ISSN 2395-695X (Print)
ISSN 2395-695X (Online)

International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Issue 4, April 2016

A FAST RAQ: A FAST APPROACH TO RANGE-AGGREGATE QUERIESE IN HETEROGENOUS ENVIRONMENT

P.SATHIYA1 , S.RAMA MOORTHY2

(Anna Univ Affiliated) Master of Engineering, Department of Computer Science and Engineering1,
(Anna Univ Affiliated) Associate Professor, Department of Computer Science and Engineering2
Mohamed Sathak Engineering College, Kilakarai, Ramanathapuram dist., TamilNadu..
1sivasathya0794@gmail.com
2ramamoorthis@gmail.com

Abstract-Big data is a broad term for data sets so large or complex that traditional data processing applications which are inadequate. There are several challenges include analysis, capture data curation, search, sharing, storage, transfer, visualization, and information privacy. Range aggregate queries are defined as that to apply a certain aggregate functions on all tuples within given query ranges. It is a challenging problem to quickly obtain range-aggregate query in the Big Data environments. To overcome these challenges, we develop an approach called FASTRAQ which divides the big data into independent partitions with balanced partitioning algorithms and generate a local estimation sketch for each partition. When a query request arrives, FASTRAQ obtains the result directly by the use of local estimation in all partitions. FASTRAQ has O (m) time complexity for data updates and O (N/ (P*B)) time complexity for range aggregate queries. The experimental results demonstrate the 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

Big data is a broad term for data sets so large or complex that traditional data processing applications are inadequate. Challenges include analysis, capture, data curation, search, sharing, storage, transfer, visualization, and information privacy. The term often refers simply to the use of predictive analytics or other certain advanced methods to extract value from data, and seldom to a particular size of data set. Accuracy in big data may lead to more confident decision making. And better decisions can mean greater operational efficiency, cost reduction and reduced risk.

Analysis of data sets can find new correlations, to "spot business trends, prevent diseases, and combat crime and so on". Scientists, business executives, practitioners of media and advertising and governments alike regularly meet difficulties with large data sets in areas including Internet search, finance and business informatics. Scientists encounter limitations in e-Science work, including meteorology, genomics, connectomics, complex physics simulations, and biological and environmental research.

Data sets grow in size in part because they are increasingly being gathered by cheap and numerous information-sensing mobile devices, aerial (remote sensing), software logs, cameras, microphones, radio-frequency identification (RFID) readers, and wireless sensor networks. The world's technological per-capita capacity to store information has roughly doubled every 40 months since the 1980s; as of 2012, every day 2.5 exabytes (2.5×10^{18}) of data were created; The challenge for large enterprises is determining who should own big data initiatives that straddle the entire organization.

Big data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time. Big data "size" is a constantly moving target, as of 2012 ranging from a few dozen terabytes to many petabytes of data. Big data is a set of techniques and technologies that require new forms of integration to uncover large hidden values from large datasets that are diverse, complex, and of a massive scale.

Analysis of data is a process of inspecting, cleaning, transforming, and modeling data with the goal of discovering useful information, suggesting conclusions, and supporting decision-making. Data analysis has multiple facets and approaches, encompassing diverse techniques under a variety of names, in different business, science, and social science domains.

The partition problem is the task of deciding whether a given multiset S of positive integers can be partitioned into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2. Although the partition problem is NP-complete, there is a pseudo-polynomial time dynamic programming solution, and there are heuristics that solve the problem in many instances, either optimally or approximately. For this reason, it has been called "The Easiest Hard Problem". There is an optimization version of the partition problem, which is to partition the multiset S into two subsets S_1, S_2 such that the difference between the sum of elements in S_1 and the sum of elements in S_2 is minimized. The optimization version is NP-hard.

Histograms are a concise and flexible way to construct summary structures for large data sets. They have attracted a lot of attention in database research due to their utility in many areas, including query optimization, and approximate query answering. They are also a basic tool for data visualization and analysis.

A histogram is a display of statistical information that uses rectangles to show the frequency of data items in successive numerical intervals of equal size. In the most common form of histogram, the independent variable is plotted along the horizontal axis and the dependent variable is plotted along the vertical axis. The data appears as colored or shaded rectangles of variable area.

Range searching and its variants have been studied extensively in the computational geometry and database communities because of their many important applications. Range-aggregate queries, such as range-COUNT, SUM and MAX, are some of the most commonly used versions of range searching in database applications. Since many such applications involve massive amounts of data stored in external memory, it is important to consider external memory (or I/O-efficient) structures for fundamental range-searching problems. In this paper, we develop an external memory data structure for answering orthogonal range-COUNT, SUM and MAX queries. Note that from these we automatically get some other aggregates like AVE and MIN.

Good histograms partition data sets into \smooth" buckets with close-to-uniform internal tuples density. In other words, the frequency variance of the tuples enclosed by such buckets is minimized, leading to accurate selectivity estimations for range queries. Unfortunately, current multidimensional histogram techniques do not always manage to produce close-to-uniform partitions of the data sets, as we discuss next. Later it reports a thorough experimental evaluation of these techniques that complements the discussion in this section.

A partition of a multidimensional data domain results in a set of disjoint rectangular buckets that cover all the points in the domain and assigns to each bucket some aggregated information, usually the number of tuples enclosed. The choice of rectangular buckets is justified by two main reasons: First, rectangular buckets make it easy and efficient to intersect each bucket and a given range query to estimate selectivity. Second, rectangular buckets can be represented concisely, which allows a large number of buckets to be stored using the given budget constraints

In this paper, we present an approach called FASTRAQ, a fast approach to the range aggregate queries in the big data environments. This approach first divides big data into independent partitions based on the balanced partition algorithm, and also generates a local estimation for those partitions. When an range aggregate query arrives FASTRAQ obtains the results directly by summarizing local estimates.

A histogram is a display of statistical information that uses rectangles to show the frequency of data items in successive numerical intervals of equal size. In the most common form of histogram, the independent variable is plotted along the horizontal axis and the dependent variable is plotted along the vertical axis. The data appears as colored or shaded rectangles of variable area.

Range searching and its variants have been studied extensively in the computational geometry and database communities because of their many important applications. Range-aggregate queries, such as range-COUNT, SUM and MAX, are some of the most commonly used versions of range searching in database applications. Since many such applications involve massive amounts of data stored in external memory, it is important to consider external memory (or I/O-efficient) structures for fundamental range-searching problems. In this paper, we develop an external memory data structure for answering orthogonal range-COUNT, SUM and MAX queries. Note that from these we automatically get some other aggregates like AVE and MIN.

Good histograms partition data sets into \smooth" buckets with close-to-uniform internal tuples density. In other words, the frequency variance of the tuples enclosed by such buckets is minimized, leading to accurate selectivity estimations for range queries. Unfortunately, current multidimensional histogram techniques do not always manage to produce close-to-uniform partitions of the data sets, as we discuss next. Later it reports a thorough experimental evaluation of these techniques that complements the discussion in this section.

A partition of a multidimensional data domain results in a set of disjoint rectangular buckets that cover all the points in the domain and assigns to each bucket some aggregated information, usually the number of tuples enclosed. The choice of rectangular buckets is justified by two main reasons: First, rectangular buckets make it easy and efficient

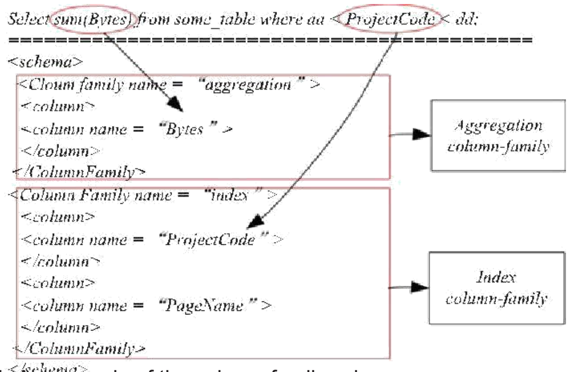

Fig. 1. An example of the column-family schema.

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 strati-fied 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 num-ber of servers changes.

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

FastRAQ has $O(1)$ time complexity for data updates and $O(P \times B)$ 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 buck-ets in a histogram. Furthermore, it produces negligible vol-ume of index data in big data environments.

We implement the FastRAQ approach on the Linux plat-form, and evaluate its performance with about 10 billions data records. Experimental results demonstrate that Fas-tRAQ 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 envi-ronments, 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 alpha-betic. One example of the range-aggregate problem is shown as follows:

Select exp(AggColumn), other ColName
where $l_{i1} < ColName_i < l_{i2}$ opr
$l_{j1} < ColName_j < l_{j2}$ opr
. . . ;

In the above query, exp is an aggregate function such as SUM or COUNT; AggColumn is the dimension of the aggre-gate operation; $l_{i1} < ColName_i < l_{i2}$ and $l_{j1} < ColName_j < l_{j2}$ are the dimensions of ranges queries; opr is a logical oper-ator including AND and OR logical operations. In the fol-lowing 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 par-tition 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 for-mulated 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 que-ried ranges, and $Sample_i$ is the sample for values of aggre-gation-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-fam-ily, 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

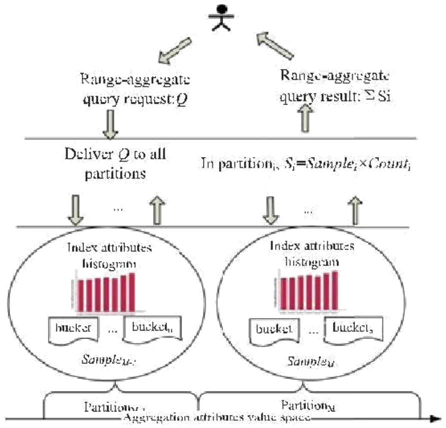

Fig. 2. The FastRAQ framework.

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 li1<ColNamei<li2 opr lj1<ColNamej<lj2.
Output: S;
S: range-aggregate query result.

Deliver the request Q to all partitions;
for each partitioni in partitions do
Compute the cardinality estimator of range li1 < ColNamei < li2 from the local histogram, and let CEi be the estimator of the ith dimensions;
Compute the cardinality estimator of range lj1 < ColNamej < lj2 from the local histogram, and let CEj be the estimator of the jth dimensions;
Merge the estimators CEi and CEj by the logical operator Opr, and compute the merged cardinality estimator CEmerged;
Counti _hðCEmergedÞ;
==_h is a function of cardinality estimation.
Compute the sample for AggColumn, and let Samplei be the sample;
SUMi Counti _ Samplei;
==SUMi is a local range-aggregate query result;
end for
Set the approximate answering of FastRAQ as S. Let
S P M i¼1 SUMi, where M is the number of partitions;
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 popula-tion, and selecting sample in each group to improve the rep-resentativeness 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:

PartitioningðRÞ ¼ ðg; pÞ ¼ ðVe; random½1; Vr&Þ; (1)

where the number of a partition p in a group g, is a random number in ½1; Vr&, and Ve 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: Ve;
Ve: the group Identifier (GID).
k logN;
if ðk ¼¼ 0Þ then
Ve < 0; 0; 0 > ;
Set the interval length of group Ve as [0,1];
return Ve;
else
if ðN _ 2k ¼¼ 0Þ then
Ve < k; 0; 0>;
Set the interval length of group Ve as ½2k ; 2k þ 1&;
return Ve;
else
l log N _ 2k ;
if ðl ¼¼ 0 k N _ 2k _ 2l ¼¼ 0Þ then
Ve < k; l; 0>;
Set the interval length of group Ve as ½2k þ 2l ; 2k þ 2l þ 1&;
return Ve;
else
m log N _ 2k _ 2l ;
Set the interval length of group Ve as ½2k þ 2l þ 2m ; 2k þ 2l þ 2mþ1 _ 1&;þ2
return Ve.
end if
end if
end if

TABLE 1
The Maximum Number of Groups in Different Value Spaces

numeric value space	½1; 2 _ 1& (10)	½1; 2 _ 1& (20)	½1; 2 _ 1& (30)
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 ¼6 0, l ¼6 0, m ¼6 0, the length of the group is ½2 þ 2 þ 2 ; 2 þ 2 þ 2 _ 1&. Other processes of cal-culations 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, 30 itwill produce 8;190 groups at most in the value space ½1; 2 _ 1&, and it is acceptable in many applications. Of course, one can increase the number of logarithm func-tions to reduce the sample error in each group, but it will pro-duce a greater number of groups.

To make data balanced on each server, the partition algorithm subdivides each group into a number of parti-tions according to the current data distributions and sends each partition onto one server. Let Vr represent the maxi-mum number of partitions in each group. The value of Vr is related to the current data distributions and the number of available servers at the same time. We design Algo-rithm 3 to compute the value of Vr for the current system. The key idea of Algorithm 3 is to calculate an average ratio of records b0 for all groups, and then set the value of Vr according to b0 and the current number of records in each group.

Algorithm 3. Numbering(G, dr)

Input: G;
G ¼ f < GIDi; nri >; 1 _ i _ M};
dr: the maximum number of partitions for a group; GIDi: the group identifier of group gi;
nri: the number of records in gi;
M: the number of groups.
Output: VP ;
VP : the partition vectors set, and
VP fVpjj1 _ j _ Mg.
1: Compute an average ratio of record for all groups,
i.e., b0 i ¼1 nri=M;
V 2:rMax number P of servers dr _, and V rMin 1;
3: for all ðgi 2 GÞ do
4: if ðgi:nri < b0Þ then
5: Vpi < gi:GID; VrMin >;
6: else
7: Vpi < gi:GID; MINf nri/b0 ; VrMaxg > ;
8: end if
9: VP VP þ Vpi;
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 nri/b0 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 Vr locates in the interval [VrMin, VrMax], where the VrMax and VrMin 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 ¼ fVpi : < Ve; Vr > j1 _ i _ Mg 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 ¼ 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-fami-lies by the defined schema;
2: Compute the GID with its value from aggregation-column by algorithm 2;
3: Get the partition vector Vpi from VP with the GID, and let Vpi ¼ < GID; Vr >;
4: Set target partition identifier,
PID < GID; random½1; Vpi:Vr& >;
5: Build the sample in partition PID, such as:
counterPID counterPID þ 1;
==counterPID is the number of record;
sumPID sumPID þ N;
//N is value of aggregation attribute from R;
SamplePID sumk;l;m;r=counterPID;
6: RID HashðPID; counterPIDÞ;
//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 avail-able severs changes, it just needs to modify the Vr in corre-sponding partition vector Vp, and the newly incoming records will be adaptively mapped into a partition in [1,Vr] randomly.

4 Range Cardinality Estimation

4.1 Clustering Based Histogram

We measure the data distributions by clustering values of all our histogram. A feature vector of clustering is expressed as ftag; vectorg, 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 fea-ture vectors from learned data set, it will produce vectors set. Let it be f < tagi; vectori > j0 < i < Ng. 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 ClusterIDs into the same bucket. If some tag occurring frequency is significantly different from others, its ClassID is different after the K-Means clus-tering, 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.

Scan the learning data source F , and generate the fre-quency features set f< tagi; vectori > j0 < i < Ng, where tag is the attributes value, vector is the fre-quency occurring on each dimension;
Cluster the features set f< tagi ; vectori > j0 < i <N g by K-Means clustering method and produce K clusters fClusterij1 _ i _ Kg;
Assign a unique ClusterID to each cluster, and scan the K clusters to generate key-value pairs list f < tagq ; ClusterID > j1 _ ClusterID _ K g;
Sort the key-value pairs list by tag in alphabetical order, and the sorted sequence is S ¼ fSi : < tagi; clusterID > j1 _ i _ Ng;
  for all Si in S do
    if ðCureent ClusterID ¼¼ Si:ClusterIDÞ then
      i þ þ; continue;
    else
      Add Si:tagi into P ;
      CureentClusterID Si:clusterID;
      i þ þ;
    end if
end for
  Add MIN VALUE, MAX VALUE into P ;
return P .

Algorithm 5 produces buckets boundary P for the histo-gram, and P ¼ fpij0 _ i _ ng, where pi is the value of tag from the feature vector. The values spreads for buckets in the histogram are ½p0; p1Þ; ½p1; p2Þ; . . . ; ½pn_1; pnÞ respectively, and p0 ¼ _1, pn ¼ þ1. In Algorithm 5, we let MIN VALUE be _1, and MAX VALUE be þ1.

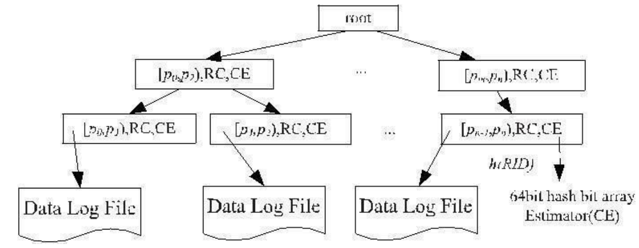

Fig. 3. A typical RC-Tree structure.

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 Algo-rithm 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. HyperLogLog-Plus uses 64 bits hash function instead of 32 bits in Hyper-LogLog 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 ½pi; pjÞ. 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 ½pi; piþ1Þ, the estimator CE of each bucket, and the bucket data file pointer. The leaf node only keeps these statistical informa-tion, 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 for-mal bucket data files. The incremental updating process will greatly improve the throughput of RC-Tree in big data envi-ronments. 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 ðli1; li2Þ cardinality, we first locate the first leaf node using random searching method. Let the first node be Nodei, such that li1 _ Nodei:pi, where ½pi; piþ1Þ 2 Nodei. Then we find other nodes sequentially from Nodei, until the last node is found. Let the last node be Nodej, and li2 _ Nodej:pjþ1, where ½pj; pjþ1Þ 2 Nodej. All the CEs from Nodei to Nodej are merged into a single CE with binary format, and the cardinality of range ½pi; pjþ1Þ is obtained from the merged CE. If the two edge nodes Nodei, Nodej do not fully cover the queried range (li1, li2), that is to say, li1 < pi and/or li2 > pjþ1. 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 Nodei_1 and/or Nodejþ1, 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.

```
for all columns in R do
  Parse value of index-columns into key-value pairs, in format of < IndexValue; RID >;
  Search in the buckets spreads P , and get the target bucket ½pi; pjÞ, such that IndexValue 2 ½pi; pjÞ
  Search in hash table and get the target node NodeH , which include bucket range ½pi; pjÞ;
    NodeH :RCNodeH :RC þ 1;
    Set RID into NodeH :CE;
    Write IndexValue into a temporary bucket data file;
  if ðhash table node number > thresholdÞ then
    for all nodes in hash table do
      Flush the nodes of hash table into T ;
      Append the temporary data files into the for-mal bucket data files.
    end for
  end if
end for
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 CEmerge 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 ðb_aÞ2 12 exclusive-inclusive principle for the logical AND operation, which is jAj T jBj ¼ jAj þ jBj _ jAj S jBj. When the size of jjAj T jBjj=MINjAj; jBj is large enough, the exclusive-inclu-sive principle can produce a satisfied accuracy estimate. There are also some discussions about how T to get a better cardinality estimation when the size of jjAj jBjj=MINjAj; jBj is small [17].

Algorithm 7. Range cardinality query algorithm

Input: (Q, T , h0);
  Q : select distinct count(*) where li1 < ColName < li2; T : the RC-Tree;
  h0: the edge range cardinality ratio.
Output: R;
  R: the range cardinality queried result.

```
According to the queried range ðli1; li2Þ, locate the first node by ColName in RC-Tree T randomly, and let the searched node be Nodei, where li1 < pi and ½pi; piþ1Þ 2 Nodei;
m i;
while ðli2 > pmþ1Þ do
  Merge Nodem. CE into cardinality estimator CEmerge;
  m++;
end while
if ð _hðNodei_1:CEÞ / h_ðCEmergeÞ _ h0Þ then
  Merge Nodei_1.CE into cardinality estimator CEmerge;
else
  Scan bucket data file of Nodei_1 to compute the exact cardinality CEi_1;
  Merge CEi_1 into cardinality estimator CEmerge;
end if
if ð hðNodejþ1:CEÞ / h_ðCEmergeÞ - h0Þ then
  Merge Nodejþ1. CE into cardinality estimator CEmerge;
else
  Scan bucket data file of Nodejþ1 to compute the exact cardinality CEjþ1;
  Merge CEjþ1 into cardinality estimator CEmerge;
end if
R _hðCEmergeÞ;
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:

$$\frac{|variable_{true} - variable_{est}|}{variable_{true}}; \qquad (2)$$

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

$$\frac{|variable_{true} - variable_{est}|}{variable_{est}}: \quad (3)$$

D is used as a notation to represent relative error of a given variable. Let Y be the exact range-aggregate result, and $\hat{Y}$ be estimated variable of Y . Their relative errors are DY and D$\hat{Y}$ respectively. Let S be the local range-aggregate

tition. We present Theorem 1 to discuss DS in each partition.

Theorem 1. D$\hat{}$ is an unbiased estimation of D in big data

S S environments.

Proof: According to Algorithm 3, the range-aggregate query

S

$$\hat{S} ¼ Count \; \bar{Sample}; \quad (4)$$

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

partition. If the

range-aggregate result S is expressed as S ¼ ^{n¼1} x^{j}.

j

estimators of two edge-buckets are produced by scan-ning bucket data files, they do not lead to extra errors of

¼1^{P}n

X

S Avgn j¼1 j S n 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

DS expressed next:

b

to Eq. (4), the expectation of can be Count _: (5)

Suppose the buckets of histogram are independent of each other, then Count is an unbiased estimation of n in big

Count

data environments [16], that is to say, n ¼ 1, thus

EðDSÞ=0. ut

We $\hat{}$ expressed as follows:

b

and it is s 2 ðDSampleÞ þ s 2 ðDCountÞ; (6)

where s ð b Þ is variance of relative error of sample for values of aggregation-column in a partition, and

2

s ðDCount Þ is variance of relative error for cardinality esti-mation in a histogram. We suppose that DSample obeys a uniform distribution, and it can be expressed as Uða; bÞ, where a and b are the minimum and maximum values of the distribution. The variance of uniform distribution is

. We omit the minus relative error in the succeeding discussions. According to Algorithm 2, a and b can be com-puted in each group within stratified sampling model, and

S

spaces are listed in Table 2.

The variance of estimated cardinality has been discussed in the work of [16], and the sðDCountÞ asymptotically

TABLE 2
The Standard Variance in Different Numeric Space

numeric value space	½1; 2^{10} _ 1&	½1; 2^{20} _ 1&	½1; 2^{30} _ 1&
maximum relative error(b)	0.07	0.07	0.07
the standard variance(ð $\hat{}$ Þ)	0.02	0.02	0.02

equals to 1:04 / $\sqrt{m}$, where m is the number of register bit array. If we set ¼ 2^{12}, sðD $\hat{}$ Þ ¼ 0 026. m S :

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

We use Theorem 2 to discuss relationship between DS and DY .

Theorem 2. DS is an unbiased estimation of DY , that is EðDY Þ ¼ EðDSÞ.

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

$$DY ¼ \frac{\sum_{i¼1}^{M} DS_i \_ S_i}{\sum_{i¼1}^{M} S_i}; \quad (7)$$

where DSi 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 fDSij1 _ i _ Mg are independent and identically distributed (i.i.d.) variables. The fDSig can be considered as a list of observations for variable DS. Let $\sum_{i¼1}^{M}$ S^{i} be a constant C,

and the expectation of DY can be written as follows:

$$EðDY Þ ¼ E \left(\frac{\sum^{M} DS_i \_ S_i}{P} \right) ¼ EðDSÞ: \quad (8)$$

Thus EðDSÞ is an unbiased estimation of EðDY Þ. ut

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

$$Y ¼ \sum^{M} \sum^{n} x_{ij}; \quad (9)$$

where M is the number of partitions, Xij is the value of aggregation-column in the queried ranges of the ith parti-tion. Let Si be the local range-aggregate query result in the ith partition, thus Y is

$$Y ¼ \sum^{M} S_i: \quad (10)$$

In Eq. (10), Y is the sum of i.i.d. variables fDSig. Accord-ing to Central Limit Theorem, if M is large enough, Y obeys a normal distribution, that is Y _ Nðm; s^{2} Þ, where m and s is the

expectation and variance of Si.

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:

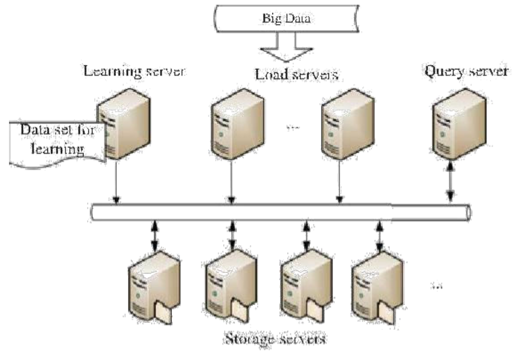

Fig. 4. System configuration used in experiments.

Then Y locates in $[z - m, z + m]$ with probability p, where $z \frac{1}{4}$ ... and z_p is p-quantile in the standard normal distribution. The final $100p$ percent confidence interval of range-aggregate query result is $[z - m, z + m]$.

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 per-formance 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 experi-ments, 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 serv-ers. The learning server fetches a certain amount of data set

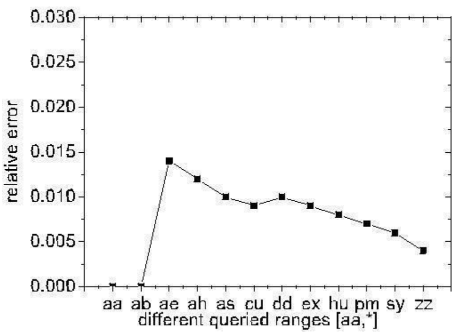

Fig. 5. The relative errors in different queried ranges.

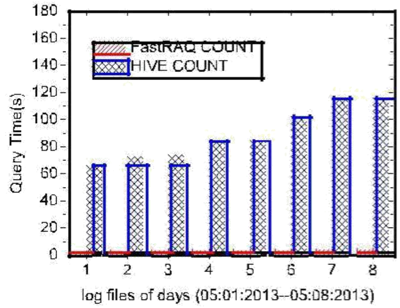

Fig. 6. Performance comparisons for count queries with eight days log files.

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 frame-work of FastRAQ is shown in Fig. 4.

In the experiments, we analyze the pagecount traffic sta-tistics files of Wikipedia [19]. We construct a table contain-ing four columns. We set projectcode and pagename columns as index columns, bytes field as aggregation-col-umn. 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 ∈ ('aa', '*')", where '*' is a random variable string changed from 'aa' to 'zz'. The relative errors in different que-ried examples are shown in Fig. 5. We just present the values of '*' on the X axis. When the '*' equals to 'aa' and 'ab', the rel-ative 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 rela-tive 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 ∈ ('aa', 'dd');
Sum query: Select sum(bytes) from pagecounts where projectcode ∈ ('aa', 'dd').

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

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

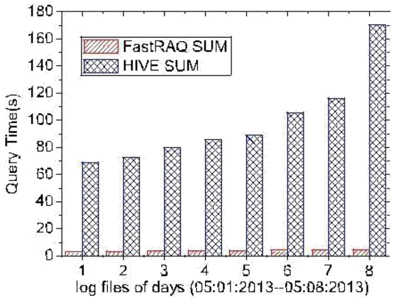

Fig. 7. Performance comparisons for sum queries with eight days log files.

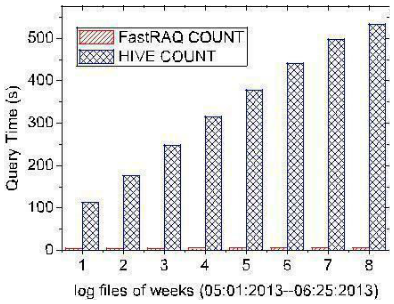

Fig. 8. Performance comparisons for count queries with eight weeks log files.

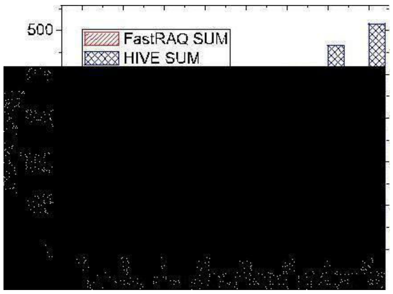

Fig. 9. Performance comparisons for sum queries with eight weeks log files.

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 uncom-pressed files) respectively. We examine the query perfor-mance 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.

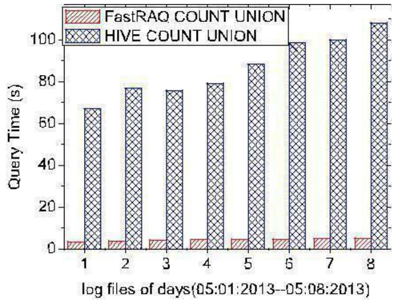

Fig. 10. Performance comparisons for count on union queries with eight days log files.

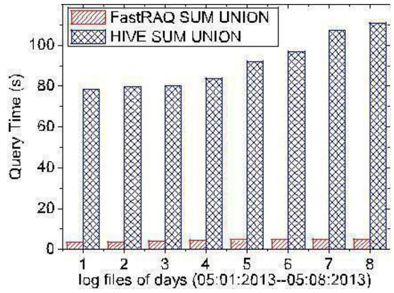

Fig. 11. Performance comparisons for sum on union queries with eight days log files.

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 mil-lionth of the input data on average. So the query time changes slightly for FastRAQ in our daily or weekly step-ping 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 sys-tems are presented in Figs. 10, 11, 12, and 13 using the pre-ceding union queries examples.

Hive predicts if the values of the two index-columns sat-isfy 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 Fas-tRAQ, 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-col-umns. 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 que-ried 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

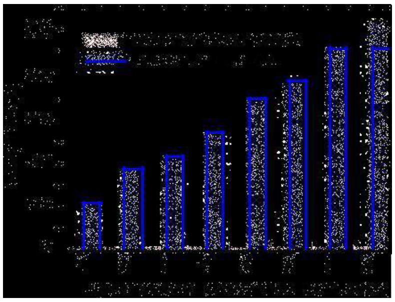

Fig. 12. Performance comparisons for count on union queries with eight weeks log files.

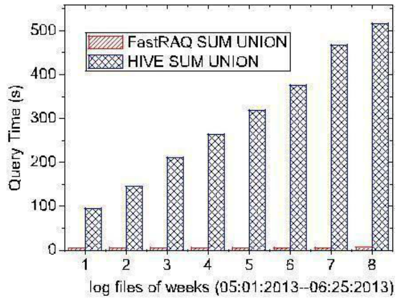

Fig. 13. Performance comparisons for sum on union queries with eight weeks log files.

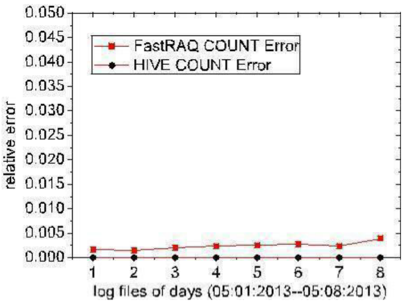

Fig. 14. Relative errors of count queries with eight days log files.

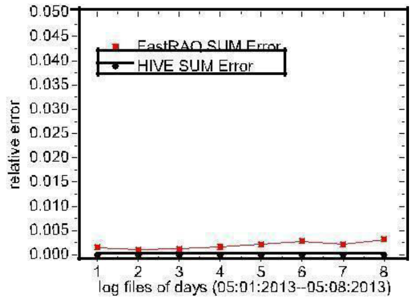

Fig. 15. Relative errors of sum queries with eight days log files.

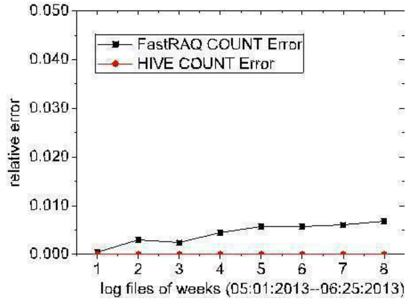

Fig. 16. Relative errors of count queries with eight weeks log files.

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 esti-mation 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 (s) 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 confi-dence 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 Fas-tRAQ in terms of update cost, query cost, and data volume of the histogram. We first define some parameters for analy-ses, 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 pre-dominates the query cost of FastRAQ to search in the histo-gram. 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 N scanning data files, the query cost can be expressed as $O(P - B) + C$. 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 $UpdateFastRAQ$. The updating process

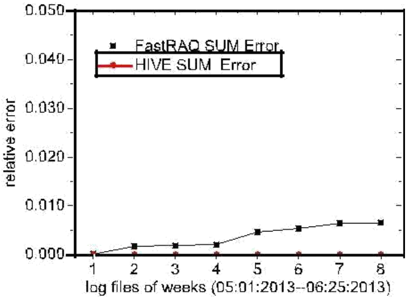

Fig. 17. Relative errors of sum queries with eight weeks log files.

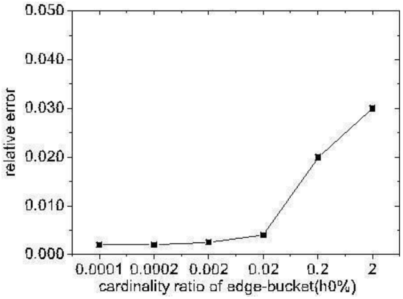

Fig. 18. Relative errors of different edge-bucket cardinality ratio (ho) with one week log files.

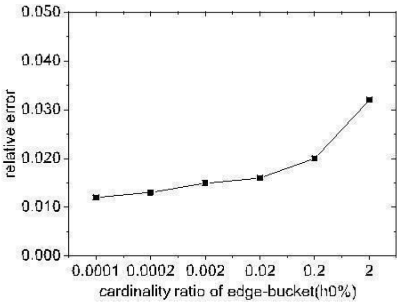

Fig. 19. Relative errors of different edge-bucket cardinality ratio (ho) with one month log files.

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 UpdateFastRAQ ¼ P _ AvgRC _Tree, where AvgRC_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

StorageFastRAQ ¼ P _ B _ NodeRC_Tree,

where NodeRC_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

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 ¼ n _ d
P	the number of partitions
B	the number of bucket for histogram

size of data files increases, the ratio

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

In the Existing System, An Approach called FASTRAQ, which is a new approximate answering approach in big data environments for accurate estimations quickly for range-aggregate queries. The approach first divides big data into independent partitions with a balanced partitioning algorithm, and then generates a local estimation for those partitions. When a query acquires, the FASTRAQ obtains result directly by summarizing local estimates from all partitions. Here the balanced partitions algorithm is used which works with a straight sampling model and divides all data into different groups with regard to their attribute the value of interest, and further separates each group into multiple partitions according to the current data distributions and number of available servers. The local estimation is a new type of multi-dimension histogram that is build for learned data distributions. Our experimental results demonstrate the 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.

DISADVANTAGES:

- As more data is processed, the estimate is progressively refined and the confidence interval is narrowed until the satisfied accuracy is obtained.
- Only works on the homogeneous Environment.

The range-aggregate query problem has been studied by Sharathkumar and Gupta [20] and Malensek [21] in compu-tational 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 stream-ing 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 accu-racy 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 meth-ods: sampling and histogram. Sampling is an important

TABLE
Storage OverLoad With RC Tree

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 esti-mation for histogram, and proposed the corresponding esti-mators with block-level samplings. Haas and K€onig€ [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 distrib-uted 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 challeng-ing 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. Fas-tRAQ 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 proc-essing in big data environments.

8 CONCLUSIONS AND FUTURE WORK

In this paper, An Approach called FASTRAQ, which is a new approximate answering approach in big data environments for accurate estimations quickly for range-aggregate queries. The approach first divides big data into independent partitions with a balanced partitioning algorithm, and then generates a local estimation for those partitions. When a query acquires, the FASTRAQ obtains result directly by summarizing local estimates from all partitions. Here the balanced partitions algorithm is used which works with a straight sampling model and divides all data into different groups with regard to their attribute the value of interest, and further separates each group into multiple partitions according to the current data distributions and number of available servers. The local estimation is a new type of multi-dimension histogram that is build for learned data distributions. Our experimental results demonstrate the 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.

In our Future work, we propose a new approach called FASTRAQ which works on the heterogeneous big data environment. That follows the answering approach in big data environments for accurate estimations quickly for range-aggregate queries. This proposed work, divides big data into independent partitions with a balanced partitioning algorithm, and then generates a local estimation for those partitions. Our experimental can be implemented in the real time environment for the effective results

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

ISSN 2395-695X (Print)
ISSN 2395-695X (Online)

International Journal of Advanced Research in Biology Engineering Science and Technology (IJARBEST)
Vol. 2, Issue 4, April 2016

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, E. 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/hllintersections-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. TABLE 5 Storage Overhead of RC-Tree Index with 1-4 Months Log Files log files of 1-4 months 1M 2M 3M 4M RC-Trees data volume (GB) 7.2 8.1 8.6 8.9 the volume ratio 0.031 0.017 0.012 0.009 YUN ET AL.: FASTRAQ: A FAST APPROACH TO RANGE-AGGREGATE QUERIES IN BIG DATA ENVIRONMENTS 217

[22] S. Chaudhuri, G. Das, and U. Srivastava, ―Effective use of blocklevel sampling in statistics estimation,‖ in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2004, pp. 287–298.

[23] P. J. Haas and C. Konig, ―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, ―Distributedonline 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.