FastRAQ: A Fast Approach to Range-Aggregate Queries in Big Data Environments

Xiaochun Yun, *Member, IEEE*, Guangjun Wu, *Member, IEEE*, Guangyan Zhang, Keqin Li, and Shupeng Wang

Abstract—Range-aggregate queries are to apply a certain aggregate function on all tuples within given query ranges. Existing approaches to range-aggregate queries are insufficient to quickly provide accurate results in big data environments. In this paper, we propose FastRAQ—a fast approach to range-aggregate queries in big data environments. FastRAQ first divides big data into independent partitions with a balanced partitioning algorithm, and then generates a local estimation sketch for each partition. When a range-aggregate query request arrives, FastRAQ obtains the result directly by summarizing local estimates from all partitions. FastRAQ has $O(1)$ time complexity for data updates and $O(\frac{N}{P\times B})$ time complexity for range-aggregate queries, where N is the number of distinct tuples for all dimensions, P is the partition number, and B is the bucket number in the histogram. We implement the FastRAQ approach on the Linux platform, and evaluate its performance with about 10 billions data records. Experimental results demonstrate that FastRAQ provides range-aggregate query results within a time period two orders of magnitude lower than that of Hive, while the relative error is less than 3 percent within the given confidence interval.

Index Terms—Balanced partition, big data, multidimensional histogram, range-aggregate query

1 Introduction

1.1 Motivation

Big data analysis can discover trends of various social aspects and preferences of individual everyday behaviours. This provides a new opportunity to explore fundamental questions about the complex world [1], [2], [3]. For example, to build an efficient investment strategy, Preis et al. [2] analyzed the massive behavioral data sets related to finance and yielded a profit of even 326 percent higher than that of a random investment strategy. Choi and Varian [3] presented estimate sketches to forecast economic indicators, such as social unemployment, automobile sale, and even destinations for personal travelling. Currently, it is important to provide efficient methods and tools for big data analysis. We give an application example of big data analysis.

Distributed intrusion detection systems (DIDS) monitor and report anomaly activities or strange patterns on the network level. A DIDS detects anomalies via statistics information of summarizing traffic features from diverse sensors to improve false-alarm rates of detecting coordinated attacks.

Such a scenario motivates a typical range-aggregate query problem [4] that summarizes aggregated features from all tuples within given queried ranges. Range-aggregate queries are important tools in decision management, online suggestion, trend estimation, and so on. It is a challenging problem to quickly obtain range-aggregate queries results in big data environments. The big data involves a significant increase in data volumes, and the selected tuples maybe locate in different files or blocks. On the other hand, real-time systems aim to provide relevant results within seconds on massive data analysis [5].

The *Prefix-sum Cube* (PC) method [4], [6] is first used in OLAP to boost the performance of range-aggregate queries. All the numerical attribute values are sorted and any range-aggregate query on a data cube can be answered in constant time. However, when a new tuple is written into the cube, it has to recalculate the prefix sums for all dimensions. Hence, the update time is even exponential in the number of cube dimensions. *Online Aggregation* (OLA) is an important approximate answering approach to speeding range-aggregate queries [7], which has been widely studied in relational databases [8] and Cloud systems [9], [10], [11], [12]. The OLA systems provide early estimated returns while the background computing processes are still running. The returns are progressively refined and the accuracy is improved in subsequent stages. But users cannot obtain an appropriate answering with satisfied accuracy in the early stages.

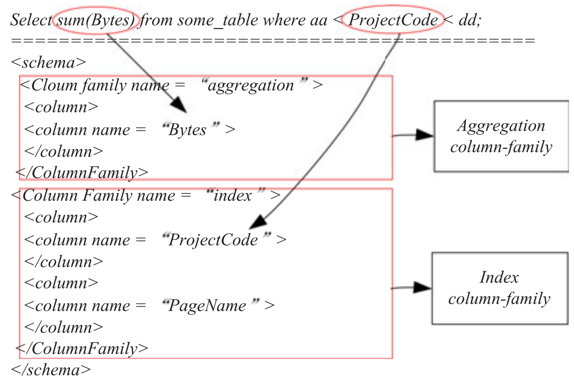
The sampling and histogram approaches have been utilized in database environments to support approximate answering or selectivity estimation. However, it can not acquire acceptable approximations of the underlying data sets, when data frequency distributions in different dimensions vary significantly.

1.2 Our Contributions

In this paper, we propose FastRAQ—a new approximate answering approach that acquires accurate estimations quickly for range-aggregate queries in big data environments.

- *X. Yun, G. Wu, and S. Wang are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100029, China. E-mail: {yunxiaochun, wuguangjun, wangshupeng}@iie.ac.cn.*
- *G. Zhang is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: gyzh@tsinghua.edu.cn.*
- *K. Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu*

Manuscript received 24 Feb. 2014; revised 26 June 2014; accepted 27 June 2014. Date of publication 29 July 2014; date of current version 10 June 2015. Recommended for acceptance by R. Ranjan, L. Wang, A. Zomaya, D. Georgakopoulos, G. Wang, and X.-H. Sun. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TCC.2014.2338325



```
Select sum(Bytes) from some_table where aa < ProjectCode < dd;
===========================================
<schema>
 <Cloum family name =   "aggregation " >
 <column>
 <column name =   "Bytes " >
 </column>
 </ColumnFamily>
<Column Family name =   "index " >
 <column>
 <column name =   "ProjectCode " >
 </column>
 <column>
 <column name =   "PageName " >
 </column>
 </ColumnFamily>
</schema>
```

Fig. 1. An example of the column-family schema.

FastRAQ first divides big data into independent partitions with a balanced partitioning algorithm, and then generates a local estimation sketch for each partition. When a range-aggregate query request arrives, FastRAQ obtains the result directly by summarizing local estimates from all partitions.

The balanced partitioning algorithm works with a stratified sampling model. It divides all data into different groups with regard to their attribute values of interest, and further separates each group into multiple partitions according to the current data distributions and the number of available servers. The algorithm can bound the sample errors in each partition, and can balance the number of records adaptively among servers when the data distribution and/or the number of servers changes.

The estimation sketch is a new type of multi-dimensional histogram that is built according to learned data distributions. Our multi-dimensional histogram can measure the quality of tuples distributions more accurately and can support accurate multi-dimensional cardinality queries. It can maintain nearly equivalent frequencies for different values within each histogram bucket, even if the frequency distributions in different dimensions vary significantly.

FastRAQ has $O(1)$ time complexity for data updates and $O(\frac{N}{P \times B})$ time complexity for ad-hoc range-aggregate queries, where N is the number of distinct tuples in all dimensions, P is the number of partitions, and B is the number of buckets in a histogram. Furthermore, it produces negligible volume of index data in big data environments.

We implement the FastRAQ approach on the Linux platform, and evaluate its performance with about 10 billions data records. Experimental results demonstrate that FastRAQ provides range-aggregate query results within a time period two orders of magnitude lower than that of Hive, while the relative error is less than 3 percent within the given confidence interval.

2 Overview of the FastRAQ Approach

2.1 Problem Statement

We consider the range-aggregate problem in big data environments, where data sets are stored in distributed servers. An aggregate function operates on selected ranges, which are contiguous on multiple domains of the attribute values. In FastRAQ, the attribute values can be numeric or alphabetic. One example of the range-aggregate problem is shown as follows:

$$\begin{aligned}&\textit{Select } exp(AggColumn), \textit{other ColName where}\\ &l_{i1} < ColName_i < l_{i2} \textit{ opr}\\ &l_{j1} < ColName_j < l_{j2} \textit{ opr}\\ &\ldots;\end{aligned}$$

In the above query, *exp* is an aggregate function such as SUM or COUNT; *AggColumn* is the dimension of the aggregate operation; $l_{i1} < ColName_i < l_{i2}$ and $l_{j1} < ColName_j < l_{j2}$ are the dimensions of ranges queries; *opr* is a logical operator including AND and OR logical operations. In the following discussion, *AggColumn* is called *Aggregation-Column*, $ColName_i$ and $ColName_j$ are called *Index-Columns*.

The cost of distributed range-aggregate queries primarily includes two parts. i.e., the cost of network communication and the cost of local files scanning. The first cost is produced by data transmission and synchronization for aggregate operations when the selected files are stored in different servers. The second cost is produced by scanning local files to search the selected tuples. When the size of a data set increases continuously, the two types of cost will also increase dramatically. Only when the two types of cost are minimized, can we obtain faster final range-aggregate queries results in big data environments.

2.2 Key Idea

To generate a local request result, we design a balanced partition algorithm which works with stratified sampling model. In each partition, we maintain a sample for values of the aggregation-column and a multi-dimensional histogram for values of the index-columns. When a range-aggregate query request arrives, the local result is the product of the sample and an estimated cardinality from the histogram. This reduces the two types of cost simultaneously. It is formulated as $\sum_{i=1}^{M} Count_i \times Sample_i$, where M is the number of partitions, $Count_i$ is the estimated cardinality of the queried ranges, and $Sample_i$ is the sample for values of aggregation-column in each partition.

Column-family schema for FastRAQ, which includes three types of column-families related to range-aggregate queries. They are *aggregation column-family*, *index column-family*, and *default column-family*. The aggregation column-family includes an aggregation-column, the index column-family includes multiple index-columns, and the default column-family includes other columns for further extensions. A SQL-like DDL and DML can be defined easily from the schema. An example of column-family schema and SQL-like range-aggregate query statement is shown in Fig. 1.

In FastRAQ, we divide numerical value space of an aggregation-column into different groups, and maintain an estimation sketch in each group to limit relative estimated errors of range-aggregate paradigm. When a new record is coming, it is first sent onto a partition in the light of current data distributions and the number of available servers. In each partition, the sample and the histogram are updated respectively by the attribute values of the incoming record.

When a query request arrives, it is delivered into each partition. We first build cardinality estimator (CE) for the queried range from the histogram in each partition. Then we calculate the estimate value in each partition, which is the product of the sample and the estimated cardinality from the estimator. The final return for the request is the sum of all the local estimates. A brief FastRAQ framework

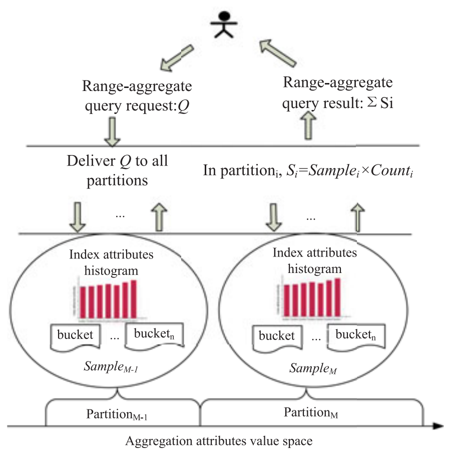

Range-aggregate query request: Q
Range-aggregate query result: Σ Si
Deliver Q to all partitions
In partition_i, S_i=Sample_i×Count_i
Index attributes histogram
bucket
bucket_n
Sample_{M-1}
Sample_M
Partition_{M-1}
Partition_M
Aggregation attributes value space

Fig. 2. The FastRAQ framework.

is shown in Fig. 2, and a multi-dimensional range-aggregate query process is presented in Algorithm 1.

Algorithm 1. FastRAQuering(Q)

```
Input: $Q$;
  $Q$: select sum(AggColumn) otherColname where $l_{i1} < ColName_i < l_{i2}$ opr $l_{j1} < ColName_j < l_{j2}$.
Output: $S$;
  $S$: range-aggregate query result.
1: Deliver the request $Q$ to all $partitions$;
2: for each $partition_i$ in $partitions$ do
3:   Compute the cardinality estimator of range $l_{i1} < ColName_i < l_{i2}$ from the local histogram, and let $CE_i$ be the estimator of the ith dimensions;
4:   Compute the cardinality estimator of range $l_{j1} < ColName_j < l_{j2}$ from the local histogram, and let $CE_j$ be the estimator of the jth dimensions;
5:   Merge the estimators $CE_i$ and $CE_j$ by the logical operator Opr, and compute the merged cardinality estimator $CE_{merged}$;
6:   $Count_i \leftarrow \hbar(CE_{merged})$;
     //$\hbar$ is a function of cardinality estimation.
7:   Compute the sample for $AggColumn$, and let $Sample_i$ be the sample;
8:   $SUM_i \leftarrow Count_i \times Sample_i$;
     //$SUM_i$ is a local range-aggregate query result;
9: end for
10: Set the approximate answering of FastRAQ as $S$. Let $S \leftarrow \sum_{i=1}^{M} SUM_i$, where $M$ is the number of partitions;
11: return $S$.
```

3 Distributed Partitioning Algorithm

Partitioning is a process of assigning each record in a large table to a smaller table based on the value of a particular field in a record. It has been used in data center networks to improve manageability and availability of big data [13]. The partitioning step has become a key determinant in data analysis to boost the query processing performance [14]. All of these works enable each partition to be processed independently and more efficiently. Stratified sampling is a method of sampling from independent groups of a population, and selecting sample in each group to improve the representativeness of the sample by reducing sampling error. We build our partitioning algorithm based on the idea of stratified sampling to make the maximum relative error under a threshold in each partition. At the same time, the sum of the local result from each partition can also achieve satisfied accuracy for any ad-hoc range-aggregate queries. We first divide the value of numerical space into different groups and subdivide each group into different partitions according to the number of available servers. The partition algorithm can be expressed as follows for data sets R:

$$Partitioning(R) = (g, p) = (V_e, random[1, V_r]), \qquad (1)$$

where the number of a partition p in a group g, is a random number in $[1, V_r]$, and V_e is a *group identifier* (GID) for the group g.

The stratified sampling is a method to subdivide the numerical value space into independent intervals with a batch of logarithm functions, and each interval stands for a group. When the number of logarithm functions is fixed, an arbitrary natural integer N can be mapped into a unique group g. The grouping model of stratified sampling is shown in Algorithm 2.

Algorithm 2. Grouping(N)

```
Input: $N$;
  $N$: an arbitrary numerical value ($N > 0$).
Output: $V_e$;
  $V_e$: the group Identifier ($GID$).
1: $k \leftarrow \log N$;
2: if ($k == 0$) then
3:   $V_e \leftarrow <0,0,0>$;
4:   Set the interval length of group $V_e$ as [0,1];
5:   return $V_e$;
6: else
7:   if ($N - 2^k == 0$) then
8:     $V_e \leftarrow <k,0,0>$;
9:     Set the interval length of group $V_e$ as $[2^k, 2^k + 1]$;
10:    return $V_e$;
11:  else
12:    $l \leftarrow \log N - 2^k$;
13:    if ($l == 0 \| N - 2^k - 2^l == 0$) then
14:      $V_e \leftarrow <k,l,0>$;
15:      Set the interval length of group $V_e$ as $[2^k + 2^l, 2^k + 2^l + 1]$;
16:      return $V_e$;
17:    else
18:      $m \leftarrow \log N - 2^k - 2^l$;
19:      Set the interval length of group $V_e$ as $[2^k + 2^l + 2^m, 2^k + 2^l + 2^{m+1} - 1]$;
20:      return $V_e$.
21:    end if
22:  end if
23: end if
```

TABLE 1
The Maximum Number of Groups in Different Value Spaces

numeric value space	$[1, 2^{10}-1]$	$[1, 2^{20}-1]$	$[1, 2^{30}-1]$
interval number	145	1775	8190

Algorithm 2 also presents the calculations for lengths of the grouping model. For example, when GID equals to $<0,0,0>$ the length of the group is [0,1]. When GID equals to $<k,l,m>$, $k \neq 0$, $l \neq 0$, $m \neq 0$, the length of the group is $[2^k+2^l+2^m,\ 2^k+2^l+2^{m+1}-1]$. Other processes of calculations are shown in Steps 5 and 15 of Algorithm 2. In Algorithm 2, it uses triple logarithmic functions to divide numerical space into independent groups. This can achieve better tradeoff between sampling errors (see Section 5) and the number of groups. The instances for the number of groups in different value spaces are listed in Table 1. For instance, it will produce 8,190 groups at most in the value space $[1, 2^{30}-1]$, and it is acceptable in many applications. Of course, one can increase the number of logarithm functions to reduce the sample error in each group, but it will produce a greater number of groups.

To make data balanced on each server, the partition algorithm subdivides each group into a number of partitions according to the current data distributions and sends each partition onto one server. Let V_r represent the maximum number of partitions in each group. The value of V_r is related to the current data distributions and the number of available servers at the same time. We design Algorithm 3 to compute the value of V_r for the current system. The key idea of Algorithm 3 is to calculate an average ratio of records b_0 for all groups, and then set the value of V_r according to b_0 and the current number of records in each group.

Algorithm 3. Numbering(G, dr)

```
Input: $G$;
  $G = \{<GID_i, nr_i>, 1 \leq i \leq M\}$;
  $dr$: the maximum number of partitions for a group;
  $GID_i$: the group identifier of group $g_i$;
  $nr_i$: the number of records in $g_i$;
  $M$: the number of groups.
Output: $VP$;
  $VP$: the partition vectors set, and
  $VP \leftarrow \{V_{pj} | 1 \leq j \leq M\}$.
1:  Compute an average ratio of record for all groups,
      i.e., $b_0 \leftarrow \sum_{i=1}^{M} nr_i / M$;
2:  $V_{rMax} \leftarrow number\_of\_servers \times dr$, and $V_{rMin} \leftarrow 1$;
3:  for all $(g_i \in G)$ do
4:    if $(g_i.nr_i < b_0)$ then
5:      $V_{pi} \leftarrow <g_i.GID, V_{rMin}>$;
6:    else
7:      $V_{pi} \leftarrow <g_i.GID, MIN\{\frac{nr_i}{b_0}, Vr_{Max}\}>$;
8:    end if
9:    $VP \leftarrow VP + V_{pi}$;
10: end for
11: return $VP$.
```

The number of partitions should be kept under some threshold in an applicable system. Some groups may hold the majority of input records, and it will make $\frac{nr_i}{b_0}$ be a very large number. We use the factor dr to bound the maximum number of partitions in each group. As shown in step 7 of Algorithm 3, the V_r locates in the interval [V_{rMin}, V_{rMax}], where the V_{rMax} and V_{rMin} are the maximum and minimum number of partitions for each group.

In big data environments, a partition is a unit for load balancing and local range-aggregate queries. FastRAQ uses the vectors set $VP = \{Vp_i : <Ve, Vr> | 1 \leq i \leq M\}$ to build partitions for all the incoming records, where M indicates the number of groups. In each partition, a dynamic sample is calculated from the current loaded records. Currently, FastRAQ uses a mean value of aggregation-column as the sample, which is $Sample = SUM/Counter$, where SUM is sum of values from aggregation-column, and $Counter$ is the number of records in the current partition. A detailed balanced partition algorithm is shown in Algorithm 4.

Algorithm 4. Partitioning(R,VP)

```
Input: ($R$,$VP$);
  $R$: an input record;
  $VP$: the partition vector set.
Output: $PID$;
  $PID$: a partition identifier for partition $p$.
1: Parse the input record $R$ into different column-families by the defined schema;
2: Compute the $GID$ with its value from aggregation-column by algorithm 2;
3: Get the partition vector $V_{pi}$ from $VP$ with the $GID$, and
   let $V_{pi} = <GID, V_r>$;
4: Set target partition identifier,
       $PID \leftarrow <GID, random[1, V_{pi}.V_r]>$;
5: Build the sample in partition $PID$, such as:
       $counter_{PID} \leftarrow counter_{PID} + 1$;
       $//counter_{PID}$ is the number of record;
       $sum_{PID} \leftarrow sum_{PID} + N$;
       $//N$ is value of aggregation attribute from $R$;
       $Sample_{PID} \leftarrow sum_{k,l,m,r}/counter_{PID}$;
6: $RID \leftarrow Hash(PID, counter_{PID})$;
       $//RID$ is the unique record identifier for $R$;
7: Send $R$ to partition $PID$;
8: return $PID$.
```

The input record R is sent to a partition represented by PID. The PID is generated from its value of aggregation-column. When the data distribution or the number of available severs changes, it just needs to modify the V_r in corresponding partition vector V_p, and the newly incoming records will be adaptively mapped into a partition in [1,V_r] randomly.

4 Range Cardinality Estimation

4.1 Clustering Based Histogram

We measure the data distributions by clustering values of all index-columns and use the learned knowledge to build our histogram. A feature vector of clustering is expressed as $\{tag, vector\}$, where tag is the attribute value, and $vector$ is the frequency for the tag occurring in each dimension. For example, the feature {*tag*=*ad*, *vector*=<10,2>} indicates that the value of ad occurs in the first index column 10 times and the second index column 2 times. After extracting the feature vectors from learned data set, it will produce vectors set. Let it be $\{<tag_i, vector_i> | 0 < i < N\}$. We use the common K-Means clustering method to analyze the vectors set and produce K clusters. A unique $ClusterID$ is assigned to each cluster. We construct a list of key-value pairs from the result of K clusters. The key-value pairs are in the format of $< tag, ClusterID >$. We sort the key-value pairs by tag in alphabetical order. The buckets in the histogram are built from the sorted pairs. The key idea is to merge the pairs with the same $ClusterID$s into the same bucket. If some tag occurring frequency is significantly different from others, its $ClassID$ is different after the K-Means clustering, and it will be put into an independent bucket in the histogram.

Algorithm 5. Building(F)

```
Input: $F$;
  $F$: learning data set.
Output: $P$;
  $P$: a bucket boundary list.
 1: Scan the learning data source $F$, and generate the frequency features set $\{<tag_i, vector_i> | 0 < i < N\}$, where $tag$ is the attributes value, $vector$ is the frequency occurring on each dimension;
 2: Cluster the features set $\{<tag_i, vector_i> | 0 < i < N\}$ by K-Means clustering method and produce $K$ clusters $\{Cluster_i | 1 \le i \le K\}$;
 3: Assign a unique $ClusterID$ to each cluster, and scan the $K$ clusters to generate key-value pairs list $\{<tag_q, ClusterID> | 1 \le ClusterID \le K\}$;
 4: Sort the key-value pairs list by $tag$ in alphabetical order, and the sorted sequence is $S = \{S_i : <tag_i, clusterID> | 1 \le i \le N\}$;
 5: for all $S_i$ in $S$ do
 6:   if ($Cureent\ ClusterID == S_i.ClusterID$) then
 7:     $i++$; continue;
 8:   else
 9:     Add $S_i.tag_i$ into $P$;
10:     $CureentClusterID \leftarrow S_i.clusterID$;
11:     $i++$;
12:   end if
13: end for
14:   Add $MIN\_VALUE$, $MAX\_VALUE$ into $P$;
15: return $P$.
```

Algorithm 5 produces buckets boundary P for the histogram, and $P = \{p_i | 0 \le i \le n\}$, where p_i is the value of tag from the feature vector. The values spreads for buckets in the histogram are $[p_0, p_1), [p_1, p_2), \ldots, [p_{n-1}, p_n)$ respectively, and $p_0 = -\infty$, $p_n = +\infty$. In Algorithm 5, we let MIN_VALUE be $-\infty$, and MAX_VALUE be $+\infty$.

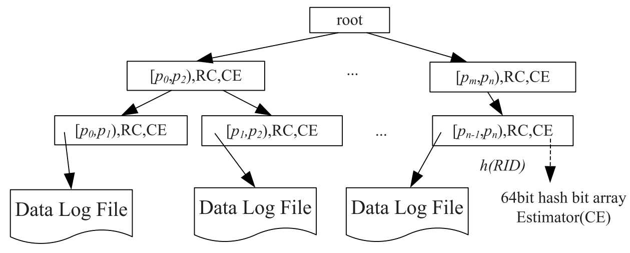


Fig. 3. A typical RC-Tree structure.

4.2 Range Cardinality Queries

FastRAQ supports multi-dimensional ranges queries, each of which may include multiple buckets of the histogram. FastRAQ uses a unique $RecordID$ (RID, as step 6 in Algorithm 4) to predict whether the cardinalities obtained from different buckets belonging to the same record. We adopt the HyperLogLogPlus algorithm to estimate the cardinality in the queried range [15]. We serialize the hash bits to bytes array in each bucket as a *cardinality estimator*. HyperLogLogPlus uses 64 bits hash function instead of 32 bits in HyperLogLog to improve the data-scale and estimated accuracy in big data environments. Readers can further refer to the references [15], [16] to learn about cardinality estimation mechanism. We establish a hierarchical tree structure to implement the histogram. A typical index structure is shown in Fig. 3. We term it *range cardinality tree* (RC-Tree).

RC-Tree includes three types of nodes, which are root node, internal nodes, and leaf nodes. The root node or an internal node points to its children nodes and keeps their values of spreads, such as $[p_i, p_j)$. A leaf node is for one bucket in the histogram. The parameters in a leaf node are values of spreads for each bucket, for example $[p_i, p_{i+1})$, the estimator CE of each bucket, and the bucket data file pointer. The leaf node only keeps these statistical information, and tuples values are stored in bucket data files. Because the buckets are independent of each other, the RC-Tree structure and its construction process are similar to the B+ Tree. We do not discuss the details further in this paper.

In order to improve throughput of RC-Tree, a hash table for newly incoming data is introduced for incremental updating process. The hash table consists of multiple nodes which are identical to the RC-Tree's leaves nodes. If a new record is coming, it first writes into the hash table, creates node if it does not exist, and then appends the tuples values into a temporary data file. When the number of nodes in the hash table reaches a threshold, the hash table flushes nodes into the RC-Tree, and appends the temporary files to the formal bucket data files. The incremental updating process will greatly improve the throughput of RC-Tree in big data environments. Algorithm 6 discusses the incremental updating process in RC-Tree.

The RC-Tree supports to search a leaf node randomly and sequentially. For example, when we query range (l_{i1}, l_{i2}) cardinality, we first locate the first leaf node using random searching method. Let the first node be $Node_i$, such that $l_{i1} \le Node_i.p_i$, where $[p_i, p_{i+1}) \in Node_i$. Then we find other nodes sequentially from $Node_i$, until the last node is found. Let the last node be $Node_j$, and $l_{i2} \ge Node_j.p_{j+1}$, where $[p_j, p_{j+1}) \in Node_j$. All the CEs from $Node_i$ to $Node_j$

are merged into a single CE with binary format, and the cardinality of range $[p_i, p_{j+1})$ is obtained from the merged CE. If the two edge nodes $Node_i$, $Node_j$ do not fully cover the queried range (l_{i1}, l_{i2}), that is to say, $l_{i1} < p_i$ and/or $l_{i2} > p_{j+1}$. There are two methods to compute the remainder edge range cardinality. The first is to scan the bucket data file to build the remainder edge cardinality estimator. The second is to use the estimators from edge nodes, which are $Node_{i-1}$ and/or $Node_{j+1}$, to directly obtain the remainder range cardinality. The second method is simpler and does not need to scan the bucket data files, but it will bring extra errors into the estimate. It is believed that if the edge bucket accounts for smaller cardinality ratio in the final queried results, the second method can quickly produce satisfied estimation.

Algorithm 6. Updating(R, P)

```
Input: ($R$, $P$);
  $R$: an input record;
  $P$: bucket boundary key set.
Output: $T$;
  $T$: the RC-Tree.
 1: for all $columns \quad in \quad R$ do
 2:   Parse value of index-columns into key-value pairs, in format of $< IndexValue, RID >$;
 3:   Search in the buckets spreads $P$, and get the target bucket $[p_i, p_j)$, such that $IndexValue \in [p_i, p_j)$
 4:   Search in hash table and get the target node $Node_H$, which include bucket range $[p_i, p_j)$;
 5:   $Node_H.RC \leftarrow Node_H.RC + 1$;
 6:   Set $RID$ into $Node_H.CE$;
 7:   Write $IndexValue$ into a temporary bucket data file;
 8:   if $(hash \quad table \quad node \quad number > threshold)$ then
 9:     for all $nodes \quad in \quad hash \quad table$ do
10:       Flush the nodes of hash table into $T$;
11:       Append the temporary data files into the formal bucket data files.
12:     end for
13:   end if
14: end for
15: return $T$.
```

To query cached data in hash table, the process is the same as Algorithm 7 to obtain cardinality estimator of the cached data, and then we merge the estimator into CE_{merge} to compute the final cardinality estimation. If the request includes multiple ranges, the queried ranges are connected by AND or OR logical operators. The logical OR operation is simple. We obtain estimators for each queried ranges respectively, and then merge the estimators into a single estimator to produce the final estimate. The logical AND operation is relatively complex. Currently, FastRAQ uses exclusive-inclusive principle for the logical AND operation, which is $|A| \bigcap |B| = |A| + |B| - |A| \bigcup |B|$. When the size of $||A| \bigcap |B||/MIN|A|, |B|$ is large enough, the exclusive-inclusive principle can produce a satisfied accuracy estimate. There are also some discussions about how to get a better cardinality estimation when the size of $||A| \bigcap |B||/MIN|A|$, $|B|$ is small [17].

Algorithm 7. Range cardinality query algorithm

```
Input: ($Q$, $T$, $h_0$);
  $Q$: select distinct count(*) where $l_{i1} < ColName < l_{i2}$;
  $T$: the RC-Tree;
  $h_0$: the edge range cardinality ratio.
Output: $R$;
  $R$: the range cardinality queried result.
 1: According to the queried range $(l_{i1}, l_{i2})$, locate the first node by $ColName$ in RC-Tree $T$ randomly, and let the searched node be $Node_i$, where $l_{i1} < p_i$ and $[p_i, p_{i+1}) \in Node_i$;
 2: $m \leftarrow i$;
 3: while $(l_{i2} > p_{m+1})$ do
 4:   Merge $Node_m$. CE into cardinality estimator $CE_{merge}$;
 5:   $m{+}{+}$;
 6: end while
 7: if $(\frac{\hbar(Node_{i-1}.CE)}{\hbar(CE_{merge})} \leq h_0)$ then
 8:   Merge $Node_{i-1}$.CE into cardinality estimator $CE_{merge}$;
 9: else
10:   Scan bucket data file of $Node_{i-1}$ to compute the exact cardinality $CE_{i-1}$;
11:   Merge $CE_{i-1}$ into cardinality estimator $CE_{merge}$;
12: end if
13: if $(\frac{\hbar(Node_{j+1}.CE)}{\hbar(CE_{merge})} \leq h_0)$ then
14:   Merge $Node_{j+1}$. CE into cardinality estimator $CE_{merge}$;
15: else
16:   Scan bucket data file of $Node_{j+1}$ to compute the exact cardinality $CE_{j+1}$;
17:   Merge $CE_{j+1}$ into cardinality estimator $CE_{merge}$;
18: end if
19: $R \leftarrow \hbar(CE_{merge})$;
20: return $R$.
```

5 Analysis of Relative Errors

FastRAQ uses approximate answering approaches, such as sampling, histogram, and cardinality estimation etc., to improve the performance of range-aggregate queries. We use relative error as a statistical tool for accuracy analysis. Relative error is widely used in an approximate answering system. Also, it is easy to compute the relative errors of combined estimate variables in a distributed environment for FastRAQ.

In this section, we analyze the estimated relative error and the confidence interval of final range-aggregate query result.

In our work, the relative error is defined as follows:

$$\frac{|variable_{true} - variable_{est}|}{variable_{true}}, \tag{2}$$

where $variable_{true}$ is the true value of a variable, and $variable_{est}$ is an estimate of the variable $variable_{true}$. Equation (3) is usually used as an acceptable substitute for the analysis of relative error,

$$\frac{|variable_{true} - variable_{est}|}{variable_{est}}. \tag{3}$$

Δ is used as a notation to represent relative error of a given variable. Let Y be the exact range-aggregate result, and $\widehat{Y}$ be estimated variable of Y. Their relative errors are ΔY and $\Delta\widehat{Y}$ respectively. Let S be the local range-aggregate result in each partition, and $\widehat{S}$ be estimated variable of S. Their relative errors are ΔS and $\Delta\widehat{S}$.

First, we discuss expectation and variance of $\Delta\widehat{S}$ in a partition. We present Theorem 1 to discuss ΔS in each partition.

Theorem 1. *$\Delta\widehat{S}$ is an unbiased estimation of ΔS in big data environments.*

Proof: According to Algorithm 3, the range-aggregate query result $\widehat{S}$ in each partition is expressed as follows:

$$\widehat{S} = Count \times Sample, \tag{4}$$

where *Count* is estimated range cardinality obtained from the histogram, *Sample* is a sample of values of aggregation-column in the queried partition. The exact range-aggregate result S is expressed as $S = \sum_{j=1}^{n} X_j$, where X is a selected tuple in the queried partition. If the estimators of two edge-buckets are produced by scanning bucket data files, they do not lead to extra errors of the estimate $\widehat{S}$. Let $Avg = \frac{1}{n}\sum_{j=1}^{n} X_j$, and $S = n \times Avg$. Suppose the selected tuples randomly distribute in the queried partition, and *Avg* approaches to *Sample* when the number of selected tuples is large enough. According to Eq. (4), the expectation of $\Delta\widehat{S}$ can be expressed next:

$$E(\Delta\widehat{S}) = E\Big(|(S - \widehat{S})/S|\Big) = E\left(\left|1 - \frac{Count}{n}\right|\right). \tag{5}$$

Suppose the buckets of histogram are independent of each other, then *Count* is an unbiased estimation of n in big data environments [16], that is to say, $\frac{Count}{n} = 1$, thus $E(\Delta\widehat{S})$=0. □

We use error transformation formula to analyze variance of $\Delta\widehat{S}$ and it is expressed as follows:

$$\sigma(\Delta\widehat{S}) = \sqrt{\sigma^2(\Delta Sample) + \sigma^2(\Delta Count)}, \tag{6}$$

where $\sigma^2(\Delta Sample)$ is variance of relative error of sample for values of aggregation-column in a partition, and $\sigma^2(\Delta Count)$ is variance of relative error for cardinality estimation in a histogram. We suppose that $\Delta Sample$ obeys a uniform distribution, and it can be expressed as $U(a, b)$, where a and b are the minimum and maximum values of the distribution. The variance of uniform distribution is $\frac{(b-a)^2}{12}$. We omit the minus relative error in the succeeding discussions. According to Algorithm 2, a and b can be computed in each group within stratified sampling model, and the standard variances ($\sigma(\Delta\widehat{S})$) in different numeric value spaces are listed in Table 2.

The variance of estimated cardinality has been discussed in the work of [16], and the $\sigma(\Delta Count)$ asymptotically equals to $\frac{1.04}{\sqrt{m}}$, where m is the number of register bit array. If we set $m = 2^{12}$, $\sigma(\Delta\widehat{S}) = 0.026$.

TABLE 2
The Standard Variance in Different Numeric Space

numeric value space	$[1, 2^{10} - 1]$	$[1, 2^{20} - 1]$	$[1, 2^{30} - 1]$
maximum relative error(b)	0.07	0.07	0.07
the standard variance($\sigma(\Delta\widehat{S})$)	0.02	0.02	0.02

Next, we discuss the relative error and confidence interval for final range-aggregate query result.

We use Theorem 2 to discuss relationship between ΔS and ΔY.

Theorem 2. *ΔS is an unbiased estimation of ΔY, that is $E(\Delta Y) = E(\Delta S)$.*

Proof According to Eq. (2), ΔY can be expressed as follows:

$$\Delta Y = \frac{\sum_{i=1}^{M} \Delta S_i \times S_i}{\sum_{i=1}^{M} S_i}, \tag{7}$$

where ΔS_i is relative error of local range-aggregate query result in the ith partition. According to Algorithm 2, the partitions are independent from each other, and $\{\Delta S_i | 1 \le i \le M\}$ are *independent and identically distributed* (i.i.d.) variables. The $\{\Delta S_i\}$ can be considered as a list of observations for variable ΔS. Let $\sum_{i=1}^{M} S_i$ be a constant C, and the expectation of ΔY can be written as follows:

$$E(\Delta Y) = E\left(\frac{\sum_{i=1}^{M} \Delta S_i \times S_i}{C}\right) = E(\Delta S). \tag{8}$$

Thus $E(\Delta S)$ is an unbiased estimation of $E(\Delta Y)$. □

We further discuss the variance of variable Y, which is expressed as follows:

$$Y = \sum_{i=1}^{M}\sum_{j=1}^{n} X_{ij}, \tag{9}$$

where M is the number of partitions, X_{ij} is the value of aggregation-column in the queried ranges of the ith partition. Let S_i be the local range-aggregate query result in the ith partition, thus Y is

$$Y = \sum_{i=1}^{M} S_i. \tag{10}$$

In Eq. (10), Y is the sum of i.i.d. variables $\{\Delta S_i\}$. According to *Central Limit Theorem*, if M is large enough, Y obeys a normal distribution, that is $Y \sim N(\mu, \sigma^2)$, where μ and σ^2 is the expectation and variance of S_i.

We can obtain the corresponding formulas to compute confidence interval of variable Y. Let Y locate in an interval with probability p, which is expressed as:

$$P(|Y - \mu| \le \zeta) = P\left(\left|\frac{\sqrt{n}(Y - \mu)}{\sigma}\right| \le \frac{\zeta\sqrt{n}}{\sigma}\right) = p. \tag{11}$$

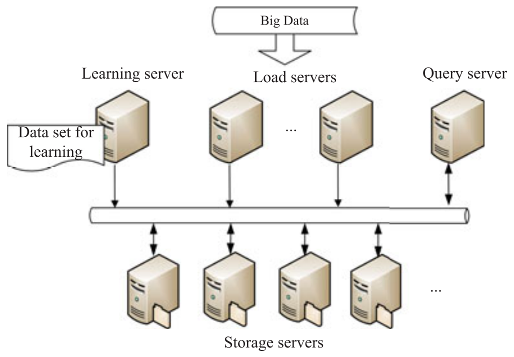


Fig. 4. System configuration used in experiments.

Then Y locates in $[\zeta - \mu, \zeta + \mu]$ with probability p, where $\zeta = \frac{z_p \sigma}{\sqrt{n}}$, and z_p is p-quantile in the standard normal distribution. The final $100p$ percent confidence interval of range-aggregate query result is $[\zeta - \mu, \zeta + \mu]$.

6 Experimental Evaluation

In this section, we present a prototype of FastRAQ, and evaluate its performance in terms of query cost, estimated relative errors, and storage overhead. We compare FastRAQ with Hive through range-aggregate query examples with real-world page traffic files from Wikipedia.

Hive is a typical data analysis tool with $O(N)$ time complexity for any ad-hoc range-aggregate queries. Hive can compile the task of an ad-hoc range-aggregate query into optimized mapreduce jobs and execute them on top of Hadoop. It is widely used to process extremely large data sets on commodity hardware in Facebook [18]. We compare against Hive in our experiment to illustrate performance improvement between FastRAQ and the $O(N)$ time complexity methods. We run our software on an eleven node cluster connected by 1 Gbit Ethernet switch. Each server has 6×2.0 GHz processors, 64 GB of RAM, and 6 SATA disks. We use Cloudera CDH4 in our experiments, which includes the packagings of Hadoop-2.0.0 and Hive-0.10.0. Hive runs with one master node and 10 slaves.

6.1 Evaluation Methodology

The framework of FastRAQ includes four types of servers: learning server, load server, query server, and storage servers. The learning server fetches a certain amount of data set

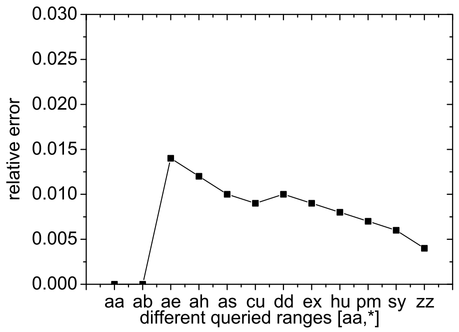


Fig. 5. The relative errors in different queried ranges.

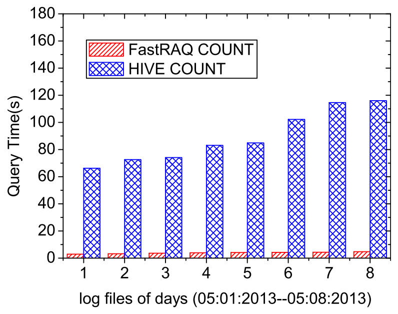


Fig. 6. Performance comparisons for count queries with eight days log files.

to learn data distributions, builds histogram and partition vectors for all partitions, and then dispatches them to other servers. The load servers receive online data sets, and deliver them to specified storage servers. The query server receives user's query request, and sends it to all storage servers. The storage servers keep RC-Tree for each partition, and respond the request independently. A typical framework of FastRAQ is shown in Fig. 4.

In the experiments, we analyze the pagecount traffic statistics files of Wikipedia [19]. We construct a table containing four columns. We set $projectcode$ and $pagename$ columns as index columns, $bytes$ field as aggregation-column. The FastRAQ stores four months of the traffic files which includes 960 GB of uncompressed data.

We first analyze the relative error in different queried examples. We use the traffic log files from Wikipedia in eight days. We set random variables in the queried examples and calculate the relative errors of different examples. The query example is "select $sum(bytes)$ from $pagecounts$ where $projectcode \in ('aa', '*')$", where '*' is a random variable string changed from 'aa' to 'zz'. The relative errors in different queried examples are shown in Fig. 5. We just present the values of '*' on the X axis. When the '*' equals to 'aa' and 'ab', the relative errors are equal to zero. The results are calculated by scanning the log files of the two edge-buckets. When the '*' grows larger, the relative error increases slightly. The relative errors are nearly constant when the '*' equals to 'cu', 'dd' and 'ex'. In our experiment, we use $('aa', 'dd')$ as our queried examples in following evaluations.

The examples of range-aggregate queries include count and sum queries, and aggregate functions on union queries. The queried examples are shown below:

Count query: Select $count(*)$ from $pagecounts$ where $projectcode \in ('aa', 'dd')$;
Sum query: Select $sum(bytes)$ *from* $pagecounts$ *where* $projectcode \in ('aa', 'dd')$.

Count on union query: Select $count(*)$ *from* $pagecounts$ *where* $projectcode \in ('aa', 'dd')$ *or* $pagename \in ('aa', 'dd')$;
Sum on union query: Select $sum(bytes)$ *from* $pagecounts$ *where* $projectcode \in ('aa', 'dd')$ *or* $pagename \in ('aa', 'dd')$;

During processing of the preceding queries, Hive returns the exact queries results, and FastRAQ returns estimated results with relative errors.

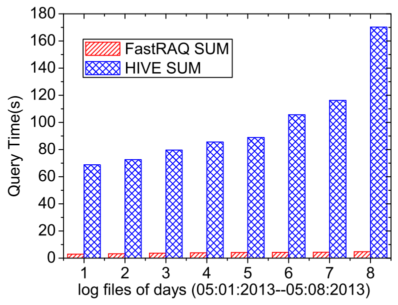


Fig. 7. Performance comparisons for sum queries with eight days log files.

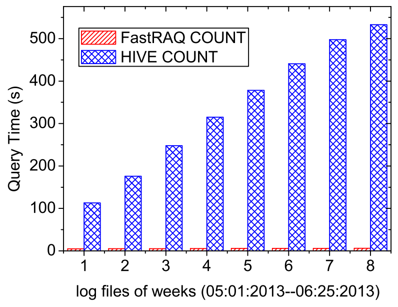


Fig. 8. Performance comparisons for count queries with eight weeks log files.

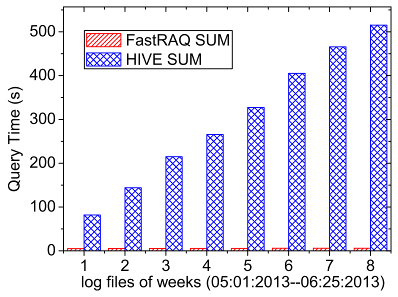


Fig. 9. Performance comparisons for sum queries with eight weeks log files.

6.2 Performance Evaluation

We analyze log files containing eight days of hourly log files (1.4 billion records, 61.6 GB uncompressed files), and eight weeks of hourly log files (9.8 billion records, 432 GB uncompressed files) respectively. We examine the query performance and corresponding relative errors in the two systems.

6.2.1 *Performance of Range Query*

Figs. 6 and 7 illustrate query time comparisons with count and sum query examples. In the testings of eight days of log files, Hive costs 114.6 s for count queries, but FastRAQ only costs 4.3 s for the same request. FastRAQ achieves 26 times of performance improvement on count queries than Hive. Figs. 8 and 9 further illustrate the phenomenon of queries performance comparisons with eight weeks log files. In the testings of eight weeks of log files, Hive costs 520 s for sum query, while FastRAQ costs 6.2 s for the same request. In other words, FastRAQ achieves 84 times of performance improvement on sum request. It is believe that, when the size of data sets increases, FastRAQ can achieve better performance improvement on range-aggregate queries than Hive.

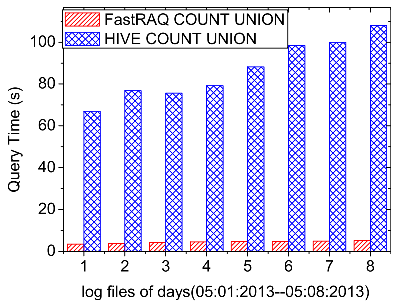


Fig. 10. Performance comparisons for count on union queries with eight days log files.

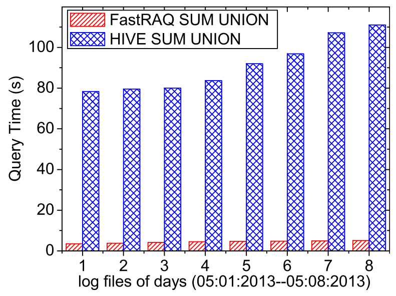


Fig. 11. Performance comparisons for sum on union queries with eight days log files.

In our experiment, we generate about 2,000 partitions and 1,000 buckets in each partition. That is to say, the amount of each data-log file accounts for less than one millionth of the input data on average. So the query time changes slightly for FastRAQ in our daily or weekly stepping tests.

6.2.2 *Performance of Union of Set Query*

Due to the fact that it needs to scan and merge massive duplicated tuples in union of set queries, we primarily focus our testings in union of set range-aggregate queries. The performance comparisons of union query in the two systems are presented in Figs. 10, 11, 12, and 13 using the preceding union queries examples.

Hive predicts if the values of the two index-columns satisfy the union statement in memory. It occupies most of time to fetch tuples from disk files to memory, thus the query time does not change much from single index-column statement to union of two index-columns statements. In FastRAQ, different index-columns of queried ranges can be searched in parallel in the RC-Tree. The overhead of union statements is to merge estimators from different index-columns. The merging overhead is negligible. Thus the query times of the two approaches are nearly the same as shown in Section 6.2.1.

6.3 Relative Errors

Hive obtains exact query result, and its relative error of queried result is 0. As discussed in Algorithm 7, it does not lead to extra errors into the estimate when we merge estimators of different queried dimensions. Thus the estimated relative errors of the union queries in multiple index-columns are the same as the errors in single index-column queries. We discuss the detailed relative errors of the range-aggregate

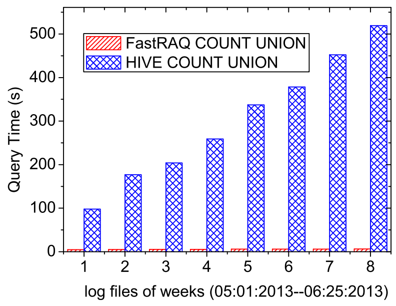


Fig. 12. Performance comparisons for count on union queries with eight weeks log files.

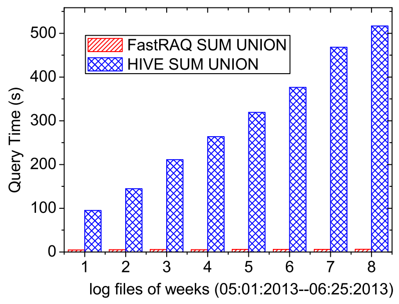


Fig. 13. Performance comparisons for sum on union queries with eight weeks log files.

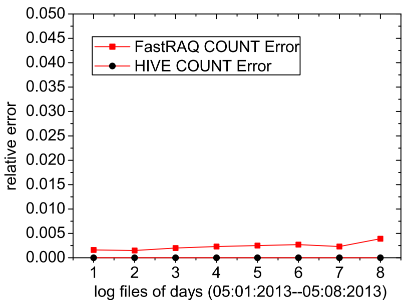


Fig. 14. Relative errors of count queries with eight days log files.

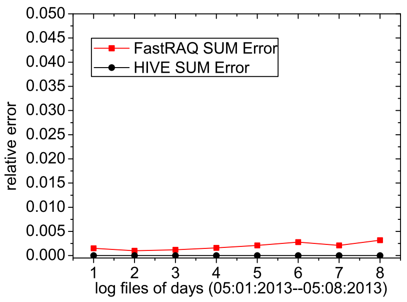


Fig. 15. Relative errors of sum queries with eight days log files.

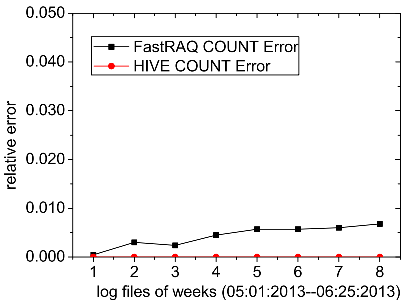


Fig. 16. Relative errors of count queries with eight weeks log files.

queries in Section 6.2. Figs. 14, 15, 16, and 17 present the estimated relative errors in the corresponding queries examples. Because when the volume of data sets is small, the estimator can achieve better cardinality estimation in each buckets [16]. Thus FastRAQ achieves more accurate cardinality estimation in small amount of data set environments. When the size of data increases, the relative error of estimator obeys standard normal distribution, and its standard variance (σ) equals to $\frac{1.04}{\sqrt{m}}$ [16]. In our experiment, we set $m = 2^{12}$, and the standard variance of relative error is 0.026, that is to say, the relative error falls into $[-0.026, +0.026]$ with given confidence interval. The experimental results are consistent with the conclusions in Section 5.

Another important factor is the edge-bucket cardinality ratio (h_0), which affects the estimated relative errors. When h_0 is greater than a threshold, the estimators are obtained directly from leaves nodes of a RC-Tree, and it will add more errors into the final estimate. We further analyze the impact of h_0 affections on the estimated relative errors. We design different query examples to make the values of h_0 changing from 0.0001 to 2 percent, and examine the relative errors caused by estimators of the edge-buckets. Figs. 18 and 19 illustrate the impact of h_0 on the estimated relative errors. It comes to the conclusion that when h_0 grows smaller, the errors caused by the estimators of edge-buckets becomes smaller correspondingly. It is clearly that when h_0 approaches to 0.02 percent the errors caused by estimators of edge-buckets are negligible. Thus for those queries whose edge-buckets cardinalities are smaller than a threshold, we can directly use all the estimators from RC-Tree to generate the final approximate answering results.

6.4 Pros and Cons

In this section, we analyze the theoretical overheads of FastRAQ in terms of update cost, query cost, and data volume of the histogram. We first define some parameters for analyses, and the notations are listed in Table 3.

First, we examine the query cost of FastRAQ. According to Algorithm 4, the records can be loaded to the servers with balanced load distribution. The queries operations can be carried out between partitions parallelly. The cost of transmitting a local result of a partition is negligible. It predominates the query cost of FastRAQ to search in the histogram. According to Algorithm 7, it costs $O(\log B)$ time to search a random node in RC-Tree. If the number of buckets B is almost fixed in the histogram, it takes nearly constant time to search a random node in the histogram. Let the constant be C. When the estimators of the edge-buckets are produced by scanning data files, the query cost can be expressed as $O(\frac{N}{P \times B}) + C$. Thus both approaches reduce the volume of data needed to scanned greatly. Of course, when the edge cardinality ratio (h_0) is small enough, we can get the estimators from RC-Tree directly, and the query cost approaches a constant even in big data environments.

Second, we analyze the update cost of FastRAQ, which is represented by $Update_{FastRAQ}$. The updating process

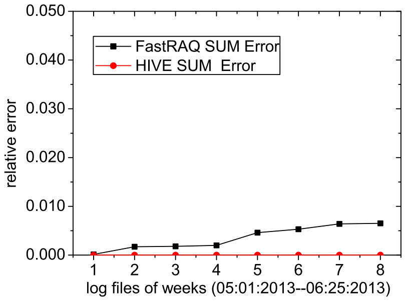


Fig. 17. Relative errors of sum queries with eight weeks log files.

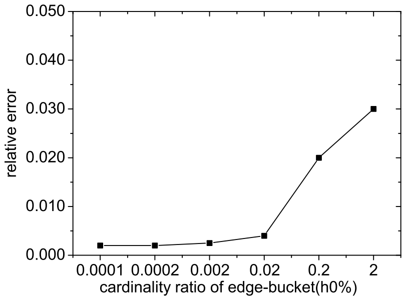


Fig. 18. Relative errors of different edge-bucket cardinality ratio (h_0) with one week log files.

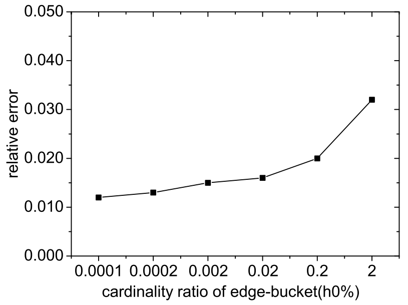


Fig. 19. Relative errors of different edge-bucket cardinality ratio (h_0) with one month log files.

includes delivering a record to a specified partition, and updates the parameters of the histogram in a partition. The delivering process can be done in constant time as discussed in Algorithm 4. When the number of nodes is almost fixed in the RC-Tree, the updating cost of RC-Tree approaches a constant. The update process can be parallelized among partitions, and the distributed throughput of FastRAQ can be expressed as $Update_{FastRAQ} = P \times Avg_{RC-Tree}$, where $Avg_{RC-Tree}$ is the average update cost in each RC-Tree. We have designed a cached hash table for incremental updating process, and it will improve the performance of throughput significantly.

Third, we discuss the storage overhead of FastRAQ. The RC-Tree is built on top of the values of index-columns. The leaf node contains estimator and values of spreads for each bucket. The tuples values of index-columns are stored in the bucket data file. The size of RC-Tree volume is expressed as $Storage_{FastRAQ} = P \times B \times Node_{RC-Tree}$, where $Node_{RC-Tree}$ is the size of leaf node in RC-Tree. We further examine the size of RC-Tree in TB-scale uncompressed data files. The testing results are shown in Tables 4 and 5. Meanwhile we present the volume ratio of RC-Tree and the uncompressed source data. When the size of data files increases, the ratio

TABLE 3
The Notations for the Analysis of Complexity

parameters	contents
n	the number of records
d	the number of index-columns
N	the number of index tuples, and $N = n \times d$
P	the number of partitions
B	the number of bucket for histogram

TABLE 4
Storage Overhead of RC-Tree Index with 1-4 Weeks Log Files

log files of 1-4 weeks	1 W	2 W	3 W	4 W
RC-Trees data volume (GB)	5.9	6.5	6.8	7.1
the volume ratio	0.11	0.06	0.04	0.03

becomes significantly small. It is believed that if the volume of data files is large enough, the storage overhead produced by RC-Tree is negligible.

7 Related Work

The range-aggregate query problem has been studied by Sharathkumar and Gupta [20] and Malensek [21] in computational geometry and *geographic information systems* (GIS). Our work is primary focused on the approximated range-aggregate query for real-time data analysis in OLAP. Ho et al. was the first to present *Prefix-Sum Cube* approach to solving the numeric data cube aggregation [4] problems in OLAP. The essential idea of PC is to pre-compute prefix sums of cells in the data cube, which then can be used to answer range-aggregate queries at run-time. However, the updates to the prefix sums are proportional to the size of the data cube. Liang et al. [6] proposed a dynamic data cube for range-aggregate queries to improve the update cost, and it still costs $O(N^{\frac{d}{3}})$ time for each update, where d is the number of dimensions of the data cube and n is the number of distinct tuples at each dimension. The prefix sum approaches are suitable for the data which is static or rarely updated. For big data environments, new data sets arrive continuously, and the up-to-date information is what the analysts need. The PC and other heuristic pre-computing approaches are not applicable in such applications.

An important approximate answering approach called *Online Aggregation* was proposed to speed range-aggregate queries on larger data sets [7]. OLA has been widely studied in relational databases [8] and the current cloud and streaming systems [9], [10]. Some studies about OLA have also been conducted on Hadoop and MapReduce [10], [11], [12]. The OLA is a class of methods to provide early returns with estimated confidence intervals continuously. As more data is processed, the estimate is progressively refined and the confidence interval is narrowed until the satisfied accuracy is obtained. But OLA can not respond with acceptable accuracy within desired time period, which is significantly important on the analysis of trend for ad-hoc queries.

Our work is related to two approximate answering methods: sampling and histogram. Sampling is an important

TABLE 5
Storage Overhead of RC-Tree Index with 1-4 Months Log Files

log files of 1-4 months	1 M	2 M	3 M	4 M
RC-Trees data volume (GB)	7.2	8.1	8.6	8.9
the volume ratio	0.031	0.017	0.012	0.009

technique for processing of aggregate queries at run time. The sampling for massive data sets includes two types: row-level sampling and block-level sampling [22]. The work in [22] analyzed the impact of block-level sampling on statistic estimation for histogram, and proposed the corresponding estimators with block-level samplings. Haas and König [23] proposed a new sampling scheme, which combines the row-level and page-level samplings in the field of relational DBMS. Data sampling is also well used in the field of distributed and streaming environments [24], [25]. Histogram is another important technique for selectivity estimation. A series of alterative techniques were presented in other articles to provide better selectivity estimation than the original equi-width method. The multi-dimensional histograms were also widely studied by researchers. The problem is more challenging since it was shown that optimal splitting even in two dimensions is NP-hard [26]. The hTree [27] and mHist [28] are the typical works to support multi-dimensional selectivity estimation. While the current works are shown that it is quite expensive to generate a multi-dimensional histogram. FastRAQ combines sampling, histogram and data partition approaches together to generate satisfied estimations in big data environments. All of the above techniques are designed for distributed range-aggregate queries paradigm, and it achieves better performance on both query and update processing in big data environments.

8 Conclusions and Future Work

In this paper, we propose FastRAQ—a new approximate answering approach that acquires accurate estimations quickly for range-aggregate queries in big data environments. FastRAQ has $O(1)$ time complexity for data updates and $O(\frac{N}{P\times B})$ time complexity for ad-hoc range-aggregate queries. If the ratio of edge-bucket cardinality (h_0) is small enough, FastRAQ even has $O(1)$ time complexity for range-aggregate queries.

We believe that FastRAQ provides a good starting point for developing real-time answering methods for big data analysis. There are also some interesting directions for our future work. First, FastRAQ can solve the *1:n* format range-aggregate queries problem, i.e., there is one aggregation column and *n* index columns in a record. We plan to investigate how our solution can be extended to the case of *m:n* format problem, i.e., there are *m* aggregation columns and *n* index columns in a same record. Second, FastRAQ is now running in homogeneous environments. We will further explore how FastRAQ can be applied in heterogeneous context or even as a tool to boost the performance of data analysis in DBaas.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments and suggestions which have helped to improve the quality of this paper. This work was supported by the National Natural Science Foundation of China (61202067, 61271275); the National High Technology Research and Development Program of China (863 Program)(2013AA01A213, 2013AA013204, 2013AA013205, 2012AA012803); the National Grand Fundamental Research 973 Program of China (2014CB340402). S. Wang and G. Wu are the corresponding authors of this paper.

References

[1] P. Mika and G. Tummarello, "Web semantics in the clouds," *IEEE Intell. Syst.*, vol. 23, no. 5, pp. 82–87, Sep./Oct. 2008.

[2] T. Preis, H. S. Moat, and E. H. Stanley, "Quantifying trading behavior in financial markets using Google trends," *Sci. Rep.*, vol. 3, p. 1684, 2013.

[3] H. Choi and H. Varian, "Predicting the present with Google trends," *Econ. Rec.*, vol. 88, no. s1, pp. 2–9, 2012.

[4] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant,, "Range queries in OLAP data cubes," *ACM SIGMOD Rec.*, vol. 26, no. 2, pp. 73–88, 1997.

[5] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin, "Fast data in the era of big data: Twitter's real-time related query suggestion architecture," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1147–1158.

[6] W. Liang, H. Wang, and M. E. Orlowska, "Range queries in dynamic OLAP data cubes," *Data Knowl. Eng.*, vol. 34, no. 1, pp. 21–38, Jul. 2000.

[7] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," *ACM SIGMOD Rec.*, vol. 26, no. 2, 1997, pp. 171–182.

[8] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *ACM SIGMOD Rec.*, vol. 28, no. 2, pp. 287–298, 1999.

[9] E. Zeitler and T. Risch, "Massive scale-out of expensive continuous queries," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1181–1188, 2011.

[10] N. Pansare, V. Borkar, C. Jermaine, and T. Condie, "Online aggregation for large MapReduce jobs," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1135–1145, 2011.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears, "Online aggregation and continuous query support in MapReduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1115–1118.

[12] Y. Shi, X. Meng, F. Wang, and Y. Gan, "You can stop early with cola: Online processing of aggregate queries in the cloud," in *Proc. 21st ACM Int. Conf. Inf. Know. Manage.*, 2012, pp. 1223–1232.

[13] K. Bilal, M. Manzano, S. Khan, E. Calle, K. Li, and A. Zomaya, "On the characterization of the structural robustness of data center networks," *IEEE Trans. Cloud Comput.*, vol. 1, no. 1, pp. 64–77, Jan.–Jun. 2013.

[14] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Integrity for join queries in the cloud," *IEEE Trans. Cloud Comput.*, vol. 1, no. 2, pp. 187–200, Jul.–Dec. 2013.

[15] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proc. 16th Int. Conf. Extending Database Technol.*, 2013, pp. 683–692.

[16] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Int. Conf. Anal. Algorithms*, 2008, pp. 127–146.

[17] [Online]. Available: http://research.neustar.biz/2012/12/17/hll-intersections-2/, 2012.

[18] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive—a petabyte scale data warehouse using Hadoop," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 996–1005.

[19] D. Mituzas. Page view statistics for wikimedia projects. (2013). [Online]. Available: http://dumps.wikimedia.org/other/pagecounts-raw/

[20] R. Sharathkumar and P. Gupta, "Range-aggregate proximity queries," IIIT Hyderabad, Telangana 500032, India, Tech. Rep. IIIT/TR/2007/80, 2007.

[21] M. Malensek, S. Pallickara, and S. Pallickara, "Polygon-based query evaluation over geospatial data using distributed hash tables," in *Proc. IEEE/ACM 6th Int. Conf. Utility Cloud Comput.*, 2013, pp. 219–226.

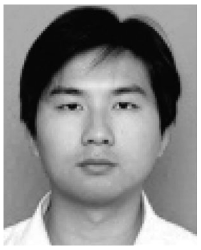
[22] S. Chaudhuri, G. Das, and U. Srivastava, "Effective use of block-level sampling in statistics estimation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 287–298.
[23] P. J. Haas and C. König, "A bi-level bernoulli scheme for database sampling," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, ACM, 2004, pp. 275–286.
[24] S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan, "Distributed online aggregations," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 443–454, Aug. 2009.
[25] E. Cohen, G. Cormode, and N. Duffield, "Structure-aware sampling: Flexible and accurate summarization," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 819–830, 2011.
[26] S. Muthukrishnan, V. Poosala, and T. Suel, "On rectangular partitionings in two dimensions: Algorithms, complexity and applications," in *Proc. 7th Int. Conf. Database Theory*, 1999, pp. 236–256.
[27] M. Muralikrishna and D. J. DeWitt, "Equi-depth multidimensional histograms," *ACM SIGMOD Rec.*, vol. 17, no. 3, 1988, pp. 28–36.
[28] V. Poosala and Y. E. Ioannidis, "Selectivity estimation without the attribute value independence assumption," in *Proc. 23rd Int. Conf. Very Large Data Bases*, 1997, vol. 97, pp. 486–495.

Xiaochun Yun received the BS and PhD degrees from the Harbin Institute of Technology, China, in 1993 and 1998, respectively. He is currently a professor at the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include network and information security. He has published more than 200 papers in refereed journals and conference proceedings. He won the Best Paper Award at ICNP 2012. Dr. Yun is a member of the IEEE Computer Society.

Guangjun Wu received the master's and doctor's degrees in computer science from the Harbin Institute of Technology, China, in 2006 and 2010, respectively. He is currently a senior engineer at the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include big data analysis, distributed storage, and information security. Dr. Wu is a member of the IEEE Computer Society.

Guangyan Zhang received the bachelor's and master's degrees in computer science from Jilin University and the doctor's degree in computer science and technology from Tsinghua University in 2000, 2003, and 2008, respectively. He is currently an associate professor at the Department of Computer Science and Technology, Tsinghua University. His current research interests include big data computing, network storage, and distributed systems.

Keqin Li received the BS degree in computer science from Tsinghua University in 1985, and the PhD degree in computer science from the University of Houston in 1990. He is currently a SUNY distinguished professor of computer science in the State University of New York at New Paltz. His research interests include design and analysis of algorithms, parallel and distributed computing, and computer networking. He has nearly 300 research publications.



Shupeng Wang received the master's and doctor's degrees from the Harbin Institute of Technology, China, in 2004 and 2007, respectively. He is currently an associate research fellow at the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include big data management and analytics, network storage.