our indexing system provides the much needed scalability with the ability to support a large number of indexes of different types. Second, in our design the indexing system is loosely coupled with the storage system while the indexes in [21, 22] ride directly on the data nodes of the storage system. Decoupling system functions of a cloud system into loosely coupled components enables each component to scale up and scale down independently.

# 3. CAYLEY GRAPH-BASED INDEXING

In this section, we present a Cayley graph-based indexing scheme and use it to support multiple types of distributed indexes, such as hash, B$^+$-tree-like and multi-dimensional index, on the cloud platform.

## 3.1 Mapping P2P Overlays to Cayley Graph

Cayley graph, which encodes the structure of a discrete set, is defined as follows.

DEFINITION 1. ***Cayley Graph*** *: A Cayley graph $\mathcal{G} = (\mathcal{S}, G, \bigoplus)$, where $\mathcal{S}$ is an element set, $G$ is a generator set and $\bigoplus$ is a binary operator, is a graph such that*

1. *$\forall e \in \mathcal{S}$, there is a vertex in $\mathcal{G}$ corresponding to $e$.*
2. *$\bigoplus : (S \times G) \rightarrow \mathcal{S}$.*
3. *$\forall e \in \mathcal{S}$ and $\forall g \in G$, $e \bigoplus g$ is an element in $\mathcal{S}$ and there is an edge from $e$ to $e \bigoplus g$ in $\mathcal{G}$.*
4. *There is no loop in $\mathcal{G}$, namely $\forall e \in S, \forall g \in G \rightarrow e \bigoplus g \neq e$.*

In a Cayley graph, we create a vertex for each element in $\mathcal{S}$. If for elements $e_i$ and $e_j$, there is a generator $g$ satisfying $e_i \bigoplus g = e_j$, then edge $(e_i \rightarrow e_j)$ is created.

Each node in the Cayley graph can be considered a peer, allowing the Cayley graph to define a P2P overlay. Many P2P overlays, including Chord [19], an expanded BATON [7], and CAN [16], can be mapped to an instance of Cayley graph [12]. We discuss how this mapping occurs.

When a cluster node assumes the role of an index node in the Cayley graph, we use a hash function to generate a unique value in $\mathcal{S}$ as its identifier. Suppose the list of index nodes is $\{n_0, n_1, ..., n_d\}$ and let $I(n_i)$ denote node $n_i$'s identifier. $I(n_i)$ refers to an element in $\mathcal{S}$ and an abstract vertex in the Cayley graph. To support index construction, we partition the element set into subsets with continuous elements based on nodes' identifiers. We first sort the nodes by their identifiers and hence, $I(n_i) < I(n_{i+1})$. The subset $\mathcal{S}_i$ is defined as

$$\mathcal{S}_i = \begin{cases} \{x | I(n_{i-1}) < x \leq I(n_i)\} & \text{if } x \neq 0 \\ \{x | 0 \leq x \leq I(n_i) \vee I(n_d) < x \leq \mathcal{S}.max\} & \text{otherwise} \end{cases}$$

where $\mathcal{S}.max$ is the maximal element in $\mathcal{S}$. Node $n_i$ is responsible for subset $\mathcal{S}_i$. The overlay mapping problem is formalized as:

DEFINITION 2. ***Overlay Mapping*** *: Given a P2P overlay $\mathcal{O}$ and a Cayley graph $\mathcal{G} = (\mathcal{S}, G, \bigoplus)$, $\mathcal{O}$ can be mapped to $\mathcal{G}$ by using the following rules:*

1. *For a node $n$ in $\mathcal{O}$, $I(n)$ is defined as an element in $\mathcal{S}$.*
2. *Given two nodes, $n_i$ and $n_j$, $n_j$ is a routing neighbor of $n_i$, iff there is a generator $g$ in $G$, satisfying $I(n_i) \bigoplus g \in \mathcal{S}_j$.*

In a Cayley graph, the element set and generator set can be defined arbitrarily, which makes the mapping problem very complex. In our proposal, we fix the element set $\mathcal{S}$ and the generator set $G$ as $\{x | 0 \leq x \leq 2^m - 1\}$ and $\{2^i | 0 \leq i \leq m - 1\}$, respectively. In this way, the overlay mapping problem is transformed into finding a proper $\bigoplus$ for each overlay. Popular P2P overlays such as Chord [19], CAN [16] and BATON [7] can be easily integrated into the above model. In fact, these three overlays are sufficient to support DBMS-like indexes similar to those commonly available in commercial centralized DBMS, namely the hash, B$^+$-tree and R-tree.

In the following example, we show how to map Chord to the Cayley graph by defining a proper *operator*. Details of mapping other overlays are shown in Appendix A.

In our framework, Chord is a $2^m$-ring since Cayley graph's element set is $\{x | 0 \leq x \leq 2^m - 1\}$ (we set $m = 30$ in our experiments to support large datasets). We define the operator $x \bigoplus y$ as $(x + y) \bmod 2^m$. Therefore, given a node $n_i$ and its identifier $I(n_i)$ in the element set $S$, we create edges between $I(n_i)$ to keys $I(n_i) + 2^k$ $(0 \leq k \leq m - 1)$, based on the generator set $\{2^i | 0 \leq i \leq m - 1\}$. Namely, each node maintains $m$ routing entries, which follows the structure of Chord. Algorithm 1 shows the routing algorithm that simulates the Chord protocol. In Algorithm 1, $n_i$ refers to the index node that receives the routing request and $start$ is the node's identifier. Basically, we iterate all generators and try to route the message as far as possible. In line 7, given a key, the function $getNodeByKey$ returns the address of the index node that is responsible for the key based on the routing table.

**Algorithm 1** `ChordLookup(Node` $n_i$`, Key` $dest$`)`

```
1. Key start= I(n_i)
2. if start == dest then
3.    return node
4. else
5.    for i=m-1 to 0 do
6.       if (start + 2^i mod 2^m) ≤ dest then
7.          Node nextNode = getNodeByKey(start + 2^i)
8.          return nextNode
```

## 3.2 Data Mapping

The indexed columns may have different value domains. Since we use the values of indexed columns as the keys to build the index, before publishing an index entry with key $k$ we need to normalize

$k$ into a value $k'$ in the element space $\mathcal{S}$. For this purpose, some key mapping functions are required. Given an element domain $\mathcal{S}$ and an attribute domain $\mathcal{D}$, the mapping function $f : \mathcal{D} \rightarrow \mathcal{S}$ maps every value in $\mathcal{D}$ to a unique value in $\mathcal{S}$.

Suppose we have $d$ index nodes, $\{n_0, n_1, ..., n_d\}$, and use $\mathcal{S}_i$ to denote the element subset of $n_i$. Given a table $T$, if the index is built for $T$'s column $c_0$, the number of index entries published to $n_i$ is estimated as:

$$g(n_i) = \sum_{t_j \in T} \Phi(f(t_j.c_0)) \quad (1)$$

where function $\Phi(x)$ returns 1 if $x \in \mathcal{S}_i$, or 0, otherwise.

A good mapping function should provide the properties of ***locality*** and ***load balance***.

DEFINITION 3. ***Locality*** *: The mapping function $f : \mathcal{D} \rightarrow \mathcal{S}$ satisfies the locality property, if $\forall x_i \forall x_j \in \mathcal{D} \wedge x_i < x_j \rightarrow f(x_i) \leq f(x_j)$.*

DEFINITION 4. ***$\epsilon$-Balance*** *: The mapping function $f : \mathcal{D} \rightarrow \mathcal{S}$ is an $\epsilon$-balance function for column $c_0$, if for any two cluster nodes, $n_i$ and $n_j$, $\frac{g(n_i)}{g(n_j)} < \epsilon$.*

Locality requirement is used to support range queries, but is not necessary for the hash index. Load balance guarantees that the workload is approximately uniformly distributed over the index nodes. Definitions 3 and 4 can be extended to support multi-dimensional (multi-column) indexes. For the $d$-dimensional case, the mapping function is defined as $f : \mathcal{D}_0 \times ... \times \mathcal{D}_d \rightarrow \mathcal{S}$, while the locality is measured by the average value of $L(x_i, x_j) = \frac{d(x_i, x_j)}{|f(x_i) - f(x_j)|}$, where $d(x_i, x_j)$ returns the Euclidean distance between two multi-dimensional points, $x_i$ and $x_j$.

DEFINITION 5. ***d-Locality*** *: The mapping function $f : \mathcal{D}_0 \times ... \times \mathcal{D}_d \rightarrow \mathcal{S}$ satisfies the locality property, if the average $L(x_i, x_j)$ is bounded by a function of $d$.*

In our current implementations, we provide two data mapping functions: a uniform mapping function and a sampling-based mapping function.

### 3.2.1 Uniform Data Mapping

The uniform mapping function assumes that the data are uniformly distributed in the key space. In the one dimensional case, given a key $k$ in original key space $[l, u]$, we map it to the new value

$$f(k) = min(2^m - 1, \lfloor \frac{(k - l)2^m}{u - l} \rfloor) \quad (2)$$

For example, suppose the original domain is $[0, 10]$ and the key space of Cayley graph is $[0, 2^2]$, keys 6 and 8 will be mapped to values 2 and 3, respectively.

THEOREM 1. *When the data distribution is uniform, Equation 2 provides a mapping function that has the properties of locality and 1-balance.*

PROOF. Equation 2 is a monotonic increasing function and scales the original domain to the new key space using the same factor, $\frac{2^m}{u-l}$. Therefore, it satisfies the two properties. □

For a string value, the uniform mapping function is defined as a hash function $h$, which maps the string to a value in $\mathcal{S}$. If the index needs to support range queries for strings, $h$ is implemented as a locality sensitive hashing [4].

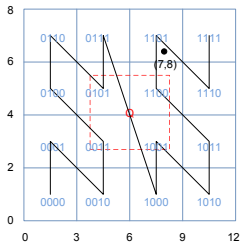


**Figure 2: Mapping Multi-Dimensional Data**

In the multi-dimensional case, we partition the space in a kd-tree style. The key space is partitioned into sub-spaces by different dimensions iteratively. In the $x$th partition, we partition each sub-space evenly by the $j$th dimension, where $j$=$x$ mod $d$. Suppose we have $2^a$ sub-spaces before partitioning. The next iteration will generate $2^{a+1}$ sub-spaces. The partitioning process terminates when $2^m$ partitions are created. Then, we assign each partition an $m$-length binary string as its ID, recording its partitioning history. The ID can be transformed back to a value in $[0, 2^m - 1]$, namely the key space of the Cayley graph. Note that in uniform mapping, sub-spaces have equal size. Suppose the Cayley graph key space is $[0, 2^4 - 1]$ and the data domains are $x = [0, 12]$ and $y = [0, 8]$, respectively. Figure 2 shows how a 2-D space is partitioned. The point $[7, 8]$ is transformed into 1101. Linking the partitions with adjacent IDs generates a multi-dimensional Z-Curve.

THEOREM 2. *Z-Curve mapping provides a mapping function that has the properties of d-locality and 1-balance for uniform distributions.*

PROOF. Hilbert-curve has been proven to satisfy d-Locality using the metric properties of discrete space-filling curve [6]. The same proof technique can be applied for Z-Curve. Although Z-Curve performs a bit worse than Hilbert-Curve, it still preserves the locality property. As we split the space into equal-size partitions, we also achieve the 1-balance property for uniform distribution. □

### 3.2.2 Sampling-based Data Mapping

If data distribution is skewed, the uniform mapping function cannot provide a balanced key assignment. Hence, some nodes may need to maintain more index data than the others, which is undesirable. Consequently, we may need to use a load balancing scheme to shuffle the data dynamically during query processing, which is costly. In our framework, a sampling-based approach is used to address this problem.

Before we map and partition the space, we first collect random samples from the base tables to get a rough estimate of the data distribution. Seshadri and Naughton [17] showed that stratified random sampling method can guarantee perfect load balancing without reducing the accuracy of the estimation. Specifically, the domain being sampled is partitioned into disjoint subsets and a specified number of samples is taken from each subset.

Based on the retrieved samples, we map the data to $\mathcal{S}$ in such a way that each partition in $\mathcal{S}$ has approximately the same number of samples. In the one dimensional case, the partitioning strategy is equivalent to building an equal-depth histogram. In the $k$-dimensional case, we apply the kd-tree style partitioning. When partitioning a space, we guarantee that the generated subspaces have the same number of samples.

THEOREM 3. *The sampling-based mapping keeps locality and provides $\log_2 N$-balance ($N$ is the total number of cluster nodes), if the samples provide an accurate estimate of the overall data distribution.*

PROOF. Since the proof of locality is similar to the uniform case, we do not discuss it. For any two cluster nodes, $n_i$ and $n_j$, $\frac{|\mathcal{S}_i|}{|\mathcal{S}_j|} < \log_2 N$ [19]. In our sampling-based mapping approach, each sub-space has the same number of samples. When we distribute the sub-spaces in the cluster, node $n_i$ will get $k$ sub-spaces, where $k$ is proportional to $|\mathcal{S}_i|$. Therefore, if the samples provide an accurate estimation for the data distribution, sampling-based mapping approach has the property of $\log_2 N$-balance. □

It is notable that bulk insertion from external data sources into cloud databases is a common operation [18]. For example, in a webshop application, the system needs to bulk insert the daily feed of new items from partner vendors into its operational table. Sampling operations are typically done during this bulk insertion process [18]. Therefore, the statistics such as data domains and data distribution could be estimated quite accurately. Note that these statistics might become obsolete due to many skewed online updates after the bulk insertion, and the distribution of the index data among index nodes will become unbalanced as a result. However, when the level of load imbalance reaches a predefined threshold, the system can activate a data migration process to redistribute the index data and update the data mapping function correspondingly.

Moreover, besides the default uniform and sampling-based mapping functions, the user can define his own mapping function by extending the interface of the framework. In this interface, the abstract mapping function is defined. The user can overload the function with his customized implementation. When a mapping function is linked with an index, our indexing framework will automatically invoke it to transform the data for indexing and querying.

**Query Mapping.** The objective of distributed indexes is to facilitate fast retrieval of a subset of data without having to scan every data node. The query optimizer of the client applications will decide if index scan or full table scan should be employed. For queries that involve a small portion of data that fall within a small range, a simple but efficient mapping solution is sufficient to handle such a query pattern. A range query $Q$, in the one dimensional case, is mapped into a single key range, while in the multi-dimensional case, is transformed into multiple key ranges. For example, in Figure 2, the query $Q = \{3.5 \leq x \leq 8.5, 3 \leq y \leq 5.5\}$ is transformed into four key ranges, $[0011, 0011]$, $[0101, 0101]$, $[1001, 1001]$ and $[1100, 1100]$. To retrieve the index for $Q$, we need to search the four key ranges in the Cayley graph.

## 3.3 Index Functions

In this section, we describe three basic functions provided by the indexing framework for client applications to build the indexes, perform search on the indexes, and update data in the indexes.

### *3.3.1 Index Building*

In our current framework, we have provided the operators for Chord [19], BATON [7] and CAN [16], as those overlays are used to build the common distributed hash, B$^+$-tree-like and R-tree-like indexes. The Cayley graph manager considers each operator as a class of overlays. In the initialization process of an index for column $c$, suppose its operator is *op*, the Cayley graph manager registers the index as an instance of *op*. In other words, each operator can have multiple instances, referring to different indexes on the same type of overlays. The Cayley graph manager also keeps the information of table name, column name, value domain and data mapping function, which it broadcasts to all index nodes to initialize them.

Following the initialization, the index is created for each incoming tuple. Algorithm 2 shows how the index is built for a tuple. First the overlay instance of the index is obtained from the Cayley graph manager, which is then used to map and publish the data. In line 6, the $lookup$ function is an abstraction of the underlying routing algorithms, which will be transformed into different implementations of the overlay. In line 7, the $getIndexData(t)$ function returns the offset of tuple $t$ in the underlying DFS or the value of tuple $t$ itself (or possibly portions of the tuple $t$) if the user opts to use covering indexes. Algorithm 2 demonstrates the extensibility of our indexing framework. It hides all the implementation details, such as how data are mapped and which routing algorithms are used, by providing a highly abstract interface for users.

**Algorithm 2** `Insert(Tuple` $t$`, CayleyManager` $M$`)`

```
1. for every column c_i of t do
2.   if M.isIndexed(c_i) then
3.     Instance I = M.getInstance(c_i)
4.     MappingFunction F = M.getMappingFunction(c_i)
5.     Key k = t.c_i
6.     Node n=I.lookup(F(k))
7.     IndexData v = getIndexData(t)
8.     publish (k, v) to n
```

### *3.3.2 Index Search*

Regardless of the underlying overlays, the $Search$ method can be abstracted as outlined in Algorithm 3. The inputs of $Search$ are a search key and the index column name. The column name is used to identify the specific index and the corresponding overlay. The query engine first asks the Cayley graph manager to get the overlay instance for the column. Then, it invokes the $lookup$ method of the overlay, which will return the index node that is responsible for the search key.

**Algorithm 3** `Search(Key` $k$`, CayleyManager` $M$ `Column` $c_i$`)`

```
1. Instance I = M.getInstance(c_i)
2. MappingFunction F = M.getMappingFunction(c_i)
3. Node n=I.lookup(F(k))
4. Array< IndexValue > values = n.localSearch()
5. for i = 0 to values.size-1 do
6.   getIndexTuple(values[i])
```

In line 4, after receiving the request, the index node performs a search on its local disk-resident indexes to retrieve the indexed data of the key, denoted as as a set of index values. If the index being used is a covering index, then the returned index values themselves are the tuples in the query result set. Otherwise, the index values are only the pointers to the base records, i.e. the offsets of the records in the underlying DFS. In this case, the system needs to invoke the interfaces of the underlying DFS to perform random accesses based on these offsets.

Range search can be processed in a similar way, except that in line 3, multiple index nodes could be returned. In this case, the query should be processed by these nodes in parallel. We note that this parallelism mechanism is especially useful for processing equi-join and range join queries. The joined columns of different tables typically share the same data semantics and the index data of these columns are normally partitioned and distributed over the index nodes in the same way. Therefore, these index nodes can process the join queries in parallel. In addition, the support of parallel scans of different indexes also facilitates correlated access across multiple indexes, which is necessary when a query accesses multiple indexed columns. Note that the indexing service only provides basic interfaces for upper layer applications to access the in-

dex data, while the join order between tables is determined by the upper layer query optimizer such as the storage manager of a cloud storage system [3] that includes the indexing service to extend its functionalities.

### 3.3.3 Index Update

Client applications update the index via the interface of the framework. Receiving the update request from clients, the indexing service realizes this update in two steps. First, the corresponding old index entries of the update are deleted from the system. Then, the update is inserted into the indexes as a new index entry. The performance of update operations is studied in Appendix D.3.

It is notable that the enforcement of consistency and ACID properties for index update is based on the requirements of applications. Since there is a trade-off between performance and consistency, the client applications, based on its consistency requirements, will determine the policy to perform index updates. The indexes could be updated under strict enforcement of ACID properties or in a less demanding bulk update approach.

In the former approach, the client applications need to reflect all modifications on the base records to the associated indexes before returning acknowledgement messages to users. This approach typically requires the client applications to implement a refresh transaction in order to bring all associated index data up-to-date with the base data. Since these data are possibly located on different machines, an efficient distributed consensus protocol is more desirable than the traditional two-phase commit protocol. Paxos-based algorithm has recently been shown as a more efficient alternative approach [15].

In the latter approach, the client applications can backlog the modification on the base records and bulk update these modifications to the associated indexes.In addition, the client applications can dynamically determine the frequency of index updates in runtime based on the system workload. Specifically, when the client application faces a peak load, i.e., a sudden increase in the input load, it can defer the index bulk update to reserve system resources for handling the user's request first, and then resume the index bulk update process after the peak load. We note that the problem of guaranteeing ACID properties in the cloud is a broad research issue itself [11] and developing an efficient strategy for the maintenance of the proposed cloud indexing service is part of our future work.

## 3.4 System Performance and Availability

It is impractical for a user or upper layer application to do performance tuning with the existence of multiple types of indexes. Therefore, we identify the factors that may affect the performances for all types of indexes and apply general strategies to improve the global performance of the system.

First, the index process routes queries based on the operators defined in the Cayley graph instances; however, it is expensive for each process to maintain active network connections to all other processes. Second, the memory capacity of the index process relative to the application size is limited and all the index entries cannot be cached. To solve these issues, regardless of the index types, our self tuning strategies optimize performance by adaptively creating network connections and effectively buffering local indexes. We describe these techniques in Appendix B.

In addition, the proposed indexing service is designed to meet service level agreements (SLA) such as 24×7 system availability – an important desideratum of cloud services, which requires the system's ability to handle failures on the index nodes. We describe the replication of index data to ensure correct retrieval of data in the presence of failures in Appendix C.

# 4. PERFORMANCE EVALUATION

We have run a series of experiments to evaluate the robustness, efficiency and scalability of our proposed indexing framework. First, we study the query performance with two query plans: *index covering* approach where the index entries contain a portion of the data records to service the query request directly and *index+base* approach where the index entries only contain pointers to the records in the base table.

Second, we compare the performance of distritbuted indexes against parallel full table scans. Third, we study the scalability of the system in terms of both the system size and the number of indexes. In Appendix D, we present additional experimental results on the effect of varying data size, the effect of varying query rate, the update performance, the ability of handling skewed data and query distribution, and the performance of equi-join and range join query.

## 4.1 Experimental Setup

We test the indexing framework in two environments. We ran a set of experiments in an in-house cluster to stress test the system with varying query and update rates. The cluster includes 64 machines with Intel X3430 2.4 GHz processors, 8 GB of memory, 500 GB of disk capacity and gigabit ethernet.

We also conduct experiments on a cluster of commodity machines on Amazon EC2 to test the robustness and scalability of the indexing framework when varying the system size from 16 to 256 nodes. Each index node in our system runs on a small instance of EC2. This instance is a virtual machine with a 1.7 GHz Xeon processor, 1.7 GB memory and 160 GB disk capacity.

We run most of experiments with TPC-W benchmark dataset[2]. TPC-W benchmark models the workload of a database application where OLTP queries are common. These queries are relatively simple and selective in nature, and thus building indexes to facilitate query processing is essential, especially in a large scale environment such as the cloud.

We generate 10 million to 160 million records (each record has an average size of 1KB) of *item* table. Thus, the total data size ranges from 10 GB to 160 GB. The data records are stored in the underlying HDFS[3] and sorted by their primary key, i.e., the *item_id* attribute. We build distributed indexes on the *item_title* attribute and the *item_cost* attribute by respectively instantiating a distributed hash index and distributed B$^+$-tree-like index based on the proposed indexing framework. We test the performance of the distributed indexes with two types of queries, namely exact match query with a predicate on the *item_title* attribute:

Q1: SELECT item_id FROM item WHERE item_title ='$\xi$'

and range query with a predicate on the *item_cost* attribute:

Q2: SELECT item_id FROM item
WHERE item_cost $> \alpha$ AND item_cost $< \beta$

By setting the values of $\alpha$ and $\beta$, we can define the selectivity of the test queries.

In addition, to test the system with a bigger number of indexes, we also synthetically generated data and indexes as follows. There are multiple tables $T_i$ with schema $T_i(a_1, a_2, a_3, a_4, p)$ where each attribute $a_i$ takes integer values that are randomly generated from the domain of $10^9$ values, and attribute $p$ is a payload of 1 KB data. Each table is generated with 10 million records. For each table $T_i$, the attribute $a_1$ is indexed with a distributed hash index, $a_2$ is indexed with a distributed B$^+$-tree-like index, and $(a_3, a_4)$ is indexed with a distributed multi-dimensional index. Thus, each table $T_i$ has 3 indexes, and we can test the effect of varying number of indexes in the system by increasing the number of testing tables.

[2]http://www.tpc.org/tpcw

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

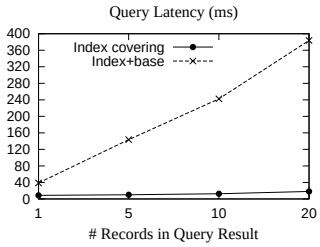


**Figure 3: Index covering vs. index+base**

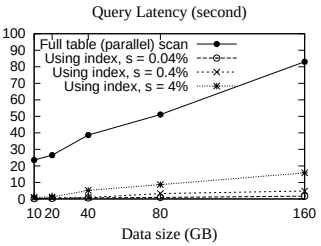


**Figure 4: Index plan vs. full table scan**

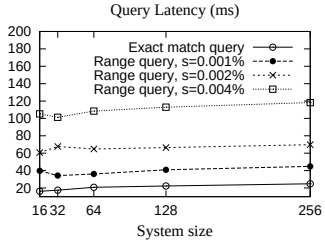


**Figure 5: Scalability test on query latency (on EC2)**

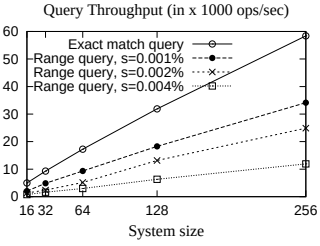


**Figure 6: Scalability test on query throughput (on EC2)**

Table 1 summarizes the default experiment configuration. The default system size for the experiments is 64 index nodes. The memory buffer for the local indexes at each node is set to 64 MB. The system uses the adaptive connection management strategy as the default setting. We test the system with the default 10 client threads at each node. Each client thread continually submits a workload of 100 operations to the system. A completed operation will be immediately followed up by another operation. We also vary the query rate and update rate submitted into the system by changing the number of client threads at each node.

Table 1 Experiment Settings

| Parameter | Default | Min | Max |
|---|---|---|---|
| System size | 64 | 16 | 256 |
| Data size | 10 GB | 10 GB | 160 GB |
| Buffer size for local indexes | 64 MB | - | - |
| Number of client threads per node | 10 | 5 | 50 |
| Number of operations per thread | 100 | - | - |
| Query type | Exact match | - | - |
| Query plan | Index covering | - | - |

## 4.2 Index covering vs. Index+base Approach

In this experiment, we study the query processing performance using distributed indexes with two alternative query plans, namely *index covering* and *index+base*. In the former approach, the index entries include the data of non-key attributes to service the queries directly. For example, if the index entries of the distributed index on *item_cost* contain the *item_id* information, then the above query Q2 can be processed efficiently with only index traversal. On the contrary, in the latter case, the query processing needs to follow the pointers in the index entries and perform random reads on the base table, which is costly when the base table is stored in the HDFS.

Figure 3 shows that the *index covering* approach outperforms the *index+base* approach, especially when the the size of the query result set is large. Note that even though the *index+base* has worse performance in this case, it still performs better than scanning the whole table to retrieve only a few qualified data records. In our experiment, even a parallel scan using MapReduce[4] on the 10 GB $item$ table takes about 23 seconds (Figure 4), which is significantly slower than the *index+base* approach. Moreover, it is also notable that the cost of both *index covering* and *index+base* approach depend only on the size of the query result set, while the cost of full (parallel) table scans increase with the table size.

In the extreme case of the *index covering* approach, users can opt to include all the data of the original records in the index entries to speed up query processing. Although this approach incurs additional storage cost, the query performance using the secondary indexes is improved considerably. Moreover, the additional storage cost is acceptable in the case the sizes of data records are relatively small or we only include a portion of a data record that is needed for common queries. Hence, we mainly test the performance of the system with the *index covering* query plan in other experiments.

[4]http://hadoop.apache.org/mapreduce/

## 4.3 Index plan vs. full table parallel scan

In this test, we vary the data set size from 10 GB to 160 GB (from 10 million to 160 million records). We compare the performance of the system to process the query Q2 using the distributed $B^+$-tree-like index with the approach that performs a full parallel scan on the $item$ table using MapReduce in a system of 64 cluster nodes.

As shown in Figure 4, when the data set size increases, the query latency of the distributed index approach also increases due to the increasing size of the result set. However, it still performs much better than the full table scan approach, whose query response time increases almost linearly along with the data set size.

The distributed index achieves better performance because it can directly identify the qualified records and retrieve them from the local indexes of the index nodes, while in the other approach the whole table is scanned. The parallel scans with MapReduce still consume a significant execution time when the table size is large. Note that the distributed index also employs parallelism to speed up query processing. When the query selectivity is set to 4% or higher, the data that qualify the query could be stored on multiple index nodes. These index nodes would perform the index scan in parallel and the query latency would not increase with the size of result set any longer.

## 4.4 Scalability test on EC2

In this experiment, we evaluate the scalability of the proposed indexing service in terms of the system size. We vary the system size from 16 to 256 virtual machines on EC2 and measure the latency and throughput of queries with different selectivities.

As can be seen from Figure 5, the system scales well with nearly flat query latency when the system size increases. In our experimental setting, the workload submitted to the system is proportional to the system size. However, more workload can be handled by adding more index nodes to the system. Therefore, the system response time for a query request with respect to a specific query selectivity is maintained nearly unchanged with different number of index nodes. Figure 5 also shows that a query with lower selectivity incurs higher latency, due to the larger result set, local processing costs, and higher communication cost.

As the number of index nodes increases, the aggregated query throughput also increases as shown Figure 6. In addition, the system query throughput scales almost linearly when the query has high selectivity, especially for the exact-match query. With a high query selectivity, the result set is small and therefore, the local processing at each index node and data transfer have less effect on the query throughput. More importantly, the indexing service achieves better throughput when there are more high selectivity queries for its ability of being able to identify the qualified data quickly rather than scanning multiple data nodes.

## 4.5 Multiple Indexes of Different Types

In this test, we experiment with 8 tables, each of which has 3 distributed indexes of the 3 different index types as described in the experimental setup. The workload, i.e., the number of client threads, submitted to the system is increased with the number of indexes in the system (1 client thread for each index). We expect more indexes in the system to be able to handle a bigger workload from the users. Figure 7 and Figure 8 plot the effect of varying the number of indexes on the query latency and system throughput respectively.

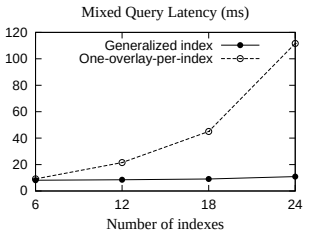


**Figure 7: Query latency with multiple indexes**

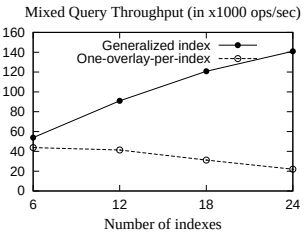


**Figure 8: Query throughput with multiple indexes**

The results confirm the superiority of our *generalized index* over the *one-overlay-per-index* approach that runs one overlay for a specific index, i.e., 24 overlays totally for 24 indexes in this test. The *generalized index* can guarantee a constant query latency and its query throughput increase steadily with the number of indexes that it instantiates. This is due to the fact that with more indexes, there will be more index traversal paths that can be used to answer the queries. In addition, the query execution load is better shared among the index nodes in the cluster.

While the *generalized index* approach only runs one index process to maintain multiple indexes and self-tunes the performance among these indexes via sharing resources such as memory and network connections, the *one-overlay-per-index* approach runs multiple processes, each for a specific overlay. When there are more indexes, the more processes need to be launched, which considerably adds overhead to the virtual machine and affects the query latency and throughput. Therefore, our approach provides the much needed scalability for supporting multiple indexes of different types in the cloud.

# 5. CONCLUSION

This paper presents an extensible indexing framework to support DBMS-like indexes in the cloud. The framework supports a set of indexes using P2P overlays and provides a high level abstraction for the definition of new indexes. The most distinguishing feature of our scheme is the ability to support multiple types of distributed indexes, such as hash, $B^+$-tree-like and multi-dimensional index, in the same framework, which significantly reduces the maintenance cost and provides the much needed scalability. To achieve this goal, we define two mapping functions to transform different indexes into the Cayley graph instances. We exploit the characteristics of Cayley graph to reduce the index creation and maintenance cost, and embed some self tuning capability. Finally, we evaluate our indexing scheme in an in-house cluster and Amazon EC2 with extensive experiments, which verify its efficiency and scalability.

# 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 and the shepherd for their insightful comments.

# 6. REFERENCES

[1] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *SIGMOD*, pages 179–192, 2009.

[2] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *PVLDB*, 1(1):598–609, 2008.

[3] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Es2: A cloud data storage system for supporting both oltp and olap. In *ICDE*, 2011.

[4] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262, 2004.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 2007.

[6] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, 5(5):794–797, 1996.

[7] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.

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

[9] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In *BNCOD*, pages 20–35, 2000.

[10] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 1981.

[11] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.

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

[13] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, pages 633–644, 2003.

[14] C. Qu, W. Nejdl, and M. Kriesell. Cayley dhts - a group-theoretic framework for analyzing dhts based on cayley graphs. In *ISPA*, pages 914–925, 2004.

[15] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, 4(4):243–254, 2011.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[17] S. Seshadri and J. F. Naughton. Sampling issues in parallel database systems. In *EDBT*, pages 328–343, 1992.

[18] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *SIGMOD*, 2008.

[19] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[20] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 3(1):506–517, 2010.

[21] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD*, 2010.

[22] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient b-tree based indexing for cloud data processing. *PVLDB*, 3(1):1207–1218, 2010.

# APPENDIX

## A. OVERLAY MAPPING

We had demonstrated how Chord [19] can be mapped to a Cayley graph in Section 3.1. Here we discuss the mapping of BATON [7] and CAN [16].

***1. BATON***

To support BATON overlay, $x \bigoplus y$ is defined as $(x+y)\%2^m$ as well since BATON can be transformed to Chord by adding a virtual node. In our case, the key space is $[0, 2^m - 1]$ and thus the maximal level of BATON tree is $m$. Given a BATON node $n_i$ at level $l$, suppose its ID at level $l$ is $x$, $n_i$ can be transformed into a node in Chord by using the following function:

$$\theta(l, x) = 2^{m-l}(2x - 1)$$

The routing neighbors of BATON are similar to those of Chord except for the parent-child and adjacent links. If $n_i$ is a left child of its parent, then the links to its parent node, right child and right adjacent node can be emulated by $I(n_i) + 2^k$. Since BATON has a tree topology, $n_i$'s left adjacent link and left child link cannot be emulated by Chord's routing fingers, and therefore we define new generators ($2^m - 2^x$, $x$ is an integer) to handle the links. However, to keep the framework generic, we choose to use the old generator set, namely $\{2^k | 0 \leq k \leq m-1\}$. Even without half adjacent/child links, we note that the queries can still be routed to the destination node using a similar routing scheme as in Algorithm 1.

***2. CAN***

CAN partitions multi-dimensional space into zones such that each zone is managed by a node. The kd-tree space partitioning provides a good basis for multi-dimensional data indexing. Compared to other overlays, supporting CAN is more challenging, as we need to establish a mapping between CAN's identifiers (multi-dimensional vector) and Cayley graph's key space (1-dimensional value). There are many works on dimensionality reduction, such as Space Filling Curve [9].The basic idea is to partition the search space by each dimension iteratively, and assign a binary ID to each sub-space, which is mapped to the Cayley graph based on its ID. Similarly, a multi-dimensional query is transformed into a set of sub-spaces, which are denoted by their IDs as well.

In CAN, we define $x \bigoplus y$ as *x XOR y*. The basic routing algorithm is shown in Algorithm 4. We first get the key space of current node (lines 1 and 2). If the search key $dest$ is covered by current node, the lookup process can stop by returning current node (lines 3 and 4). Otherwise, we handle the lookup as in the following two cases. First, if this is the first node receiving the search request, the initial key $start$ is empty. We iterate all keys in the node's key space to find the one, which has the longest common prefix with the search key $dest$. That key will be used as the initial key $start$ (lines 5-10). Second, if the node is not the first node in the search route, the initial key $start$ has already been decided by the last node in the route (lines 12 and 13). In either case, we attempt to reduce the difference between $dest$ and $start$ by routing to a property neighbor (lines 14 and 15).

## B. PERFORMANCE SELF TUNING

In Section 3.4 we identified the factors that influence the performance of the proposed framework. Here we discuss our solutions to these issues.

### B.1 Index Buffering Strategy

Each index process maintains the index data for different Cayley graph instances. To support efficient retrieval of index data, given a Cayley graph instance $g$, its index data, denoted as $\mathcal{I}(g)$, are stored in a local disk-resident index structure of the index nodes. Based on the type of $g$, different index structures are built for $\mathcal{I}(g)$. For example, if $g$ is a Chord [19] overlay, then a hash index is created. Otherwise, if $g$ is an instance of CAN [16], we create an R-tree index for $\mathcal{I}(g)$. Similarly, a B$^+$-tree index is created if $g$ is an instance of BATON [7].

Figure 9 shows how the local indexes are maintained. In this example, each index node is responsible for maintaining the index data for three Cayley graph instances. We use the instance ID (IID) to identify a specific instance in the Cayley graph manager.

**Algorithm 4** `CANLookup(Node` $n_i$`, Key` $start$`, Key` $dest$`)`

1. Key $k_0$= $I(n_i.predecessor)$
2. Key $k_1$= $I(n_i)$
3. **if** $dest > k_0$ and $dest \leq k_1$ **then**
4. return $n_i$
5. **if** $start == NULL$ **then**
6. **for** $i = k_0 + 1$ to $k_1$ **do**
7. $p = getCommonPrefix(i, dest)$
8. **if** $p.length > maxlength$ **then**
9. $maxlength = p.length$
10. $start = i$
11. **else**
12. $p = getCommonPrefix(start, dest)$
13. $maxlength = p.length$
14. Node $nextNode$ = $getNodeByKey(start$ *XOR* $2^{maxlength+1})$
15. return $nextNode$

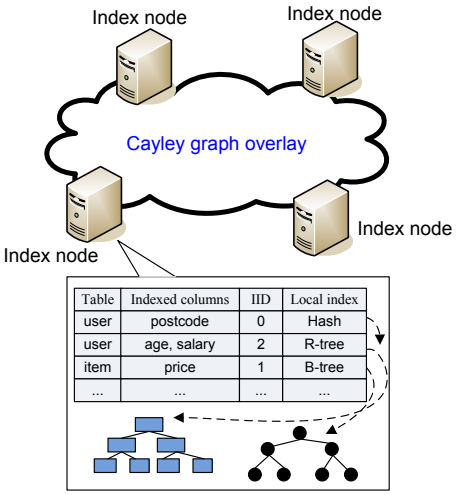


**Figure 9: Local Indexes**

To improve the performance of the local disk-resident indexes of each index node, we buffer some index entries, i.e., the nodes of B$^+$-tree and R-tree, or buckets of hash index, in memory. However, the available memory of the virtual machines hosting the indexing service relative to the application size is limited. Therefore, each index process establishes a buffer manager to manage the buffer dynamically.

Let $Idx = \{e_1, e_2, ..., e_m\}$ be all index entries on the disk, where $e_i$ represents a node of B$^+$-tree and R-tree, or a bucket of hash index. We define two functions to measure the importance of an index entry. $f(e_i)$ returns the number of queries involving

$e_i$ in last $T$ seconds and $g(e_i)$ is the size of $e_i$. Suppose $M$ bytes are used to buffer the local indexes, we define a buffering vector $v = (v_1, v_2, ..., v_m)$. When $e_i$ is buffered in memory, $v_i$ is set to 1. Otherwise, $v_i$ equals to 0. The buffer manager tries to find a buffer vector that maximizes

$$\sum_{i=1}^{m} v_i f(e_i)$$

and satisfies

$$\sum_{i=1}^{m} v_i g(e_i) \leq M$$

This is a typical knapsack problem. We solve it using a greedy algorithm. After every $T$ seconds, we periodically run the above algorithm to select the index entries for buffering. The old entries are replaced by new ones to catch the query patterns.

## B.2 Adaptive Network Connection

In our framework, the index process routes queries based on the operators defined in the Cayley graph instances. The approach to maintaining a complete connection graph is not scalable since each index process can only maintain a limited number of open connections. Therefore, in our system a connection manager is developed to manage the connections adaptively. It attempts to minimize the routing latency by selectively maintaining the connections. The connection manager classifies the connections as *essential connections* and *enhanced connections*. An essential connection is an active network connection established between two index processes $(I_p, I'_p)$ where $I'_p$ is a routing neighbor of $I_p$ by the definition of any Cayley graph instance in the Cayley graph manager. An enhanced connections is established at runtime between two frequently communicating processes.

By maintaining essential connections, we keep the overlay structures defined by Cayley graph instances. Suppose $K$ types of indexes are defined in the framework for a cluster of $N$ nodes, each node will maintain at most $Klog_2N$ essential connections, with $log_2N$ connections for each type. That is, even if we have thousands of indexes defined for the tables using these $K$ types of indexes, we will need only $Klog_2N$ essential connections. Queries can then be routed based on the overlay routing protocols with the essential connections.

Enhanced connections can be considered as shortcuts for essential connections. When routing a message, the index process first performs a local routing simulation using the Cayley graph information. Then, it checks the enhanced connections for shortcuts. If no shortcut exists, it follows the normal routing protocols and forwards the message via an essential connection. Otherwise, it sends the message via the available enhanced connections, which is adaptively created during query processing as follows.

To route a query, the connection manager performs a local routing simulation based on the Cayley graph information, and get a path $P$ which is a set of essential connections. We generate enhanced connections by replacing the chain of essential connections in $P$ by a shortcut connection. Specifically, connection $c = (I_p, I'_p)$ is a candidate enhanced connection for $P$ if there exists a shortcut path $P'$ of $P$, satisfying that $P'$'s starting node is $I_p$ and its ending node is $I'_p$.

The connection manager keeps a counter for each candidate connection, recording the number of appearances of the connection during query processing. After every $T$ seconds, the connection manager discards current enhanced connections and adds the top frequently used connections into the set of enhanced connections. The counters are then reset to 0 and a new tuning iteration starts.

# C. FAILURES AND REPLICATION

While there could be fewer failures in a cloud environment relative to the churn experienced in a P2P system, machine failures in large clusters are more common. Therefore, our indexing service employs replication of index data to ensure the correct retrieval of data in the presence of failures. Specifically, we use the two-tier partial replication strategy to provide both data availability and load balancing, as proposed in our recent study [20]. The first tier of replication, which consists of totally $K$ copies of the index data, is designed to guarantee the data reliability requirement ($K$ is commonly set to 3 in distributed systems with commodity servers [5, 15]). At the second tier, additional replicas of frequently accessed data are adaptively created in runtime based on the query workload as a way to distribute the query load on the "hot" queried data across their replicas.

Given a Cayley graph instance $g$, its index data, denoted as $\mathcal{I}(g)$, are partitioned and distributed on multiple index nodes. The index data on an index node (master replica) are replicated to the successors (slave replicas) of that index node. Note that the successors of an index node regarding to a specific index are determined by the type of $g$ (e.g. Chord [19], BATON [7], CAN [16]), or more specifically, the operator of the Cayley graph (cf. Section 3.1). When the master replica fails, we apply the Cayley graph operator to locate its successors and retrieve the index data from one of these replicas. Hence, the query processing of our indexing service is resilient to the failures of index nodes.

Paxos-based replication algorithm [15] has been shown to be feasible to maintain strict consistency of replicas. However, the tradeoff is the complexity of the system and the performance of write operations. When the relaxed consistency is acceptable for client applications, the replicas can be maintained asynchronously instead. The master replica is always updated immediately, while the update propagation to slave replicas can be deferred until the index node has spare network bandwidth and its peak load has passed.

To avoid the "lost updates" problem, i.e., the master replica crashes while the update has not been propagated to other slave replicas, the system performs write-ahead logging before updating the master replica. Hence, in our indexing service, the updates to the master replica of the index data are durable and eventually propagated to the slave replicas. Note that if the master replica fails during the processing of an update, its immediate successor will be promoted to take over the mastership. Therefore, when an index node recovers from a failure, it can retrieve back all latest updates from the slave replica that previously has been promoted to the mastership. We studied the complete method for system recovery from various types of node failures in [20].

# D. ADDITIONAL EXPERIMENTS

In this section, we present additional five sets of experimental results: 1) the effect of varying data size 2) the effect of varying query rate 3) the performance of update operations 4) the ability of handling skewed data and query distribution and 5) the performance of equi-join and range join query. The default experimental configuration is the same as described in Section 4.1.

## D.1 Effect of Varying Data Size

In this experiment, we perform the scalability test on data size and study the query performance in the system. Specifically, we measure the latency of exact-match queries on the $item\_title$ attribute, a non-key attribute of the $item$ table. By instantiating a distributed hash index on this attribute based on the proposed indexing framework, the system can support queries on this attribute efficiently without the need of full table scan. As shown in Figure

10, the query response time using distributed indexes is not affected by the database size.

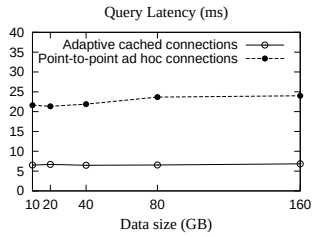


**Figure 10: Effect of varying data size**

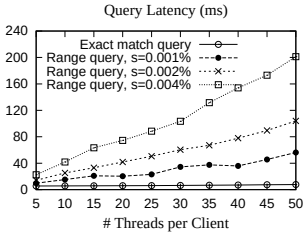


**Figure 11: Effect of varying query rate**

In addition, the advantage of the proposed adaptive connection management strategy is well demonstrated. The system can guarantee low query latency for users with the use of adaptive cached connections. In this approach, each index node in the cluster 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 which is the norm in the case of ad-hoc point-to-point connection approach.

## D.2 Effect of Varying Query Rate

In this test, we study the performance of exact match queries and range queries using the distributed indexes when we vary the query selectivity and the query input rate.

Figure 11 demonstrates that the system has better query latency with higher query selectivity. This is due to the fact that with high query selectivity, the result set is small and the system does not need to spend much time to scan the local disk-resident index at the index nodes in the cluster and retrieve the qualified records.

The results also show that the latency of exact-match queries is less affected by input query rate than that of range queries. As discussed above, range queries incur more local disk scans than an exact-match query. When the input query rate is high, more queries will compete with each other for the disk I/Os. Thus, the latency of range queries increases with the query input rate.

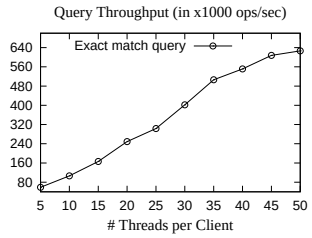


**Figure 12: Query throughput**

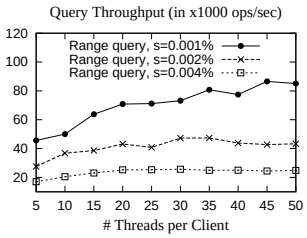


**Figure 13: Query throughput**

More importantly, the advantage of our proposal is well demonstrated in Figure 12. The system achieves better load when there are more concurrent exact match queries. With the use of indexes, we facilitate better load distribution, since we do not have to scan all nodes just to get the exact match tuples, and due to the ability of being able to identify the storage node that contains the tuple quickly, we only search that node, and search it efficiently. Therefore, the system can admit more queries and the throughput increases linearly along with the input load.

The system is also able to server better load when there are more high selective range queries, e.g., 0.001%, as shown in Figure 13. However, range queries with lower selectivities incur more local disk scans, and thus the throughput of range queries is more constrained by the input load. Specifically, for range queries with selectivities 0.002% and 0.004%, when the input load reaches the threshold of 15 client threads per node, more queries will compete for disk resources and the system throughput does not increase with the input load any longer.

## D.3 Update Performance

An update to the distributed indexes supported in our indexing framework is performed in two steps. First, the corresponding old index entries (if exists) of the update will be deleted from the system. Then, the update will be inserted into the indexes. Since the old and the new index entry might reside on different machines we need to perform two rounds of index traversal to process an update request, which increases the network cost.

In addition, update operations might also need to modify the local disk-resident index pages. Thus, an update operation in the indexes is much costlier than a search operation during query processing. Another source of latency cost of update operations is the concurrency control on the local indexes, which we employ a similar approach as $B^{link}$-tree [10], to guarantee the correctness of concurrent updates.

In summary, the latency cost of an update to a distributed index in our framework consists of three factors: the network cost, the local index update cost, and the concurrency cost. We note that regarding to the three types of distributed indexes implemented in our framework, namely distributed hash, $B^+$-tree, and R-tree indexes, the network cost and concurrency cost of update operations are similar. The only difference is the local index update cost, which is dependent on the type of the local index, e.g., the local hash table[5], $B^+$-tree[6] and R-tree[7] implemented for the distributed indexes. Therefore, we shall present here the update performance of the distributed hash index. We get similar observations on the update performance of the distributed $B^+$-tree and R-tree index.

In this experiment, we test the update performance of the distributed hash index built on the $item\_title$ attribute of the $item$ table in the 10 GB TPC-W benchmark data set. We perform scalability test with different system sizes on an in-house cluster and measure the performance of update operations in term of the update latency and update throughput. For each system size, we also vary the input update load. Specifically, we launch from 5 to 50 client threads at each node and each client thread continuously submits update requests to the distributed index. Each completed operation will be followed up by the another request.

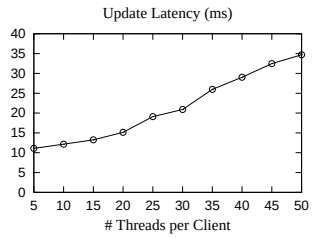


**Figure 14: Update latency**

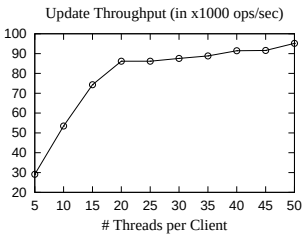


**Figure 15: Update throughput**

Figure 14 and Figure 15 plot the update performance of the distributed hash index in a system of 64 nodes. As we have discussed, the update operation suffers from three considerable factors of latency cost. That is the reason why, in this test, the system is saturated when the input update rate is high, i.e., there is a large number of client threads. In particular, the update throughput becomes stable when we increase the input rate to the level of 20 client threads per node. When the input update load gets larger, there will be

[5]http://sna-projects.com/krati/

[6]www.oracle.com/technetwork/database/berkeleydb/overview/index.html

[7]http://www.rtreeportal.org/code.html

more number of concurrent update operations in the system. These operations compete each others for the resources such as disk I/Os and concurrency lock holding. Therefore, given a fixed amount of resources, i.e., 64 nodes in the cluster, the update performance is constrained by a threshold of input load (about 20 client threads per node in this experiment).

## D.4 Handling Skewed Multi-Dimensional Data

In this test, we study the efficiency of our proposed indexing system in the presence of skews both in data and query distribution. We apply the Brinkhoff data generator[8] to generate a dataset of 10 million skewed 2-d moving objects based on the city map, which represents the real-time traffic.

**System storage load distribution.** The storage load of an index node is measured by the amount of index data maintained by that index node. Since the data has skewed distribution, the use of uniform data mapping function will assign some index nodes with much more data than other nodes, leading to an unbalanced system storage load distribution. This is the case where our proposed sampling-based data mapping takes its effect. As Figure 16 demonstrates, when the sampling-based data mapping function is used, the system storage load is well distributed, i.e., a certain percentage of the number of index nodes (in the system of 64 nodes) services the corresponding percentage of the index data.

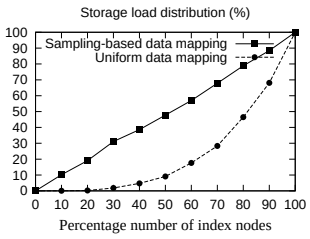


**Figure 16: Distribution of load under skewed data distribution**

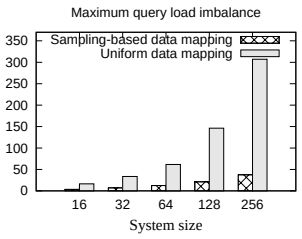


**Figure 17: Load imbalance under skewed query distribution**

**System execution load imbalance.** The maximum query load imbalance is defined as the ratio between the query execution load of the heaviest-loaded node divided by the query execution load of the lightest loaded node in the system. Figure 17 shows the maximum query load imbalance of different system sizes under the skewed query distribution (Zipf factor = 1).

In our experimental setup, a larger number of nodes in the system results in a higher query workload input. The situation becomes worse when the query distribution is skewed: an increasing number of queries will be directed to some hot data. If the uniform data mapping function is used while the data stored in the system has skewed distribution, the system will end up with high imbalance in query execution load.

On the contrary, the sampling-based data mapping function proposed in our indexing framework can roughly estimate the data distribution and distribute the data over the index nodes. Thus, the incoming queries on the skewed data is also distributed over the index nodes, leading to less query load imbalance in the system.

## D.5 Performance of Equi-joins and Range Joins

The joined columns of different tables typically share the same data semantics and the index data of these columns are normally partitioned and distributed over the index nodes in the same manner. Therefore, to speed up the join query processing these index nodes scan their local indexes and join the records in parallel. As discussed in Section 3.3.2, the join order between multiple tables is determined by the upper layer query optimizer, e.g. the storage manager of a cloud storage system [3] that uses the indexing service to extend its functionalities.

**Equi-join.** In this test, we compare the equi-join performance of two approaches: index join using the distributed indexes and MapReduce-based sort merge join. The MapReduce join can be implemented as follows. In the map phase, the mappers scan through the two joining tables and with each tuple $t$ the mappers generate an intermediate records $(k_t; t')$ where $k_t$ is the joining key and $t'$ is tuple $t$ tagged with the name of the table that it belongs to. The mappers then partition these intermediate records based on the value of the joining key (a hash function is normally used to guarantee load balance) and shuffle them to the reducers. In the reduce phase, the reducers just need to form groups of the same key value and yield the final join results.

To compare the performance of the two approaches, we measure the latency of the following equi-join query based on the TPC-H[9] schema:

```
SELECT  O.orderkey, orderdate, L.partkey, quantity, shipdate
FROM    Orders O, Lineitem L
WHERE   O.orderkey = L.orderkey
        and O.orderpriority='1-URGENT'
```

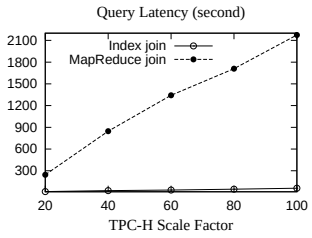


**Figure 18: Index join vs. MapReduce join**

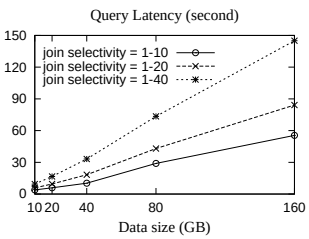


**Figure 19: Range join performance**

Figure 18 plots the experimental results when we vary the scales of TPC-H benchmark in a cluster of 64 nodes. As expected, the index join approach outperforms the MapReduce join approach because the indexes are able to identify the joinable records quickly and join them in parallel on each index node. On the contrary, the main shortcoming of the MapReduce join approach is the need to transfer the whole base tables from the mappers to the reducers, which incurs a great deal of overhead including network bandwidth and storage capacity.

**Range join.** Since range join queries are not popularly tested in benchmarks such as TPC-H, in this test we synthetically generate data for two tables $T_1$ and $T_2$ with the same schema $(rid, val, p)$ where $rid$ is the record id, $val$ takes its values from the domain of $10^9$ values and $p$ is a payload of 1KB. These two tables are stored in HDFS and we instantitate secondary indexes for both tables on the $val$ attribute using the distributed B$^+$-tree of the proposed indexing framework. We measure the latency of the following range join query on the $val$ attribute:

```
SELECT  T1.rid, T2.rid    FROM  T1, T2
WHERE   T1.val between (T2.val + α) and (T2.val + β)
```

We define the join selectivity of the test queries, i.e., the average number of the joining $T_2$ records per $T_1$ record, by setting the value of $\alpha$ and $\beta$. Figure 19 shows the performance of using indexes for processing range join queries with different selectivities and data sizes in a system of 64 nodes. Note that the MapReduce-based sort-merge join is inefficient for processing equi-join queries as shown previously. Extending it to support range join queries would only reduce the performance, so we do not show its result in this test.

[8] http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/

[9] http://www.tpc.org/tpch/