Just-In-Time Query Retrieval Over Partially Indexed Data on Structured P2P Overlays

Sai Wu[§], Jianzhong Li[†], Beng Chin Ooi[§], Kian-Lee Tan[§]
School of Computing, National University of Singapore, Singapore[§]
Harbin Institute of Technology, Harbin, China[†]
{wusai, ooibc, tankl}@comp.nus.edu.sg
lijzh@hit.edu.cn

ABSTRACT

Structured peer-to-peer (P2P) overlays have been successfully employed in many applications to locate content. However, they have been less effective in handling massive amounts of data because of the high overhead of maintaining indexes. In this paper, we propose PISCES, a Peer-based system that Indexes Selected Content for Efficient Search. Unlike traditional approaches that index all data, PISCES identifies a subset of tuples to index based on some criteria (such as query frequency, update frequency, index cost, etc.). In addition, a coarse-grained range index is built to facilitate the processing of queries that cannot be fully answered by the tuple-level index. More importantly, PISCES can adaptively self-tune to optimize the subset of tuples to be indexed. That is, the (partial) index in PISCES is built in a Just-In-Time (JIT) manner. Beneficial tuples for current users are pulled for indexing while indexed tuples with infrequent access and high maintenance cost are discarded. We also introduce a light-weight monitoring scheme for structured networks to collect the necessary statistics. We have conducted an extensive experimental study on PlanetLab to illustrate the feasibility, practicality and efficiency of PISCES. The results show that PISCES incurs lower maintenance cost and offers better search and query efficiency compared to existing methods.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed databases; C.2.4 [**Distributed Systems**]: Distributed databases

General Terms

Algorithms, Design, Experimentation, Management, Measurement, Performance

Keywords

Peer-to-Peer, Partial Indexing, Just-In-Time, Self-Tuning, Sampling, BATON, CAN

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

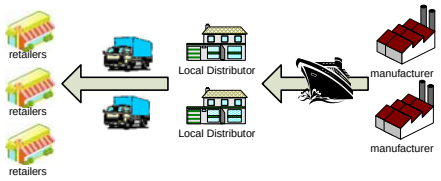


Figure 1: Supply Chain Management

1. INTRODUCTION

Peer-to-Peer (P2P) technologies have been deployed to locate content in many applications. In particular, structured P2P overlays such as Chord[34], CAN[30], Pastry[31], PGrid[6] and BATON[21]) have been developed to guarantee a bound on search performance. However, the "success" stories remain restricted largely to file-sharing based systems.

Consider an enterprise business application such as the supply chain management (SCM) system in Figure 1. This application needs to handle a large amount of data distributed over many companies (peers) participating in the corporate network. Each organization presumably has its own enterprise resource planning (ERP) or database system managing its own data. The traditional SCM cannot effectively handle a large number of participants, and in fact slows down information dissemination by creating a long chain (the bullwhip effect). The implication of this is that the corporation may not have all the necessary information in time for sound decision making.

As a remedy to the bottleneck of large-scale systems, P2P technology offers a promising platform for business applications. To effectively share information, organizations can participate in an SCM P2P network, where each organization has an application server acting as a peer node. In such applications, each node manages its local data and exchanges information in the form of tables and tuples. The data can thus be shared in a large-scale network more efficiently.

We note that existing P2P systems for database applications (e.g., Mercury [10] and PIER [19]) have not been designed to support the above applications. These systems adopt a full indexing strategy where all database tuples are indexed in the network. This is clearly not practical in our case as the amount of information to be shared is massive in volume, and the number of tables and attributes is huge. To index all tuples, the indexing process will consume too much network bandwidth and take hours to complete. Also, as nodes join or leave the network, the corresponding index needs to be reconstructed or removed, which incurs considerable overhead.

As cycle time is a major metric of a supply chain [20], the real-time supply chain, which provides up-to-second information, becomes the next-generation supply chain [12]. In this new type

of supply chains, data are updated in high and unpredictable frequency. For illustration, FedEx receives about 2.5 million packages daily, and Sun Microsystems gets 480% spikes for a product in demand. Hence, the update overhead to keep indexes up-to-date and/or monitor the peers becomes impractical. The full indexing strategy also cannot cope with the case where each peer wants to share huge amounts of data and the network suffers high update and churn rates.

Fortunately, we observe that, in practice, only a small portion of data is actually frequently accessed, and most of the data receive few or no accesses at all.

EXAMPLE 1.1. *As shown in [22], both offline and online sales (from EBay) of Sotheby's (http://www.sothebys.com/) commodities follow a highly skewed distribution in prices. In online sales, most transactions focus on the items with price from $ 200 to $ 400. In such a case, Sotheby's should monitor the sales of items in a specific range more frequently to adjust its storage. As a result, the following query becomes popular:*

SELECT *count(sale)*
FROM *sale*
WHERE *price>x and price<y*

As illustrated in Example 1.1, the focus is about goods in a price range. Retailers should not bother to index information of goods in other price ranges: The best strategy is to index what the user requires.

In this paper, we examine the ambitious goal of designing *practical* peer-based data management systems (PDMS) to support enterprise-quality business processing that involves a large amount of data and peers. In particular, we propose PISCES, a *P*eer-based system that *I*ndexes *S*elected *C*ontent for *E*fficient *S*earch. PISCES selectively picks a subset of tuples based on some criteria, such as query frequency, update frequency and index cost, to build a partial index. This reduces both the cost to insert tuples into the network, and the maintenance overhead. To handle queries that cannot be fully facilitated by the tuple-level index, we employ a coarse-grained range index to direct the queries to candidate nodes. The coarse-grained range index is light-weight and its maintenance cost is low. In addition, PISCES can adaptively tune itself to optimize the subset of tuples to be indexed based on the query access patterns. The tuples are indexed Just-In-Time (JIT): An index is built for a query set whenever it is necessary. We also introduce a light-weight monitoring scheme for structured networks to collect the necessary statistics. We have conducted an extensive experimental study on PlanetLab to illustrate the feasibility, practicality and efficiency of PISCES.

The contributions of the paper are summarized as follows:

1. We propose a partial indexing strategy to selectively index the tuples of databases based on the cost model of structured overlays.
2. To handle the case of missing indexes in the partial indexing strategy and to pull data for constructing new tuple-level indexes, a new kind of index, the approximate range index, is proposed.
3. A light-weight sampling scheme is applied to monitor query distribution in the network.
4. Experiments in PlanetLab illustrate the effectiveness and scalability of our scheme in different network configurations.

The rest of this paper is organized as follows. Section 2 reviews related work and Section 3 gives an overview of our system. In Section 4, we introduce our partial indexing strategy. To collect approximate network statistics, a new monitoring scheme is proposed in Section 5. We test the effectiveness of our system by a series of experiments in Section 6. Section 7 concludes the paper.

2. RELATED WORK

A partial index entails only marginal extra complexity and can help solve a collection of problems that a database management system (DBMS) faces. It was first adopted in PostgreSQL. Stonebraker [35] listed possible applications of the partial index. Seshadri and Swami [33] proposed the scheme of building a partial index based on statistics collected in various granularity . Through analysis of statistics, partial indexes are constructed for frequently accessed data. Experiments show that the partial index is a promising way to improve query efficiency. Our scheme extends the idea of [33] to a distributed and dynamic environment.

The database community has proposed a series of PDMS to enhance the usability of P2P networks, such as Piazza [37] in unstructured networks and PIER [19] and Mercury [10] in structured networks. Compared to unstructured networks, structured ones provide better search efficiency but incur more maintenance overheads. In the case of business applications, where data are in high volume, maintenance messages dominate network cost in structured networks. [38] and [24] proposed a kind of partial indexing strategy by combining structured networks and unstructured networks. Specifically, only rare items are indexed in structured networks whereas the popular ones are searched via flooding in unstructured networks. In [25], a similar scheme is proposed to speed up data dissemination in structured MANETs and support approximate similarity search for images. The above schemes are aimed at improving search latency and reducing maintenance overheads. In this paper, we start from a different motivation. Our scheme is inspired by the query patterns while the others focus on data patterns. Our partial index is built in a JIT manner and popular queries are expected to be answered by the index directly.

3. PRELIMINARIES

PISCES distinguishes itself from traditional P2P networks in three ways. First, nodes employ database systems to organize their data. Nodes can adopt different DBMS and customize their database designs. Data sharing amongst nodes is also via tuples of databases. Second, PISCES establishes some mapping servers outside P2P networks. These mapping servers translate various users' schemas into a uniform mediated one. Third, in PISCES, nodes apply a partial indexing strategy to disseminate their data, and hence greatly reduce the overhead of disseminating data.

3.1 Schema Mapping Servers

Before a new node starts sharing its data, it asks a mapping server to obtain the uniform mediated schema. Mapping servers are classified by the type of the schemas they can handle. The node selects a corresponding mapping server and sends its database schema and necessary meta-data. The server invokes a local schema mapping algorithm, such as those proposed by Doan et al. [17, 16], and returns a mediated schema to the node.

Schema mapping is a challenging task. Since the focus of our work is developing a partial indexing strategy for P2P systems rather than addressing schema matching problems, we assume that the mediated schemas of all domains have already been defined. The node only needs to perform the mapping process when it joins the network. Even if the node leaves and rejoins the network, it does not need to recreate the mapping relations if the local schema has not been changed. Thus, the schema mapping servers are only lightly loaded.

3.2 Basic Tuple-Level Indexing Strategy

In this paper, we propose indexing of important data based on the scheme proposed in [19]. We shall first review how this scheme

works: Once a node obtains the mediated schema and mapping relations, it indexes the data according to the mediated schema. Suppose we are building an index for attribute A of relation R. Then the index is given a namespace of $R.name + A.name$. Let v_k be a value of the indexed attribute of A. An index message including the namespace, v_k and ip address of the owner node is sent via the P2P protocol to the corresponding node (responsible for value v_k). After the index is established, it can be applied to answer the query (exact query or range query) involving A. Each index entry is assigned a timestamp. The node responsible for the index may discard the index entry if its timestamp has expired. To guarantee that the nodes responsible for the index are still alive, the owner node should send refreshment messages to those nodes periodically. If the node is still alive, it will refresh the timestamp. Otherwise, the index will be recreated.

In PIER, all tuples in the database are published for sharing. This full indexing strategy is impractical for a corporate network as business databases are data intensive. In PISCES, a partial indexing strategy is proposed to construct the index adaptively, which can significantly reduce cost.

3.3 Histogram-based Approach

To index only a subset of the database, we collect the query frequency for each tuple to determine if it is beneficial to index it. Even in a traditional database, storing the precise query information for every tuple is impractical due to the data size. Hence, we adopt an approximation approach. Suppose the domain of attribute A is $[L, U]$, we partition the range into m cells of equal length. The cell is used as a unit for index construction and query distribution monitoring. In index construction, once PISCES decides to index a cell, all tuples bounded by this cell will be indexed. To monitor query distribution, we build a query frequency histogram, where each cell acts as a bucket of the histogram. In this way, the statistics maintenance cost is greatly reduced.

The granularity of cells affect the accuracy of the estimated information. In this paper, the length l of cells for attribute A is computed in the following way: Suppose initially, k nodes declare that they have data for attribute A after schema mapping. The schema mapping server requires each of the nodes to send s sampling tuples. Then, we have ks samples of attribute A. We sort them in ascending order and remove the duplicate values.

Suppose the remaining values are $\{v_1, ..., v_n\}$. Let I_i and I represent $v_{i+1} - v_i$ and $\frac{l}{n}$ respectively , we define the distribution variance of cell c_i as:

$$var(c_i) = \sum_{i=1}^{n-1} E((I_i - E(I))^2)$$

We require that l should satisfy:

$$\forall i(var(c_i)) < \epsilon l^2 \wedge l \leq l_0 \quad (1)$$

ϵ is a tunable parameter; currently, we set $\epsilon = 0.01$ and l_0 is a threshold of maximal cell length. The intuition of the above definition is to guarantee that data in a cell almost follow a uniform distribution. In this way, the histogram method can provide good performance.

4. PISCES APPROACH

In this section, we introduce our proposed PISCES strategy, which constructs an index for online databases in a JIT manner. As we aim to reduce the maintenance overhead of a PDMS, we first analyze the major cost of such a system. Then a new type of index, the approximate range index, is introduced to facilitate tuple-level index building and reduce network cost. Based on the cost model and network statistics, our partial indexing strategy switches between tuple-level index and approximate range index adaptively. While the proposed indexing and querying strategy are independent of the underlying structured overlay, we use BATON [21] in our discussion. We have developed and implemented the proposed method on both BATON and CAN [30]. We note that the same idea can be extended to any structured overlays that support range search.

4.1 Cost of a PDMS

Suppose node n_i has a database with N_t tuples and we build an index for its attribute. The index maintenance cost is mainly composed of three parts: refresh cost, update cost and churn cost.

Refresh messages are used to guarantee the data are consistent and to defend against data loss from network corruption (a node leaving without notifying others). On average, node n_i sends refresh messages to the nodes responsible for its indexes every T seconds. In BATON and many other structured overlays, the routing cost of a message in an overlay of N nodes is $O(\log N)$. With a full indexing strategy, node A incurs approximately a cost of:

$$C_{refresh} = \frac{N_t \log N}{T} \quad (2)$$

per second. If only αN_t ($\alpha < 1$) tuples are indexed in a partial indexing strategy, the cost is reduced to $\alpha C_{refresh}$.

For its duration in the P2P network, node A may insert, delete or update tuples, which in turn affects the indexes. Once a tuple is inserted or deleted, the corresponding index entries should be updated as well. Suppose I tuples are inserted or deleted per second. In the full indexing case, because all data are indexed in the network, the index update strategy will incur a cost of:

$$C_{update} = I \log N \quad (3)$$

In the partial indexing case, whether the update affects the index depends on the indexing strategy. In our case, because the partial indexing strategy is query driven, the number of affected tuples can be expressed as a function of query distribution P_q, $f(I, P_q)$, where $P_q(v)$ represents the probability of value v being queried and thus $0 \leq f(I, P_q) \leq I$. The update cost for the partial indexing strategy is:

$$C'_{update} = f(I, P_q) \log N \quad (4)$$

If a node joins/leaves a P2P network, the indexes should be moved accordingly. In a structured P2P network, if a node leaves, the indexes stored at the node should be passed to its neighbors (the adjacent node in BATON or CAN). If a new node joins the network, its neighbor splits its indexes and transfers the corresponding part to the new one. The new node will also publish the data in its local database to the network. Suppose, on average, N_u nodes join or leave the network per second and the total number of nodes is kept constant. The average cost caused by churn in the full indexing strategy is:

$$C_{churn} = \frac{1}{N} N_t N_u + N_u N_t \log N \quad (5)$$

For the partial indexing strategy, the cost is reduced to αC_{churn}.

The total cost of maintaining the index for the full and partial indexing strategies are:

$$C_{full} = C_{refresh} + C_{update} + C_{churn} \quad (6)$$

$$C_{partial} = \alpha C_{refresh} + C'_{update} + \alpha C_{churn} \quad (7)$$

respectively. Note that all the costs are estimated. In our case, accurate statistics is not necessary because the model is only used as a hint for building an efficient index.

Although partial indexing reduces maintenance overhead, it also leads to lower query recall. Some queries cannot be answered by the current index as the corresponding tuples are not indexed. In this case, an alternative is required, where our approximate range index is applied. A natural question is which indexing strategy is better? We omit the detailed analysis here. Instead, we use an example to illustrate the important role of query distribution.

Suppose no index is updated and no node joins or leaves the network. Under this assumption, different index entries incur the same maintenance cost. In uniform query distribution, each tuple has the same probability of being queried. The partial index can answer α percent of queries by indexing α percent of tuples. There is no benefit compared to the full indexing strategy. In Zipfian query distribution, the distribution function is:

$$f(k; s, N) = \frac{1}{k^s H_{N,s}}$$

where $H_{N,s} = \sum_{n=1}^{N} \frac{1}{n^s}$. Assume that $s = 1$ and we want to pick the top-K terms in order that $2H_{K,1} > H_{N,1}$. We know that

$$limit_{N->\infty} H_{N,1} = \log N$$

So we assume that $H_{K,1} = \log K + C_0$ and $H_{N,1} = \log N + C_1$. To achieve $2H_{K,1} > H_{N,1}$, we need

$$2\log K + 2C_0 > \log N + C_1$$
$$\log K > \log \sqrt{N} + \epsilon \quad (8)$$

ϵ is a small constant when both K and N are large. By indexing the most popular $\sqrt{N}$ tuples, we can answer about half of the queries. Thus, the partial indexing strategy outperforms the full indexing one significantly. In a more general case, if the query follows skewed distribution, the partial indexing strategy can reduce maintenance cost.

4.2 Approximate Range Index

As mentioned, when only part of the data are indexed, queries whose involved tuples are not indexed cannot be answered using the index. In this case, flooding the entire network is an alternative. This is clearly undesirable and impractical for large scale network because of the significant communication overhead. In this paper, a new kind of index, the approximate range index, is used to handle this problem. Suppose the max and min values of attribute A in node n_i are v_{max} and v_{min} respectively. n_i can build a range index $[v_{min}, v_{max}]$ to indicate that it may answer queries which overlap with this range. We refer to this kind of index an *approximate range index*. However, as such an approximate index introduces false positive (i.e., a query is forwarded to a node that contains no answer to the query) if applied for query processing, it must be carefully designed for good performance. For example, in an extreme case, assume node n_i's tuples have the same A value, v_{min}; then, the index will always return a false candidate if used to answer the queries in $(v_{min}, v_{max}]$. In our system, a range index entry is in the format of *(namespace, min, max, number of unique values)*. To measure the effectiveness of an approximate range index, we define *False Positive Factor* as:

Definition 4.1. ***False Positive Factor***
The false positive factor (ρ) of a range is defined as the probability of returning a false positive for an arbitrary query (range or exact query) that overlaps with the range.

As we use cell as a unit for index building, two kinds of False Positive Factor should be considered, outer cell factor and inner cell factor. To simplify the notation, we use $|R|$ to represent the total number of cells in the range R.

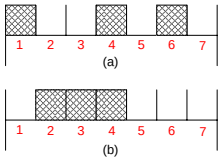


Figure 2: Comparison of Different False Positive Factor

Definition 4.2. ***Outer Cell Factor***
Outer cell factor is used to estimate the false positive of queries with range greater than cell length l. Suppose $f(i)$ returns the frequency of consecutive i empty cells, outer cell factor ρ_0 is computed as:

$$\rho_0 = \frac{2}{|R|(|R|+1)} \sum_{i=1}^{|R|} f(i) \quad (9)$$

For example, in Figure 2, let the shaded cells represent non-empty cells. $f(1)$ returns 4 for both Figure 2(a) and Figure 2(b) (4 empty cells in each case), while $f(2)$ returns 1 ($r_1 = (c_2, c_3)$) for Figure 2(a) and 2 ($r_1 = (c_5, c_6)$, $r_2 = (c_6, c_7)$) for Figure 2(b) respectively. Finally, the outer cell factor of Figure 2(a) and Figure 2(b) are estimated as 17.8% and 25% respectively. Note that the two ranges with the same length and same number of non-empty cells in Figure 2 have different false positive factors due to their distinct data distributions.

Definition 4.3. ***Inner Cell Factor***
Suppose $P(r)$ represents the possibility of issuing a range query with length $r < l$ (l is the cell length), and n_i is the number of unique values in cell i, the inner cell factor for range R is computed as:

$$\rho_1 = \frac{1}{|R|} \sum_{cell_i \in R} \int_0^l P(r)(1 - \frac{r}{l})^{n_i} d(r) \quad (10)$$

Inner cell factor gives a more precise description for possible false positives of small range queries (exact query are 0-length range query). If the query distribution is unknown, we assume that queries of different ranges are issued in the same probability. Then the cell's inner false positive factor can be calculated as $\frac{2}{n+2}$. Finally, the false positive factor of range R is estimated as:

$$\rho(R) = (1 - \frac{l^2}{R^2})\rho_0 + \frac{l^2}{R^2}\rho_1 \quad (11)$$

To incrementally compute the false positive factor, we introduce the following properties.

Lemma 4.1. *If range r is chosen to be indexed and c_i is a border cell of r, c_i must be non-empty cell. Otherwise, we can get a better indexing strategy.*

Proof. (sketch) If a range r has empty cells as its borders, we can remove the empty cells and obtain an indexed range with lower cost. The range can answer the same set of queries but the maintenance cost is reduced since the number of false positives is decreased. □

As shown in Figure 3, the original indexing strategy has two entries, [L, a] and [a, U]. But we can shrink them to get two ranges, [L, b] and [c, d], with lower cost. The removed empty range from the index entries must be the maximal empty range (all the cells in the range are empty cells and its adjacent cells are non-empty).

Theorem 4.1. *Suppose a range R is partitioned into a maximal empty range r_0 and two non-empty ones, r_1 and r_2. The outer cell factors of ranges satisfy:*

$$\rho_0(R) = \frac{(|r_0|+1)|r_0|}{(|R|+1)|R|}\rho_0(r_0) + \frac{(|r_1|+1)|r_1|}{(|R|+1)|R|}\rho_0(r_1) + \frac{(|r_2|+1)|r_2|}{(|R|+1)|R|}\rho_0(r_2)$$

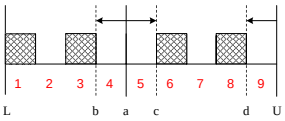


Figure 3: Shrunk Index Range

PROOF. (sketch) The query, which cannot be answered by R, cannot be answered by r_0, r_1 and r_2 either. For r_0, no queries can be answered because it is an empty range. As indicated by Lemma 4.1, both r_1 and r_2's border cells to r_0 are non-empty. If the query can be answered by r but cannot be answered by r_1, it must be answered by r_2 and vice versa. So the outer cell factors of R is actually the sum of outer cell factors of r_0, r_1 and r_2 with normalization of query distribution. □

As a simple example, suppose the cells in Figure 2(a) are partitioned into three ranges, r_0={cell 2,cell 3}, r_1={cell 1} and r_2={cell 4, cell 5, cell 6, cell 7}. From Equation 9, we compute $\rho_{r_0} = 1$, $\rho_{r_1} = 0$ and $\rho_{r_2} = 0.2$. Assume query distribution is uniform for R, the probability of being queried for the subranges are $p_q(r_0) = 0.107$, $p_q(r_1) = 0.0357$ and $p_q(r_2) = 0.357$. So the estimated false positive factor of r is $1 \times 0.107 + 0 \times 0.0357 + 0.2 \times 0.357 = 0.178$.

THEOREM 4.2. *Suppose a range R is partitioned into range r_0 and r_1. The inner cell factors of ranges satisfy:*

$$\rho_1(R) = \frac{1}{|R|}(|r_0|\rho_1(r_0) + |r_1|\rho_1(r_1))$$

PROOF. (sketch) This can be verified by Equation 10. □

4.2.1 Optimal Approximate Range Index Strategy

The cost of approximate range index includes index maintenance cost and false positive cost. When used during query processing, the approximate range index may return false positives, i.e., some queries may be forwarded to nodes that will not contribute any answer tuples. Let x be the number of queries for the range R. The overhead of the approximate index for range R is estimated as:

$$C_{range} = \rho(R)x + C_{entry} \tag{12}$$

C_{entry} is the average maintenance cost of a single range index entry. It can be derived from Equation 6.

$$C_{entry} = \frac{C_{full}}{N_t} = \frac{\log N}{T} + \frac{I \log N}{N_t} + \frac{N_u}{N} + N_u \log N \tag{13}$$

Because the update frequency of approximate range index is far lower than that of tuple-level index, we discard the term for updates in Equation 13 when computing cost for range entries.

Now we face the problem of building an optimal approximate range index, which can be formalized as

DEFINITION 4.4. ***Optimal Range Index Problem***
Given query distribution P_q, for attribute A with domain [L, H], node n wants to set up an approximate range index I with k entries. For any non-empty cell c_i of n, $\exists r_j \in I \rightarrow c_i \in r_j$, and the cost

$$\sum_{i=1}^{k} \rho(r_i)x_i + kC_{entry} \tag{14}$$

is minimized.

The optimal range index problem is an NP-hard problem, if the range is partitioned in a continuous way. Fortunately, as we use cell as a basic unit for indexing, the optimal indexing strategy can be solved by dynamic programming. We search the possible solutions of different partitioning strategies and select the best one. Let $f(p, q, k)$ denote the optimal indexing strategy by partitioning the range from cell p to cell q into k entries ($q \geq p$). The optimal solution can be computed as the best combination of the sub-solutions, which is represented as:

$$f(p,q,k) = min(f(p,x,y)+f(x+1,q,k-y)), (p \leq x < q \wedge 1 \leq y \geq k-1) \tag{15}$$

Algorithm 1 lists the detail of selecting the optimal indexing strategy. At first, the cost table is initialized to compute the basic cost (lines 1 to 5). In line 5, *shrinkcost*(i, j) removes the empty cells as suggested in Lemma 4.1 and returns the shrunk cost of the range from cell i to j. Then we compute strategies with different number of partitions and return the one with the least cost (lines 7-11). In line 8, a recursive function f is invoked to compute the minimal cost of partitioning the whole space into a specific number of entries. The detail of f is illustrated in Algorithm 2. If the cost of the strategy has already been recorded, we return it directly (lines 3-4). Otherwise, the minimal cost is computed by Equation 15 (line 7-11). And the new cost is updated in the cost table. The complexity of Algorithm 1 is $O(e - s)^3$. Once, the entries of cost table have been filled up, the algorithm terminates.

Algorithm 1 OptimalIndex(int s, int e)

```
//s: index of start cell
//e: index of end cell
 1: initialize T as a (e-s+1)×(e-s+1)×(e-s+1) table
 2: for i = 1 to e-s do
 3:     for j = 1 to e-s do
 4:         T[i,i,j]=cost of cell i
 5:         T[i,j,1]=shrinkcost(i,j)
 6: min=possible max value, minindex = 1
 7: for i = 1 to e-s do
 8:     c=f(s, e, i)
 9:     if c<min then
10:         min=c, minindex=i
11: return strategy recorded by minindex
```

Algorithm 2 f(s, e, k)

```
//k: number of partitions
 1: if k>e-s+1 then
 2:     return invalid
 3: if T[s,e,k] is not null then
 4:     return T[s,e,k]
 5: else
 6:     min=possible max value, mini=1, minj=1
 7:     for i=1 to k-1 do
 8:         for j=s to e-1 do
 9:             c=f(s, j, i)+f(j+1, e, k-i)
10:             if c<min then
11:                 min=c, mini=i, minj=j
12:     T[s,e,k]=min
13:     return min
```

THEOREM 4.3. *Algorithm 1 returns an optimal indexing strategy.*

PROOF. (sketch) We first prove that for a specific k, *f(p,q,k)* returns the minimal cost. Then, as our algorithm iterates all possible ks, it can get the optimal one. When *k=1*, there is only one indexing strategy and thus *f(p,q,1)* is the best strategy. Assume for *k<n*, *f(p,q,k)* always returns optimal strategy, now we prove that for *k=n*, the above conclusion also stands. Suppose *f(p,q,n)* is not the optimal one, and we get another best strategy *f'(p,q,n)*. In *f'(p,q,n)*, we can get two sub partitions, *f'(p,q-x,n-y)* and *f'(q-x+1,q,y)*. *f'(p,q-x,n-y)* and *f'(q-x+1,q,y)* must be the suboptimal solutions, other-

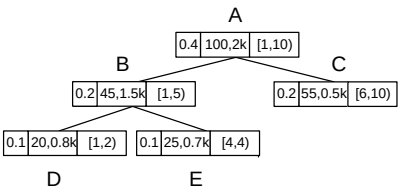


Figure 4: Maintenance Tree

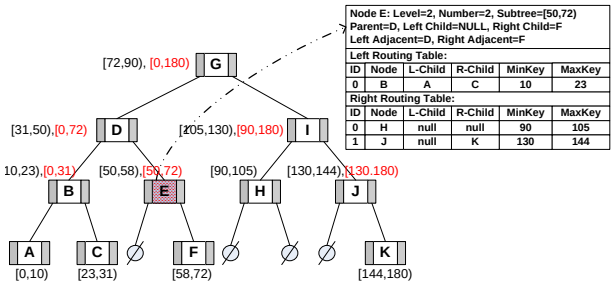


Figure 5: BATON Tree

wise *f'(p,q,n)* is not the optimal one either, which means *f'(p,q-x,n-y)=f(p,q-x,n-y)* and *f'(q-x+1,q,y)=f(q-x+1,q,y)*. So the cost of our solution *f(p,q,n)* is no more than that of *f'(p,q,n)* and it must be the optimal one. □

4.2.2 Maintenance of Approximate Range Index

With Algorithm 1, given the query distribution, we can compute the optimal range indexing strategy in $O(n)^3$, where n represents the number of cells. Once the query distribution evolves, the indexing strategy should be recomputed. However, we do not want to invoke Algorithm 1 frequently. Although cell number is far smaller than the tuple number, $O(n)^3$ is still a remarkable cost. Moreover, indexing range entries incurs additional network overheads. In this section, we propose our light-weight maintenance scheme. The scheme is based on the observation that the query distribution changes slowly.

After Algorithm 1 computes the optimal indexing strategy for the current query distribution, we construct a binary tree to organize the index entries. The leaf node of the tree refers to an index entry, whereas the inner node represents a set of entries. The tree node is represented in the format of:

$$\{errorbound, (cost, tupleNumber), cellrange\}$$

Error bound is applied to limit the estimation error, cost represents the maintenance cost of the range entries computed by Algorithm 1, tuple number is the approximate number of tuples in the range and cell range describes the start and end cell IDs.

DEFINITION 4.5. ***Estimation Error***
For a tree node n, suppose its initial cost computed by Equation 12 is c. After a time t, if the new observed cost is c', the estimation error is defined as $\frac{|c-c'|}{c}$.

As shown in Figure 4, the root node represents the domain of the attribute (cell 1 to cell 10). The total error bound is 0.4 and its current cost is 100. The node splits its range into two children such that their cost variance is minimized. The error bounds of the children are half of their parents. Note that the combination of children's ranges may not equal to the parent's range, because some empty cells are excluded from the index entries.

If the query distribution of an index entry changes, the corresponding leaf node will check if the change results in violation of the error bound. If not, the change will be discarded. Otherwise, the node will ask its parent for help. The parent node computes its estimation error. If the change is still bounded, the parent node will invoke Algorithm 1 to compute an optimal indexing strategy for its range, and all descendant nodes update their ranges and initial cost. Otherwise, the parent will forward the request to its own parent. The process may be recursively invoked until the root node is reached, where the total optimal indexing strategy is recomputed. The total error bound is a tunable parameter. A small error bound makes the index more sensitive to query distribution, while a large bound leads to lazy update. In the experiments, we set it to 0.4.

4.2.3 Index Range Entry in BATON

Structured P2P networks support exact key index. Given a key, the protocol can locate the peer responsible for the key in $O(logN)$ time. For overlays that support range queries, it is easy to extend them to support range index. In this paper, we use BATON [21] as an example, and BATON and CAN [30] in our implementation; other overlays, such as P-ring [14], P-Grid [6] and HotRod [28], can also adopt the same scheme.

Figure 5 illustrates the structure of BATON. Besides the original indexes, an internal node of BATON also records the range of its subtree. For example, node E's own range is $[50, 58)$ and its subtree's range is $[50, 72)$. Once a node joins the network, it gets a subtree range from its parent. The subtree range needs to be updated only when nodes leave or join or when BATON needs balance the system load. Moreover, even if a node drops out of the network without notification, the subtree range can be recovered by asking its parent and children.

Algorithm 3 shows how to locate the node responsible for a specific range. We depend on BATON's routing protocol to locate the node for a key (line 1). Here, the mid-point of the range is used as the key, which turns out to be more effective than the minimum and maximum values. The request is continuously forwarded to the upper level nodes until a node whose subtree range fully contains the range is reached. The complexity of Algorithm 3 is $O(logN)$ (N is the number of nodes), for the tree height is $O(logN)$[21].

The range indexing algorithm is illustrated in Algorithm 4. To publish one range index entry, the node invokes Algorithm 3 to find the candidate node for its entry. To reduce the cost of query processing, we keep a soft status for each node. The node is marked as left active, right active, left-right active or non-active (indicating that it indexes ranges overlapping with left subtree, right subtree, both subtrees or none of them). After indexing a range entry R, the node checks whether its status need to be changed and notifies its descendants. The notification processing is omitted here because of space constraints. In summary, the node will change its status and forward the notification to its children if it is in a different status. Otherwise, the notification process is terminated. On the other hand, if the last range index is removed from the node, it will invoke the notification process as well.

Algorithm 5 lists the range query processing via approximate range index. The algorithm checks the status of the parent node to decide whether to forward the request or not.

4.3 Tuning the Partial Index

In this section, we present our JIT indexing strategy: only index what is beneficial to the current system. If a data range becomes popular, the nodes responsible for the range will receive more queries. These nodes trigger an index requirement to data owners. This behavior is similar to the publish/subscribe system. The approximate range index can be considered as a subscription,

Algorithm 3 search(range r)

```
1: node m=lookup(r.mid)
2: if m.subtreeLow≤r.low and r.up≤m.subtreeUp then
3:     return m
4: else
5:     m=m.parent
6:     return m.search(r)
```

Algorithm 4 index(range r, string ip, string namespace)

```
1: node m=search(r)
2: m.buildIndex(r, ip, namespace)
3: if !m.isLeftActive and isOverlap(r,m.leftSubtree) then
4:     send left notification to m.leftChild
5:     m.isLeftActive=true
6: if !m.isRightActive and isOverlap(r,m.rightSubtree) then
7:     send right notification to my rightChild
8:     m.isRightActive=true
```

registering what data the node can provide. After receiving the notification, the owner node publishes its tuple-level index for the corresponding cells. Thus the partial indexes are built in a query driven manner.

For a cell c, suppose there are k nodes having data in this cell and node i has approximately n_i tuples in the cell ($\sum_{i=1}^{k} n_i = n$). Let n_q be the average number of queries per second for this cell, if:

$$nC_{entry} - n_q \sum_{i=1}^{k} \int_0^l P(r)(1 - \frac{r}{l})^{n_i} d(r) \tag{16}$$

is negative, the cell should construct tuple-level index, because even the smallest range index (1 cell length) will incur too much overhead. The integration based on query distribution can be computed by Monte Carlo method. S samples are taken to estimate the values. To emphasize the importance of recent queries, n_q is calculated as:

$$n_q = q + \frac{n'_q}{f} \tag{17}$$

where q represents the query number in this second, n'_q is the last computed value of n_q and f is an aging factor to degrade the effect of old queries. This aging scheme is also adopted in web cache management [29]. In our implementation, the query number is computed periodically and the average value is applied. To get the approximate values of k and n_i in Formula 16, when we search the index of a specific range in Algorithm 5, we piggy back the number of tuples registered in the range indexes to the leaf node (recall the range index format). Then the average number of tuples in a cell of the node can be computed. This piggy back process is invoked periodically.

The tuple-level index is constructed in three steps

Algorithm 5 query(range r)

```
1: node m=search(r)
2: return m's local index for r
3: while true do
4:     n = m.parent
5:     if (m=n.leftChild and n.isLeftActive) or (m=n.rightChild
       and n.isRightActive) then
6:         m=n
7:         return m's local index for r
8:     else
9:         break
```

1. Node responsible for the hot range decides to index tuples for some cells according to Formula 16.
2. Algorithm 5 is invoked to find all nodes overlapping with the cell. And a notification message is sent out to these nodes.
3. After publishing their data, the nodes update their local maintenance trees of approximate range indexes.

The first two steps of creating partial index guarantee that all online nodes achieve the same indexing strategy (for a tuple, it will be indexed by all nodes or none of them). In step 2, once a node m sends index notification messages, it records the indexed cell's ID. For the newly joining node, when it publishes its range index to a node m, m will check its record for indexed cells and inform the joining node about the global decision.

In step 3, the node, after sending its data for indexing, will update its maintenance tree to reflect the new cost. For example, in Figure 4, if tuple-level index is created for cell 1, the cost of node D, B and A need to be recomputed. The cost of indexed cells are removed from the range entry cost. If the estimated error exceeds the error bound, the indexing strategy should be modified.

Similar to the index construction, if a node observes that an indexed cell is no longer beneficial (formula 16 is positive), the node will discard the index of the cell and inform the corresponding nodes. The maintenance trees of the involved nodes are updated as well.

4.4 Load Balancing

Load balancing is a well studied topic in P2P system. In this paper, load balancing is accomplished mainly via BATON's protocol, which has been shown to be effective for data indexes. However, indexing range intervals incurs new load balancing problem. As discussed in [23], arbitrarily small ranges may be mapped to arbitrarily large interval. For example, range *[70, 80)* is mapped to interval *[0, 180)*(node G) in Figure 5. To solve this problem, we adopt a partitioning strategy.

DEFINITION 4.6. ***Range Index Entry Partitioning***
Suppose node n's minimal value and maximal value for tuple-level index are T_{min} and T_{max} respectively, and its subtree's range is [L, U). A range entry R=[l, u) at node n should be partitioned i.f.f. $T_{min} - l < \frac{1}{2}(T_{min} - L)$ *and* $u - T_{max} < \frac{1}{2}(U - T_{max})$

With this definition, range *[70, 80)* will be partitioned into subranges *[70,72)* and *[72,80)*, which are indexed at nodes F and G respectively.

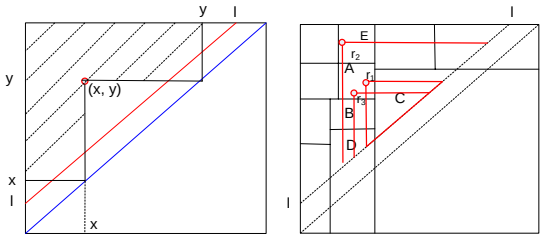
THEOREM 4.4. *A range entry is partitioned at most once according to the Definition 4.6.*

PROOF. (sketch) After one partition, suppose the subrange r' is routed to node m', it can no longer be partitioned. Otherwise, we can find a child of m' whose subtree range also contains r'. This conflicts with Algorithm 3, which returns the first node that contains the range. □

As BATON is a balanced tree, in most cases, search space is partitioned evenly. Range partition defined in Definition 4.6 is effective to balance the load. In addition, the partial indexing strategy also lightens the load of large range interval. As the index is designed to answer the hot queries, the approximate range index is therefore seldom used.

4.5 Implementation in Other Overlays

Our partial indexing scheme is not limited to BATON. In this section, we discuss how to extend the approach to other overlays.



(a) Index Range in CAN (b) Notification Area

Figure 6: Range Indexing in CAN

In particular, we have also implemented our scheme on CAN [30]. To support partial indexing scheme in CAN, we only need to modify CAN to support range index. In [32], Sahin et al. propose a method on CAN to process range queries. We modify their scheme to support the approximate range index. For a range $R = [x, y]$, it can be represented as a point *p=(x, y)* in a 2-dimensional CAN. The range $R' = [x', y']$ that overlaps with R satisfies:

$$x' \leq x \leq y' \vee x' \leq y \leq y'$$

The search area for the corresponding point of R' is indicated as the shaded area in Figure 6(a). Hence, to retrieve the range index for a range query, we should search the query's shaded area.

For an exact query $q = x$, we use a point (x, x) in the diagonal to represent it. Then, only the space in the left-upper corner is used (the space above the blue line), because values of y-axis are always greater than those of x-axis. In our scheme, we use the cell as a unit to store the data. So the value v is actually stored at the node responsible for the point $(\lfloor \frac{v}{l} \rfloor, \lfloor \frac{v}{l} \rfloor + l)$, where l is the cell length. Thus, the usable data space in CAN is the space above the line $Y = X + l$, the red line in Figure 6(a).

To optimize the search efficiency and ensure load balancing, two strategies are applied.

1. Searching the whole shaded area of a query incurs too much overhead. Instead, we use skyline points [11] to prune the search space. As shown in Figure 6(b), the red triangle represents the dominated range of range index r_1. Only queries within the triangle need to search r_1. To prune the search space, when publishing a range index r_1, we invoke Algorithm 6 to inform the corresponding nodes. Specifically, we keep a skyline point set for each node and their neighbors respectively. Figure 6(b) shows the idea of Algorithm 6. The first range index r_1 is indexed at node A and A will update its R[A], the skyline point set for itself, as $\{r_1\}$ and send notification to its neighbors, B and C. Then, the second range index r_2 is indexed at E and the notification is sent to A. Node A updates both skyline point sets for neighbor E (R[E]) and itself (R[A]) as $\{r_2\}$. The notification is sent to A's neighbors as well. Finally, the third range index r_3 comes to A. But A finds that r_3's point is dominated by $r_2 \in R[A]$. So A will not send notification to its neighbors. Once receiving a query q, A first processes it locally and then checks the skyline point sets of its neighbors. Query q is forwarded to neighbor E, only if q is dominated by a point in R[E].
2. As only the space above the line $Y = X + l$ is used, to balance the load, we modify CAN's partitioning strategy to split the space above $Y = X + l$ evenly, instead of partitioning the whole space evenly. Figure 6(b) illustrates a result of such partition.

The cost model of CAN is slightly different to BATON, as the routing cost in CAN is $O(\sqrt[d]{N})$. However, we can follow the same analysis and the details are therefore omitted due to space constraint.

Algorithm 6 Notify(node n, node m, range r)

//n receives the notification sent by m for range r
1: p=getPoint(r)
2: **if** p is not dominated by points in *n.R[n]* **then**
3: add p to *n.R[n]* and *n.R[m]*
4: *n.R[n]*=getSkyline(*n.R[n]*)
5: *n.R[m]*=getSkyline(*n.R[m]*)
6: Notify n's right-down neighbors about r

Other overlays such as P-Grid [6] and Pastry [31] can apply the scheme proposed in [15] to process the range index. The routing request is always sent to the parent node which maintains a larger search range. Our scheme does not need any modification to be integrated to these overlays.

5. MONITORING P2P NETWORK

PISCES requires approximate global statistics, e.g. total number of nodes N, total query number N_q, query distribution P_q and average number of nodes joining/leaving N_u. However, in P2P systems, it is not only costly to collect these statistics, the dynamism of the system will render any accurate statistics obsolete very quickly.

Sampling techniques have been applied to unstructured P2P networks in [8, 36]. The sampling algorithm in unstructured P2P networks try to sample each peer in the same probability. In DHT network, this issue is no longer a problem for the use of consistent hashing. By sampling the nodes through randomly generated IDs, we will sample each node in the same probability. As shown in Manku's scheme [27, 26], the estimation error can be bounded by a constant factor. However, for the overlays supporting range queries [21, 6, 7], hash function is not adopted and the above schemes cannot be applied in our case. Thus, in this paper, we introduce a new sampling scheme to address this problem.

Previous sampling schemes periodically send sampling messages to the nodes in a random manner. This strategy introduces additional cost to P2P network. Instead, we decide to exploit the overlay's protocol to do the sampling. Our scheme can be applied to any overlays. For simplicity in description, we again use BATON as an example.

5.1 Sampling Scheme

5.1.1 Footprint

To count the number of queries, the nodes record the corresponding messages. One choice is to make the nodes, which process the queries or updates, store the performed operations. But this scheme leads to biased sampling result when some hotspot exists. To better distribute the query and update information in the network, we adopt a "footprint" scheme.

In BATON, a query will be processed in about $O(\log N)$ messages, and about $6 \log N$ messages are required for node joining and $4 \log N$ messages for node leaving. In the original protocol, after receiving these messages, the routing node will process the corresponding request (update routing finger table or forward the message to other nodes). But no information about the actions are recorded. In this paper, we exploit these messages to help the sampling process.

After a query is issued from the user node, it will reach the destination in about $O(\log N)$ nodes. Now, we require the node, which routes a specific query Q, to add the following entry to its local storage.

$$(Q.ID, Q.time, Q.range)$$

The query ID is generated by the system automatically and each query has a unique ID. The node records when it receives the query routing request. And after a time threshold T, if $Q.time + T < current\_time$, the record of query Q will be discarded. The query range is used to infer the involved cells. In this way, a query will generate about $O(\log N)$ footprints in the network.

Following the same idea, when a node joins or leaves the network, we require it to mark its footprint in the corresponding nodes. In BATON, once joining or leaving, the node sends notification messages to nodes in its level and its parent level to correct the routing fingers. If a node receives such notification message, it will generate the following records:

$$(Event, Event.time)$$

Event indicates whether it is a join or leave operation. And the node records the message time as well. The old records will be discarded periodically. In BATON, node joining and leaving will create about $6log(N)$ and $4log(N)$ footprints in the network respectively.

5.1.2 Iterative Sampling

In the BATON protocol, the node periodically sends ping messages to one of its routing fingers to fix incorrect fingers (other overlays such as Chord [34] also perform this kind of stabilization). This stabilization process is light-weight and performed frequently (averagely 30 seconds in Chord protocol). Our sampling scheme exploits the finger ping messages to accomplish the statistics collection. Once the ping messages are received, the routing node will reply with a summary of the corresponding statistics records.

Algorithm 7 stabilization(node n)

```
//n_q, n_l, n_j : local records of query number, number of nodes leaving and joining
//n̂_q, n̂_l, n̂_j : current global estimated statistics values
//n̄_q, n̄_l, n̄_j : iterative estimated values
1:  while true do
2:    for every i in the routing table do
3:      if finger[i]!=null then
4:        send ping message to finger[i] for stabilization and collecting statistics
5:        wait for reply
6:        n̄_q+ = finger[i].n̄_q
7:        n̄_j+ = finger[i].n̄_j
8:        n̄_l+ = finger[i].n̄_l
9:      wait for t seconds
10:   do the same sampling scheme for adjacent nodes, parent node and child nodes
11:   n̄_q, n̄_l, n̄_j are computed as the average values of neighbors
12:   if time for new estimation then
13:     n̂_q = n̄_q, n̂_l = n̄_l, n̂_j = n̄_j
14:     n̄_q = n_q/logN, n̄_l = n_l/4logN, n̄_j = n_j/6logN
```

In Algorithm 7, three versions of estimated values are maintained. Local records denote the query or churn footprints recorded by the node in the local database. It is the raw statistics observed by the node. The current global estimated values are used for range index construction. And the iterative values are used to compute the next global estimation values. The node collects the estimation values from its routing fingers, adjacent nodes, parent node and child nodes. And its own estimations are computed as the average values of the received values (lines 2-11). After sometime, the iterative values are used as the new global estimated values and we start the processing for computing new statistics. (lines 12-14).

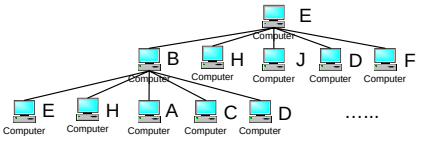


Figure 7: Sampling Process

5.1.3 Analysis of the Sampling Scheme

Figure 7 shows how the sampling process works for a BATON tree as illustrated in Figure 5. Node E collects statistics from its routing fingers, which will recursively extract statistics from their routing fingers. In this way, an information aggregation "tree" is constructed (only the left-most subtree is displayed). This "tree" is actually a graph as a node may have different parents. In an iterative way, the global information will be aggregated at the root node. But what is the expected statistics after some iterations?

We define two matrixes, R and P, to represent the relationship between nodes.

$$R = \begin{pmatrix} r_0 \\ r_1 \\ \dots \\ \dots \\ r_{N-1} \end{pmatrix} P = \begin{pmatrix} p_{0,0} & p_{0,1} & \dots & \dots & p_{0,N-1} \\ p_{1,0} & p_{1,1} & \dots & \dots & p_{1,N-1} \\ \dots & & & & \\ \dots & & & & \\ p_{N-1,0} & p_{n-1,1} & \dots & \dots & p_{N-1,N-1} \end{pmatrix}$$

R is a $1 \times N$ matrix. r_i represents the corresponding local statistic data of ith node in the network. P is an $N \times N$ matrix, representing the routing finger relationship. If node j does not exist in the finger table of node i, $p_{i,j} = 0$. Otherwise, $p_{i,j}$ is a non-negative value representing the weight of node j in node i's finger table. Based on Algorithm 7, $p_{i,j} = p_{i,k}$ if both of them are non-negative. In addition, the diagonal element $p_{i,i}$ is always non-negative and the matrix P satisfies:

$$\sum_{j=0}^{N-1} p_{i,j} = 1$$

Let the number of non-negative elements in row i and column j be pr_i and pc_j respectively. In Chord, BATON or most other structured overlays, pr_i and pc_j are therefore $O(\log N)$.

Suppose all nodes perform Algorithm 7 to collect the statistics. Let $1 \times N$ matrix R_i represent the estimated statistics after i iterations. That is, the element of row j denotes node j's estimation. R_n is represented as:

$$R_n = \begin{cases} R & \text{if } n = 0 \\ P^n R & \text{otherwise} \end{cases}$$

Theorem 5.1. *The estimated statistics computed by Algorithm 7 converges to the approximate average global statistics of the network.*

Proof. (sketch) In fact, we only need to prove that $L = lim_{n\to\infty} P^n$ exists and equals to a $N \times N$ matrix S where $\forall i \forall j (a_{i,j} \in S) \rightarrow (|a_{i,j} - \frac{1}{N}| < \epsilon)$. SR will produce a global statistics which approximates $\frac{1}{N}(r_0, r_1, ..., r_{N-1})$ for each node.

Because $\sum_{j=0}^{N-1} p_{i,j} = 1$ and $0 \leq p_{i,j} \leq 1$, P is actually a markov stochastic matrix, generated by considering nodes and their links as a directed graph. In BATON network, each node can locate arbitrary node in $O(\log N)$ step, which means that we can find a constant c, the elements of matrix P^c are all non-zero. Hence, the Perron-Frobenius theorem [9] guarantees that the limit L always exists. And there is a stationary probability vector π that does not change under application of the transition matrix. π is associated with eigenvalue 1. That is $\pi(P - I) = 0$. We get $\pi P = \pi$. The

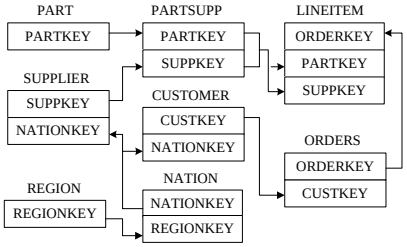


Figure 8: Schema of TPC-H

element $a_{i,j}$ of L is equal to jth element of π. Because for each row and column, there is a similar number of non-negative elements with same value, vector π is approximate to $(\frac{1}{N}, \frac{1}{N}, ..., \frac{1}{N})$. □

Theorem 5.1 indicates the correctness of the sampling scheme in a static environment (the initial statistics, matrix R, does not change). In our scheme, we do not need to wait for the matrix to converge. An approximate estimation is sufficient for building a good range index. To handle the dynamism of the network, in Algorithm 7, we will recompute the global statistics periodically. The new statistics are taken into account when new computation starts.

5.2 Optimization

If we piggy back the detail query distribution in each pong message, the network cost will increase for the large size of pong messages. To keep the pong message as compact as possible, we adopt some kind of optimization. First, we have two observations about query distribution in real systems. 1) Most tuples receive zero or few queries. 2) Queries are always focused on a hot area. In our scheme, a cell is used as a bucket in histogram construction. To further reduce the size of data to be transferred, we adopt the clustering strategy. Cells are clustered by their received query numbers. And the medians of the clusters are sent via pong messages. In addition, bloom filter [13] of each cluster is created and sent along to verify the membership of the cluster.

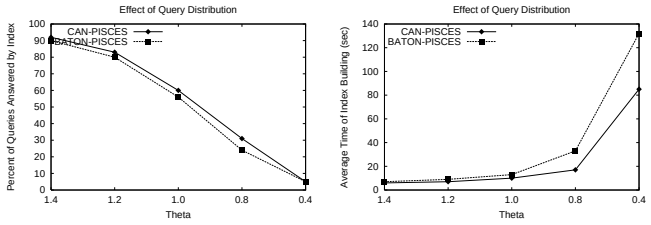
6. EXPERIMENTAL EVALUATION

In this section, we evaluate our PISCES on PlanetLab[2], an open platform for deploying distributed systems. 256 nodes are selected from different continents in the world. We test our system in different network configurations to demonstrate its flexibility and scalability. And the results show that PISCES outperforms the full indexing scheme significantly, especially in the case of large data size, high update rate and high churn rate.

To simulate the real data, TPC-H[4] generator is used to generate data for the nodes. TPC-H is a decision support benchmark, and its schema is shown in Figure 8. Specifically, the following query is used as the test query:

SELECT sum(TOTALPRICE) as totalSales
FROM ORDERS
WHERE TOTALPRICE>x and TOTALPRICE<y

The tuples in *ORDERS* table are grouped according to the *SUPPKEY* of *LINEITEM* (this is done by joining the *ORDERS* and *LINEITEM* tables and grouping tuples by *SUPPKEY*). Then the orders for the same *SUPPKEY* is disseminated to the same node, which simulates the real scenario. Each node is assigned between 1k to 10k tuples. Every experiment is run for one hour (the network initialization time is not included). The queries with selectivity of 0.01 are generated based on the Zipfian distribution with θ ranging from 0.4 (mildly skewed) to 1.4 (highly skewed). This allows us to evaluate the performance of PISCES under the conditions when



(a) Index Effectiveness (b) Index Building Time

Figure 9: Effect of Query Distribution

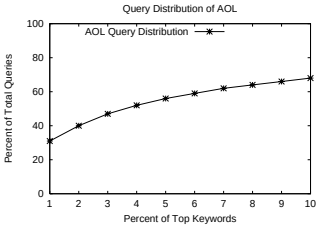


Figure 10: Statistics of AOL

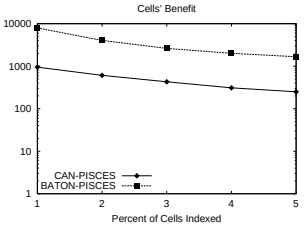


Figure 11: Cells' Benefit

there are some popular data in the dataset and the query distribution changes after a time interval.

Parameters	***Default Values***
Network Size	256
Data Size	1000 per node
Update Rate	0.1 per second
Churn Rate	0.001 per second
Interval of Changing Query Distribution	500 second
Zipfian θ	1.0
Total Time	3600 second
Query Interval	300 msecond

The above table lists default configuration of the experiments. We use CAN-PISCES, CAN-FULL, BATON-PISCES and BATON-FULL to denote the PISCES in CAN, full indexing scheme in CAN, PISCES in BATON and full indexing scheme in BATON respectively. We count the total number of messages each node received in an experiment, including query messages and index maintenance messages (including index messages caused by node joining or leaving to create/delete new/old indexes, migration of indexes, etc). Then, the average message number in a minute (per node) is used as our main performance metrics.

6.1 Effect of Query Distribution

As we discussed before, the query distribution plays an important role in the index selection. In this subsection, we evaluate PISCES by issuing queries of different distributions. Two metrics are defined in this experiment. In Figure 9(a), we measure the percentage of queries that can be directly answered by the index without JIT data pulling from the individual ERP or database system. In Figure 9(b), the delay to construct the index is computed (measured as the interval between the data becoming popular and the index for the data being constructed) When the query distribution is highly skewed, about 90% of queries can be directly answered by the index. Moreover, PISCES is also sensitive to the changing of query distribution. The new index entries can be incorporated into the existing index in less than 10 seconds. However, as the distribution becomes more uniform (θ close to 0), the index cannot answer most of the queries and it also takes a long time for PISCES to observe the query distribution.

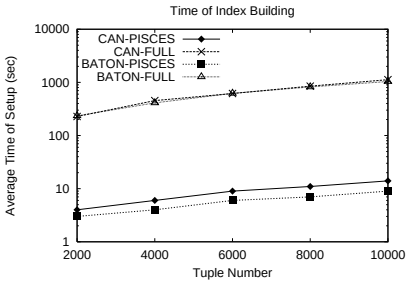


Figure 12: Average Setup Time of Node

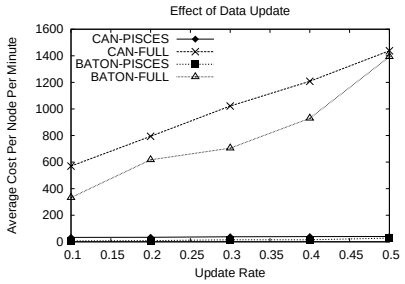


Figure 13: Effect of Update

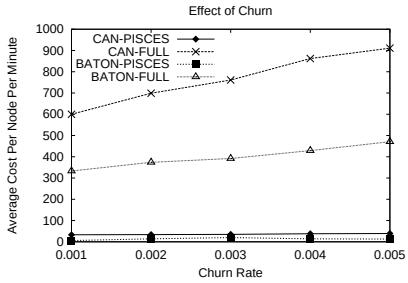
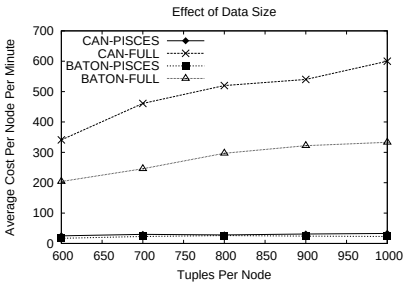
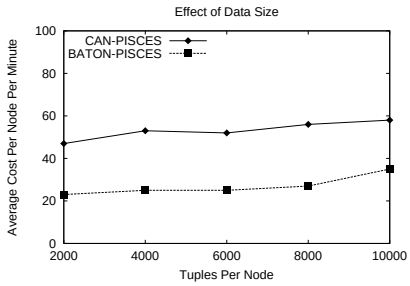


Figure 14: Effect of Churn



(a)



(b)

Figure 15: Effect of Data Size

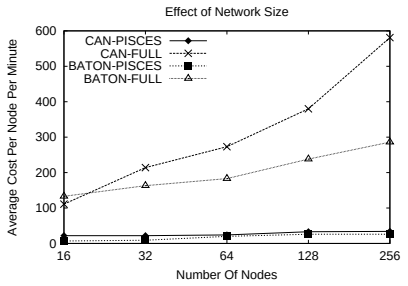


Figure 16: Effect of Network Size

This indicates that PISCES is most efficient in skewed query distribution. In real cases, queries often follow some patterns. Figure 10 shows the statistics about the search of AOL [1] from March 2006 to May 2006. The top 5% keywords attract about 60% of total queries. In Figure 11, we compute the benefit of indexing data within a cell. The benefit is defined as:

Number of Answered Queries / Indexing Cost

1000k queries following Zipfian distribution with $\theta = 1$ are injected to the system and we sort the cells by their benefits in descending order. Figure 11 shows the average benefit of indexing top 1% to 5% cells. The average benefit decreases sharply as more cells are indexed, which indicates that only the top cells are worth indexing.

6.2 Average Setup Time

In this experiment, we measure the average time for building indexes. When a node joins the network, its index is constructed to allow its content to be searched. Full indexing strategy requires the node to publish all the local sharable data, which may last for hours. Instead, PISCES can significantly reduce the joining time. We do note that peers are likely to be more stable in a corporate network compared to a social network, and this is even more so, when the collaboration affects the companies' profits and losses (P&L). However, being a peer-to-peer network, we do not discount node churning and respect the autonomy of peers.

As shown in Figure 12, setup time in both strategies increases as we increase the data size(the time is shown in log scale). However, data size has less effect on PISCES and it has a setup process with less than 10 seconds in most cases.

6.3 Effect of Different Network Configurations

Several factors, such as the update rate, churn rate, data size and network size, affect the maintenance cost of structured overlays. In this subsection, we evaluate the performance of our scheme in different network configurations. The major metric is the average message number of a node per minute. In Figure 13, we vary the update rate to test the performances of different strategies. The update rate is defined as the number of tuples that are deleted/inserted per second. In the experiment, the node uniformly picks a tuple to update. In PISCES, the index should be updated only if the updates involve the indexed data. On the contrary, in the full indexing strategy, all updates should be reflected. In Figure 13, as the update rate increases, the costs of CAN-FULL and BATON-FULL increase linearly, whereas the update rate has minor effect on CAN-PISCES and BATON-PISCES. BATON based schemes perform better than those based on CAN because the routing costs in BATON and CAN are *O(logN)* and $O(\sqrt[d]{N})$ respectively. For a 2-Dimensional CAN, the routing cost is much higher compared to other overlays. This can be solved by creating some random routing fingers, as in MURK [18].

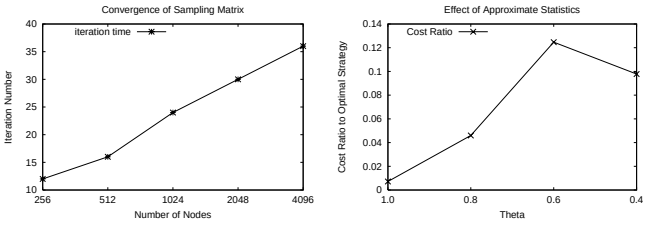
We compare the effect of churn in Figure 14. Even in a highly dynamic network when one peer leaves or joins in about every three minutes (churn rate of 0.005), PISCES remains cost effective. This confirms the robustness of PISCES.

As shown in Figure 15(a), the strategies based on full indexing incur high overheads when publishing the node's data, while PISCES generates much less cost because it only needs to publish the most popular data and a small range index. In Figure 15(b), we only show the results of PISCES to focus on their trends. The cost of PISCES grows slowly as the data size increases.

From Figure 16, we observe that as network size increases (ranging from 32 to 256), the routing cost for all strategies (partial and full indexing) also increase. However, because PISCES publishes much fewer data than its full indexing counterpart, the cost increases at a much slower rate.

6.4 Convergence of Iterative Sampling

Figure 17(a) shows how many iterations are required for the sampling matrix to converge. Figure 17(b) illustrates the effect of approximate statistics. In the experiment, each node only performs 4 iterations of sampling (about 12 iterations are required for convergence) and then a range index strategy is computed based on inaccurate statistics. The query cost ratio of the approximate index strategy to that of the optimal index strategy is computed as $\frac{RangeIndexCost}{OptimalCost} - 1$. The cost ratio is effected by the query distribution and our scheme is more robust in skewed query distribution. We



(a) Convergence (b) Cost Ratio

Figure 17: Cost and Effect of Sampling

also observe that the sampling scheme is fairly effective in mildly skewed distributions, because in such a case, query patterns can still be detected.

7. CONCLUSION

We have proposed PISCES (Peer-based system that Indexes Selected Content for Efficient Search) to reduce the maintenance cost of corporate networks, where an index is built JIT (Just-In-Time). PISCES is self-tuning to the changing of query distribution. It identifies a subset of tuples to index based on some criteria (such as query frequency, update frequency, importance of the content, etc.). We have also introduced the approximate range index, a new type of index for processing queries that cannot be fully answered by the current index. The approximate range index is also used for notifying nodes about new index construction. To support our cost estimation, we have proposed a light-weight iterative sampling scheme to collect global statistics about the whole network. It exploits common maintenance messages of the network to keep the overhead low. Experiments in PlanetLab indicate that our schemes can significantly reduce the cost of P2P networks and PISCES is effective in answering popular queries. In summary, we have proposed an efficient and practical indexing strategy that will make PDMS more acceptable as a solution for supporting enterprise-quality business processing and data sharing that involves a large amount of data and peers.

8. ACKNOWLEDGMENT

This research is in part funded by ASTAR SERC Grant 072 101 0017 of S3 project [5], National Grand Fundamental Research 973 Program of China Grand 2006CB303000 and Key Program of National Natural Science Foundation of China Grant 60533110.

9. REPEATABILITY ASSESSMENT RESULT

Code and/or data used in the paper are available at [3].

10. REFERENCES

[1] http://data.aolsearchlogs.com.
[2] http://www.planet-lab.org.
[3] http://www.sigmod.org/codearchive/sigmod2008/.
[4] http://www.tpc.org/tpch.
[5] S3:scalable,shareable and secure p2p based data management system. In *http://www.comp.nus.edu.sg/ s3p2p/.*
[6] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3), 2003.
[7] J. Aspnes and G. Shah. Skip graphs. In *SODA*, 2003.
[8] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Distributed uniform sampling in unstructured peer-to-peer networks. In *HICSS*, 2006.
[9] O. Axelsson. *Iterative solution methods*. Cambridge University Press, 1994.
[10] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, Portland, OR, Aug. 2004.
[11] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.
[12] S. Boyson and T. Corsi. Managing the real-time supply chain. In *HICSS*, 2002.
[13] A. Broder and M. Mitzenmacher. Network application of bloom filters: A survey. In *Internet Mathematics*, 2004.
[14] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: an efficient and robust p2p range index structure. In *SIGMOD*, 2007.
[15] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *P2P*, 2005.
[16] R. Dhamanka, Y. Lee, A. Doan, A. Halevy, and P. Domingos. imap: Discovering complex semantic matches between database schemas. In *SIGMOD*, 2004.
[17] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine learning approach. In *SIGMOD*, 2001.
[18] P. Ganesan, B. Yang, and H. G. Molina. One Torus to Rule Them All: Multidimensional Queries in P2P Systems. In *WebDB*, 2004.
[19] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, Sept. 2003.
[20] M. Hugos. *Essentials of Supply Chain Management*. John Wiley & Sons, Inc., 2006.
[21] H. Jagadish, B. C. Ooi, and Q. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.
[22] E. Kazumori. Selling online versus offline: theory and evidences from sotheby's. In *Proceedings of the 4th ACM conference on Electronic commerce*, 2003.
[23] A. Kothari, D. Agrawal, A. Gupta, and S. Suri. Range addressable network: A p2p cache architecture for data ranges. In *P2P*, 2003.
[24] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *VLDB*, 2004.
[25] M. Lupu, J. Li, B. C. Ooi, and S. Shi. Clustering wavelets to speed-up data dissemination in structured manets. In *ICDE*, 2007.
[26] G. S. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *PODC*, 2004.
[27] G. S. Manku, M. Naor, and U. Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized p2p networks. In *STOC*, 2004.
[28] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, load balancing and efficient range query processing in dhts. In *EDBT*, 2006.
[29] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4), 2003.
[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.
[31] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, November 2001.
[32] O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, 2004.
[33] P. Seshadri and A. N. Swami. Generalized partial indexes. In *ICDE*, 1995.
[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, San Diego, California, Aug. 2001.
[35] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4), 1989.
[36] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger. On unbiased sampling for unstructured peer-to-peer networks. In *IMC*, 2006.
[37] I. Tatarinov, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The piazza peer data management project. In *SIGMOD Record 32(3)*, 2003.
[38] R. Zhang and Y. C. Hu. Assisted peer-to-peer search with partial indexing. In *INFOCOM*, Miami, USA, March 2005.