Distributed Online Aggregations

Sai Wu [#1], Shouxu Jiang [§2], Beng Chin Ooi [#3], Kian-Lee Tan [#4]
[#]*School of Computing, National University of Singapore, Singapore, 117590*
[1,3,4]{wusai, ooibc, tankl}@comp.nus.edu.sg
[§]*School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China 150001*
[2]jsx@hit.edu.cn

ABSTRACT

In many decision making applications, users typically issue aggregate queries. To evaluate these computationally expensive queries, *online aggregation* has been developed to provide approximate answers (with their respective confidence intervals) quickly, and to continuously refine the answers. In this paper, we extend the online aggregation technique to a distributed context where sites are maintained in a DHT (Distributed Hash Table) network. Our Distributed Online Aggregation (DoA) scheme iteratively and progressively produces approximate aggregate answers as follows: in each iteration, a small set of random samples are retrieved from the data sites and distributed to the processing sites; at each processing site, a local aggregate is computed based on the allocated samples; at a coordinator site, these local aggregates are combined into a global aggregate. DoA adaptively grows the number of processing nodes as the sample size increases. To further reduce the sampling overhead, the samples are retained as a precomputed synopsis over the network to be used for processing future queries. We also study how these synopsis can be maintained incrementally. We have conducted extensive experiments on PlanetLab. The results show that our DoA scheme reduces the initial waiting time significantly and provides high quality approximate answers with running confidence intervals progressively.

1. INTRODUCTION

Today's enterprise business applications such as the supply chain management (SCM) handle large amount of data that are distributed over many companies. Each organization (presumably) has its own enterprise resource planning (ERP) or database system managing its own data. To facilitate decision making, data from these sources have to be accessed, combined and summarized.

One traditional approach is to consolidate the data from the various data sources into a database warehouse to be managed centrally by a powerful (and expensive) server. However, such an approach is not cost-effective and has limited scalability. More importantly, it cannot provide timely information adequately for critical decision making. More recently, some distributed systems have been proposed to support large scale data intensive applications, such as Google's MapReduce [12] and Microsoft's Dryad [19]. These systems exploit parallelism from low-cost workstation clusters to achieve comparable performance to high-end performance server. Compared to the centralized systems, the distributed ones offer better scalability and reliability at an affordable cost.

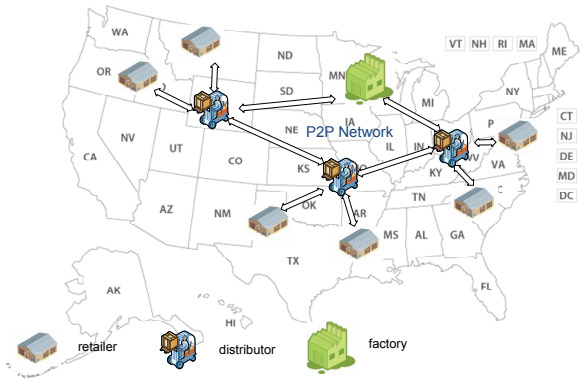


Figure 1: Corporate Network

In this paper, we adopt the distributed approach by organizing the participating sites (data sources or processing nodes) in a Peer-to-Peer (P2P) network to facilitate parallelism. Figure 1 shows a typical corporate network, where sites join the P2P network to form a logical network overlay. Each site is responsible for its local data and participates in query processing. Compared to other infrastructures, P2P network is a promising alternative for corporate networks because sites are autonomous and loosely connected, and the system is scalable. Among the various types of P2P networks, the structured ones are the most feasible for business applications [30]. Besides their search efficiency, the participating companies collaborate with common goals – cost reduction and profitability. In this paper, we adopt Chord [27] as the overlay in view of its simplicity and popularity.

Now, in a P2P-based corporate network, it is not uncommon for the users (decision makers) to issue aggregate queries that provide summarized statistics for daily or long term business activities. For example, by collecting the sale's report from its branches, WalMart can plan its orders for the forthcoming week and arrange for the necessary transportation more efficiently. To answer an aggregate query, a large proportion of the databases needs to be accessed. Compared to non-aggregate queries, aggregate queries are expensive and require long processing time. While parallel and/or distributed processing techniques can be employed to speed up the processing of such queries, the users may still find the long waiting time (before any answers are returned) unacceptable.

In [16], Hellerstein et. al. argued that aggregate queries are typ-

VLDB '09, August 24-28, 2009, Lyon, France

ically used to get a "rough picture" from a large amount of data. As such, instead of producing a precise answer, an "approximately correct" answer suffices. This prompted Hellerstein et. al. to design *online aggregation* techniques to evaluate aggregation query progressively as follows: as soon as a sufficient amount of data is examined, an approximate answer and its corresponding running confidence intervals can be presented to the user; as more data are processed, the answer and the confidence intervals are refined. In this way, an user receives immediate feedback on his/her aggregate query. Moreover, the user can terminate the evaluation prematurely (and hence saving computation overhead) if the approximate answer suffices for his/her decision making or he/she can observe the progression of the approximate/refined answer into the precise answer when all the data have been processed.

In this paper, we extend the online aggregation technique to a distributed context where sites are maintained in a DHT (Distributed Hash Table) network. Our Distributed Online Aggregation (DoA) scheme iteratively and progressively produces approximate aggregate answers as follows: in each iteration, a small set of random samples are retrieved from data sites and distributed to processing sites; at each processing site, a local aggregate is computed based on the allocated samples; at a coordinator site, these local aggregates are combined into a global (approximate) aggregate. We present how the running confidence intervals for the global aggregates are obtained from the local aggregates/intervals. DoA adaptively grows the number of processing sites as the sample size increases. This is achieved using LH (Linear Hashing) [23] to adaptively split the samples and balance the load across different processing sites.

To further reduce the sampling overhead, the samples are retained as a precomputed synopsis over the network to be used for processing future queries. To reduce the maintenance cost, the size of the synopsis data is bounded. Consequently, samples from different tables compete for space. We propose an algorithm for optimizing the synopsis data with regard to the query distribution. A ranking algorithm is used to measure the importance of the samples, and samples that are less beneficial are discarded from the synopsis. We also study how these synopsis can be maintained incrementally.

We have implemented DoA and conducted extensive experiments on PlanetLab. The results show that our DoA scheme reduces the initial waiting time significantly and provides high quality approximate answers with running confidence intervals progressively.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of related work. In Section 3, we describe the system architecture and data flow. We present our sampling technique for the P2P systems in Section 4. In Section 5, we propose our basic query processing scheme. And in section 6, we discuss how to maintain the samples in the synopsis for processing future queries. We report results of an evaluation of the proposed schemes in Section 7. Finally, Section 8 concludes the paper.

2. RELATED WORK

2.1 Approximate Query Processing

In real systems, such as decision support systems (DSS), exact answers to queries incur long response time, and is not always required. To provide early feedback and reduce processing cost, approximate query processing is proposed to process aggregate queries. There are two types of approximate query processing: online aggregation [15, 16, 29] and precomputed synopsis [4, 25]. Online aggregation retrieves samples at query time and provides a gradually refined answer under the user's control. Once satisfied, the user can stop the processing immediately. On the contrary, the precomputed synopsis scheme constructs and stores the synopsis prior to query time. And the stored synopsis can be applied to process incoming queries.

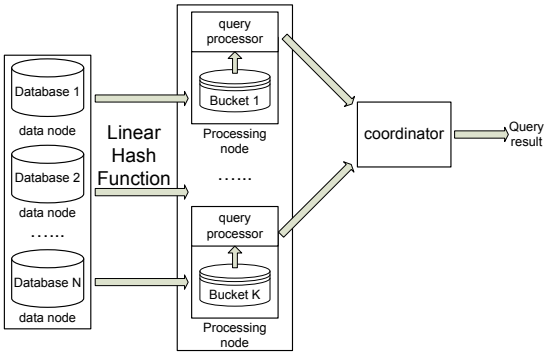


Figure 2: Data Flow of the System

In our scheme, we combine the two types of approximate query processing to address the data analytical problem in the distributed OLAP systems. When a query cannot be processed by the synopsis, we retrieve the samples on-the-fly; otherwise, the query will be directly answered by the synopsis. Previous work on distributed OLAP systems [5, 6] focuses on how to generate a distributed query plan and does not support online aggregations.

2.2 Sampling Techniques

Sampling techniques have been well studied by the database community [13]. When precise result is not necessary or too costly to compute, representative samples are selected to provide an approximate estimation. Recently, the sampling approach has been applied to distributed systems. Bash et al. [10] and Henzinger et al. [17] address the uniform sampling problem in sensor networks and Internet environments respectively.

Sampling is extremely useful in P2P networks, where global statistics, such as the average degree of peers and the total number of peers, are impossible to calculate precisely due to the high overhead. Most existing work is based on unstructured P2P network. The basic idea is to apply random walks [14] to sample the peers in the network uniformly. However, as peers may have different sizes of data and various degrees of connectivity, random walks cannot guarantee unbiased result. Different schemes [9, 11] have been proposed to address the problem of generating unbiased result in the unstructured P2P networks. Based on the sampling approach, Arai et al. [7, 8] proposed their approximate query processing strategy for aggregate queries in unstructured P2P networks. The query is processed via the sampled tuples from the peers' databases.

Our scheme focuses on structured P2P networks, as they provide better search efficiency and are more feasible for business applications. Since the routing index exists, sampling in structured P2P networks are more manageable than in unstructured ones. However, as databases are maintained by each peer individually and the global distribution is unknown, it is challenging to retrieve unbiased samples as well.

3. DISTRIBUTED ONLINE AGGREGATION

3.1 System Overview

Figure 2 shows the architecture and data flow of our DoA system. The data/processing nodes are connected using a DHT overlay. The data nodes retrieve random samples continuously from the local

databases and apply a linear hash function to map the samples to some specific processing nodes. The processing nodes process the queries in parallel and report the results to the query coordinator. The coordinator further combines the answers from the processing nodes to produce a global approximate answer that is presented to the user. The coordinator continues to refine its aggregate (and confidence interval) as the local counterparts are refined. Once the user is satisfied with the result, the processing can be terminated. As an example, to compute the average sales of Walmart, each branch may send its samples to the processing nodes which calculate the average sales (together with a confidence interval) for the allocated sampled data. The local averages (and confidence interval) are then sent to the coordinator, which derives an aggregated summary that is returned to the user. In our system, the data nodes also act as the processing nodes. They cooperate with each other to speed up the query processing.

To improve the performance of our DoA system, we introduce several optimizations. First, DoA adaptively tunes the number of processing nodes based on the size of the samples. In the initial stage of processing, the number of samples are small, and so a few processing nodes are sufficient to handle the workload; as more samples are required, the number of processing nodes is correspondingly increased. To facilitate the varying number of processing nodes, we adopt a linear hash function, as it can dynamically increase its bucket number when more samples are required. We map one bucket to one processing node and thus, more buckets indicate more processing nodes get involved. Hence, the query performance can be guaranteed.

Second, after the samples are disseminated to the buckets, we will keep the buckets as a precomputed synopsis in the network. In other words, the processing node will share part of its storage to maintain the synopsis. When the local databases are updated, we also reflect such updates in the synopsis (through an incremental maintenance strategy that refines the synopsis to ensure its randomness). Note that we assume that updates to the local databases are in batches rather than on-the-fly. Hence, the synopsis is stable most of the time. We also study how these synopsis can be maintained incrementally. To reduce the overhead of maintenance, the number of buckets and the bucket size are bounded.

The samples in the buckets can be exploited to answer future queries. Given a new query, we will first check the samples in the synopsis. If the synopsis can be used to answer the query directly, we do not need to access the local databases. Otherwise, the processing nodes will retrieve samples from the data nodes until the user terminates the processing or the query is processed completely. In this way, we reduce the cost of mapping process by maintaining the synopsis continuously.

The implementation details of the system are presented in the following sections. Before we delve into the implementation details, we briefly introduce our network overlay structure and the distributed indexing strategy.

3.2 Relational Index in DHT

The processing nodes and data nodes are connected in a DHT network. Specifically, Chord [27] is used as an example to demonstrate the idea; other DHT overlays can be easily extended to support our scheme. Figure 3 shows a 2^4-Chord, where four nodes join the network with id 1, 4, 8 and 13 respectively. In Chord, each node is responsible for a key space starting from its predecessor's id to its own id. Suppose the id of the peer is x. For routing purpose, the peer keeps the IP addresses of the peers responsible for the key $x + 2^i$ $(0 \leq i < m)$ in a 2^m Chord. Hence, the peer will keep m routing entries in its routing table.

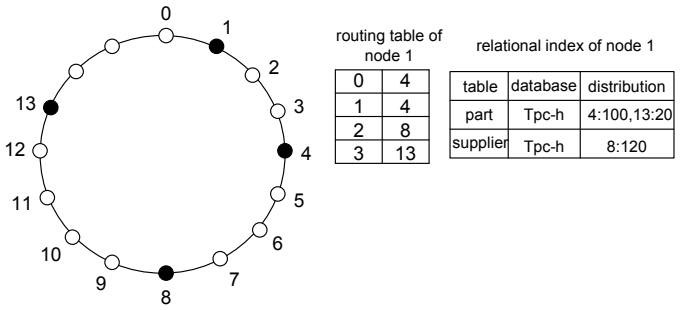


Figure 3: Chord and Relational Index

In our system, the nodes manage their data in local databases. As the nodes may adopt different schemas, we map all data to a global schema. Schema mapping is a complicated problem and is beyond the scope of this paper. In this paper, we assume that the data nodes have already built their mapping relations. By employing the DHT network, the nodes are connected together to provide a global view of the data, and user queries are answered based on the global view. Therefore, given a query, we need to forward the query to all corresponding data nodes. To efficiently discover the data nodes, we build a *relational index* in the DHT network. As shown in Figure 3, there are three attributes in the relational index, where "database" and "table" indicate the data source and are used as the key to publish the index and "distribution" gives a coarse estimation of the data distribution among nodes. For example, for "part" table, the distribution column indicate that node 4 holds 100 tuples of the table, while node 13 holds 20 tuples. We do not maintain the ranges/values of these tuples. To balance the load, we apply the second hash function method proposed in [28] to create r replicas.

To retrieve samples for a specific table, the system looks up the index by combining the table name and database name as the key. Then, it will have an approximate estimation about the data distribution, based on which, a uniform sampling can be performed. Finally, an approximate answer is generated based on the retrieved samples. Error bounds and confidence intervals are provided as well.

The local updates may affect the validity of the index. In this paper, we adopt the lazy update scheme. The query processor retrieves samples based on the data distribution in the index. Therefore, the data node will receive an estimated data distribution from the index via the query processor. After the query is processed, the data node computes the difference between the estimated statistics and its local computed one (normally stored in the local histograms). And if the difference is large enough, an update message is triggered to modify the index entries.

More sophisticated index schemes can be proposed to optimize the query processing. However, the above index scheme is enough for processing the aggregate query approximately. In this paper, we focus on optimizing and processing aggregate queries in parallel.

4. ADAPTIVE RANDOM SAMPLING

One of the key challenges in our design of DoA is to ensure that the samples are randomly picked from among the distributed nodes. We present in this section our approach to realize this.

4.1 Local Sampling

In our system, the node must provide an interface for picking random samples from its database. The order of samples should not

be affected by their values. Hellerstein et al. [16] proposed three methods: heap scans, index scans and sampling via indices. In this paper, we adopt the approach of sampling via indices [24] since it can provide better random samples, even when attributes are not independent. Moreover, sampling via indices can be implemented based on the API provided by major DBMSs. Specifically, Postgres is employed as our underlying DBMS.

4.2 Distributed Sampling

Let T be a global table. Let P nodes contain tuples of T. Let the set of tuples of T at node i be T_i ($1 \leq i \leq P$). Recall that in our DoA framework, the cardinality of T_i can be easily obtained from the relational index in the DHT network. Based on this information, the following naive scheme, called GetSample(Table T, int k), can be used to generate k unbiased sample for T: (a) Each node i provides the number of samples that is proportional to its cardinality, i.e., node i provides ($\frac{|T_i|}{|T|} \cdot k$) samples, where $|T|$ refers to the cardinality of table T. (b) Each node independently samples its data locally. In our work, each local site's samples are provided by the local Postgres. (c) All the samples are collected to produce an unbiased sample for T.

THEOREM 4.1. *Suppose each node provides an unbiased set of samples for its local data. Given the precise data distribution, algorithm GetSample(T,k) generates k unbiased samples for the table T.*

PROOF. A sampling of T is unbiased if each tuple in T has the same probability of being picked, which is $\frac{1}{|T|}$. Under algorithm GetSample, for each node i with $|T_i|$ tuples of T, if node i randomly picks one sample from the T_i, each tuple of T (in T_i) has the probability $\frac{1}{|T_i|}$ of being sampled. Since we essentially pick node i with probability $\frac{|T_i|}{|T|}$, each of T_i's tuples has the same probability $\frac{1}{|T_i|} \times \frac{|T_i|}{|T|} = \frac{1}{|T|}$ of being sampled. □

To get random samples from a table, we have to know the data distribution, specifically, the number of tuples hosted by each data node. As the relational index is built for each table, we only have an estimation about the data distribution. In particular, due to the lazy update strategy, the estimation is only an approximation. This means that the samples may not be as truly unbiased (unless the estimated values reflect the true values). To handle this problem, we propose a self-adaptive sampling scheme.

The proposed self-adaptive sampling scheme, shown in Algorithm 1, generates samples in two phases. In the first phase, the query processor checks the relational index to determine the set of nodes, P, that hold tuples of T (line 1). Recall that the index contains only estimated sizes of the table in these nodes (namely $t_e^1, t_e^2, \cdots, t_e^{|P|}$), from which we can determine the estimated size of T, i.e., t_e (line 2). Then, based on the indexed information and Theorem 4.1, each node retrieves and returns a number of random samples (lines 5-6). At the same time, the actual cardinality of T is computed based on the responses of the nodes (line 7).

In the second phase, we refine the samples that have already been computed in the first phase as follows. If enough samples have been retrieved (i.e., $t > t_e$), we recalculate the necessary number of samples from those already retrieved and return the combined sampling data (lines 9–13). Otherwise, the algorithm issues another message to retrieve more samples from the nodes (lines 14–19).

Algorithm 2 illustrates the actions at a node, when receiving the sampling requirement. If $isAdaptive$ is set to false, the node just returns s_e samples as the estimation requires. If $isAdaptive$ is set to true, the node will recalculate the sample size according to the actual size of the table in the node. The real number of tuples in a table can be obtained by searching the histogram or meta table in the local database. A simple optimization is to retrieve δ more samples than required. And if δ is large enough, the second phase (in Algorithm 1) is not necessary.

Algorithm 1 SelfAdjustSample(Table T, int k)

```
// T : table to be sampled
// k : number of required samples
 1: P = lookupRelationalIndex(T)
 2: t_e = Σ_{i=1}^{|P|} t_e^i
 3: t = 0
 4: for ∀n_i ∈ P do
 5:    s_e=estimated number of samples from node n_i
 6:    S[n_i] = AdaptiveSampling(T, n_i, k, s_e, t_e, true)
 7:    t = t + S[n_i].n
 8: S = ∅
 9: if t ≥ t_e then
10:    for ∀n_i ∈ P do
11:       s_e = S[n_i].n×k / t
12:       S = S ∪ {s_e random samples from S[n_i].s}
13:    return S
14: else
15:    for ∀n_i ∈ P do
16:       s_e = k×S[n_i].n / t − k×S[n_i].n / t_e
17:       reval = AdaptiveSampling(T, n_i, k, s_e, t, false)
18:       S[n_i].s = S[n_i].s ∪ reval.s
19:    return ∪_{∀n_i∈P} S[n_i].s
```

Algorithm 2 AdaptiveSampling(Table T, Node n_i, int k, int s_e, int t, bool $isAdaptive$)

```
//k : total number of required samples
//s_e : estimated number of samples from n_i
//t : estimated total number of tuples in table T
 1: reval.s = ∅, reval.n = s_e
 2: if isAdaptive = false then
 3:    reval.s = {s_e local samples from T_i}
 4: else
 5:    reval.n = actual cardinality of T_i
 6:    reval.s = { reval.n / t × k samples from T_i}
 7: return reval
```

THEOREM 4.2. *Suppose each node provides an unbiased set of samples for its local data. Algorithm 1 gets k unbiased samples for table T in a distributed network.*

PROOF. For node n_i, let t and t_i represent the total number of tuples in T and the size of T in n_i respectively. Let t_e and t_e^i be their estimated values. Unbiased sampling indicates that each tuple should have the same probability $\frac{1}{t}$ to appear in the samples. In Algorithm 1, in phase 1, we first retrieve $\frac{t_i \times k}{t_e}$ samples from node n_i. Thus, each tuple in n_i has the probability $\frac{1}{t_i}$ of being sampled. Then, there are two cases (in phase 2):

1. if $t \geq t_e$, we just pick $\frac{t_i \times k}{t}$ samples (out of the $\frac{t_i \times k}{t_e}$ samples retrieved in phase 1). The sample number of each peer is proportional to its data size. For peer n_i, we have probability $\frac{t_i}{t}$ of picking its samples. Hence, for a tuple of n_i, it has probability $\frac{1}{t} = \frac{t_i}{t} \times \frac{1}{t_i}$ of being sampled.

2. if $t < t_e$, we already have $\frac{t_i \times k}{t_e}$ samples from each node, but we wanted $\frac{t_i \times k}{t}$ samples from each node. So, we need to pick an additional number which is given by the difference. So for node n_i, its samples will also have the probability $\frac{t_i}{t}$ of being selected.

Finally, it is easy to verify that the total number of returned samples equals k. □

5. ONLINE AGGREGATE QUERY PROCESSING

In this section, we discuss how to retrieve samples from the data nodes to process the aggregate queries approximately. Our query processing scheme can be split into three phases. In the first phase, samples are pulled from the data nodes and disseminated to some processing nodes. In the second phase, the processing nodes process the queries in parallel and send the results to a coordinator. Finally, in the third phase, the local results are combined into a global answer.

Algorithm 3 QueryProcess(Query Q)

```
1: Coordinator C=getCordinator(Q)
2: Processing Nodes S = ∅
3: while TRUE do
4:    getSamples(Q,k)
5:    update S
6:    for ∀s ∈ S do
7:       s processes Q based on the samples
8:    C collect results from S
9:    C computes the global answer
10:   if result satisfies user's requirement then
11:      break
```

Algorithm 3 illustrates our general idea. We generate a unique id for each query. Based on the id, we assign the query to a specific node, which acts as the query coordinator. Initially, there are no processing nodes for the query. Then, we retrieve k samples for the query in each iteration. The samples are disseminated to some processing nodes. The number of nodes may change dynamically as the size of the samples increases. After all the samples have been received, the processing nodes process the query in parallel and return the partial results (with corresponding confidence intervals) to the coordinator. The coordinator computes the global results. This process of sampling, local processing, and integrating of local answers into a global results continue until the user is satisfied with the approximate answers (in which case, the query may be terminated prematurely).

In order to adaptively tune the number of processing nodes, we adopt linear hash function to distribute samples across the processing nodes. If a large number of samples need to be processed, more processing nodes will get involved. In the following subsections, we first discuss how to disseminate samples in a DHT network, and then we present our distributed online aggregation technique for a DHT network.

5.1 Namespace

In DHT networks, we need a key to publish and retrieve a sampled tuple. In previous work [18], the database name, table name and values of the key attributes are typically combined together and used as the keys to disseminate tuples. We adopt the same scheme for single relation queries. However, for queries involving (equi-) joins, we need special care to guarantee that tuples from different tables that can be joined together are sent to the same processing node. Therefore, we need to generate the same key for these tuples. To do so, we need to discover the relationships between the tables. We define the joining group concept below for this purpose.

DEFINITION 5.1. *Joining Group*
A joining group is a set of attributes $g = \{T_1.a_1, T_2.a_2, \cdots, T_k.a_k\}$ *such that (a)* $\forall T_i, T_j (1 \leq i, j \leq k)$, $T_i \bowtie_{Ti.ai=Tj.aj} T_j$ *is a valid operation, and (b) for any* $T.a \notin g$, $T_i \in g$, $T \bowtie_{T.a=Tj.aj} T_j$ *is not a valid operation.*

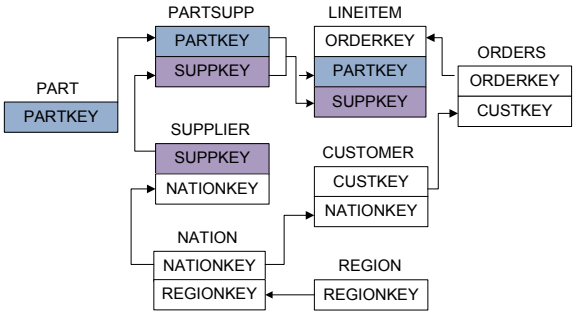


Figure 4: TPC-H Schema

A joining group describes the equi-join relationships between different tables. Before we publish the tuples, we can generate the joining groups based on the global schema. Figure 4 shows the TPC-H [3] schema. Two joining groups, g_1={PART.PARTKEY, PARTSUPP.PARTKEY, LINEITEM.PARTKEY} and g_2={SUPPLIER.SUPPKEY, PARTSUPP.SUPPKEY, LINEITEM.SUPPKEY}, are generated for the attribute "PARTKEY" and "SUPPKEY" respectively.

We define a namespace for each joining group and use it as the key in the DHT network to disseminate the tuples.

DEFINITION 5.2. *Joining Group's Namespace*
For a joining group $g = \{T_1.a_1, T_2.a_2, \cdots, T_k.a_k\}$*, we sort the attributes based on their names. The Namespace of g is defined as the string concatenation of all the attributes' names.*

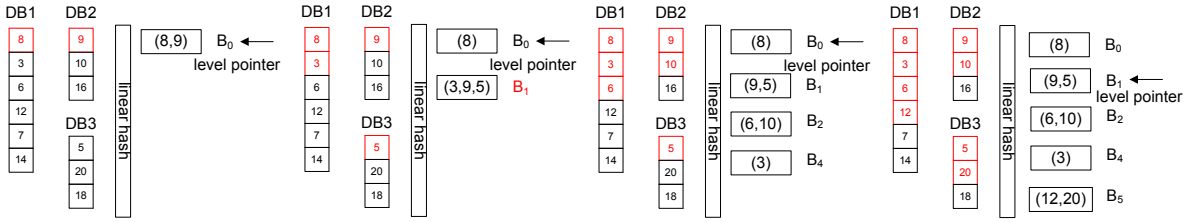
A table T may get involved in multiple joining groups. For example, in Figure 4, *PARTSUPP* participates in two joining groups, g_1 and g_2. To disseminate samples in T, we can randomly select one of the joining groups or use all of the joining groups to create multiple replicas. A better strategy employed in this paper is to select the joining groups based on the query, as the samples are published for answering queries. For the single relational queries, we pick a random joining group and use it to generate the namespace for publishing our samples. For the multiple relational queries, we select the joining groups based on the equi-join conditions. For example, suppose the following query is issued to compute the average retail price of a specific supplier.

```
SELECT AVG(RETAILPRICE)
FROM PART, PARTSUPP, SUPPLIER
WHERE PART.PARTKEY=PARTSUPP.PARTKEY
AND PARTSUPP.SUPPKEY=SUPPLIER.SUPPKEY
GROUP BY SUPPKEY
```

Table *PART* and *SUPPLIER* will use joining groups g_1 and g_2 respectively, while table *PARTSUPP* can select any of the joining groups. Different selections of *PARTSUPP* leads to different query plans.

5.2 Samples Dissemination

To dynamically adapt the number of processing nodes on-the-fly, we employ linear hashing such that each bucket is assigned to a processing node. In this way, as the number of buckets grows dynamically, the number of processing nodes also increases correspondingly. Briefly, in linear hashing, the system has initially m buckets - bucket 0 to bucket $(m-1)$. When a bucket is full (this



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Figure 5: Dissemination of samples.

can occur in *any* bucket), a new bucket m is created. Data in bucket 0 is then rehashed into buckets 0 and m. A new bucket may also be added to accommodate the overflow in the bucket that is full. Subsequently, when the next bucket overflows, bucket 1 will be split between bucket 1 and a newly created bucket $m+1$. This process continues until all the initial m buckets have been split, after which the level of the hash function increases by 1. For detail algorithms of linear hashing, readers can refer to [23]. In this paper, we focus on how to apply linear hashing to disseminate the samples.

When K samples are required from a specific table T, we invoke algorithm 1 to retrieve the samples from the data nodes in parallel. Each node, after receiving the request, starts to publish its samples into the network. It first generates a namespace based on the joining group and query. Then, it employs the linear hash function to disseminate its samples. Figure 5 illustrates the data dissemination process.

In Figure 5(a), we have three databases (DB1, DB2 and DB3) maintained by three data nodes. Suppose 8 samples are required for processing a query, we will retrieve 4, 2 and 2 samples from DB1, DB2 and DB3 respectively. Let the size of the bucket in the linear hash function be 2. And the hash function is defined as *h(k)=k mod* 2^i, where i is the level of the linear hash. Figure 5(a) shows the case when 8 and 9 are published. At first, there is only one bucket and all data are inserted into the bucket. After 8 and 9 are inserted, the bucket is full and cannot accept more samples. When new value 3 is inserted, the bucket splits and increases its level by 1. Now, the hash function becomes *h(k)=k mod* 2. And thus, 8 is kept in B_0 and 9 and 3 are stored in B_1.

When new value 5 is inserted into B_1, B_1 becomes overloaded as shown in Figure 5(b). However, we cannot split it as the level pointer is set to B_0. In Figure 5(c), after samples 6 and 10 are inserted, B_0 becomes overloaded and splits half of its data to the new bucket B_2 based on the hash function *h(k)=k mod* 4. The level of B_0 increases by 1 and the level pointer moves to B_1. As B_1 already satisfies the split condition, it creates the new bucket B_2 and increases its level by 1. And the level pointer is reset to B_0. In the end, Figure 5(d) shows the final status of the buckets.

We give each bucket a unique id (combination of the namespace and the bucket number) and use the id as the key in the DHT network to locate a processing node. The processing node is responsible for maintaining the bucket. It stores the bucket data, bucket level and bucket number. In this way, we map the samples in the same bucket to a specific processing node. All data within the bucket have the same hash value.

We note that when the level (and hash function) is changed, the data nodes are notified. The delay in updating the data nodes is not a serious issue as data sent to the split bucket will be rehashed accordingly.

5.3 Optimizing the Bucket Size

By employing linear hashing to disseminate the data, we can efficiently balance the load between processing nodes. The number of processing nodes is proportional to the number of samples that need to be processed. Each processing node is responsible for a specific bucket of the linear hash function. Therefore, the size of bucket affects the workload of the node. In this section, we try to compute the optimal bucket size. Table 5.3 shows the parameters used. Suppose S samples are required to process the query and the bucket size is set to B. On average, there are $x = \frac{S}{\alpha B}$ buckets, where α is the load factor of the linear hash function and is between 65% and 70%[23]. This also indicates that x processing nodes are needed to process the query in parallel.

For a single query, increasing the number of processing nodes will increase the parallelism and thus reduce the processing time. x should be set as large as possible. However, when multiple queries are processed in the system simultaneously, the queries will compete for the processing nodes. When buckets from different queries are mapped to the same processing node, the node will process the data in the FIFO (First-In-First-Out) order. The new query must wait for the old one to be processed. Its query time thus includes its own processing time and the waiting time. We present a model to determine the optimal bucket size below.

Table 5.3 Parameters

Name	Description	Name	Description
x	number of buckets	S	average number of samples
B	bucket size	α	LH load factor
λ	query arrival rate	N	number of nodes
ν	processing rate	H_x	xth harmonic number

Assume the query arrival rate follows a poisson distribution with mean λ. Let the number of processing nodes in the system be N. Because we use hash function to locate the processing nodes, for each processing node, the queries arrive in a poisson distribution with rate $\frac{\lambda x}{N}$, where x is the average number of buckets per query (poisson distribution is always used to model the stochastic events). Suppose q is a query starting at time t_0 and $\{B_0, B_1, ...B_{x-1}\}$ are its buckets. The query is finished only when its last bucket returns the final result (Here, a query is considered to be completed as long as the error bound and confidence intervals has reached a certain level of accuracy. In our experimental study, this happens when the error bound is within 1% and the confidence level is at 98%. We also ignore the overhead at the coordinator as it is negligible given its task is a simple integration process.) Hence, the processing time of q is determined by the longest job queue of B_i in time t_0. In linear hashing, the bucket's data are changed over time. As data are inserted randomly, we assume the number of data in the bucket follows an exponential distribution with mean αB. Moreover, as the processing time is proportional to the data in the bucket, the processing time of a bucket also follows an exponential distribution with mean $\alpha B\nu$, where ν denotes the unit processing

time. Based on the above assumptions, we compute the expected processing time as:

$$E(T) = \alpha B\nu x \sum_{m=0}^{+\infty} mP\{N(t_0) = m\}P^{x-1}\{N(t_0) < m\}$$

where $P\{N(t_0) = m\}$ denotes the probability of a m-length job queue at time t_0.

This problem is an instance of the famous fork-join queue problem in parallel computing. There is no exact solution, when the allowed number of processing nodes is larger than 2. In this paper, we adopt the result in [26], i.e., if each query is mapped to x buckets, the mean processing time can be approximated by:

$$E(T) \approx (\frac{H_x}{H_2} + \frac{4}{11}(1 - \frac{H_x}{H_2})\rho)\frac{12 - \rho}{8\mu(1 - \rho)} \quad (1)$$

where H_x is the xth harmonic number, μ is the mean service time and $\rho = \frac{\lambda x}{\mu N}$ is the utilization of the processing node. In our scenario, $\mu = \frac{1}{\alpha B\nu}$ and $\rho = \frac{\alpha\lambda x B\nu}{N}$. When x is large enough, $H_x \approx ln(x)$. Suppose the average number of processed samples for a query is S, we have $x = \frac{S}{\alpha B}$. The final solution for $E(T)$ is

$$\begin{aligned} E(T) &\approx (\tfrac{2ln(x)}{3} + \tfrac{4\alpha\lambda x B\nu}{11N}(1 - \tfrac{2ln(x)}{3}))\tfrac{12\alpha B\nu N - (\alpha B\nu)^2\lambda x}{8N - 8\alpha\lambda x B\nu} \\ &\approx (\tfrac{2ln(\frac{S}{\alpha B})}{3} + \tfrac{4\lambda S\nu}{11N} - \tfrac{8\lambda S\nu ln(\frac{S}{\alpha B})}{33})\tfrac{12\alpha B\nu N - \alpha\lambda B S\nu^2}{8N - 8\lambda S\nu} \end{aligned}$$

To simplify the representation, we define three parameters a, b and c as:

$$a = \frac{2ln(\frac{S}{\alpha})}{3} + \frac{4\lambda S\nu}{11N} - \frac{8\lambda S\nu ln(\frac{S}{\alpha})}{33}$$

$$b = \frac{8\lambda S\nu}{33} - \frac{2}{3}$$

$$c = \frac{12\alpha\nu N - \alpha\lambda S\nu^2}{8N - 8\lambda S\nu}$$

And thus, the expected mean time can be approximated by

$$E(T) \approx acB + bcBln(B) \quad (2)$$

If $ac < 0$ and $cb > 0$, by differentiation, we get the B value that minimizes the response time.

$$B = e^{-1-\frac{a}{b}} \quad (3)$$

If $ac > 0 \wedge cb < 0$ or $ac < 0 \wedge cb < 0$, there is no minimal value for $E(T)$. As the bucket size increases, the estimated processing time decreases. Thus, we set the bucket size as large as possible (map the data to one processing node).

Finally, if $ac > 0 \wedge cb > 0$, the bucket size should be as small as possible. The optimal bucket size is $\frac{S}{N}$, which indicates that all processing nodes are involved in the query processing.

The optimal bucket size is determined by a series of parameters, such as query arrival rate and average sample size. In the experiment, we can set these parameters as the predefined values. In the real system, we employ the peer gossip method to estimate the parameters. Each peer has a local estimation for the parameters based on the past query processing. It will periodically exchange the estimation with its neighbors (peers in its routing table). The efficiency and effectiveness of the gossip based method have been verified in [20, 21]. We will not discuss the detail in this paper.

5.4 Query Processing

Once the data dissemination process finishes, we can start the query processing at the corresponding processing nodes. The processing at each node is similar to the original online aggregation [16], except that the queries in our system are processed in multiple nodes. In our case, each processing node will produce a partial result based on its current samples. Then, the results along with some meta-data are sent to a coordinator for producing the final result.

5.4.1 Confidence Computation

If the samples are selected randomly and in an unbiased way, we can provide an error bound and confidence interval. The result v with error bound ϵ and confidence interval c means that v is within $\pm\epsilon$ of the real result v_r with probability approximately c. Or we can say that the real answer v_r lies in the range $[v - \epsilon, v + \epsilon]$ with probability approximately c. In this paper, ϵ and c at each processing node can be derived in the same manner as that in [16].

Consider a typical single relational aggregate query such as

SELECT $op(expression(x_i))$ FROM T

Let μ be the expected aggregate value (average, sum or count) and $\overline{Y}$ be the estimated value. Generally, we obtain a sample set S by selecting k random samples from relation T.

1. $\overline{Y} = \frac{|T|}{k}\sum_{\forall t_i \in S} c(expression(t_i))$, where $|T|$ is the size of table T, $c(x_i) = 1$ if $op = count$ and $c(x_i) = x_i$ if $op = sum$.
2. $\overline{Y} = \frac{1}{k}\sum_{\forall t_i \in S} expression(t_i)$, when $op = avg$.

Suppose σ^2 is the estimated variance of table T. Then by the Central Limit Theorem, $\frac{\sqrt{k}}{\sigma}(\overline{Y} - \mu)$ is distributed approximately as a standardized normal distribution. This assertion also holds if σ is replaced by the estimators $\sigma_1^2 = k^{-1}\sum_{i=1}^{k}(op(expression(x_i)) - \overline{Y})^2$.

Given an error bound ϵ, the confidence can be computed by the following formula:

$$P|\overline{Y} - \mu| \le \epsilon \approx 2\phi\left(\frac{\epsilon\sqrt{n}}{\sigma_1}\right) - 1 \quad (4)$$

For a typical multi-relational aggregate query such as

SELECT $op(expression(x_i, x_j))$ FROM T_1, T_2 WHERE $T_1.a = T_2.b$

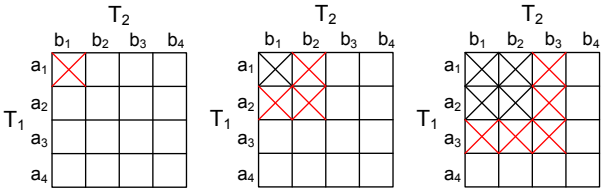
Suppose we have sample set S_1 and S_2 for table T_1 and T_2 respectively.

1. $\overline{Y} = \frac{|T_1||T_2|}{|S_1||S_2|}\sum_{t_1 \in S_1, t_2 \in S_2} c(expression(t_1, t_2))$, where $c(x_i) = 1$ if $op = count$ and $c(x_i) = x_i$ if $op = sum$.
2. $\overline{Y} = \frac{1}{k'}\sum_{t_1 \in S_1, t_2 \in S_2} expression(t_1, t_2)$, where k' is the number of tuples after joining and $op = avg$.

To estimate the confidence and error bound of the multi-relational query, we need to estimate the corresponding variances. Based on the analysis of [15], the variance for sum and $count$ can be computed as:

$$\sigma = \frac{\sigma_{T_1}}{|T_1|} + \frac{\sigma_{T_2}}{|T_2|} \quad (5)$$

where σ_{T_i} is the variance of table T_i. And the variance of avg can be estimated based on the variance of sum and $count$.



(a) iteration 1 (b) iteration 2 (c) iteration 3

Figure 6: Incremental Computation for Join

5.4.2 *Result Collection at Coordinator*

After the processing nodes finish their processing, the partial results will be gathered at the coordinator, where a final result is computed for the query. Suppose k processing nodes get involved and X_i is the node n_i's estimation. Let σ_{X_i} denote the variance of X_i and $Cov(X_i, X_j)$ represent the covariance of X_i and X_j. The variance for the final (approximate) result is estimated as:

$$\sigma = \sum_{i=1}^{k} w_i \sigma_{X_i} + \sum_{i=1}^{k} \sum_{j=1, j\neq i}^{k} Cov(X_i, X_j) \quad (6)$$

We assign a weight w_i to each individual estimation ($\sum_{i=1}^{k} w_i = 1$). Based on the analysis of [22], the covariance can be ignored and the optimal estimation is obtained by setting

$$w_i = \frac{1}{\sigma_{X_i} \sum_{j=1}^{k} \frac{1}{\sigma_{X_i}}} \quad (7)$$

And thus, the final result is computed as:

$$X = \sum_{i=1}^{k} w_i X_i \quad (8)$$

Because the sum of normal distributed variants follows normal distribution as well, we can use the same rule to estimate the confidence and error bound of the result.

In most cases, the workload of the coordinator can be ignored. However, if a large number of groups are generated in the processing nodes, the coordinator may be overloaded by combining the results. Therefore, if the initial coordinator receives too many groups, it will start up some other coordinators to share its load. The groups are partitioned between the coordinators and the processing nodes will forward the results of a group to the corresponding coordinator.

5.4.3 *Incremental Computation*

As described in Algorithm 3, in each iteration, we retrieve k samples from the corresponding data nodes. If these samples can generate a satisfied result for the user, we can stop the processing. Otherwise, additional k samples will be retrieved. This process continues until the user terminates the processing or the query has been processed completely.

In each iteration, the processing node will generate a partial result based on the local samples. The local samples are the samples mapped to the bucket maintained by this node. Now, as more samples are inserted, it would be ideal if the work done earlier can be salvaged. In other words, the computation of the aggregates and the corresponding confidence intervals can be incrementally computed. However, this is only possible for buckets that are not split. For a bucket that has been split during an iteration, we need to scan the sample set completely again to re-produce a new partial result.

Here, we discuss the case when incremental computation can be exploited. Suppose N_i, σ_i, $\bar{x}_i$ and X_i denote the number of samples, variance, average value and the estimated result in the ith iteration, respectively. Let S_i represent the new retrieved samples in the ith iteration. Hence, $N_{i+1} = |S_{i+1}| + N_i$. Based on the following equations, we can incrementally compute the partial result and its error bound. For the average value, we have

$$\bar{x}_{i+1} = \frac{1}{N_{i+1}}(N_i \bar{x}_i + \sum_{x_j \in S_{i+1}} x_j) \quad (9)$$

And the variance can be computed in a similar way.

$$\begin{aligned} \sigma_{i+1} &= \frac{1}{N_{i+1}} \sum_{\forall x_j} x_j^2 - \bar{x}_{i+1}^2 \\ &= \frac{1}{N_{i+1}}(N_i \sigma_i + N_i \bar{x}_i^2 + \sum_{x_j \in S_{i+1}} x_j)) - \bar{x}_{i+1}^2 \end{aligned} \quad (10)$$

Incremental computation can be easily adopted for the single relational query. In the new iteration, we only need to scan the new samples and apply the above formulas to generate the new results. In the case of multi-relational query, not only the new samples but the old samples need to be scanned for producing the result.

Figure 6 shows how the multi-relational query is processed in an iterative way. Suppose we retrieve one block samples from tables T_1 and T_2 in each iteration. In the first iteration, samples in the block a_1 and b_1 are joined together. In the second iteration, samples in the block a_2 need to join with samples in the block b_1 and b_2. And similarly, samples in b_2 need to join with samples in a_1 as well. This strategy is similar to the ripple join processing. The difference is that as we apply the linear hash function, the samples in the same bucket can join with each other with high probability.

In the single relational query, the processing of each iteration is almost constant, as only the newly inserted samples need to be processed, while in the multi-relational case, the processing of each iteration increases linearly. In our system, the partial results are sent to the coordinator for producing the final result. If the partial results are sent after each iteration, the final result will be updated in an irregular rate. To avoid such problem, when performing join, we adopt the block-based update strategy. After the processing node finishes joining samples in one block with another block, it will report its partial result to the coordinator. For example, in Figure 6, instead of waiting for the end of the iteration, the node will report its report after joining block b_2 with a_1 in iteration 2.

6. MAINTAINING SAMPLES AS A PRECOMPUTED SYNOPSIS

Sampling is an expensive operation. In DoA, instead of sampling the data nodes for each query, we maintain the samples as a precomputed synopsis over the network. After the samples are used for processing the queries, we keep them in the processing nodes. In other words, we keep the buckets of the linear hash function after the query processing.

The existence of the synopsis changes the strategy of query processing. We need to modify two lines of Algorithm 3 to reuse samples in the synopsis. First, in line 2, we can directly search for the processing nodes for a specific query. The potential processing nodes for a query is the nodes storing the corresponding buckets for the query. As discussed before, we apply the joining group's namespace to discover such nodes. A challenging problem is how to find all the candidate nodes. Fortunately, based on the protocol of linear hash function, we can efficiently retrieve all the potential processing nodes.

Initially, the query request is forwarded to the node responsible for the first bucket of the linear hash function. The node maintains the basic information about the bucket, such as the bucket level. Suppose its bucket level is L, the total number of buckets is between 2^{L-1} to 2^L. Then, this node forwards the query request to the first 2^{L-1} processing nodes. After receiving the request, the node responsible for the bucket x will process the query based on the data in x and if its level is L, it will further forward the query to the node responsible for the bucket $x + 2^{L-1}$. In this way, all the corresponding processing nodes will receive the query and start their processing.

The second modification of Algorithm 3 is the sample retrieval process in line 4. In the early stage of query processing, we apply the samples in the synopsis to process the query. Therefore, there is no network cost for the data nodes in this stage. However, if existing samples cannot generate a satisfied result, more samples will be retrieved from the data nodes on-the-fly. In our scheme, each processing node also "remembers" the samples that have been maintained as synopsis. In this way, any additional requests will always retrieve different tuples.

6.1 Sample Replacement

Each processing node shares part of its local storage for maintaining the synopses. These synopses essentially are the buckets of different tables/namespaces that have been hashed to the processing node. In other words, each processing node maintains synopses for a number of buckets from different namespaces. To reduce the maintenance overheads, the system sets up an upper bound for the shared storage. However, the amount of space allocated for each namespace may be different. Thus, it is necessary to manage this storage carefully for optimal performance, especially when the storage capacity has been reached. When more samples of a namespace are retrieved, we need to "victimize" samples from another namespace to hold these additional samples. Our solution is described below.

For a processing node n_i, suppose there are k buckets $B_1, ..., B_k$ with different namespaces stored at n_i. Let $|B_i|$ represent the size of the bucket (number of samples in the bucket). As mentioned above, based on the system parameters, we have computed an optimal bucket capacity $|B|$. Recall, $|B_i| < |B|$ since each bucket is typically 65% full. Let S be the query set processed by n_i in a time period θ. We rank each bucket B_i as follows:

$$r(B_i) = \frac{f(S, B_i)}{|B_i|}$$

where $f(S, B_i)$ returns the number of queries processed by B_i.

Let $g(B_i)$ denote the bytes required for storing a sample at B_i. When the size of samples exceeds the defined threshold $\mathcal{T}$ (e.g. $\sum_{i=1}^{k} |B_i| g(B_i) > \mathcal{T}$), we need to remove some samples from the synopsis to accept the newly inserted ones. Given the condition:

$$\sum_{i=1}^{k} x_i g(B_i) \leq \mathcal{T}$$

where x_i denotes the size of bucket after adjustment, we want to maximize the benefit of synopsis, which is estimated as

$$Y = \sum_{i=1}^{k} x_i r(B_i)$$

This problem can be solved by a greedy algorithm. Each time, we pick the bucket with highest rank and insert it into the reserved list until the size of the list reaches the storage threshold. Then, for the rest of the buckets, we will remove all the data in the buckets. Note that the meta-data of these buckets are kept as a requirement of linear hash function. The last bucket in the reserved list needs special care. We remove part of its samples from the bucket until the storage requirement is satisfied.

6.2 Synopsis Update

Different from online transaction systems, in the data warehouse system, the data are inserted in a batch manner. After one day's commercial activity, the company will backup its business transactional data into the data warehouse system for analysis. This observation indicates two facts. First, the system is updated periodically in a batch manner. Second, most of the update operations are insertions. These characteristics allow us to propose an efficient update scheme for the synopsis.

To facilitate the update of samples, each peer maintains a sample table recording its samples in the synopsis. The table has three columns, namely "Namespace", "Bucket Size" and "Bucket Level". Namespace is the corresponding joining group's namespace for publishing the samples. Bucket size is the precomputed optimal bucket capacity and bucket level is the current level of the buckets. Bucket level can be estimated by asking the first bucket of the linear hash function. Suppose the returned level is L, then the buckets level must be L or $L - 1$.

Suppose table T has t_0 tuples in the data warehouse and t_1 tuples are inserted during the update process. We will select some random samples from the newly inserted data to replace the old ones. First, the maximal number of samples in the synopsis is estimated as $2^L B$, where B is the optimal bucket size. Then, we retrieve $S = \frac{2^L B t_1}{t_0 + t_1}$ samples from the newly inserted data and disseminate them according to the linear hash protocol. Based on algorithm GetSample, we can compute the number of samples required in each data node. As mentioned before, the data in the synopsis are dynamically adjusted based on the query pattern. Thus, S is an upper bound for the required samples. We need to adjust the number of new samples in each bucket.

For a bucket B_i, it needs to replenish some new samples and remove some old ones. Due to the limitation of storage, the size of bucket after updating is kept to be less than $|B_i|$. Suppose there are x new samples hashed to the bucket. Based on x, there are two cases:

1. If $\frac{|B_i| t_1}{t_0 + t_1} \leq x$, we get enough new samples to update the bucket. Specifically, $\frac{|B_i| t_1}{t_0 + t_1}$ old samples are removed and the same amount of new samples are inserted. In this case, the number of data in the bucket is unchanged, but we do not accept all the new samples.

2. If $\frac{|B_i| t_1}{t_0 + t_1} > x$, there are not enough new samples to update the bucket. In this case, we will shrink the size of the bucket. $|B_i| - \frac{t_0 x}{t_1}$ old samples are removed from the bucket and x new samples are fully accepted. The bucket size is changed to $\frac{t_0 x}{t_1} + x$.

In both cases, we keep the ratio between the old samples and the new samples in the bucket. And hence, the synopsis still maintains a uniform sample set for the data nodes. The correctness of applying the synopsis to process the queries is guaranteed.

When updating the synopsis, we apply the batch insertion. Instead of inserting the new samples one by one, we group them based on the hash value in the data nodes. Then, the samples to the same bucket are sent together. This batch insertion is only valid in the update case for two reasons. First, the number of new samples is small and the local data node can group the samples efficiently

Table 1: Experimental parameters

Parameters	Default Value	Range of Values
error bound	$\pm 1\%$	$\pm 1\% - \pm 5\%$
confidence	95%	$80\% - 98\%$
data size per group	1G	1G - 5G
λ (avg query per second)	10 (Q1)/ 1 (Q2)	10 (Q1)/ 1 (Q2)
shared storage per node	10M	10M

(buffer them in the memory). Second and the most important reason is that the buckets are never split during the update process as we keep the bucket size. Then, we can compute the samples for each bucket beforehand.

7. EXPERIMENT EVALUATION

To evaluate the performance of our DoA mechanism, we deploy our system on PlanetLab [2]. Specifically, we construct a corporate network with 128 nodes on the Chord overlay. The TPC-H toolkit [3] is employed to generate the test data. Two tables, *lineitem* and *orders*, are used in the experiments. We generated a 1G TPC-H dataset which includes 6000K and 1500K tuples from tables *lineitem* and *orders*, respectively. To simulate a multi-corporate scenario, we group the nodes into 6 groups. Each group is given a specific namespace and hosts one dataset. All groups, except the last one, will have 20 data nodes. The data is evenly partitioned among the nodes. Therefore, for the 1G TPC-H dataset, each node in the group will manage about 600K tuples of *lineitem* or 150K tuples of *orders*. Each node provides 10M memory for maintaining the synopsis. We find that in most cases, the sample size is quite small and even a small-size memory can buffer all the samples.

Q1=
SELECT avg($l\_extendedprice \times l\_discount$) AS revenue
FROM lineitem l
WHERE $l\_shipdate < x$+ 1 year AND $l\_shipdate \geq x$ AND
$l\_discount < y + 0.01$ AND $l\_discount \geq y - 0.01$ AND
$l\_quantity < z$

Q2=
SELECT avg($l\_quantity$)
FROM lineitem l, orders o
WHERE $l\_shipdate < x$+ 1 year AND $l\_shipdate \geq x$ AND
$l\_orderkey = o\_orderkey$ AND $o\_orderpriority$ like '%s'

The query is issued to a specific node group. In the experiment, two query templates, shown above, are employed to generate different types of queries. Q1 represents the typical single-relational query and Q2 is used to produce the multi-relational query. Parameters *x, y, z* and *s* are random values in the corresponding attributes' domains. More experiment parameters can be found in Table 1.

For each experiment, 10K aggregate queries are generated and injected into the system. The average query response time is adopted as the major metric.

7.1 Optimal Bucket Size

In DoA, linear hashing is applied to dynamically manage the samples. If more samples are retrieved and maintained in the synopsis, more processing nodes will get involved. Specifically, the bucket size of the linear hash function determines the number of processing nodes. In this paper, we have proposed a method to compute the optimal bucket size based on previous work [26]. In this experiment, we assume queries arrive to the system in a poisson distribution with rate λ. We set $\lambda = 10$ for Q1 and $\lambda = 1$ for Q2. In other words, on average, we issue 10 single-relational queries or 1 multi-relational query to the system per second. The average number of retrieved samples for Q1 and Q2 are approximately 33,380 and 120,000 for each table (with error rate less than 1% and confidence higher than 95%), respectively. Assume the processing node can process 1000 tuples per-second. We can compute that the optimal bucket size is about 4500 and 13000 for Q1 and Q2 respectively.

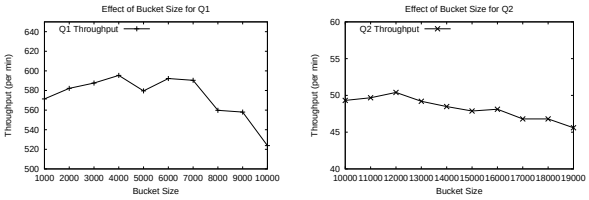


Figure 7: Effect of Bucket Size

Figure 7 shows the effect of bucket size on the throughput (per min). First, we observe that there is a certain optimal bucket size for both types of queries. When the bucket size is small, it means that more processing nodes are involved. This results in more contention for resources (though each node may be doing lesser work). On the other hand, when the bucket size is large, only very few processing nodes are involved. These nodes process more data and hence take a longer time to complete their processing. Second, we observe that our estimated bucket size is almost optimal for both query types. However, as Equation 1 is only an approximation for the fork-join model and we use estimated values for the parameters, our computed bucket size is not the exact optimal one. As shown in Figure 7, Equation 3 can result in a good enough bucket size.

7.2 Effect of Data Size

We evaluate the performance of the system by varying the data size. We increase the data size for each node group from TPC-H 1G dataset to 5G dataset. In that way, each node hosts about 600K to 3000K tuples of *lineitem* or 150K to 750K tuples of *orders*. Here, we compare four processing schemes. *AQP with synopsis* is the approximate query processing strategy with samples maintained in the synopsis. The synopsis is applied to process the future queries. In *AQP without synopsis*, we will retrieve the samples for each query on-the-fly. For both the above schemes, we simulate user satisfaction such that the processing of the queries stop as soon as the error bound of the aggregate is within 1% and a confidence interval of 98%. We also look at the case when the precise answer is returned. This is represented as the *AQP with precise* method. In all the above 3 schemes, answers are returned and refined progressively until the query is terminated. Finally, *CQP* processes the query completely. *CQP* adopts different processing strategies for Q1 and Q2. For Q1, *CQP* forwards the query to each node to compute a partial result. And then the final result is generated based on the partial results in the query coordinator and returned to the user. For Q2, *CQP* generates two sub-queries, one for nodes storing *lineitem* and one for nodes storing *orders*, to retrieve the tuples for join processing. After receiving the sub-query, the node will process the sub-query and return the results to the query coordinator. The coordinator collects all the result tuples and processes the join operation.

Figure 8 and Figure 9 show the performances of different approaches for Q1 and Q2, respectively. Note that the figures are shown in logarithmic scale. As expected, *AQP with precise* is clearly unacceptable because of its long response time. The per-

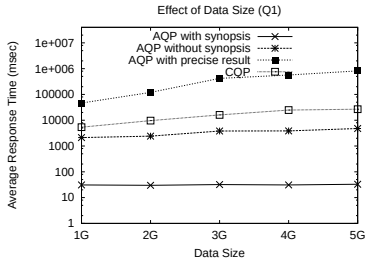


Figure 8: Effect of Data Size (Q1)

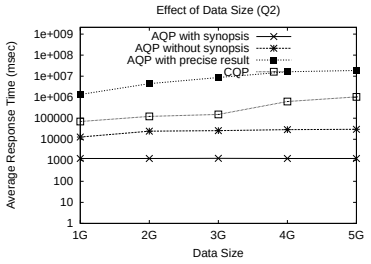


Figure 9: Effect of Data Size (Q2)

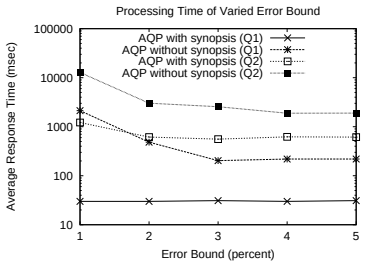


Figure 10: Processing Time of Varied Error Bound

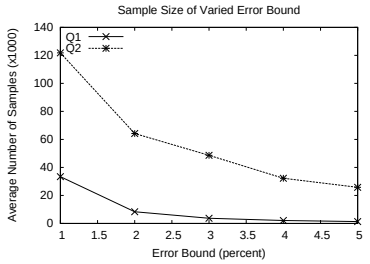


Figure 11: Sample Size of Varied Error Bound

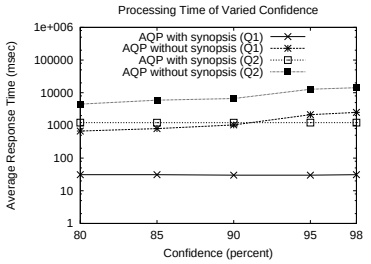


Figure 12: Processing Time of Varied Confidence

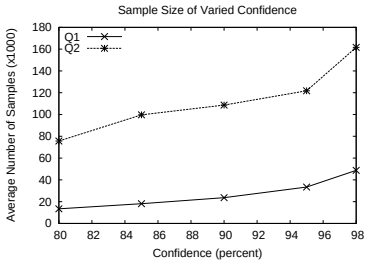


Figure 13: Sample Size of Varied Confidence

formance of *CQP* degrades tremendously, especially for the multi-relational query. The response time of *AQP without synopsis* also increases for the larger dataset, as sampling for large datasets incurs more overheads. On the contrary, by reducing the number of samples retrieved on the fly, *AQP with synopsis* is more scalable for the large datasets.

7.3 Effect of Error Bound

In this experiment, we evaluate the approximate approaches by changing the estimated error bound. For an error bound ϵ and approximate answer v, the precise result is estimated to be in the range of $[v - \epsilon, v + \epsilon]$ with high probability. A smaller ϵ indicates a higher quality result and thus more samples are required. In this paper, we set ϵ to be $v \times \frac{p}{100}$, where p ranges from 1 to 5. As shown in Figure 10, when the approximate answer is prone to the exact answer, the processing time improves exponentially. This can be explained by Figure 11. Figure 11 shows the average number of retrieved samples. To improve the estimated error rate from 2% to 1%, we need much more samples than the ones we retrieve for improving error rate from 3% to 2%. If the user requires an answer with 99.9% precision, *AQP without synopsis* may be even worse than *CQP*.

To reduce the overheads of online sampling, *AQP without synopsis* maintains the used samples in the synopsis. As the samples are random tuples for the whole table, it can be employed to answer the future queries as well. Figure 10 verifies that *AQP with synopsis* has a better response time, even for high accuracy requirement.

7.4 Effect of Confidence c

Confidence is another indicator of how good the approximate result is. It gives a clue of the likelihood that the precise answer falls in our estimated range. To guarantee a high confidence, more tuples are retrieved from databases in the approximate query processing strategies. In this experiment, by changing the estimated confidence, we evaluate the performance of different strategies. The error bound is set to 1% to provide a good enough result. As Figure 12 and Figure 13 show, the number of retrieved samples increases linearly as the confidence improves, which result in a longer processing time. Again, *AQP with synopsis* outperforms *AQP without synopsis* by applying the synopsis to process queries.

7.5 Precision of Estimation

The central limit theorem (CLT) is applied to compute the estimated error bound and confidence. In our approach, we use the variance computed based on the samples to simulate the precise variance. In this experiment, we test how good the approximation is. Suppose the result of *AQP with synopsis* and CQP is v and v' respectively, the real error rate is calculated as $\frac{|v-v'|}{v'}$. Figure 14 shows the difference between the estimated error bound and the real error rate. As we can see, the estimation is quite accurate. The real error rate is strictly bounded by the estimated one. For example, for point (4, 1.2) in Q1, it means that DoA's error bound of 4% turns out to be only 1.2% in the actual answer.

7.6 Effect of Insertion

In this experiment, we test the performance of samples' insertion. We generate 10K to 50K insertions for each node group. In Q1, insertions are evenly distributed over the *lineitem* tables. In Q2, insertion happens with a same probability for each *lineitem* and *orders* table. Figure 15 shows the average number of newly inserted samples. Only a small number of samples need to be updated to reflect the new data. Figure 16 shows the result's error rate after insertion. The estimated error bound is set to 1%. After updating, the approximate result is still bounded by the error bound.

7.7 Load Balancing

In query processing, we generate a namespace for each query and apply DHT protocols to balance the load among the nodes. Figure 17 shows the query load distribution in the network. We use the average number of tuples processed in each node as the metric.

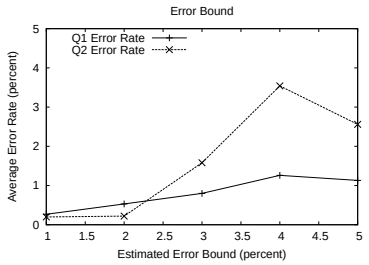


Figure 14: Real Error Rate VS Estimated Error Bound

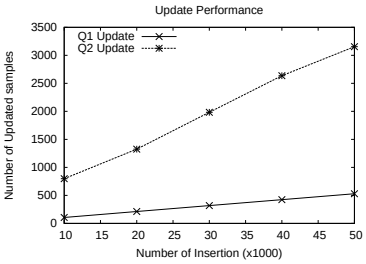


Figure 15: Update Cost

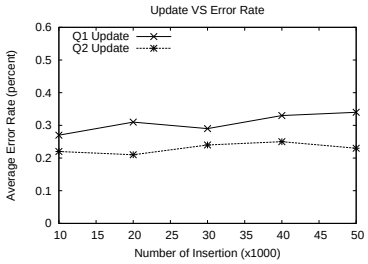


Figure 16: Error Rate VS Updates

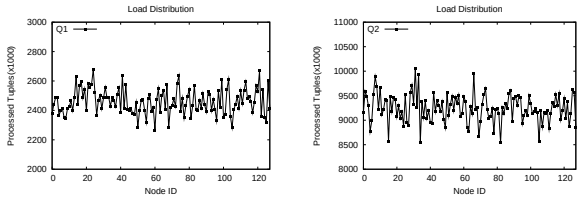


Figure 17: Load Distribution

As analyzed in [27], the ratio of maximal load to minimal load is bounded by $O(\log N)$, where N is the number of nodes. Figure 17 verifies that DHT can effectively balance the load among the nodes.

8. CONCLUSIONS

In this paper, we have extended the online aggregation technique to distributed systems and present a parallel query processing strategy to improve the query performance. To facilitate the parallelism, the nodes are organized in a DHT (Distributed Hash Table) network. The samples are retrieved uniformly from the local databases and disseminated to the processing nodes via linear hashing. The query is processed in parallel based on the retrieved samples. A statistic model is employed to compute the approximate result and estimate the error bound. After the query is processed, its samples are maintained as a precomputed synopsis to be used for future queries. Extensive experiments conducted on PlanetLab showed the effectiveness of the DoA framework.

9. ACKNOWLEDGMENT

This research is in part funded by ASTAR SERC Grant 072 101 0017 of S3 project [1], National Grand Fundamental Research 973 Program of China Grand 2006CB303000 and Key Program of National Natural Science Foundation of China Grant 60533110.

10. REFERENCES

[1] http://www.comp.nus.edu.sg/ s3p2p/.
[2] http://www.planet-lab.org.
[3] http://www.tpc.org/tpch.
[4] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. *SIGMOD Rec.*, 28(2), 1999.
[5] M. O. Akinde, M. H. Böhlen, T. Johnson, L. V. S. Lakshmanan, and D. Srivastava. Efficient olap query processing in distributed data warehouses. *Information Systems*, 28(1-2):111–135, 2003.
[6] J. Albrecht and W. Lehner. On-line analytical processing in distributed data warehouses. In *IDEAS*, 1998.
[7] B. Arai, G. Das, D. Gunopulos, and V. Kalogeraki. Approximating aggregation queries in peer-to-peer networks. In *ICDE*, 2006.
[8] B. Arai, S. Lin, and D. Gunopulos. Efficient data sampling in heterogeneous peer-to-peer networks. In *ICDM*, 2007.
[9] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Distributed uniform sampling in unstructured peer-to-peer networks. In *HICSS*, 2006.
[10] B. A. Bash, J. W. Byers, and J. Considine. Approximately uniform random sampling in sensor networks. In *DMNS*, 2004.
[11] S. Datta and H. Kargupta. Uniform data sampling from a peer-to-peer network. In *ICDCS*, 2007.
[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 2008.
[13] O. F. Random sampling from databases. In *PhD Thesis, University of California*, 1993.
[14] C. Gkantsidis, C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks: algorithms and evaluation. *Perform. Eval.*, 63(3), 2006.
[15] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.
[16] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.
[17] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. On near-uniform url sampling. In *WWW*, 2000.
[18] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*.
[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 41(3), 2007.
[20] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware*, 2004.
[21] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3), 2005.
[22] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A disk-based join with probabilistic guarantees. In *SIGMOD*, 2005.
[23] W. Litwin, M. anne Neimat, and D. A. Schneider. Lh*—a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21, 1996.
[24] F. Olken. Random sampling from databases. 1993.
[25] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *ICDE*, 1992.
[26] R.Nelson and A. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. 37, 1988.
[27] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transaction Network*, 11(1), 2003.
[28] G. Swart. Spreading the load using consistent hashing: A preliminary report. In *ISPDC*, 2004.
[29] K.-L. Tan, C. H. Goh, and B. C. Ooi. Online feedback for nested aggregate queries with multi-threading. In *VLDB*, 1999.
[30] S. Wu, J. Li, B. C. Ooi, and K.-L. Tan. Just-in-time query retrieval over partially indexed data on structured p2p overlays. In *SIGMOD Conference*, 2008.