SPKV: A Multi-dimensional Index System for Large Scale Key-Value Stores

Qi Wang, Hailong Sun, Yu Tang, and Xudong Liu

School of Computer Science and Engineering, Beihang University,
Beijing, China 100191,
{wangqi,sunhl,tangyu,liuxd}@act.buaa.edu.cn

Abstract. A number of key-value databases have emerged with the development of cloud computing, which provide the ability of large scale data storage, but they do not efficiently support the multi-dimensional range queries and kNN queries which are important in online applications. Thus, we introduce the Sliced Pyramid Index for Key-Value Stores (SPKV), an index system that bridges the gap between data scale and querying functionality for highly available and scalable distributed key-value databases. SPKV implements a distributed index system with an improved pyramid index scheme called SP-Index, which allows efficient multi-dimensional query processing. In our experiments, SPKV achieves dozens of times faster than other index systems for key-value databases.

1 Introduction

In recent years, the Internet data has been growing rapidly with the development of large scale Internet applications, such as social networking, e-commerce and so on. As a result of the requirements of big data and cloud computing, the data storage system is expected to achieve a series of new requirements which the traditional relational database cannot satisfy. Thus, a plenty of distributed NoSQL databases are developed. Key-value database is the most important category of NoSQL. They efficiently support simple queries based on the primary key, but most of the key-value databases do not efficiently support range queries, kNN queries and other complicated queries based on non-primary keys because of lacking of efficient indexes. For these queries, the whole dataset has to be scanned, which leads to excessive costs. This defect makes their application scenario mainly limited to simple applications or some offline data analysis applications with the help of MapReduce[2]. And it is not suitable for complex query requirements of online applications.

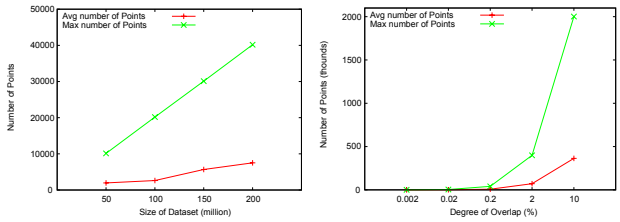
Although key-value stores follow a schema-free design, in practice developers prefer to store schema specific application data for convenient data processing with them. Therefore, in recent years, there have been many research on the indexing technology of key-value database for schema data, such as CCIndex[4] and BIDS[7]. However, existing indexing technologies are basically designed for specific architecture of key-value databases and are highly coupled with the underlying database storage engine, which limits the application scope of existing

index technologies. Therefore, the index technology for key-value databases still needs to be further studied to be adapted to more general application scopes.

In this paper, we propose the design of SPKV, an index system which provides efficient multi-dimensional query processing for different key-value databases. The key of SPKV is an efficient multi-dimensional index scheme called SP-Index, which is designed on the basis of the pyramid technique[1]. Since the pyramid technique mainly uses only one dimension to calculate the index value, which causes that multiple data points are mapped to a single pyramid value on large-scale datasets with less distinct values. In order to understand the impact of pyramid value, we perform an experiment to evaluate how the pyramid technique performs with changing number of distinct values and size of the dataset. Fig 1 shows that the number of candidate points to be scanned in a point query can be up to 400,000 for the dataset containing 20 million points with 50 distinct values in each dimension. The linear scan for so many points on the disk severely degrades the query performance. In order to decrease the number of points to be scanned, we specify the dimensions with less distinct values and divide the pyramid space much finer based on the information of all dimensions. Theoretical analysis shows that our division strategy leads to exponential improvement about the dimensionality on query performance compared to the original pyramid technique.

Then we apply SP-Index to Cassandra, a popular open-source key-value database, to implement a prototype system of SPKV. Experiments on synthetically generated dataset and the dataset of TPC-H benchmark[11] show SPKV can efficiently process complex multi-dimensional range queries and kNN queries. And it greatly outperforms some other index methods on key-value databases with tens of times faster.

The rest of the paper is organized as follows. In next section, we conduct a literature review of related works. In Section 3, we describe the design of SP-Index and its query processing. The implementation details of SPKV are



(a) Points vs dataset size (b) Points vs overlap degree

Fig. 1: The *overlap degree* means the percentage of the points with the same pyramid value in total points, which is negatively correlated to the number of distinct values. The number of distinct values in (a) is 50 and the dataset size of (b) is 20 million.

presented in Section 4. In Section 5, we evaluate the performance of SPKV and conclude this paper in Section 6.

2 Related Work

Nowadays, there are various key-value databases. Dynamo[3], Hbase[5] and Cassandra[6] are representative key-value stores. They cannot efficiently deal with complex queries based on non-primary key because of lack of efficient secondary index. In production environment, the solution is using MapReduce technology to scan the whole database in parallel to establish special data tables as indexes. 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. Besides, it is not a general method which can bring excess work to database users.

Complemental Cluster Indexing technology (CCIndex)[13, 4] is proposed based on Hbase and Cassandra. However, CCIndex need to store a replica for every dimensions in a row which results in large amounts of disk space consumption in high-dimensional cases, and it does not support for kNN query. BIDS[7] achieves very low space cost and provides efficient multi-dimensional range queries and join queries with highly compressed bitmap index. But it is more suitable for offline data analysis applications with rare updates rather than online applications because of the defect of the bitmap index in updating.

Pyramid technique[1] is proved to be an efficient multi-dimensional index structure. The pyramid technique adopts the strategy of the non-uniform space division and filter to distribute the data points into space pyramids. It calculates the pyramid index value (Pv) according to the multi-dimensional values. Then it builds one dimensional index in B^+-tree according to the Pv. The final query results are obtained through filtering the candidate points which are searched from the B^+-tree with the Pv of the query. The performance of pyramid technology far exceeds the tree-like indexing methods[1]. But the pyramid index calculates the Pv only considering the value of the dimension which is the furthest from the space center and ignoring the information of other dimensions, incurring that two points whose values which largely differ from each other in some dimensions have the same Pv. Other index technology derived from the pyramid technique, such as P^+-Tree[12], has no obvious performance improvement under the uniform datasets because they mainly focus on the query for the skewed distributed or clustered datasets[12]. So far there is no fundamental solution to the high cost in filtering the candidate points corresponding to the query.

3 The SP-Index

In order to cope with the query requirements of huge volumes of data in various types, we propose SP-Index on the basis of the pyramid technique. So we take a look at pyramid technique before introducing the SP-Index.

The pyramid technique divides the d-dimensional data space into $2d$ pyramids that share the center point of the space as their top (Fig 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 $Pv_X = (p + h_X)$.

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 number of Pv becomes less when there are less distinct values in each dimensions. These problems make the data points corresponding to a Pv will out of range of a leaf node in B^+-tree. Large numbers of candidate points increase the times of disk I/O 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.

Therefore, the main objective of SP-Index is to reduce 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 with the decreasing of the needed disk scanning.

3.1 Space Division of SP-Index

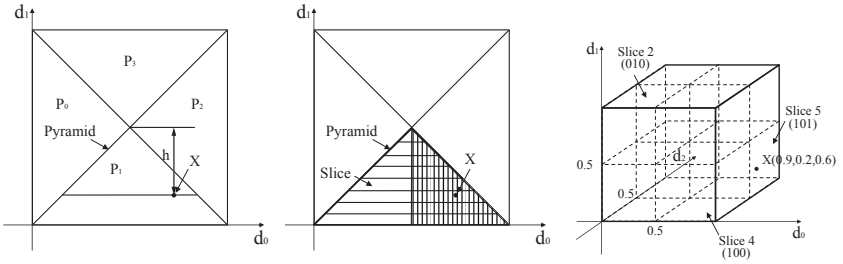
With this consideration, the basic idea of SP-Index is to enlarge the pyramid value scope of each pyramid and to provide finer division on the pyramid space so as to make each Pv corresponds to as less data points as possible. First, the d-dimensional data space is divided into $2d$ pyramids as original pyramid method does. Second, we specify the columns with less distinct values and perform the slice division to insurance the index items with the same Pv can be stored in a disk page. Then each corresponding pyramid will be further split into 2^{d-1} slices. Since we set the interval size of each slice to 1, the Pv of some points in a pyramid may greater than the upper bound of this pyramid which leads to collisions with the points in next pyramid. So we need to extend the interval of each pyramid to avoid the collisions. 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 will be addressed through a triple $< pyramidid, sliceid, height >$ (Fig 2(b)). The detailed method is described as following steps:

S1 We assume the dimension of data is d, a row of data is presented by a point $X = (x_0, x_1, ..., x_{d-1})$ in d-dimensional space. Normalize X according to the range of the value of each dimension and put it into d-dimensional [0,1] space. Then we get point $X' = (x'_0, x'_1, ...x'_{d-1})$.

S2 Get the id p of the pyramid which X' is belonging to as below.

$$p = \begin{cases} j_{max} & x'_{j_{max}} < 0.5) \\ j_{max} + d & x'_{j_{max}} \geq 0.5) \end{cases} \tag{1}$$

j_{max} is the number of the dimension which has the biggest value of $|x'_j - 0.5|$.



(a) Pyramid (b) Sliced Pyramid (c) Sliced Division

Fig. 2: Space division of Pyramid and SP-Index

S3 In this step, we determine pyramids need to be divided into slices. Considering the general case, the number of points in each node and in each dimension is similar with 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 to be the disk page size and S is the size of an index item. Generally, K is 4KB and S is about 64B. Then we can get the following formula.

$$\frac{N}{ndV} \geq \frac{K}{S} \rightarrow V \leq \frac{NS}{ndK} \tag{2}$$

In Formula (2), V is the max distinct values number of the columns which need to be further divided. Specifically, 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 finer 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 low bound of its interval, which is defined as $sp = p \cdot 2^s$.

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

However, a larger s would produce too many slices since the volume of slices grows exponentially with dimensions. Since large numbers of slices will increase the times of scans and reduce the performance in range queries, we set T to 8 based on experiment results, which is not presented in this paper because of the length limitation of the article, to limit the size of s.

S5 For the pyramids corresponding to the dimensions needn't to be slice divided, we calculate the height as $h = |0.5 - x'_{p\%d}|$ directly. And the pyramid

value of the points in this dimensions is defined as $Pv_X = sp+h$, which is similar with original pyramid technique.

On the other hand, for the pyramids corresponding to the dimensions with less distinct values, we will get the slice id $q_{X'}$ of point X'. At first, we need to determine the dimensions to be used in slice division. So we insert dimension numbers into empty collection S, which contains the number of dimensions which will be used, based on following regulations.

When $d \leq T$, it means all dimensions will be used in slice division, so we add the d dimensions to the collection S by ascending order of dimension number;

If $d > T$, it means some dimensions will not be used in the division. More distinct values lead to more Pv which means much finer division, so we get the first s dimensions with the most number of distinct values and insert them to collection S by ascending order of dimension number.

We judge the normalized value of all dimensions in S except dimension $p\%d$ and set the corresponding bit of q to 1 if the value $x' \leq 0.5$, otherwise we set the bit to 0. For example, there is a 4-dimensional point $X' = (0.9, 0.2, 0.6, 0.95)$, we can get $p = 7$ and $s = 3$ according to S2 and S4. Then we put X' into a 3-dimensional space showed in Fig 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. And so on, we set the 2nd and 3rd bit of q to 0 and 1 because $x'_1 < 0.5$ and $x'_2 > 0.5$. In this way, we know that X' is located in *Slice* 5 because $q_{X'} = 5$.

S6 In last step, we not only divide 4-dimensional space into $2d = 8$ pyramids but also divided each pyramid into $2^s = 8$ slices. Inside the 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 of SP-Index is defined as $Pv_X = sp + q + h$.

In order to dynamically insert a point X, we first determine the pyramid value Pv_X of the point through the steps above and then insert the point into B^+-tree or other data structures which efficiently support point and range queries to construct the SP-Index using Pv_X as the key.

3.2 Query Processing

The process of queries for the pyramids corresponding to the dimensions without slice division is similar with origin pyramid technique. So we mainly discuss on the queries in sliced pyramids in this section.

Point Query The process of point query based on column values is simple, we compute the Pv_Q of query point $Q = (x_0, x_1, ..., x_{d-1})$ using the methods in section 3.1 and querying the B^+-tree with Pv_Q, then we will obtain a set of candidate points sharing the Pv_Q. We can scan the set and determine whether the point is satisfied with conditions of each dimension of Q to obtain the final result.

Range Query Given a range query $Q = ((x_{0_{min}}, x_{0_{max}}), ..., (x_{d-1_{min}}, x_{d-1_{max}}))$, it will be processed as following steps.

S1 At first, we normalize Q to d-dimensional $[0, 1]$ space. Get the pyramids intersected with the query range. A pyramid p is intersected with the query Q

if and only if it satisfies the following formula. And we set the *sliced pyramid* id to $sp = p \cdot 2^s$.

$$\begin{aligned} &x_{i_{min}} \leq x_{j_{max}}, x_{i_{min}} \leq 1 - x_{j_{min}} && j = 0, 1, ..., d-1, p < d, i = p \\ &1 - x_{i_{max}} \leq x_{j_{max}}, 1 - x_{i_{max}} \leq 1 - x_{j_{min}} && j = 0, 1, ..., d-1, d \leq p < 2d, i = p\%d \end{aligned}$$

S2 In this step, we determined the slices need to be queried by the following algorithm in each *sliced pyramid*. Let x be the id of the slice need to be queried. We determine the upper bound and lower bound of query range in each dimension and set the corresponding bit of x to 0 if the upper bound is less than 0.5 or set the bit to 1 if the lower bound is greater than 0.5. If the point 0.5 is included 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 example, the query range is $((0.3, 0.4), (0.4, 0.7), (0.5, 0.6), (0.0, 0.1))$. The pyramid id $p = 3$, and we can get two slice number $x_1 = 1(001)$ and $x_2 = 3(011)$ because the query range of 2nd dimension includes the point 0.5. We will get the slices to be queried after all dimensions except the dimension $p\%d$ are judged.

S3 In the last step, we need to determine the query ranges $r = (r_{low}, r_{high})$ within each slice to be queried according to Formula (3).

$$\begin{aligned} f &= 0.5 - q_{x\%d_{min}} * 0.5 - q_{x\%d_{max}} \\ r_{low} &= \begin{cases} sp + q + Min(|0.5 - x_{p\%d_{min}}|, |0.5 - x_{p\%d_{max}}|) & f \geq 0 \\ sp + q & f < 0 \end{cases} \\ r_{high} &= \begin{cases} sp + q + Max(|0.5 - x_{p\%d_{min}}|, |0.5 - x_{p\%d_{max}}|) & f \geq 0 \\ sp + q + |0.5 - x_{p\%d_{min}}| & f < 0, p < d \\ sp + q + |0.5 - x_{p\%d_{max}}| & f < 0, p \geq d \end{cases} \end{aligned} \tag{3}$$

Through the steps above, we finally obtain a set of one dimensional ranges for each slice q of each intersect pyramid p. The points outside the ranges can be ensured exclusion from the query rectangular, and every point within the range is the candidate points to be processed. Thus, we can scan the set obtained from several range queries on B^+-tree and determine whether the point is satisfied with the query range of Q to obtain the final result.

kNN Query To find the kNN of a query point $X = (x_0, x_1, ..., x_{d-1})$, we adopt a kNN search algorithm with modified decreasing radius strategy[10]. We use a priority queue A to contain k candidate nearest neighbors sorted by the distance from X in decreasing order. Let $D(v, X)$ be the Euclidean distance between a candidate point v and point X, and D_{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 after following steps.

S1 A is initialized to be empty, and we calculate the $Pv_X = p_i + q_j + h$ using the method in section 3.1. We search the B^+-tree to locate the leaf node which has the key equal to Pv_X, or the largest key less than Pv_X. After locating the leaf node, we check the data points in the node towards both to the left and right, meanwhile, we calculate the $D(v, X)$ to determine if the point v is one

of the k nearest neighbors, and update A accordingly. The search process stops when the key of the leaf node is less than $\lfloor Pv_X \rfloor$ or greater than $\lfloor Pv_X \rfloor + 0.5$, or there are 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_{max}.

S2 If the size of queue A is less than k after we finish searching the interval $[\lfloor Pv_X \rfloor, \lfloor Pv_X \rfloor + 0.5]$ then we need to repeat S1 in the slice which is the nearest from the point X. We find dimension l which has the smallest $|0.5 - x_l|$ from the collection S which is mentioned in section 3.1. Then we invert the l-th bit of the slice number q_j to get q'_j and repeat S1 with $Pv_{X'} = p_i + q'_j + h$.

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 square W enclosing $C(X, r)$ to perform an 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 neighbors by compared to the 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 will get the finally results when all the pyramids are checked.

4 Implementation of SPKV

We implement SPKV with SP-Index and deploy SPKV on the nodes of the key-value database to support the complex queries on huge amounts of multi-dimensional data. In this paper, we use Cassandra as the data layer of SPKV. Cassandra is a popular open-source key-value database, in addition, it provides the SQL-like CQL language which introduces a schema-like data model and friendly query interfaces. It is valuable and necessary to apply SPKV to Cassandra to improve its query performance limited by its original inefficient secondary index. Moreover, it is convenient to apply SPKV to other key-value databases.

In the design of SPKV, we can adopt the same partition strategy and replication strategy with database layer basing on the current node states and the partition information of the cluster. The index layer in each node only indexes the local data partition by consistent hashing of Cassandra or other partition methods of underlying databases. Besides, SPKV ensures the availability of index layer when some nodes are subject to failures by the replication of SP-Index.

The persistent storage of index tables is implemented with MapDB[8], which is a B-tree-like storage engine. When inserting new data, we first compute the Pv value by the method described in section 3.1. Then we insert it to MapDB using "$Pv : RowKey$" as the key. We implement replication and partition functions for index tables based on the strategy of underlying key-value database to guarantee the high availability and scalability of the index system. Considering the inefficient insert performance of B-tree-like data structure, we provides memtable and commitlog for SP-Index to improve the insert performance. Besides, we divide the non-index columns into two parts. The columns which are

mainly queried by multi-dimensional conditions, in terms of *Index Content*, are stored in the index table. And the other columns which mainly queried by the rowkey are stored only in the underlying database in order to control the space cost of SPKV. Although we store the columns into different tables, we can get all columns of the row through an additional low-cost key-value query in database layer or point query in SP-Index.

5 Evaluation

In this section, we present a set of evaluation results to show the performance of SPKV. Specifically, we evaluate SPKV through the comparison with MySQL Cluster 7.3.2[9], CCIndex for Cassandra[4] and SPKV implemented with original pyramid technique in terms of multi-dimensional point query, range query and kNN query. Our experimental cluster has 10 nodes. Each node has 2.0 GHz quad-cores CPU, 16 GB memory and more than 200 GB HDD. All nodes are connected by 1Gbps Ethernet. SPKV is implemented on Cassandra 1.2.8.

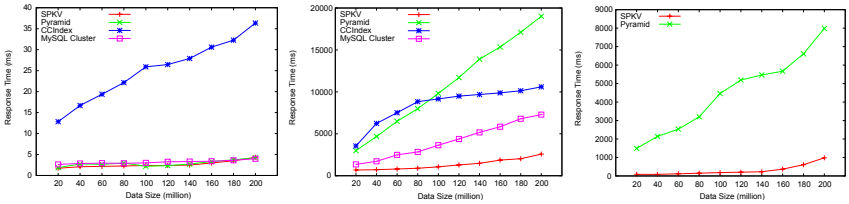
We both use the synthetically generated random datasets (The number of distinct values in each column ranges from 100 to 100,000) and the dataset of TPC-H in our evaluation. The total size of a row in generated dataset ranges from 128 to 164 Bytes according to the dimensions. We use the dataset of TPC-H, an acknowledged database benchmark which can simulate real business applications, to test the performance of SPKV in real world datasets. In order to test the performance of index, all experiments do not return the non-*IndexedContent* so as to avoid the influence of massive key-value query in Cassandra.

5.1 Effects of Data Size

In this section, we measure the performance with varying number of rows compared to other system. We perform point queries, range queries with 50,000 rows of selectivity and kNN queries with k=50 in a 6-dimensional dataset. Figure 3 displays the result of the experiments. The kNN query of original pyramid is implemented based on [10]. However, CCIndex and MySQL Cluster don't supported kNN queries so we skip the comparison with them. As is shown in Figure 3, all techniques perform well in the point query. Thanks to the finer division, SPKV outperformed other systems with speed factor 4∼8 in range query and speed factor 10∼20 in kNN query with every data sizes and performed well in large volume of data.

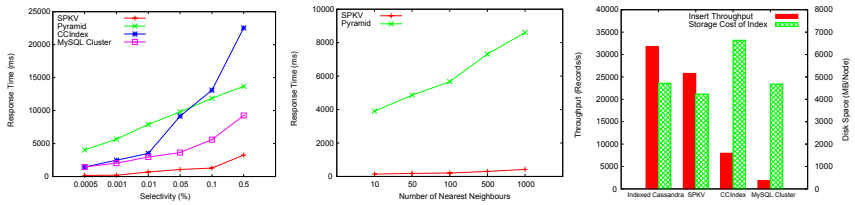
5.2 Effects of Selectivity

In this section we see how the selectivity and the number of K effect the performance of SPKV. In these experiments, we use 6-dimensional dataset and let the data size to be 100 million. Figure 4(a)(b) shows the results. SPKV always performs best and achieves the largest speedup factor in low selectivity or low K value. The reason is that the smaller query ranges lead to less slices need to



(a) Point query (b) Range query (c) kNN query

Fig. 3: Effect of Data Set Size



(a) Range query (b) kNN query (c) Maintenance Cost

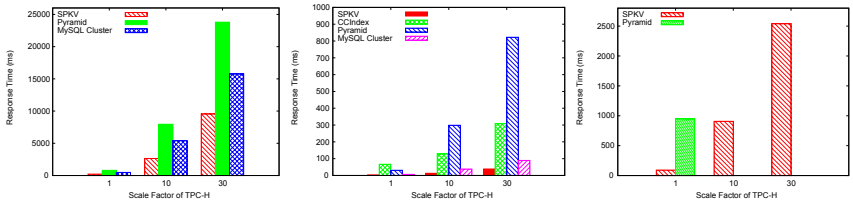
Fig. 4: Effect of Selectivity and Throughput

be query which decrease both the query times and the size of rows to be scan. When selectivity becomes larger the performance decreases, nevertheless, SPKV still outperforms other technique with 7∼20 times faster and can be competent to various of selectivity and K value.

5.3 Insert Throughput and Storage Cost

In the evaluation of insert throughput, we use 10 concurrent clients to insert rows to the system which has already stored 100 million rows of 6-dimensional data and the experiment result is showed in Figure 4(c). The insert throughput of SPKV is lower than Indexed Cassandra because of the additional write operations and network communication cost brought by the index table. However, SPKV performs far better than CCIndex which need to write 6 index tables and MySQL Cluster whose insert throughput is limited by the relational and transactional storage engine.

Figure 4(c) also indicates that each node of SPKV costs about 4GB disk space for a 6-dimension dataset of 100 million rows, which is less than the cost of 6GB disk space with CCIndex. Besides, it is 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 are acceptable while great query performance improvement is achieved.



(a) Range query with QA (b) Point query with QB (c) kNN query with QB

Fig. 5: Performance with TPC-H Dataset

5.4 TPC-H Benchmark

In this section, we use $Q6$ in TPC-H as QA to illustrate the practical effect on range queries of SPKV. We also define a simple query QB to evaluate the performance of point queries and kNN queries. For the queries we build a 5-dimensional index on the *Lineitem* table in TPC-H.(We change some columns of the table in order to adapt to Cassandra's data model.)

QA is defined as:

```
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:

```
SELECT extendedprice FROM Lineitem WHERE shipdate=sd AND commitdate
=sd+1 month AND discount=d AND tax=t AND quantity=q
```

The result is showed in Figure 5. SPKV still performs well but a bit lower than the experiments on generated dataset since the number of distinct values of index columns in *Lineitem* table are much smaller. In addition, the QA only specify the query range for 3 dimensions, that is, the other 2 dimensions of the queries are full domains. Even so, SPKV still achieves remarkable performance and outperforms other system because of the slice division on the dimensions whose query ranges are specific. The performance of CCIndex and pyramid technique rapidly deteriorates when the distinct values are less. They respectively fail to respond in 30 seconds during the range and kNN query, so we skip them in Figure 5(a)(c). All of above show that SPKV can well perform on columns with both less and more distinct values.

6 Conclusions

In this paper, we introduce an efficient multi-dimensional index technique named SP-Index. Through the much finer division on the pyramid space, we significantly improve the query performance on huge amounts of data. We present the design of SPKV that builds SP-Index over the partitioned key-value store, which allows efficient multi-dimensional query processing. In our experiments, the results demonstrate that SPKV can handle high scale of data using a modest 10 node

cluster, while efficiently processing multi-dimensional range queries and nearest neighbor queries and outperform some other multi-dimension methods.

7 Acknowledgments

This work was supported by National Natural Science Foundation of China (No. 61370057, No. 61103031), China 863 program (No. 2012AA011203), A Foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. 201159), Beijing Nova Program(No. 2011022) and Specialized Research Fund for the Doctoral Program of Higher Education (No. 20111102120016).

References

1. Berchtold, S., Böhm, C., Kriegal, H.P.: The pyramid-technique: towards breaking the curse of dimensionality. In: ACM SIGMOD Record. vol. 27, pp. 142–153. ACM (1998)
2. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP. vol. 7, pp. 205–220 (2007)
4. Feng, C., Zou, Y., Xu, Z.: Ccindex for cassandra: A novel scheme for multi-dimensional range queries in cassandra. In: Semantics Knowledge and Grid (SKG), 2011 Seventh International Conference on. pp. 130–136. IEEE (2011)
5. Hbase, A.: `http://hbase.apache.org/`
6. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
7. Lu, P., Wu, S., Shou, L., Tan, K.L.: An efficient and compact indexing scheme for large-scale data store. In: Data Engineering (ICDE), 2013 IEEE 29th International Conference on. pp. 326–337. IEEE (2013)
8. MapDB: `http://www.mapdb.org/`
9. MySQLCluster: `http://dev.mysql.com/downloads/cluster/`
10. Shi, Q., Nickerson, B.: Decreasing radius k-nearest neighbor search using mapping-based indexing schemes. Tech. rep., Tech. rep., University of New Brunswick (2006)
11. TPC-H: `http://www.tpc.org/tpch/`
12. Zhang, R., Ooi, B.C., Tan, K.L.: Making the pyramid technique robust to query types and workloads. In: Data Engineering, 2004. Proceedings. 20th International Conference on. pp. 313–324. IEEE (2004)
13. Zou, Y., Liu, J., Wang, S., Zha, L., Xu, Z.: Ccindex: a complemental clustering index on distributed ordered tables for multi-dimensional range queries. In: Network and Parallel Computing, pp. 247–261. Springer (2010)