Handling Multi-Dimensional Complex Queries in Key-Value Data Stores

†Yu Tang, ‡Hailong Sun, †Qi Wang, †Xudong Liu
School of Computer Science and Engineering, Beihang University
Beijing, China 100191
†{tangyu,wangqi,liuxd}@act.buaa.edu.cn, ‡sunhl@buaa.edu.cn

Abstract
With the advent of the era of cloud computing and big data, in order to cope with vast amounts of data, a number of key-value databases have emerged. These systems provide the ability of large scale data storage and effective data operations based on primary keys, but they do not efficiently support the range and k-Nearest Neighbour (kNN) queries on multi-dimensional datasets. In this paper, we introduce, SPIKE, a sliced Pyramid-based index system for key-value data stores. SPIKE bridges the gap between the data scale and querying functionality for highly available, scalable distributed key-value data stores. We first present SP-Index, the kernel indexing scheme. The SP-Index is designed as a two-level index mechanism consisting of a sliced pyramid space partition index and a distributed B-Tree index. On the basis of SP-Index, we have designed and implemented SPIKE on Cassandra, which provides efficient multi-dimensional complex query processing. We have conducted a set of comprehensive experiments with three types of datasets including synthetic datasets, TPC-H benchmark datasets and a real-world dataset. The experiment results show that SPIKE can efficiently handle multi-dimensional complex queries on large-scale key-value datasets. Evaluation results in comparison with existing systems demonstrate that SPIKE outperforms the comparing work including the original Pyramid, MySQL Cluster and CCIndex by dozens of times in complex query processing.

Keywords: Database; Key-value store; NoSQL; Range query; kNN query; Multi-attribute query; Index; Multi-dimensional dataset
1. Introduction

In recent years, the data scale has been growing rapidly with the development of various data intensive applications including social networking, e-commerce, smart city applications and so on. In return, the data storage system is expected to meet a series of new requirements, such as high scalability, fault-tolerance and easy management. As traditional relational databases cannot satisfy these requirements, a plenty of distributed NoSQL databases are developed. NoSQL sacrifices fully ACID-compliant transactions and constraints of normal form in traditional relational databases so as to achieve high scalability and highly concurrent read/write performance. Key-value databases, e.g. HyperDex [1], are one of the representative category of NoSQL databases. These database systems can efficiently support simple queries based on primary keys, but most of them do not well support complex queries like range queries, kNN queries and other complicated queries based on non-primary keys. For executing those complex queries, the whole dataset has to be scanned, which incurs excessive costs. This problem limits the application scenarios mainly to offline data analysis applications with data-parallel systems like MapReduce [2, 3] and Spark[4]. And NoSQL systems are not suitable for handling complex query requirements of online applications.

The key to support complex queries is to design proper index mechanisms. There have been many research efforts concentrating on the indexing technology of key-value databases, such as MD-HBase [5], CCIndex [6, 7] and BIDS [8]. For instance, MD-HBase [5] uses KD-Tree [9] and Quad-Tree [10] to implement a multi-dimensional index on HBase. However, existing indexing mechanisms are mainly designed for specific key-value databases, thus are highly coupled with the underlying database storage engine. This limits the application scope of existing index technologies.

In this paper, we present SPIKE, a sliced Pyramid-based index system for key-value stores, to support efficient multi-dimensional complex query processing for various key-value databases. The core of SPIKE is a two-level distributed multi-dimensional index scheme called SP-Index. The first level aims at dealing with the multi-dimension challenge, which is achieved through a sliced pyramid space partitioning technique. The original pyramid technique [11] uses only one dimension to calculate the index value, which can cause frequent collisions of the same pyramid value for queries on large-scale datasets with less distinct values. Here by collisions, we mean that
multiple data points are mapped to a single pyramid value. The collision is closely related to the value distribution of indexed dimensions and the size of a dataset. In order to understand the impact of pyramid value collision, we conducted an experiment to evaluate how the pyramid technique performs with varying degree of data overlapping and size of the dataset, where the degree of overlap means the percentage of the points with the same pyramid value in total points in each dimension and it is negatively correlated to the number of distinct values. In Figure 1, our experimental results show that the number of candidate points to be scanned in a query can be up to 400,000 for the dataset containing 20 million points with 50 distinct values (the degree of overlap is 2%) in each dimension. The linear scan for so many points on the disk severely degrades the query performance. Therefore, in order to decrease the number of points to be scanned, we divide the pyramid space much finer based on the information of all dimensions in considering the features of queries with low selectivity and less distinct values in many online applications. With the sliced pyramid value in the first level, we build a B-Tree index as the second level index to improve search efficiency. In addition, we present the algorithms of range queries and kNN queries on SP-Index. Theoretical analysis shows that our division strategy leads to exponential improvement about the dimensionality on query performance compared with the original pyramid method.

In the design of SPIKE, the SP-Index is persisted in a B-linked-tree [12] to achieve the efficient point and range queries with index values. In order to guarantee the high availability and scalability of the index system, we implement replication and partition mechanisms for index tables based
on the strategy of the underlying key-value database. In addition, SPIKE splits the non-index columns into two parts and store them in index table and data table respectively in consideration of query characteristics. Then we apply SPIKE to Cassandra to implement a prototype database that can provide support for multi-dimensional complex queries of key-value data. Experiments on various types of datasets show SPIKE can efficiently process complex multi-dimensional range queries and k nearest neighbour queries. And it greatly outperforms the implementation of SPIKE with original pyramid technique [11] and other multi-dimensional index methods on key-value databases.

This paper makes the following contributions:

- We propose SP-Index, a two-level indexing scheme, to support efficient complex queries of large scale multi-dimensional datasets on the basis of the original pyramid index technique by reducing the number of data items to be scanned.

- We have designed and implemented SPIKE with SP-Index and demonstrate its replication and querying mechanisms. Besides, we have applied SPIKE to Cassandra to implement a practical key-value store that support efficient multi-dimensional complex queries.

- We have performed an extensive set of experiments based on synthetically generated datasets, TPC-H benchmark datasets and a real world dataset. The results show that SPIKE can achieve dozens of times better performance in comparison with existing indexing methods.

- We have discussed the application scenarios that SPIKE is suited for, and the limitations of the proposed approach.

The rest of the paper is organized as follows. In Section 2, we describe the design of SP-Index and the query process with SP-Index. In Section 3, we analyze the performance of SP-Index compared to original pyramid technique. The overview of Cassandra and the technical details of SPIKE’s implementation are presented in Section 4. In Section 5, we show experimental evaluation results of SPIKE and comparing systems. In Section 6, we discuss the application scenarios of SPIKE. Section 7 presents a literature review of some related works. And we conclude the paper in Section 8.
2. The SP-Index

In order to cope with the complex query requirements of huge volumes of data in various types, we propose SP-Index. The first level index of SP-Index is for further splitting the data space on the basis of the original pyramid index. So we first review the original pyramid technique [11] before introducing SP-Index.

The pyramid technique divides the \(d\)-dimensional data space into \(2^d\) pyramids that share the center point of the space as their top (Figure 2(a)), and the \((d-1)\)-dimensional surfaces of the space are their bases. Each pyramid has a pyramid id \(p\) according to some rule. The distance between a point \(X\) and the center in dimension \(p\) (or \(p-d\) if \(p \geq d\)) is defined as the height of the point, \(h_X\). Then, the pyramid value of \(X\) is \(Pv_X = (p + h_X)\). Finally, the \(d\)-dimensional data point \(X\) and \(Pv_X\) will be maintained by a B\(^+\)-tree in the according data page.

Two problems of the pyramid technique make it fail to preserve its excellent performance when facing a huge amount of multi-dimensional data. First, the number of points corresponding to each \(Pv\) increases with the increasing amounts of data. Second, the less distinct values in each dimension reduce the number of \(Pv\). These problems make the number of data points corresponding to one \(Pv\) continuously grow with the increasing scale of data size. As a result, the data points with respect to a \(Pv\) cannot be stored in one leaf node in B\(^+\)-tree. Large numbers of candidate points increase the disk I/O overhead because one query operation has to search data across multiple disk pages, which results in longer response time and degrades the performance of the index.

In light of the observations above, the main objective of SP-Index focuses on reducing the number of data points with the same pyramid value \((Pv)\) in the case of large-scale datasets or datasets with less distinct values, which can improve query processing efficiency through reducing the needed disk scanning.

2.1. Sliced Pyramid Based Space Partition

The basic idea of SP-Index is to expand the pyramid value scope of each pyramid and to provide finer division the pyramid space so as to make each \(Pv\) correspond to as less data points as possible. First, the \(d\)-dimensional data space is divided into \(2d\) pyramids as the original pyramid method does. Second, we determine the columns with less distinct values. For the pyramids
corresponding with these columns, we need to further split them into much finer slices so as to ensure all the index items with the same \( P_v \) can be stored in one disk page. Since we set the interval size of each slice to 1, the \( P_v \) of some points in a pyramid may be greater than the upper bound of this pyramid which leads to collision with the points in next pyramid. So we need to extend the interval of each pyramid to avoid the collision. And in each slice, a data point is identified by the height of the point, which is similar to original pyramid method. Above all, in SP-Index, a data point is addressed through a triple \( < \text{pyramidid}, \text{sliceid}, \text{height} > \) (Figure 2(b)). The detailed method is described as the following steps:

\[ \text{S1} \] We denote the dimension of data as \( d \), a row of data is represented by \( X = (x_0, x_1, ..., x_{d-1}) \) in \( d \)-dimension space. We normalize \( X \) based on the range of the value of each dimension and convert it to a point in a \( d \)-dimensional unit hypercube, i.e. \( X' = (x'_0, x'_1, ..., x'_{d-1}) \).

\[ \text{S2} \] Compute \( p \), the id of the pyramid which \( X' \) belongs to with Equation (1).

\[
p = \begin{cases} 
j_{\text{max}} & x'_{j_{\text{max}}} < 0.5 \\
j_{\text{max}} + d & x'_{j_{\text{max}}} \geq 0.5 
\end{cases}
\]  

where \( j_{\text{max}} \) is the dimension which has the biggest value of \( |x'_j - 0.5| \).

\[ \text{S3} \] In this step, we determine the pyramids that need to be divided into slices. Considering the general case, the number of points in each node and in each dimension is similar to other nodes and dimensions. We set \( N \) to be the estimated number of rows in the dataset, \( n \) is the number of nodes in the cluster and \( V \) is the number of distinct values. So the average number of points corresponding to a pyramid value is \( \frac{N}{ndV} \). In order to optimize the
query performance, the number is expected to be less than the max number of index items in a disk page. Let $K$ be the disk page size and $S$ be the size of an index item. In most cases, $K$ is 4KB and $S$ is about 64B. Then we can have Equation (2).

\[
\frac{N}{ndV} \geq \frac{K}{S} \\
V \leq \frac{NS}{ndK}
\] (2)

where $V$ is the maximum number of distinct values corresponding to the columns which need to be further divided. For instance, given a 200 million 6-dimensional dataset and a cluster with 10 nodes, $V$ is about 50,000, which means the pyramids corresponding to the columns with less than 50,000 distinct values are required for further division.

S4 We divide each pyramid into multiple slices in the process of calculating the height of point $X$. As mentioned above, we need to extend the interval of each pyramid to avoid the collision of $Pv$ with the points in next pyramids. We set the size of the interval of each sliced pyramid to $2^s$ and uniformly divide the interval $[0, 2^s]$ into $2^s$ slices, the interval of each slice is 1. And the sliced pyramid id $sp$ is denoted by the lower bound of its interval, which is defined as $sp = p \cdot 2^s$.

So the interval of sliced pyramid $p$ is $[p \cdot 2^s, (p + 1) \cdot 2^s]$, where $s$ indicates the times that we divide the pyramid and it is defined as $s = \text{Min}(d - 1, T)$.

However, a larger $s$ would produce too many slices because the volume of slices grows exponentially with dimensions. As large numbers of slices will increase the times of scans and reduce the performance in range queries, we set an appropriate threshold value $T$ based on experiment results to limit the size of $s$.

S5 For the pyramids corresponding to the dimensions that need not to be divided into slices, we calculate the height as $h = |0.5 - x'_p|_{d}$ directly. And the pyramid value of the points in this dimension is defined as $Pv_X = sp + h$, which is similar with the original pyramid technique.

In addition, for the pyramids corresponding to the dimensions with less distinct values, we need to determine the slice id $q_{X'}$ of point $X'$. First, we need to determine the dimensions used in slice division. And we insert dimension ids into an empty collection $S$ with the following two rules.

R1 If $d \leq T$, which means all dimensions will be used in slice division, we add all the $d$ dimensions to the collection $S$ by ascending order of dimension number;
If \( d > T \), which means some dimensions will not be used in the division, we get the first \( s \) dimensions with the most distinct values and insert them to collection \( S \) by ascending order of dimension IDs.

We check the normalized value of all dimensions in \( S \) except dimension \( p\%d \). The corresponding bit of \( q \) is set to 1 if \( x' \leq 0.5 \); otherwise the bit is set to 0. For example, there is a 4-dimensional point \( X' = (0.9, 0.2, 0.6, 0.95) \), we have \( p = 7 \) and \( s = 3 \) according to S2 and S4. Then we put \( X' \) into a 3-dimensional space as shown in Figure 2(c). The 4th dimension is excluded because it has been used to determine the pyramid id. Since \( x'_0 > 0.5 \), we set the 1st bit of \( q \) to 1. Likewise, we set the 2nd and 3rd bit of \( q \) to 0 and 1 respectively because \( x'_1 < 0.5 \) and \( x'_2 > 0.5 \). Then we can know that \( X' \) is located in Slice 5 because \( q_{X'} = 5 \). In the end, we not only divide a 4-dimensional space into \( 2d = 8 \) pyramids but also divide each pyramid into \( 2^s = 8 \) slices. Inside a slice, the distance from a data point to the inner edge of each slice is defined as height \( h = |0.5 - x'_{p\%d}| \).

Finally, the pyramid value with SP-Index is defined as \( P_{vX} = sp + q + h \).

Given a new data point \( X \), we first determine the pyramid value \( P_{vX} \) of the point through the steps above, then insert the point into \( B^+ \)-tree or other data structures which efficiently support point and range queries to construct the SP-Index using \( P_{vX} \) as the key.

2.2. The Index of Distributed Storage Layer

In our design, the persistent storage layer of SP-Index is implemented with MapDB [13], a B-tree-like storage engine. When inserting a row of data, we first compute the \( P_{v} \) value with the method described in Section 2.1. Then we insert it to MapDB using “\( P_{v} : RowKey \)” as the key. In order to guarantee the high availability and scalability of SP-Index, we distribute the index tables to the data nodes based on the replication and partition strategy of underlying key-value database. Considering the inefficient insert operations of B-tree-like data structure, we provide memtable and commitlog for SP-Index to improve the insert performance. More details about the index layer will be described in Section 4.

2.3. Query Processing

2.3.1. Point Queries

The processing of point queries on column values is simple to implement. We first compute the \( P_{vQ} \) of query point \( Q = (q_0, q_1, ..., q_{d-1}) \) using the meth-
ods described in Section 2.1, and search the B\textsuperscript{+}-tree with $PvQ$. With that, we will obtain a set of candidate points with the specified $PvQ$. Then, we can scan the set and determine whether the point is satisfied with conditions of each dimension of $Q$ to determine the final results.

2.3.2. Range Queries

Given a range query $Q = ((x_{0\text{min}}, x_{0\text{max}}), ..., (x_{d-1\text{min}}, x_{d-1\text{max}}))$, it is processed as below.

**S1** At first, we normalize $Q$ to the unit d-dimensional space. Then we compute which pyramids intersect with the specified query range. A pyramid $p$ is intersected with the query $Q$ if and only if it satisfies the following conditions. And we set the sliced pyramid id to $sp = p \cdot 2^s$.

$$q_{i\text{min}} \leq q_{j\text{max}}, q_{i\text{min}} \leq 1 - q_{j\text{min}} \quad j = 0, 1, ..., i - 1, i, i + 1, ..., d - 1,$$

$$p < d, i = p$$

$$1 - q_{i\text{max}} \leq q_{j\text{max}}, 1 - q_{i\text{max}} \leq 1 - q_{j\text{min}} \quad j = 0, 1, ..., p - 1, i, i + 1, ..., d - 1,$$

$$d \leq p < 2d, i = p \%d$$

**S2** In this step, we determine the slices that need to be queried in each sliced pyramid. Let $x$ be the id of a slice that needs to be queried. We determine the upper bound and lower bound of the query ranges in each dimension respectively. Then we set the corresponding bit of $x$ to 0 if the upper bound is less than 0.5; and set the bit to 1 if the lower bound is greater than 0.5. If the point 0.5 is contained in the range, we need to search two slices and the corresponding bit of the id of the first slice is set to 0 and the other slice is set to 1. For instance, suppose the query range is $((0.3, 0.4), (0.4, 0.7), (0.5, 0.6), (0.0, 0.1))$. The pyramid id $p = 3$, then we can get two slice number $x_1 = 1(001)$ and $x_2 = 3(011)$ because the query range of the 2nd dimension contains the point 0.5. We will obtain the slices to be queried after all dimensions except the dimension $p \% d$ are checked.

**S3** Lastly, we need to determine the query ranges $r = (r_{low}, r_{high})$ within
each slice to be queried according to Equation (3).

\[ f = 0.5 - q_xd_{\text{min}} \times 0.5 - q_xd_{\text{max}} \]

\[
\begin{align*}
    r_{\text{low}} &= \begin{cases} 
        sp + q + \min(|0.5 - x_p d_{\text{min}}|, |0.5 - x_p d_{\text{max}}|) & f \geq 0 \\
        sp + q & f < 0 
    \end{cases} \\
    r_{\text{high}} &= \begin{cases} 
        sp + q + \max(|0.5 - x_p d_{\text{min}}|, |0.5 - x_p d_{\text{max}}|) & f \geq 0 \\
        sp + q + |0.5 - x_p d_{\text{min}}| & f < 0, p < d \\
        sp + q + |0.5 - x_p d_{\text{max}}| & f < 0, p \geq d 
    \end{cases}
\end{align*}
\]  

With the steps above, we will obtain a set of 1-dimensional ranges for each slice \( q \) of each intersected pyramid \( p \). The points outside the ranges are not needed to be considered as they are out of the query rectangular while every point within the ranges is the candidate points to be further processed. Next we can scan the set obtained with several range queries on \( B^+ \)-tree and determine which points satisfy the query range of \( Q \).

2.3.3. kNN Queries

To obtain the kNN points of a certain data point \( X = (x_0, x_1, \ldots, x_{d-1}) \), we adopt a kNN search algorithm with a modified decreasing radius strategy [14]. We use a priority queue \( A \) to store the \( k \) candidate nearest neighbours sorted by the distance from \( X \) in decreasing order. Let \( D(v, X) \) be the Euclidean distance between a candidate point \( v \) and \( X \), and \( D_{\text{max}} \) be the maximum distance between the points in \( A \) and point \( X \). Besides, let \( C(X, r) \) be a circle centered at \( X \) with a radius \( r \). We will get the result in queue \( A \) with the following steps.

**S1** \( A \) is initialized to be empty, and we calculate the \( P_{vX} = p_i + q_j + h \) using the method in Section 2.1. We search the \( B^+ \)-tree to locate the leaf node that has the same key as \( P_{vX} \), or the largest key that is smaller than \( P_{vX} \). After locating the leaf node, we scan the data points in the node leftward and rightward simultaneously. Meanwhile, we calculate the \( D(v, X) \) to determine if the point \( v \) is one of the \( k \) nearest neighbours, then update \( A \) accordingly. The search process stops when the key of the leaf node is less than \( \lfloor P_{vX} \rfloor \) or greater than \( \lfloor P_{vX} \rfloor + 0.5 \), or there are already \( k \) data points in \( A \) and the difference between the current key value in the node and the pyramid value of \( X \) is greater than \( D_{\text{max}} \).

**S2** If the size of queue \( A \) is less than \( k \) after we finish searching the interval \( [\lfloor P_{vX} \rfloor, \lfloor P_{vX} \rfloor + 0.5] \) then we need to repeat S1 in the slice that is
the nearest from the point $X$. We find dimension $l$ which has the smallest $|0.5 - x_l|$ from the collection $S$ described in Section 2.1. Then we invert the $l$-th bit of the slice number $q_j$ to get $q'_j$ and repeat S1 with $P_{v_{X'}} = p_i + q'_j + h$.

**S3** We get a big enough query range (radius) through the first 2 steps and the query range will gradually decrease after the range queries in each pyramid. We generate a query rectangular $W$ enclosing $C(X, r)$ to perform a range search, which guarantees the correctness of the query results. We assume there are $k$ data points in $A$ after the first two steps. We examine the rest of the pyramids one by one. If the pyramid intersects $W$, we perform a range search to check if the points in this pyramid are among the $k$ nearest neighbours by comparing with $D_{max}$. The side length of $W$ and $D_{max}$ is updated after each pyramid is examined. If the pyramid does not intersect $W$, we can prune the search in this pyramid. We can get the finally results when all the pyramids are checked.

3. **Performance Analysis**

In this section, we will provide theoretical analysis of the performance of SP-Index in comparison with the original pyramid technique in terms of point and range queries. To simplify the problem, here we mainly consider the general uniform datasets. As mentioned in Section 2, a query process can be divided into two main stages:

1. Obtain a set of candidate leaf nodes from the $B^+$-tree corresponding to the $P_v$ of the query;
2. Scan the candidate leaf nodes to get the final results that match the query constraints.

It is already known that the time complexity of a primary key search in $B^+$-tree is $O(\log n)$, where $n$ denotes the number of $P_v$. And the computation complexity of $n$ is $O(N^{\frac{1}{d}})$ in the case of uniform distribution, where $N$ is the number of data points. As the candidate points that share the same $P_v$ are randomly stored on disks, so we need to sequentially scan all candidates to filter out the results, where the time complexity is $O(N)$. Therefore, the time complexity of point queries with the original pyramid technique is as follows.

$$O(\log n) + O(N) = O\left(\frac{1}{d} \cdot \log N\right) + O(N)$$

Range queries can be understood as a certain set of point queries within each intersected pyramid. The time complexity of the query for each inter-
sected pyramid is $O(\log n) + O(N)$ and the number of pyramids is up to $2d$. Therefore, the time complexity of a range query with the original pyramid technique is

$$2d \cdot (O(\log n) + O(N)) = O(\log N) + O(d \cdot N) \quad (5)$$

Since $n$, the number of $Pv$, with SP-Index is $2^s$ times greater than the original pyramid index, the number of points corresponding to each $Pv$ will be $(\frac{1}{2})^s$ times less when the data scale $N$ is constant. Point queries only need to query a unique $Pv$ value. Hence the time complexity of S1 is $O(s \cdot \log n)$, the time complexity of S2 is $O((\frac{N}{2^s}))$. And the total time complexity is:

$$O(s \cdot \log n) + O(\frac{N}{2^s}) = O(\log N) + O(\frac{N}{2^s}) \quad (6)$$

It is more complex for range queries. Suppose a range query $Q = ((q_{0_{min}}, q_{0_{max}}), \ldots, (q_{d-1_{min}}, q_{d-1_{max}}))$, where $(q_{min}, q_{max})$ of each dimension can fall into three cases:

1. $q_{min} \leq 0.5, q_{max} > 0.5$;
2. $q_{min} \leq q_{max} \leq 0.5$;
3. $0.5 < q_{min} < q_{max}$.

In Section 2.1, we divide a pyramid into two parts in each dimension whether the data value of the dimension is greater than 0.5. So we only need to query one part for case (2) and (3). We may need to query for both the two parts for case (1), involving two times of query in the corresponding dimensions. We consider the query range is uniformly randomly specified in all the dimensions for the convenience of discussion. We assume the selectivity is less than 0.1%, because even such low selectivity will get a large result set when there are billions of rows, and we assume that each dimension has the same selectivity which means the selectivity of each dimension is $\sqrt[0.001]{1}$. When the number of dimensions is less than 10, the query range of each dimension is not more than 50%. With this condition, the probability of the happening of case (1) $p1$ is $\frac{1}{3}$. The probability of half the points to be scanned is $\frac{2}{3}$ for each dimension and the probability of double the query times is $\frac{1}{3}$ for each dimension. So the reduction factor $F$ of points to be scanned in the case of multi-dimension is defined as $F = f(i) = \frac{1}{2}$. 

12
In the case of multi-dimension, the variable $i$ follows the binomial distribution $B(s, \frac{2}{3})$, so the expectation of $F$ is computed as Equation (7).

$$E(F) = E(f(i)) = \sum_{i=0}^{s} \left( \frac{1}{2} \right)^i \cdot C_s^i \left( \frac{1}{3} \right)^{s-i} \left( \frac{2}{3} \right)^i = \left( \frac{2}{3} \right)^s$$  \hspace{1cm} (7)

The number of candidate points need to scan for each query is the number of candidate points in the original pyramid technique multiplied by $F$. In other words, the costs with SP-Index is exponentially reduced in comparison to the original pyramid. The time complexity of each query is defined as follows:

$$O(s \cdot \log n) + O\left( \left( \frac{2}{3} \right)^s \cdot N \right)$$  \hspace{1cm} (8)

The times of queries in each dimension is $T = t(j) = 2^j$. The variable $j$ follows the binomial distribution $B(s, \frac{1}{3})$, so the expectation of $T$ is computed as below.

$$E(T) = E(t(j)) = \sum_{j=0}^{s} 2^j \cdot C_s^j \left( \frac{2}{3} \right)^{s-j} \left( \frac{1}{3} \right)^j = \left( \frac{4}{3} \right)^s$$

Thus, the total time complexity is

$$O\left( \left( \frac{4}{3} \right)^s \cdot s \cdot \log n \right) + O\left( \left( \frac{8}{9} \right)^s \cdot N \right) = O\left( \left( \frac{4}{3} \right)^s \cdot \log N \right) + O\left( \left( \frac{8}{9} \right)^s \cdot N \right)$$  \hspace{1cm} (9)

Since our improvement reduces the time complexity of range query, and a kNN query can be decomposed into multiple range queries, thus the kNN query time will also decrease.

In summary, the query performance of SP-Index on these queries has exponential improvement with respect to dimension $d$ compared with the original pyramid technique. Although the time complexity of S1 has increased, but this process is logarithmic time complexity about $N$. Besides, searching non-leaf nodes in B$^+$-tree spends relatively small space so as to be loaded to memory in our implementation to avoid disk I/O operation. So the increasing of the time complexity in S1 has less negative effect on the final performance. However, processing S2 requires slow disk I/O operations to load leaf nodes from disk to memory, which is the main bottleneck of processing a query. The decrease of the number of candidates can reduce disk pages to read and leads to effective improvement of query performance, which is verified by the experimental results presented in Section 5.
4. Implementation of SPIKE

We have implemented SPIKE with SP-Index and have applied SPIKE to Cassandra, a popular open-source key-value database, to support the complex queries on large amount of multi-dimensional data. Note that SPIKE serves as the index layer and is totally independent on the underlying database layer. Actually, SPIKE can be applied to any other distributed database system.

4.1. Cassandra Overview

Cassandra is designed as a peer-to-peer system that generates the replicas of data and distributes the replicas among a group of nodes. Data is organized with tables and identified by a primary key. The primary key determines which node in the token ring the data is stored on. Inside a node, Cassandra use LSM-Tree [15] in its storage engine to ensure write performance.

Cassandra is a row-oriented key-value database and it allows users to access data using CQL language with similar syntax like SQL. The SQL-like user interface makes Cassandra not only support the relational and schema-like data model but also provide the corresponding interface for the purpose of executing the query based on non-primary keys. The v0.7 and later versions of Cassandra provide secondary indexes to support multi-dimensional range queries, but there is a limitation that the queries require at least one equal operator on a configured index column in the query expression. Besides, the implementation of the secondary index leads to poor performance during complex queries. Therefore, it is of significant implications to apply SPIKE to Cassandra to improve its query performance limited by its original inefficient secondary index mechanism.

4.2. Architecture of SPIKE

SPIKE consists of client and logical index cluster. The underlying data cluster is set up with Cassandra. The architecture of SPIKE is displayed in Figure 3, which shows SPIKE is a lightweight index system that can be deployed non-intrusively in existing database systems.

The client of SPIKE is implemented on the basis of Hector [16] and we implement a set of APIs for index maintenance and data queries. Complex multi-dimensional queries are handled by SPIKE while simple one-dimensional queries or queries with row keys are directly processed by the underlying Cassandra engine.
SPIKE works between the underlying key-value database and the client. It consists of three modules: SP-Index manager, Communication Manager and Cluster Manager. As shown in Figure 3, SP-Index Manager is established in the system to maintain the SP-Index, which is specifically responsible for building index table, computing index values and handling the read and write request. Communication Manager is responsible for interacting with clients, database layer and SPIKE of other nodes. Cluster Manager is responsible for maintaining the partition information of underlying database and states of the nodes in SPIKE in real time. For the update and delete requests, SPIKE first forwards them to the database layer for processing. Then it will update the index according to the processing results of the underlying database so as to maintain the consistency between the indexes and the data.

4.3. Data Model of Index Table

As shown in Table 1, a row of data is composed of three parts: row key, indexed columns and zero or several non-indexed columns. And Table 2 presents an index entry of SPIKE. The primary key of the index table is formed by concatenating the $Pv$ with the original rowkey of the row. Index
Table 1: A row of data

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Index Column</th>
<th>Index Content</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Col 1 ... Col d</td>
<td>Part of non-index columns</td>
<td>Other columns</td>
</tr>
<tr>
<td>rowkey1</td>
<td>231 ... 652</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: An entry of index table

<table>
<thead>
<tr>
<th>Key of Index Table</th>
<th>Pv:Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Column</td>
<td>[Col1, Col2,...,Col d]</td>
</tr>
<tr>
<td>Timestamp</td>
<td>Timestamp in Cassandra API</td>
</tr>
<tr>
<td>Index Content</td>
<td>non-index columns mainly queried by multi-dimensional conditions</td>
</tr>
</tbody>
</table>

table stores the value of each index column for filtering the candidates during the query. The timestamp is equal to the timestamp in Cassandra API and is used to determine the data version. Then we divide the non-index column into two parts. One part in terms of Index Content is the columns which are mainly queried by multi-dimensional conditions, the other part is the columns mainly queried by the rowkey. We store the content of Index Content in the index table instead of data table and we store columns of latter part only in data table in order to reduce the space costs of SPIKE. We can divide the columns according to the requirements of applications. For example, the requirement of some product tracking system based on RFID is querying the details of product by its RFID tag(rowkey in key-value databases), and querying the inventory information by multi-dimensional conditions. We regard the columns to be queried by SP-Index, for example, inventory name, arriving time and leaving time of each inventory as the Index Content and store them in index table. And other columns such as details of product are stored in data table for efficient key-value queries. Although we store the columns into different tables, we still can get all columns of a row through an additional low-cost key-value query in database layer or point query in SP-Index.

4.4. Partition and Replication of Index

In order to ensure high availability and achieve load balance we deploy the index layer and Cassandra together in the same cluster. Each index node only indexes the local data partition by consistent hashing of Cassandra. We use a separate thread to communicate with the database layer of SPIKE to obtain the node status and current information of partition in the cluster. Thus we
can adopt the same partition strategy and replication strategy as database layer in order to keep the consistency of the data distribution between index and database. The node receiving the client’s requests forwards the write request to related nodes based on the partition strategy and current states of the nodes. The consistency strategy of writes in Cassandra is eventual consistency and we adopt strong consistency for the writes of index. In other words the coordinator returns success only if all replica nodes succeed in writing.

On the other hand, because of the existence of the replicas, the results returned from all the nodes may contain repeated data. The unnecessary repeated data will incur extra costs of bandwidth and latency. Normally the replication factor is 3, which means three pieces of data need to be saved in each index node, one for master data (determined by consistent hashing) and two other replicas (determined by the replication strategy), which is shown as Figure 4. We create a separate index table for each data replica. When an update request is received, the node distinguishes which piece of data to update in the request and updates the corresponding index table based on the method described in Section 4.5. When the cluster of SPIKE works normally, the coordinator only sends Master Data Query Request to all nodes of SPIKE during queries. After receiving a Master Data Query Request, each node only need to query the index table of master data and return the corresponding results. It avoids searching the replica data and transmitting the repeated index entries in replica data so as to decrease the cost of query index table and network transmission. Another benefit of this design is that it can reduce the number of index entries to be scanned during data transition when a certain node joins the cluster or exits from the cluster. The affected nodes only need to scan and modify the replica tables related to the changing nodes when perform the data migration.

In order to ensure the availability of the system when some nodes are subject to failures, SPIKE will find two nodes containing the replica data of the failed node’s master data. SPIKE detects whether a node in the cluster is available by the monitor thread in cluster management module or the coordinator during the query process. Then the coordinator randomly selects a node from the two nodes which contain the replica data and sends a Replica Data Query Request to obtain the results of the crashing nodes. After receiving the request, the node will query the index table of replica data according to the replication information enclosed in the Replica Data Query Request and return the query results. This design ensures the high
availability when some nodes fail and avoid the redundant query overhead when there are no failure nodes in the cluster.

4.5. Insertion Operations

The persistent storage of index tables is implemented with MapDB[13] which is based on B-Linked-tree [12]. When a data is to be inserted, the $Pv$ value is first computed based on the value of each index column. Then other data elements of the index entry is obtained by the method described in Section 4.4. Then the index entry is inserted to MapDB using $Pv : RowKey$ as the key. If the data in update request is already existent in the current index table, SPIKE checks whether it expires by comparing the timestamp in index table to the timestamp in data table during the query.

The write performance of B-tree-like data structure degrades greatly when the data size is large, which will lead to serious decrease in write throughput [17]. To solve this problem, we employ CommitLog and Memtable that are used in Cassandra and Bigtable in the implementation of the index layer of SPIKE. That is, the write request is processed only after writing to an append-log and updating the Memtable in memory. At the same time, the update operations on B-Tree-like structure is executed asynchronously. Our experiment results presented in Section 5.1.6 demonstrates that the write performance of SPIKE is comparable with other work.

4.6. Update with Schemaless Data Model

It is widely known that most of the key-value databases adopt the schemaless data model and non-in-place update strategy in order to improve their performance and scalability. This means that an update request may not contain the whole index columns. However, the calculation of the index value of SP-Index asks for the values of all index columns. Therefore, in the implementation of SPIKE, we need an additional query operation on database layer when an update request with partial index columns is received. In this way, SP-Index can acquire all the needed information to calculate the index value. The details of the operation are described in Algorithm 1.

4.7. Query Processing

In this section, we describe how a client query is getting processed. The node that receives the request of complex multi-dimensional queries from the client is considered as the coordinator. Since each index node only indexes the local data, so the coordinator needs to forward the query request to all
Algorithm 1 Update with Scheme-less Data Model

function \textsc{PartialUpdate}(\textit{request})
\begin{itemize}
    \item \textbf{if} the \textit{request} does not contain all index columns then
        \begin{itemize}
            \item query the database layer based on the rowkey in the \textit{request};
            \item \textbf{if} the database layer returns other index columns then
                \begin{itemize}
                    \item merge the index columns returned by the database with the index columns in the \textit{request};
                \end{itemize}
            \end{itemize}
        \end{itemize}
    \end{itemize}
\begin{itemize}
    \item set the normalization value of unspecified columns to -1;
    \item calculate the $Pv$ and insert into the index table;
    \item delete the old index item asynchronously with the $Pv$ returned by the database;
\end{itemize}
end function

nodes within the cluster of SPIKE, and then aggregate the returned results to generate the final results.

After receiving a query request, a node will search its local index table to retrieve the eligible results meeting the query constraints, which is achieved with the method described in Section 2.3. Then the index entries in the result set with the same rowkey need to be compared by their timestamps. Only the entry with the largest timestamp is added to the result set. If a query is about non-indexed columns that are excluded from \textit{Index Content}, SPIKE needs to query the corresponding column values from database layer according to the rowkey in index table. As a result, the queries based on rowkey will produce a large number of random reads, which will be processed by Cassandra instead of by SP-Index.

4.8. Using SPIKE with other Key-Value Databases

An important feature of SPIKE is that it can be applied to various types of key-value databases. This can be attributed to two facts. First, the index layer of SPIKE is implemented with independent storage engine based on B-linked-tree, freeing it from restricting by the features of the storage engine of key-value databases. Second, as a distributed index system, SPIKE has an independent communication module based on the non-invasive design principle, which avoids being affected by the architecture of the key-value database layer. In our implementation, we need to specify the partition
strategy of the underlying key-value database, such as the consistent hash and range partitioning. We need to configure SPIKE with the corresponding partition parameters so as to ensure the consistency of data distribution between the index and the database. We have described the implementation method with Cassandra, which is based on the consistent hash partition. Using SPIKE with other databases is almost similar except for some minor difference due to partition strategies.

For instance, the replication of HBase is implemented by the underlying HDFS and HBase adopts the range partition strategy. At first, we obtain the distribution feature of the primary key according to the dataset characteristics. Second, we specify and fix the key range of each RegionServer node when we create the data table in HBase by the API of HBase client. Third, we set the partition strategy of SPIKE to range partition and specify the corresponding fixed key ranges for each index node in SPIKE. The rest of implementation is similar to that with Cassandra.

5. Experimental Evaluation

In this section, we present a set of evaluation results to evaluate the performance of SPIKE. Specifically, we evaluate SPIKE through the comparison with MySQL Cluster 7.3.2 [18], CCIndex for Cassandra [7] and SPIKE implemented with original pyramid technique instead of SP-Index in terms of multi-dimensional point queries, range queries and kNN queries. In our experiments, we used the following two metrics for performance evaluation:
throughput, which is the number of queries executed successfully per second and response time, which is defined as the average time for processing a query.

The experiments were conducted on a 32-node cluster where each node had an 8-core 2.0GHz CPU and 16GB memory and all nodes were connected with 1Gbps local network. Most of the experiments except the evaluation of scalability were conducted on 10 nodes. We implemented SPIKE on the basis of Cassandra 1.2.8 and used Hector 1.0.5 as the client to send requests to the query engine. Moreover, as the default replica factor for Cassandra is 3 and MySQL Cluster only supports 2 replicas at most, we set the replica factors of MySQL Cluster and the other three systems to 2 and 3 respectively.

Three types of datasets were used in our experiments including: (1) synthetically generated uniform datasets; (2) datasets generated with TPC-H benchmark; (3) a real-world Check-In dataset contributed by Cheng [19].

In the synthetic datasets, we varied the number of data dimensions from 3 to 15. For each dimension, the data values were uniformly generated and there were 1 million distinct values. For each row of data, there were one 32-byte rowkey, multiple 4-byte index fields and one 16-byte data field. Consequently, the total size of each row of data varied from 60 bytes to 108 bytes.

As TPC-H is an acknowledged database benchmark and it can simulate real business application scenarios, we chose it to test the performance of SPIKE in a relatively real dataset. With TPC-H benchmark, we generated multiple datasets with different scale factors, where a scale factor $f$ indicates the generated dataset has $f$ GB data. In our experiments, we only used the LINEITEM table because it is the fact table among all the TPC-H data tables.

The check-in dataset is actually a typical non-uniform real-world dataset, as the latitudes and longitudes in the dataset indicate that users are clustered in active areas. In order to test the performance of index fairly and to avoid the influence of value accessing, all systems in the experiments did not return the non-indexed columns.

Finally, in our experiments, the default selectivity for range queries was 0.05%, and the default k value in kNN queries was 50.

5.1. Experiments with Synthetic Data
5.1.1. Effect of the Threshold Parameter $T$

In the first experiment, we aims at evaluating how the threshold $T$ presented in S4 of Section 2.1 affect the performance of SPIKE. Then we can
choose an optimal value of $T$ for the rest of evaluation. Intuitively, the more slices there are, the less data a slice will contain. With a larger number of slices, we can prune a larger portion of the data space that does not intersect the query, thus reduce the candidate points in filtering. However, the times of the accessing in each query and the overhead of searching for each accessed slice will increase because the number of slices gets larger with the increasing of data dimensions. We need to determine the optimal value of $T$ through experiments. Specifically, we used a 15-dimensional synthetically generated dataset with 100 million data points to serve client queries in this experiment.

Figure 5 presents the effects for different $T$ with varying the selectivity of range queries. We found that the performance difference with different $T$ was not big at low selectivity, but it got larger at high selectivity. And there was an optimal $T$ for the dataset with high dimension. The response time decreased as $T$ grew while it increased when the $T$ was larger than 12. Therefore, according to the experimental results, we chose 12 as the $T$ of SP-Index in the rest of experiments with synthetic datasets.

5.1.2. Effects of Data Size

In this experiment, we evaluated the performance of SPIKE and the comparing systems through varying the data size. We performed point queries, range queries with 50,000 rows of selectivity and kNN queries with $k=50$ in a 6-dimensional dataset. The kNN queries of original pyramid was implemented based on [14]. As CCIndex and MySQL Cluster do not support kNN queries directly, we skip the comparison with them in terms of kNN queries.
However, as kNN queries can be implemented on the basis of range queries, the comparison of range queries should justify the performance difference of SPIKE from other comparing systems.

Figure 6 shows the result of the experiments. All the systems except CCIndex exhibited good performance for point queries. The reason why the response time of point query for CCIndex was much larger than the others is that CCIndex needs to transmit candidate points to CCIndex clients, while other schemes do not need that. As the number of candidate points was small in point queries, the response time was relatively short for all the systems. However, as Equation (4) and (6) indicate, the response time of point query for SPIKE and pyramid technique will increase as data size increases, but the response time remains nearly the same for different data sizes in the experiments. The main reason is that Equation (4) and (6) mainly take the local search time in each server into consideration, but do not consider the cost of communication between servers, which dominates the response time of point query. However, for range queries and kNN queries, as the number of candidate points was much larger, it took more time to process the queries as the data size grew. As in Equation (5) and (9) demonstrate, when the dimensionality is fixed, the response time of range queries will increase as
data size increases and exhibit a approximately linearly incremental tendency, which matches the experimental results. Thanks to the finer division, SPIKE outperformed other systems with a speed factor of 4~8 in range queries and a speed factor of 10~20 in kNN queries and performed well in large volume of data.

5.1.3. Effects of Dimension

In this experiment, we were concerned with how the data dimension affected the performance of SPIKE. Intuitively, when dimensionality becomes larger, the data size for each row will also become larger, thus data transmission will cost more time. Besides, all systems need to examine more dimensions to judge whether a candidate point can satisfy the query conditions, which leads to longer response time.

In this experiment, the selectivity of this experiment was 0.01% and the dataset contained 100 million rows of data. Figure 7 shows the experimental results corresponding to different dimensions ranging from 3 to 15. Similar to Section 5.1.2, CCIndex was much slower than others. Though SPIKE also spent more time to process range and kNN queries for higher dimensional data, the increase rate was rather low and the increased time was bound within just several seconds. And SPIKE was 2.5~8 times faster than CCIndex and MySQL Cluster, and 4~10 times faster than the original pyramid technique in range queries. SPIKE overwhelmingly outperformed other schemes mainly because it simultaneously leverages multiple dimensions and designs finer division strategy to reduce the number of candidate data points to search.

5.1.4. Effects of Selectivity

In this experiment, we studied how the selectivity and the $K$ can influence the performance of SPIKE. We used a 6-dimensional dataset containing 100 million rows of data. Figure 8 plots the corresponding results. SPIKE still performed the best and it achieved the largest speedup compared with other systems in low selectivity and small $K$ value. The reason is that the smaller query ranges lead to less slices to search, which reduces both the query times and the size of rows to be scan. When selectivity became larger, the performance gain decreased, nevertheless, SPIKE still outperformed other techniques with 7~20 times faster and can be competent to various of selectivity and $K$ values.
5.1.5. Scalability

This experiment evaluated the scalability of SPIKE and other systems. Like previous experiments, we still used a 6-dimensional dataset containing 100 million rows of data. As range queries and kNN queries are more time-consuming and kNN queries can be realized with range queries, we chose to mainly present the results of range queries. The data selectivity for the queries was 0.05% and the results are shown in Figure 9. We can see that all systems except CCIndex scaled quite well as the number of nodes increased from 4 to 32. The reason that CCIndex did not benefit from more cluster nodes is that CCIndex needs to linearly scan all candidate items on the client side. SPIKE achieved 2.5~10 times faster query processing and about 5 times higher throughput compared with other systems. We also conducted experiments with point queries and kNN queries. As for the point queries, the response time was about a few milliseconds and did not differ much for those systems. And for the kNN queries, SPIKE was over 20 times faster than pyramid technique with different number of nodes.

Moreover, we inserted 1 billion 6-dimensional, around 100GB data items into the systems running on 32 nodes. For point queries, SPIKE achieved the similar performance as pyramid technique, and was about 30% faster than MySQL Cluster. Moreover, we found that SPIKE still outperformed other systems for range queries and kNN queries. SPIKE was about 66% faster than the second fast scheme MySQL Cluster for range queries, and about 36 times faster than pyramid technique for kNN queries.
5.1.6. Insert Throughput and Maintenance Costs

In this experiment, we provided some comparison analysis and experimental evaluation of the extra overhead in terms of insert and index maintenance incurred by SP-Index. In the evaluation of insert throughput, we used 10 concurrent clients to insert data rows to the system which had already stored 100 million rows of 6-dimensional data. The experiment results are shown in Figure 10. The insert throughput of SPIKE was lower than Indexed Cassandra because of the additional write operations and network communication cost brought by the index table. However, SPIKE performed far better than CCIndex which need to write 6 index tables and MySQL Cluster whose insert throughput was limited by the relational and transactional storage engine. On the other hand, for time costs, SPIKE only performed an in-memory write to a memtable for an update and flushes the memtable to disks asynchronously. In the experiment, the latency of building index for 400 thousand new rows in the SPIKE which had stored 60 million rows
was about 5s, bringing in about 8% overheads compared with the update in Cassandra.

Figure 10 also indicates that the average disk space usage for each node of SPIKE was about 4GB for a 6-dimension dataset of 100 million rows, which was less than CCIndex’s 6GB usage. Besides, it was similar to that of the secondary index in Cassandra and the b-tree index of MySQL cluster. According to the above analysis and experimental results, the costs were acceptable while great query performance improvement was achieved.

5.1.7. Update Performance

We evaluated the update performance of SPIKE and other schemes in this experiment. Before issuing update operations, we had inserted 100 million rows of 6-dimensional item into the system. In this experiment, each client only operated one data item in each update, and every client executed 100,000 update operations. We varied the number of clients and obtained the corresponding update throughput. Figure 11 demonstrates the experimental results. As shown in the figure, in terms of update throughput, SPIKE overwhelmingly outperformed CCIndex and MySQL Cluster, but it was slightly worse than the pyramid technique. CCIndex needs to update items in 6 index table except updating items in the data table, and therefore its update performance was relatively poor. The update performance of MySQL Cluster was limited by the relational and transactional storage engine. The throughput of SPIKE was slightly lower than that of the pyramid technique, as SPIKE needs to utilize more dimensional information to build indexes.
Table 3: The Schema of Lineitem Table

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_orderkey</td>
<td>Primary Key</td>
</tr>
<tr>
<td>L_linenumber</td>
<td></td>
</tr>
<tr>
<td>L_partkey</td>
<td>6,000,000 distinct values</td>
</tr>
<tr>
<td>L_suppkey</td>
<td>300,000 distinct values</td>
</tr>
<tr>
<td>L_quantity*</td>
<td>integer in [1,50]</td>
</tr>
<tr>
<td>L_discount*</td>
<td>decimal in [0.000,0.499]</td>
</tr>
<tr>
<td>L_tax</td>
<td>decimal in [0.00,0.08]</td>
</tr>
<tr>
<td>L_shipdate*</td>
<td>date in [1992,1998]</td>
</tr>
<tr>
<td>L_commitdate*</td>
<td>date in [1992,1998]</td>
</tr>
<tr>
<td>L_receiptdate*</td>
<td>date in [1992,1998]</td>
</tr>
<tr>
<td>L_extendedprice†</td>
<td>decimal &lt; 1 million</td>
</tr>
<tr>
<td>L_returnflag</td>
<td>[N,R,A]</td>
</tr>
<tr>
<td>L_linestatus</td>
<td>[O,F]</td>
</tr>
<tr>
<td>L_shipInstruct</td>
<td>4 distinct values</td>
</tr>
<tr>
<td>L_shipMode</td>
<td>7 distinct values</td>
</tr>
<tr>
<td>L_comments</td>
<td>arbitrary string</td>
</tr>
</tbody>
</table>

* Indexed Columns
† Index Content

5.2. Experiments with TPC-H Benchmark
5.2.1. Query Performance with Different Scale Factors

In this experiment, we used Q6 in TPC-H benchmark as QA to illustrate the practical effect on range queries of SPIKE. We also defined a simple query QB to evaluate the performance of point queries and kNN queries. For the queries we built a 5-dimensional index on the Lineitem table in TPC-H. We changed some columns of the table in order to adapt to Cassandra’s data model. The index columns of table Lineitem are shown in Table 3.

QA is defined as:

```sql
SELECT sum(extendedprice*discount) as revenue
FROM Lineitem
WHERE shipdate ≥ x AND shipdate < x+1 year AND discount ≥ y AND discount < y+0.02 AND quantity < z
```

We also define QB as below:

```sql
SELECT extended*price FROM Lineitem
WHERE shipdate = sd AND commitdate = sd+1 month AND
```
discount=d AND tax=t AND quantity=q

The scale factor is used to define the database size in TPC-H. For instance, when the scale factor is 1, the size of the total dataset is approximately 1GB and the Lineitem table takes about 740MB. The table size will linearly increase as the scale factor increases. The results are plotted in Figure 12. SPIKE was slightly lower than that in the experiments on the previous synthetic datasets since the number of distinct values of index columns in Lineitem table were much smaller, but SPIKE still performed the best. In addition, the QA only specified the query range for 3 dimensions, that is, the whole domain values of the other 2 dimensions of the queries needed to be examined. Even so, SPIKE still achieved remarkable performance and outperformed other systems because of the slice division on the dimensions whose query ranges were specific. The performance of CCIndex and pyramid technique rapidly deteriorated when there were less distinct values in each index column. As CCIndex and pyramid technique respectively failed to respond in 30 seconds during the range and kNN queries, we skip them in Figure 12(b)(c). The experiments show that SPIKE also performed well on index columns with less distinct values.

5.2.2. Scalability

In this experiment, we evaluated the scalability of SPIKE and compared it with pyramid and MySQL Cluster for the TPC-H dataset. We set the scale factor of TPC-H as 10. Figure 13 demonstrates the corresponding results for range queries with QA. All systems scaled well in the experiments. Especially SPIKE outperformed other systems: SPIKE costed only 34%~49% response time and achieved about 1.4~3.1 times higher throughput compared with MySQL Cluster. The response time of point queries was about several milliseconds and did not differ much for those systems. As to kNN queries,
SPIKE showed 3.2~4.4 times faster than pyramid.

5.3. Experiments with the Check-in Dataset
5.3.1. Query Performance with Different Size of Data

In this section, we present the experiment results with a real-world Check-in dataset. This dataset containing spatial data has been widely used in the research on location sharing services. Since the range queries and kNN queries on the spatial data are commonly required [20, 21, 22], it makes much sense to evaluate the performance of range queries and kNN queries for SPIKE with the check-in dataset. Our check-in dataset is contributed by Cheng [19], and contains check-in data mainly crawled from the location sharing status on Twitter [23] and foursquare [24]. The dataset is composed of 20 million rows of check-in records by over 220 thousand users. And its size is 3.2GB.

We evaluated the performance of range queries with 0.05% selectivity and kNN queries with k=50 in our experiments. Each row of the dataset was represented as \{UserId, CheckInId, Longitude, Latitude, Date, Time, Twitter-Content\}. The schema of the data table based on the check-in dataset is shown in Table 4. We also changed some columns of the table in order to adapt to Cassandra’s data model and constructed a 4-dimensional SP-Index.

The results are shown in Figure 14, which further demonstrates the advantages of SPIKE in comparison with other methods. In this dataset, the number of distinct values is much larger than TPC-H dataset and the data distribution is more skewed and clustered than the random generated dataset. The skewed and clustered distribution leads to more data points sharing the

Figure 13: Scalability for TPC-H dataset.
Table 4: The Table Schema of Check-in Data

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckInId</td>
<td>Primary Key</td>
</tr>
<tr>
<td>UserId</td>
<td>Integer</td>
</tr>
<tr>
<td>Latitude*</td>
<td>double in [-90, 90]</td>
</tr>
<tr>
<td>Longitude*</td>
<td>double in [-180, 180]</td>
</tr>
<tr>
<td>Date*</td>
<td>integer in [0, 800]</td>
</tr>
<tr>
<td>Time*</td>
<td>integer in [0, 90000]</td>
</tr>
<tr>
<td>Twitter Content</td>
<td>String</td>
</tr>
<tr>
<td>PlackId</td>
<td>Integer</td>
</tr>
</tbody>
</table>

* Indexed Columns

![Graph A](image1.png)  
(a) Range queries

![Graph B](image2.png)  
(b) kNN queries

Figure 14: Performance with Check-in Dataset

same $P_v$, thus should increase the scan costs. Even so, SPIKE still outperformed other systems because of its fine slice split strategy. The performance of original pyramid technique rapidly deteriorated when the distinct values were less. In all, this experiment shows that SPIKE not only performed well with uniform datasets, but also with skewed datasets.

5.3.2. Scalability

We also evaluated the scalability of SPIKE and compared it with pyramid and MySQL Cluster for the check-in dataset. Figure 15 demonstrates the corresponding results for range queries with 0.05% selectivity. The performance did not differ too much for SPIKE and MySQL Cluster when the number of servers was no more than 8. However, as the number of servers increased, the performance speedup of SPIKE was higher than that of MySQL Cluster, and the performance advantage of SPIKE was further enlarged. Similar
to previous scalability experiments, there was not much difference for point queries, while for kNN queries, SPIKE only costed 38%~56% of the response time of the pyramid scheme.

6. Discussion

According to the description about the design of SP-Index, we can conclude that the design of SP-Index is not bound with the underlying databases. Nonetheless, as NoSQL systems like Cassandra usually do not provide complex and efficient secondary indexes, building SP-Index on top of NoSQL can simplify the implementation and in return greatly benefits NoSQL systems. Through our theoretical analysis and experimental evaluation, we find that SPIKE, combined with Cassandra, achieves good query performance and meanwhile inherits the high scalability of Cassandra, with acceptable insert overheads. Theoretically, SP-Index also can be applied to relational database to provide efficient multi-dimensional query.

Actually, applications that require multi-attribute queries support on huge amounts of data can also benefit from SPIKE. For instance, a RFID-based anti-counterfeit cloud service, where our work was motivated. This service is required to manage huge amounts of commodity product data, where a key-value store is a good choice to achieve the scalability. In addition, it requires the function of complex queries including multi-attribute range and kNN queries to implement the function of warehouse management and commodity tracking.

As the experimental results demonstrate, SPIKE outperforms other techniques in range query and kNN query for three datasets, which includes a
uniform synthetic dataset, a relatively dense dataset generated by TPC-H benchmark, and a skewed and clustered check-in dataset. However, we cannot conclude that SP-Index is the best choice for all application scenarios.

Firstly, SP-Index is more suitable for datasets that are not heavily skewed or clustered. This limitation actually originates from Pyramid-Technique. Skewed or clustered datasets will lead to suboptimal space partition, resulting in performance degradation. As in the experiments, the performance gap between SPIKE and MySQL Cluster is smaller in the check-in dataset compared with that in the other two datasets. The phenomenon indicates that SPIKE performs better for data that is uniformly distributed, though SPIKE can be competent for data that is skewed and clustered, just like the check-in dataset. Moreover, an extended Pyramid scheme[11] is also proposed to deal with skewed datasets. The rationale behind is to convert the coordinates of the points so that the new center of the Pyramids approximates the center of the skewed dataset. And the extended Pyramid scheme can be incorporated to SPIKE for improving its performance on skewed datasets. On the other hand, the density of dataset can influence the performance of SPIKE, but it can also affect the performance of other techniques, as the experimental results evaluated on the dataset generated by TPC-H benchmark demonstrate.

Secondly, derived from Pyramid-Technique, SP-Index also inherits the characteristic that it is mainly suitable for range queries in hypercube shape. As in the experiments, in terms of point query, the performance of SP-Index is close to that of MySQL Cluster. Specifically, for one-dimensional point query or range query, SP-Index will degenerate to Pyramid-Technique.

Thirdly, the structure of indexed columns should be determined in advance. Up till now, SP-Index does not support the modification of index structure, such as adding a new indexed column, or modifying the value range of an existing indexed column. In practice, the value range of data fields can be easily determined in advance by application developers. For instance, most columns have deterministic ranges in the Lineitem table of TPC-H, and as in the check-in dataset, longitudes and latitudes on the earth also have a determinate range. Besides, SP-Index supports many common data types, including integer, float and string types. Additionally, string values can be represented by numeric id for using SP-Index, which is similar to what is done with relational databases.

Finally, the SPIKE prototype is now built on Apache Cassandra, a highly available NoSQL. Therefore, SPIKE does not support complete SQL query and transactional semantics. Nonetheless, we think the idea of SP-Index can
be also applied to traditional SQL databases.

7. Related Work

Nowadays, there are many databases adopting the key-value model, which are commonly known as key-value stores, such as BigTable [25], Dynamo [26], HBase [27] and Cassandra [28]. Although most of them support the relational model to some extent, their index methods can only support some simple queries based on the primary key while they cannot efficiently deal with complex queries based on non-primary key due to lack of efficient secondary indexes.

In practice, people choose to use data-parallel approaches to process complex queries through parallel scanning the data and building indexes. In production environment, the solution is using MapReduce technology to scan the whole database in parallel to build indexes stored in some special tables. This scheme can satisfy the requirements of the query, but the index can only be established in batch and cannot be updated in real time. For example, SpatialHadoop [29] is proposed to deal with range query and kNN query for spatial data, and outperforms many other Hadoop-based schemes. However, Hadoop-based schemes, including SpatialHadoop, employ MapReduce [2] framework to handle range query and kNN query, and therefore are more suitable for offline data analysis workloads as MapReduce is not designed to handle incremental updates. While SPIKE can be used to serve online Internet applications, where updates is not rare. Besides, SpatialHadoop is mainly designed to analyze spatial data and supports two-dimensional data types [30], while SPIKE can be applied to handling high-dimensional data.

In order to make key-value databases support multi-dimensional range queries, Chen et al. [31] propose an index framework for databases in the cloud. Besides, Complemental Cluster Indexing technology (CCIndex) [6, 7] is proposed based on HBase and Cassandra. CCIndex’s main idea includes: (1) Re-organize replicas as several Complemental Clustering Index Tables containing full row data, to convert the slow random reads to fast range scan. (2) Leverage the region-to-server mapping information to estimate the result size. (3) Introduce Complemental Check Table for record level replica to support incremental data recover. However, CCIndex needs to store a replica in the index tables of all dimensions which results in large amounts of disk space consumption in high-dimensional cases, and it does not pro-
vide the support for kNN queries. Besides, CCIndex is not suit for large dataset where the number of distinct values for every index column is small. [5] proposed MD-HBase after the analysis of the storage structure of HBase. Firstly, they divide the data space with KD-tree and Quad-tree, two typical multi-dimensional indexing structures. Then they employ the linear division of space to implement a scalable multi-dimensional indexing structure, and implement a prototype system based on HBase to ensure the support of multi-dimensional point query, range query and kNN query. However, the efficiency is poor in high dimensions due to the defects of KD-tree and Quad-tree. BIDS [8] implements multi-dimensional queries with highly compressed bitmap indexes. BIDS achieves very low space costs and can provide efficient range queries and join queries. It is more suitable for offline data analysis applications with rare updates rather than online Internet applications because of the defect of bitmap index in updating.

In recent years, many efficient indexes have been proposed to process multi-dimensional queries. Query processing using these indexes mainly has the following two steps: filtering and candidate verification. First, filtering uses the index to eliminate part of the false results and to produce a candidate answer set. Then, candidate verification tests whether each candidate indeed fits in the query range. Dimension reduction is a common method in the filtering step, such as interpolation method in [32] and pyramid technique. Pyramid technique [11] and P⁺-Tree [33] are proved to be efficient for multi-dimensional index structure. The pyramid technique distributes the data points into space pyramid adopting the strategy of the non-uniform space division and filter. It calculates the pyramid index value \( P_v \) according to the distance between the data points and the center of the pyramid. Then it builds one dimensional index in \( B^+ \)-tree according to \( P_v \). The final query results are obtained through filtering the candidate points which are searched from the \( B^+ \)-tree with the \( P_v \) of the query. The performance of pyramid technology far exceeds the tree-like indexing methods such as R-tree [34] and X-tree [35] under the condition of high dimension or massive distinct values [11]. But the pyramid index calculates the \( P_v \) only considering the dimension whose value is the furthest from the space center and ignoring the information of other dimensions, which can incur that two points whose values which largely differ from each other in some dimensions have the same \( P_v \) value. This problem is more serious for the dimensions with less distinct values because less distinct values reduce the number of \( P_v \) and increase candidate points of each \( P_v \). Other index technology derived from
the pyramid technique, such as P+–Tree and iMinMax [36], etc. has no obvious performance improvement under the uniform datasets because they mainly focus on the query for the skewed or clustered datasets [33]. So far there are no effective solutions to the high costs in filtering the candidate points corresponding to a query.

Querying data is also an important issue in other computing systems such as Peer-to-Peer and grid systems for handling resource discovery. DHT (Distributed Hash Table) technologies, such as Chord [37] and CAN [38], are successfully used in Peer-to-Peer systems, but neither of them can support range and multi-attribute queries without additional efforts. Some approaches, such as [39, 40, 41] have been proposed to handle range queries in P2P or grid computing environment. However, they are not designed to deal with high dimensional data. Space-Filling Curves (SFC) can be used to map a multi-dimensional space to an 1-dimensional space [42], and the idea is also adopted in the Pyramid-Technique and SP-Index. However, experimental results [11] demonstrate the performance of the Pyramid-Technique is much better than that of the SFC based strategies [43]. k-Nearest Neighbor (kNN) search in high dimensional data has been a popular research topic. Tree based approaches, such as R-tree and KD tree [9], often perform well when data dimensionality is not high, but their performance will rapidly degrade for high dimensional query [44]. The iDistance technique [45] presents a method for kNN search in a multi-dimensional space, and it also transforms data points into a single dimensional value. However, in a data partition (similar to the (d-1)-dimensional surface of a pyramid in the SP-Index), the transformation strategy of the iDistance technique mainly considers Euclidean distance, which means other characteristics in each dimension of data points are lost, while SP-Index considers the value characteristics in each dimension of data points respectively. Therefore, the iDistance technique is mainly suitable for kNN queries, while SP-Index is qualified for both range and kNN queries. Another category to support kNN and range queries is leveraging hash-based approaches, which trade accuracy for efficiency. For example, LSH [46] utilizes locality preserving hash functions to hash close data, with a high probability, to the same bucket. Some works [47, 48] based on LSH have arised, and they can support kNN query and range query in Peer-to-Peer overlay networks. However, these approaches cannot guarantee exact answers for queries.

Our earlier work entitled “SPKV: A Multi-dimensional Index System for Large Scale Key-Value Stores” in APWeb 2014[49] presents the core idea of
SPIKE. In this extended journal paper, we provide theoretical analysis of the performance of SP-Index and the original pyramid technique, present more description on implementation and discuss about the application of SPIKE. Besides, we also conduct more extensive experiments to evaluate SPIKE: firstly, we evaluate the performance of SPIKE on a skewed and clustered real-world check-in dataset; secondly, we present how to choose the optimal threshold $T$ for SP-Index via experimental methods; thirdly, we evaluate the scalability via experiments for the synthetic dataset, TPC-H dataset and check-in dataset by varying the number of servers; fourthly, we evaluate systems’ update performance on synthetic dataset.

8. Conclusions

This paper presents, SPIKE, an efficient two-level multi-dimensional index system for handling complex queries. First, we propose an index scheme called SP-Index that improves the original Pyramid index by splitting the multi-dimensional space much finer. With SP-Index, it enables the data stores to better support complex queries on large-scale datasets or datasets with less distinct values. Second, we present the design of SPIKE that builds SP-Index over the partitioned key-value store, which allows for efficient query processing on multi-dimensional dataset. We apply our prototype system to Cassandra with minimal changes to it and provide the Index−Content strategy to avoid the redundant store of the part of the non-Index columns. In addition, we add the process of partial update request to handle the non-in-place updates and optimize the process of the update to reduce the additional update costs caused by the index layer. Finally, we evaluated the performance of SPIKE with extensive experiments. The results demonstrate that SPIKE can efficiently handle complex queries in comparison with existing work.

In our future work, on the one hand we hope to address the limitations discussed in Section 6; on the other hand, we will work towards an open source project so that we can obtain helpful feedback from the community contributors.

9. Acknowledgment

This work was supported partly by National Key Research and Development Program of China (2016YFB1000804), partly by National Natural Science Foundation of China (No. 61370057, No. 61421003) and partly by
China 863 program (No. 2015AA01A202). The preliminary version of this work was published in the proceedings of the 16th Asia-Pacific Web Conference.


