Efficient Index-based Processing of Join Queries in DHTs

Qiang Wang q6wang@uwaterloo.ca
Reza Akbarinia* reza.akbarinia@univ-nantes.fr
M. Tamer Özsu tozsu@uwaterloo.ca

Technical Report CS-2008-15

ABSTRACT

Massively distributed applications require the integration of heterogeneous data from multiple sources. Peer-to-peer (P2P) is one possible network model for these distributed applications and among P2P architectures, distributed hash table (DHT) is well known for its routing performance guarantees. Under a general distributed relational data model, join query operator, an essential component to integrate data from multiple relational tables in centralized DBMS, can be realized over DHT to support data integration tasks in P2P networks.

In this paper, we propose an efficient and adaptive index-based join query operator over DHTs. With attribute-value storage approach, we build decentralized join indices over DHT, facilitating join query processing with reduced bandwidth consumption. Join index information regarding each join query operator is maintained across multiple indexing peers via an adaptive scheme based on peer capacities, alleviating load-balancing problem. Moreover, we develop an algorithm to access distributed indices with proven performance guarantees. Based on the join indices, a semi-join-alike approach is exploited to handle join query processing at indexing peers concurrently, effectively realizing intra-operator parallelism and decreasing query processing latency. Through theoretic analysis and extensive simulation, we demonstrate the effectiveness and efficiency of our approach.

1. INTRODUCTION

In modern massively distributed networks, data normally originate from multiple sources and their integration for large-scale sharing is an important issue. For instance, emerging mashup Web services such as Chicago Crime Map[1] and NaviTraveller[2] join their Web service data with the geographic data obtained from GoogleMaps[3] to support enhanced services. Such data integration tasks are also essential for peer-to-peer (P2P) networks, which is one possible model for modern distributed applications.

P2P systems adopt a completely decentralized approach to data sharing and thus can scale to a very large number of data and users. Initial research on P2P systems has focused on improving the performance of query routing in unstructured systems, such as Gnutella[4] and KaZaa[5], which rely on flooding mechanism. In contrast, this work led to structured systems based on distributed hash tables (DHT), *e.g.* CAN [19] and Chord [21], which provide an efficient solution for data location and lookup in large-scale networks. While there are significant implementation differences between DHTs, they all map a given key onto a peer p using a hash function and can look up p efficiently, usually in $O(\log N)$ routing hops where N is the number of peers in the network. DHTs typically provide two basic operations [21]: *put(key, data)* stores a pair *(key, data)* in the DHT using certain hash function; *get(key)* retrieves the data associated with *key* in the DHT. These operations support exact-matching queries only. Recently, much work has been devoted to supporting more complex queries in DHTs such as range queries [11], top-k queries [2] and join queries [11]. However, much more effort is needed to develop efficient solutions for complex queries, in particular for join queries, which are crucial for data integration in large scale systems.

The following example demonstrates the use of join queries in P2P data integration. Consider a P2P publication/subscription (pub/sub) system [1, 3] over distributed video data. Each peer may act as a publisher that shares home-made video or as a subscriber that intends to browse the video data of others. Suppose that subscribers declare their preferences via attribute *"pref"*, and publishers mark up their video data with semantic categorizations, denoted by attribute *"cat"*. Once the semantic categorization of certain data hosted on a publisher peer matches the preference of a subscriber peer, the latter will maintain the physical address of the former for subsequent data fetching purpose. P2P networks are usually dynamic and peers may join and leave the network arbitrarily: leaving publishers may become unavailable such that the data published at these peers are lost; conversely, peers joining the network may become publishers that share new data. In addition to network churn, publishers may change their semantic categorizations while subscribers may also update their preferences. All these behaviors may invalidate the current subscriptions. Thus peers should periodically update their subscriptions to improve user experiences and system performance. Conceptually, denote by S the logical relation that covers all the tuples on subscriber information while denote by P the logical relation that unions the tuples of publisher information. Then the subscription updating process

[1] http: //www.chicagocrime.org

[2] http://www.navitraveller.com

[3] http://maps.google.com

[4] http://www.gnutella.com

[5] http://www.kazaa.com

*This work has been done by the author as a postdoc fellow at University of Waterloo in 2008.

is equivalent to the running of the following join query, where *ipAddr* denotes the physical address of the publisher peers. Note that, in this example, the join query operation is enforced only over the metadata rather than actual video data.

```
SELECT S.ipAddr, P.ipAddr, S.pref
FROM   S, P
WHERE  S.pref = P.cat
```

In the context of PIER system [11], hash-join-based approaches have been proposed in order to support join queries over DHT. However, a major problem of these approaches is that, to evaluate each query all tuples of any relevant relations need to be re-hashed and be stored again in the DHT, which incurs very high bandwidth cost.

Realizing that data indexing has effectively facilitated query processing with reduced processing overhead in centralized data management systems, in this work we explore how to apply indexing techniques for join query processing in DHTs. There are several issues which we need to handle such as: (1) which information to employ as join query processing indices; (2) which peers maintain the join index, and how the other peers can access the index without using centralized catalogs; (3) how to adapt the part of the index maintained by a peer to its storage capacity, to avoid overloading of the peers; and (4) how to develop algorithms that efficiently evaluate join queries in DHTs by exploiting the available indices.

In this work, we tackle these issues, and propose a novel index-based approach that efficiently deals with the evaluation of join queries in DHTs. Our main contributions are as follows.

- We first propose a new mechanism for index construction in DHTs. The indices are distributed across multiple peers in the DHT. Our index construction mechanism takes into account the load balancing issues, and adjusts dynamically the number of indexing peers (*i.e.* those peers that maintain the index) based on their capacity. Moreover, the proposed mechanism guarantees efficient retrieval of the index information for query processing purpose.
- We propose an effective join query processing approach to process join queries over DHTs. Based on distributed semi-join mechanism, the approach can process join queries efficiently, especially favoring bandwidth consumption.
- Through rigorous analysis and extensive simulations, we show that the proposed approach is both effective and efficient for distributed join query processing in DHTs. The performance evaluation results, which we obtained by using TPC-H benchmark, show that our index-based approach outperforms the PIER's approaches [11] by factors of 4 and 2.5 in terms of bandwidth cost and query processing latency respectively.

The organization of the remainder of the paper is as follows. We review related work in Section 2. The problem is more formally addressed in Section 3. The construction of the distributed join index is detailed in Section 4. Section 5 discusses index-based join query processing. We introduce an adaptive dynamic partitioning scheme to maintain indexing peers and detail an algorithm for efficient index access in Section 6. Performance evaluation results are presented in Section 7 and the paper is concluded in Section 8.

2. RELATED WORK

Various distributed join query processing schemes have been developed in distributed DBMS [15]. For example, a semi-join approach has been proposed under client-server settings [7]. Its variant Bloom join has also been developed [5, 24]. These schemes depend on catalogs that can provide data placement information of all data sources, which is hard to obtain in purely decentralized P2P networks (especially under high network churn). Recently, mutant query processing approach has been proposed to handle join queries in P2P networks [16], which propagates specific query plans among peers and conducts partial query processing at each peer. The approach assumes that data placement information is explicitly specified through URL, but does not address how to obtain such information. Similarly, a join query processing scheme has been proposed in AmbientDB system [10], where each peer keeps the data placement information of all other peers, which is only suitable for small-scale networks such as home entertainment network system. Recently, self-join query operation has been studied in P2P networks [18]. However, we are interested in the more general equi-join query operations over multiple join attributes.

The problem of decentralized data integration over P2P networks has been studied in literature [12, 25], focusing on homogeneous data under a common schema. In contrast, we consider distributed data with heterogeneous schema. There are also systems that deal with the integration of heterogeneous data in P2P networks [4], which solve the problem from schema-mapping perspective. In contrast, we address how peers conduct join query processing efficiently over distributed data under heterogeneous schema.

In [13], complex query processing in DHTs has been studied, especially range and k-nearest-neighbor queries. The authors also discussed the possibility of processing multi-way join queries. In contrast, this work exclusively discusses join query processing and develops a novel index-based strategy to facilitate the query processing.

Distributed hash join approach and its variants over P2P networks have been proposed in PIER system [11]. These approaches are built over a DHT routing infrastructure. Oblivious to data distribution information over join attributes, peers ship all tuples (or their reduced forms such as Bloom-filters) to remote peers, irrespective of whether they produce final query results, potentially incurring non-trivial communication cost. Our approach solves this problem by building distributed join indices to facilitate query processing with substantially reduced bandwidth consumption.

Partition-based join query processing techniques have been widely used in distributed DBMSs [15, 9]. In P2P literature, Triantafillou *et al.* [23] have employed ``range guards" to handle join queries in parallel over the data that satisfy the corresponding range constraints, which is similar to the main idea that we propose here. However, ``range guards" replicate complete tuples instead of building compact indices as in our approach.

3. PROBLEM DEFINITION

In this work, we assume relational data. Multiple relations are managed by peers in a decentralized fashion: each peer hosts a set of tuples that belong to one or more relation. The relation here can be either physical or logical, depending on the interpretation of different scenarios. For example, in P2P database management systems, a large relational table may be horizontally fragmented among multiple peers in a top-down fashion; thus a physical global relation exists. Instead, tuples may be generated by each peer independently (*e.g.,* in a P2P pub/sub system) and a union of all the tuples in the network constitutes a logical relation from a global perspective. The approach proposed in this work does not distinguish physical relation from logical relation, which enhances its viability in various scenarios.

We assume that certain knowledge of join attributes is known by peers, either through a-prior information over data schema (*e.g.,* foreign key constraints) or being abstracted from upcoming query load. For instance, when a foreign key constraint is defined, it is probable that this key will potentially be used as a join attribute in join queries, so that we will build indices over it. Instead, when query load information is known in advance, join attributes can be directly extracted from the query statements. Join attribute information may be populated to each peer during the time it joins the overlay network. Alternatively, such information can be propagated to all peers through multicast, as employed in the PIER system [11].

Equi-join queries with multiple attributes are considered in this work. Following common practice, our approach focuses on exact join query processing with respect to "static snapshot" [6], which ignores the manipulation of query results (*e.g.,* data insertion or deletion) during query processing.

Generally, denote by p_i an arbitrary peer in the network. For simplicity, we assume that p_i hosts a horizontal fragment $F_{i,j} \in R_j$, where R_j denotes the j_{th} physical (or logical) relation. Then a join query can be represented as $R_1 \bowtie_{\alpha_1} R_2 \bowtie_{\alpha_2} \ldots \bowtie_{\alpha_i} R_i \ldots$, which is equivalent to $(\cup F_{1,j}) \bowtie_{\alpha_1} (\cup F_{2,j}) \bowtie_{\alpha_2} \ldots \bowtie_{\alpha_i} (\cup F_{i,j}) \ldots$, where α_i denotes join attribute. Since the tuples belonging to each relation R_i may be distributed across multiple peers, queries and data need to be routed across different peers for data integration purposes.

Since DHT has been widely used in existing systems and it provides proven routing performance guarantees on query and data shipping, we consider it as the underlying routing protocol in this work. In comparison to the approaches studied in the literature, the approach developed in this work is novel in using join index information to facilitate query processing (Section 4) and an adaptive and load-balanced approach to manage the and access indexing peers (Section 6).

4. DISTRIBUTED INDEX CONSTRUCTION OVER DHTs

To facilitate join query processing, join index information is deployed on peers via DHT data placement protocol. Due to the scalability guarantees of DHT, the lookup of indexing peers can be conducted efficiently, usually logarithmic to the network size (*e.g.,* via Chord routing protocol [21]).

We employ attribute-value storage technique that has been developed in [2]. Attribute-value storage stores individually the attributes that may appear in the equality predicate of an equi-join query. Thus, like database secondary indices, attribute-value storage allows checking for the existence of tuples using attribute values. Our attribute-value storage method has two important properties: (1) after retrieving an attribute value from the DHT, peers can retrieve easily the corresponding tuple of the attribute value; and (2) attribute values that are relatively "close" are stored at the same peer. To satisfy the first property, the key used for storing the entire tuple, referred to as *tuple storage key*, is stored along with the attribute value. The second property is satisfied by using the concept of *domain partitioning* as follows. Consider an attribute α and let $D\alpha$ be its domain of values. Assume there is a total order relation $<$ on $D\alpha$, (*e.g., $D\alpha$* is numeric, string, date, *etc.*) $D\alpha$ is partitioned into n nonempty sub-domains d_1, d_2, …, d_n such that their union is equal to $D\alpha$, all sub-domains are disjoint so that the intersection of any two different sub-domains is empty, and for each $v_1 \in d_i$ and $v_2 \in d_j$, if $i<j$, we have $v_1<v_2$.

Given a value v, the sub-domain to which v belongs is denoted by $sd(a, v)$, and the lower bound value of the corresponding sub-domain is denoted by $lb(sd(a,v))$. Since the range definition of a domain is subject to change, our approach does not require the number of sub-domains of an attribute to be consistently known by all peers in all the time. In contrast, the lower-bound of the domain is normally stable. Thus we assume that all peers cache this lower bound and employ it to explore other index information via the index access algorithm detailed in Section 6 shortly.

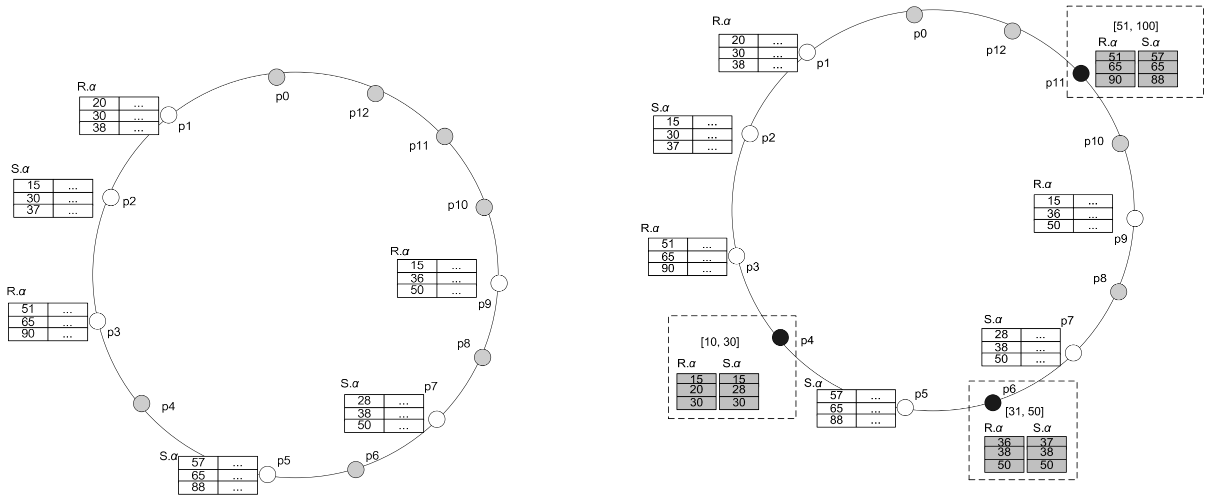
Specifically, given an attribute α of relation R and an attribute value v in a tuple $t \in R$, any peer can locally compute $sd(\alpha, v)$. The key used for storing an attribute value in the DHT is constructed as follows. Denote by $h(lb(sd(\alpha, v)))$ the key for storing v in the DHT. Thus, the attribute values that belong to the same sub-domain are stored with the same key at the same peer. The values of the attributes that are involved in attribute-value storage are stored twice in the DHT (*i.e.,* once at the source peer and the other at the indexing peer). Although this introduces potential data consistency issue, the redundancy is constrained and it is beneficial to join query processing because join index information is fully captured by the indexing peers.

```
Algorithm attribute_value_store ($R_t$, $\alpha$, $sd$)
Begin
  // allocate tuples into sub-domains
  For each tuple $t \in R_t$ Do Begin
    Let $v$ be the projected value of $t$ over $\alpha$;
    //correspond ($\alpha$,$v$) to sub-domains through a map $SB$
    $SB[lb(sd(v))]$.insert($v$);
  END;
  // store attributes on indexing peers
  Generate a combination join attribute name "A";
  For each $sb \in SB$ Do Begin
    Compute the hash id hash("A"+lb(sd(v))); // concatenation
    Locate the peer p' that is responsible for the hash id via DHT;
    Send attribute value of sb and the tuple storage key of
  the corresponding tuple to p' directly;
  End;
End;
```

Figure 1. Attribute-value Storage

Each peer issues the attribute-value storage process independently. For clarity, such a process (regarding any peer p with a relation partition R_t) is sketched in Figure 1. Note that, before the attribute-value storage process, each peer checks the join attribute names and generate a combination join attribute name if the relevant partner tables use different attribute names. For example, the join query operation may be imposed over different attribute names such as $S.\alpha$ and $R.\beta$. However, it is direct to concatenate them to produce an identical but unique join attribute name, *e.g.,* "α_β", so that the index information over this join operator from different partner tables is guaranteed to be shipped to the same (indexing) peer.

At the indexing peer, the join attribute value information of each partner table is maintained separately, so that they can be retrieved and updated independently. As illustrated in Figure 2, the original data distribution is shown in Figure 2(a), where each entry corresponds to one tuple. The layout of join index information over $R.\alpha$ and $S.\beta$ is presented in Figure 2(b). Tuple storage keys are attached with each indexing entry, which are omitted from the figure for simplicity. Note that, the tuples on a specific peer may contain join attribute values that belong to multiple sub-domains. For example, peer p_1 makes attribute-value storage at indexing peers p_4 and p_{11}. However, data locality is taken into consideration and all the attribute values that belong to the same sub-domain will correspond to the same indexing peer. This is significantly different from the hash-join-based approaches in that, in hash join approach each distinct value is potentially re-hashed to a distinguished peer in DHT.



(a) indices on indexing peers (b) data distribution

Figure 2. Attribute-value Storage

Without loss of generality, peers are shown to interconnect via a ring topology, although other topologies (*e.g.,* torus, hyper-cube and others) can be directly employed in DHTs. Also for simplicity, the long-range links that are usually established in DHTs for faster routing are omitted from the figure.

Since attribute values are replicated, consistency management needs to be enforced when original attribute values are updated at source peers. For simplicity, we do not allow indexing peers to change attribute values. Depending on applications, the consistency can be maintained in proactive or lazy fashion: in a proactive fashion, any data updates are immediately shipped and reflected at indexing peers, while in a lazy fashion, data updates can be buffered for each specific sub-domain such that the updates are materialized periodically at the indexing peers in a batch. Since each join attribute value is replicated at exactly one indexing peer, it takes $O(logN)$ routing hops (where N is network size) to ship the updated value to the responsible indexing peer for maintenance. Thus, the maintenance overhead is not significant for those applications that does not change attribute values frequently.

5. INDEX-BASED JOIN QUERY PROCESSING

5.1 Join Query Processing

Given a query q, it will be executed concurrently at all the indexing peers that store the join attribute values. Since only join-attribute values are maintained at these peers, our approach is similar to semi-join approach that has been developed in the literature [7]. However, a difference from the classic semi-join approach is that an attribute-value storage process is initially executed such that, for certain join query operators (*i.e.,* the first join query operator), all the join attribute value information is already stored at indexing peers; thus a phase to fetch join attribute values from one partner table for semi-join is unnecessary. We will first address how to handle two-way join queries that contain exactly one join attribute, and then extend the discussion to multi-way join queries that involve more than one join attributes.

Given a join query q that includes exactly one join attribute α, q is first propagated to all the indexing peers that host the join index information over α. This is easily realized via DHT routing protocol. Then the indexing peers scan the local join indices concurrently and figure out those attribute values that are contained by both partner relational tables. Denote by JS the set of these attribute values. Since the tuples from the partner relational tables join with each other only when their projected values over the join attributes are identical, all those attribute values that do not belong to JS will be pruned from subsequent join query processing without affecting the completeness of the query results. For each value belonging to JS, the corresponding tuple storage key is obtained form the local join index and the indexing

```
Algorithm semi_join($\alpha$)
BEGIN
  Compute the intersection set $JS$ from the join index over $\alpha$;
  FOR each $v \in JS$ Do Begin
    Retrieve the tuple hash ids corresponding to $v$ for both partner tables;
    Fetch the original tuples and execute join query operation over the  tuples;
  END;
END;
```

Figure 3. Semi-join Query Processing over Attribute α

peer fetches the original relational tuples from peers via DHT to produce the final results.

Conceptually, without joining the original tuples together, the projected values of the tuples over join attributes are joined first to remove unnecessary tuples from subsequent processing; then the remaining tuples are fetched to produce the join query results. Since all indexing peers obtain the query statements, it is straightforward for them to fetch only the projection of the original tuples over the *output attributes* declared by the SELECT clause in the query statement, saving bandwidth cost. For clarity, the semi-join query processing over a specific join attribute α is illustrated in Figure 3.

When the join query contains one join attribute, once an indexing peer completes producing the query results, it simply ships them to the query issuer. A union of all query results will produce the final complete query processing results, which will be proved shortly in Section 5.2.

Instead, consider multi-way join queries that involve more than one join attributes, denoted by $A=\{\alpha_1, \alpha_2, ..., \alpha_n\}$. After the completion of any join attribute α_i $(i \in [1, n))$, an intermediate relation is produced at each indexing peer after the semi-join process (presented in Figure 3). If non-empty, the intermediate relation is expected to contain the projected values (of the fetched tuples) over all the attributes that will be involved in the subsequent join query processing, including the remaining join attributes and the output attributes. Denote by I_i the set of indexing peers over α_i. Each indexing peer belonging to I_i then ships (via DHT) the intermediate tuples to those indexing peers (denoted by set I_{i+1}) that are responsible for the sub-domains over the next join attribute α_{i+1} , where the indexing peers belonging to I_{i+1} maintain temporary indices over α_{i+1}. By treating the temporary relation as a partner table that is distributed across the indexing peers belonging to

I_i, the temporary index can be built to facilitate the semi-join operation for α_{i+1}. Such semi-join-alike process iterates until all the join attributes are processed. When the processing of the final join attribute α_n completes, the relevant indexing peers simply ship the results to the query issuer, which aggregates them as the final query results. The general process of join query processing over a set (A) of join attributes is sketched in Figure 4.

It is obvious that the processing order of join query operators may decide the size of intermediate relations, thus significantly affecting the query processing performance (*e.g.,* bandwidth cost). Recall that each indexing peer maintains the join attribute values of the corresponding sub-domain. It is straightforward that the cardinality of any partner relational table over each sub-domain can be computed locally. Then the overall cardinality of relation can be computed by aggregating the cardinalities over the sub-domains. The compact cardinality information is propagated among all indexing peers. For communication among indexing peers, they can invoke index access process (see Section 6 for the algorithm) that crosses the whole domain, so that all indexing peers will be notified of the cardinality information. The obtained physical addresses can then be cached for the subsequent communication in this query processing session.

```
Algorithm join_query_processing($A$)
Begin
 FOR each attribute $\alpha_i \in A$ Do BEGIN
  invoke semi_join($\alpha_i$);
  IF $\alpha$ is the last join query attribute to process THEN BEGIN
   All indexing peers return the produced relations to the query issuer;
  ELSE
   All indexing peers ship the projected values over $\alpha_i$ to the indexing peers ($X$) responsible for $\alpha_{i+1}$ ;
   Each indexing peer in $X$ invokes semi_join($\alpha_{i+1}$) recursively;
  END;
 END;
END;
```

Figure 4. Join Query Processing

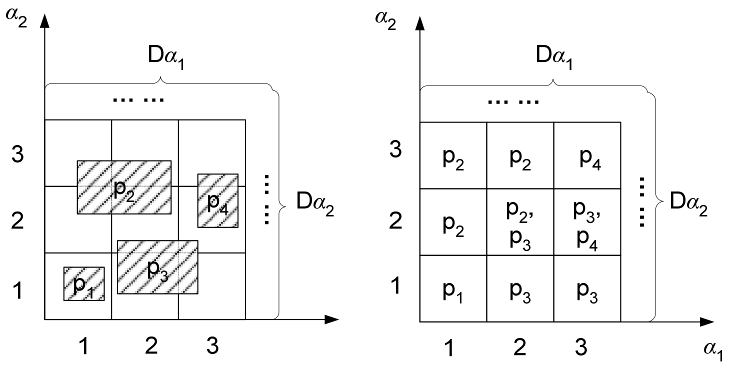
Once the overall cardinality of a relation regarding a join attribute is obtained, each indexing peer computes the order of the join attributes when available through standard join selectivity computation strategies that have been established in the literature [14]. Since all indexing peers obtain consistent cardinality information, they are able to conduct the sorting of the join query operators uniformly without requiring further communications among them.

5.2 Correctness Guarantee

The join query processing is initiated at the indexing peers, and for two-way join query, it is obvious that the results that aggregate those produced by the indexing peers are complete.

For multi-way join queries involving more than one join attributes, each indexing peer forwards the tuples belonging to intermediate relations independently to the indexing peers relevant to the subsequent join query attributes. It is needed to check whether such process fulfills the correctness guarantee, as required by the "exact join query processing semantics" introduced in Section 3.

Since the join index over each distinct join attribute is built independently, the overall join attribute space can be treated as being divided among multiple indexing peers such that each indexing peer is affiliated with the sub-domain partitions over the corresponding join attributes. Conceptually, the whole join attribute space can be modeled as a multi-dimensional space, with each dimension corresponding to one join attribute. For example, as illustrated in Figure 5, a three-way join query involving two join attributes (*e.g.,* α_1 and α_2) corresponds to a two-dimensional space with multiple partitions (over the domains $D\alpha_1$ and $D\alpha_2$) involving different indexing peers (*i.e., p_1, p_2, p_3, and p_4*). Specifically, Figure 5 (a) shows the data distribution of the join attribute values at these indexing peers while Figure 5 (b) shows the indexing peers that are potentially involved in the join query processing over each partition.



(a) data distribution (b) space partitions

Figure 5. Join Attribute Space

Since each indexing peer hosts a sub-domain over a specific join attribute, a partition of the overall space will contain all the indexing peers that are involved in the contribution of the final query results over that partition. Now we prove that, a union of the results over all partitions constitutes complete final results for multi-way join queries.

Theorem 1. *The union of query results obtained over all partitions produces complete join query results.*

Proof. Without loss of generality, consider an *l*-way join query $R_0 \rhd\lhd_{\alpha_1} R_1 \rhd\lhd_{\alpha_2} \ldots \rhd\lhd_{\alpha_{l-1}} R_i$ with join attributes, denoted by α_1, α_2, …, and α_l respectively. Generally, suppose that each join attribute domain α_i is partitioned into k_i non-overlapping sub-domains. Thus an arbitrary relation R_i is equivalent to a union of sub-relations defined over corresponding sub-domains, denoted by $R_i = R_i^{\alpha_j^1} \cup R_i^{\alpha_j^2} \cup \ldots \cup R_i^{\alpha_j^{k_j}}$, where $0 \le i \le l$ and α_j^x represents the x_{th} sub-domain over the join attribute α_j ($1 \le x \le k_j$). According to the *distributive property* of join query operation, $R_0 \rhd\lhd_{\alpha_1} R_1 \rhd\lhd_{\alpha_2} \ldots \rhd\lhd_{\alpha_{l-1}} R_i = \bigcup_{i_1\in[1,k_1],\ldots,i_l\in[1,k_l]} (R_1^{\alpha_1^{i_1}} \rhd\lhd_{\alpha_1} R_1^{\alpha_1^{i_1},\alpha_2^{i_2}} \rhd\lhd_{\alpha_2} \ldots \rhd\lhd_{\alpha_{l-1}} R_1^{\alpha_{l-1}^{i_l}})$ where $R_j^{\alpha_j^{i_j},\alpha_{j+1}^{i_{j+1}}}$ represents the sub-domain of relation R_j over both the consecutive join attributes $\alpha_j^{i_j}$ and $\alpha_{j+1}^{i_{j+1}}$. Each item $R_1^{\alpha_1^{i_1}} \rhd\lhd_{\alpha_1} R_1^{\alpha_1^{i_1},\alpha_2^{i_2}} \rhd\lhd_{\alpha_2} \ldots \rhd\lhd_{\alpha_{l-1}} R_1^{\alpha_{l-1}^{i_l}}$ is bijective (*i.e.,* one-to-one correspondent) to a disjoint partition of the data space over all join attributes, and the union of all the items cover the whole data space. Thus when the query results obtained over each partition are correct (*i.e.,* complete and sound), the union of these results produces the complete answer to the join query.

6. DYNAMIC AND ADAPTIVE INDEX CONSTRUCTION

To construct the distributed index over the DHT, in the previous sections we assumed that the domain of each attribute is partitioned by system designers. We also assumed that all peers are aware of the details of this partitioning such that they can directly send join attribute values to the indexing peers responsible for the corresponding sub-domains. In this section, we relax these assumptions by proposing a distributed indexing mechanism that dynamically and gradually partitions the attribute domain. The partitioning adapts to the capacity of peers that maintain the distributed index, and the peers are not required to have accurate knowledge about the partitioning at the moment that they join the system. We will focus on storage capacity for index information, although the scheme can be extended over other capacities such as the network bandwidth, local processing power and so on. Our mechanism provides proven performance guarantees for accessing a given value in the distributed index.

6.1 Indexing Mechanism

Our dynamic indexing mechanism works as follows. To index the values of each attribute α, there is a set of peers that maintain the index on α. This set is denoted by IM_α, referred to as the set of α's index maintainers. Each peer $p \in IM_\alpha$ has a sub-domain of α, and is responsible for indexing the values that belong to the sub-domain. Initially, there is only one peer involved in IM_α. That peer is denoted by p_0, and called the *first index maintainer*. Thus, initially we have $IM_\alpha = \{p_0\}$. For determining p_0 we use a hash function h, and apply it on the identification of attribute α. Formally, p_0 is the peer that is responsible for the key $k=h(\alpha)$ in the DHT. Initially, the sub-domain of p_0 is equal to the domain of α. However, by increasing the number of indexed data, the sub-domain of p_0 and that of other index maintainers can be split as follows. Let $p \in IM_\alpha$ be an index maintainer peer, and c_p be the maximum space capacity of p for maintaining the α's index data. When the amount of the α's index data gets higher than c_p, the peer p partitions the domain of α into t sub-domains, keeps one of them for it self and find a responsible for the other sub-domains as follows. Let $[v_0 .. v_e]$ be the sub-domain for which p is responsible, then the t new sub-domains are denoted by $[v_0 .. v_1 + \lambda_1)$, $[v_1 + \lambda_1 .. v_2 + \lambda_2)$, $[v_2 + \lambda_2 .. v_3 + \lambda_3)$, …, $[v_{t-1} + \lambda_{t-1} .. v_e)$. The values $v_1, v_2, .., v_{t-1}$ depend on the partitioning strategy. For instance, if the sub-domain is partitioned into equi-sized sub-domains, then we have $v_i = v_0 + i*((v_e - v_0)/t)$, where $1 \le i \le t-1$. The numbers λ_1, λ_2, …, λ_{t-1} are small randomly generated numbers that are used for locating appropriate peers that are responsible for the sub-domains as follows. Let [$v_i + \lambda_i$ .. $v_{i+1} + \lambda_{i+1}$) be one of the *t* new sub-domains, then the responsible for this sub-domain must the peer *q* that is responsible for the key $k=h(v_i + \lambda_i)$ in the DHT, *i.e.,* we apply the hash function on the lower bound of the sub-domain. If the capacity of *q* is not adequate for maintaining the index data which belong to [$v_i + \lambda_i$ .. $v_{i+1} + \lambda_{i+1}$) then *p* generates another random value for λ_i. It continues until finding a λ_i such that the peer that is responsible for the key $k=h(v_i + \lambda_i)$ has adequate capacity to maintain the index data that belongs to the sub-domain [$v_i + \lambda_i$ .. $v_{i+1} + \lambda_{i+1}$). Then, *p* finds the peer that is responsible for *k* by performing a lookup in the DHT, and sends to it the sub-domain [$v_i + \lambda_i$ .. $v_{i+1} + \lambda_{i+1}$) and its corresponding index data.

To partition a sub-domain, index maintainers use a strategy that uniformly partitions the sub-domain (*i.e.* the probability that a randomly chosen value falls in each of the *t* new sub-domains is the same for all of them). For this, index maintainers use histogram information that describes the distribution of attribute values. This information is provided by peers that own the attribute values, and is sent to an index maintainer before sending the attribute values to it. After each sub-domain partitioning, the index maintainer transfers the histogram information to the peers which are responsible for the new sub-domains.

For clarity, Figure 6 shows the algorithm of indexing an attribute value in the DHT using our dynamic indexing mechanism.

6.2 Accessing Data in Distributed Index

Given a value *v* and an attributeα, accessing *v* in the index of α means to find the address of the peer who is responsible for the sub-domain to which *v* belongs. Let us now describe our algorithm for accessing values in the indices. Let α be an indexed attribute. Each peer *p* has some local information about the sub-domains of α. However, this information may be out of date. Formally, at *p* there is a set $S_{p,\alpha}$ of couples *(q,* [v_i .. v_j]*)* such that each [v_i .. v_j] is considered by *p* as one of the of α's sub-domains, and *q* is the address of the peer that *p* knows as the responsible for [v_i .. v_j]. Initially, when *p* joins to the DHT, it has no information about the sub-domains of α, thus $S_{p,\alpha} = \Phi$. After each access to α's index, *p* obtains new information about the sub-domains of α and improves $S_{p,\alpha}$. To access a value *v* in the α's index, *p* performs as follows. If $S_{p,\alpha} \neq \Phi$, then *p* searches in $S_{p,\alpha}$ and chooses the smallest sub-domain to which *v* belongs, as well as the

```
Algorithm Index_Value (v, p, q)
// This algorithm Indices at peer p the value v which is owned by peer q;
Begin
  Let $D_p$ be the set of index data stored at p;
  Let $c_p$ be the space capacity of p;
  $D_p = D_p + \{(v, q)\}$;
  If $sizeOf(D_p) \le c_p$ then return;
  Else Begin
    Let sd be the sub-domain for which p is responsible;
    Partition sd into t  sub-domains $sd_1, sd_2, \ldots, sd_t$, and find t-1 peers to
       be responsible for $sd_2, \ldots, sd_t$;
    For i=2  to  t  Do Begin
      Remove from $D_p$ all couples (v', q') such that $v' \in sd_i$
      Send the removed couples to the peer that is responsible for $sd_i$;
    End;
    Return partitioning information to q;
  End;
End;
```

Figure 6. Indexing an Attribute Value in the DHT

peer that is responsible for the sub-domain, say q_0. Otherwise, if $S_{p,\alpha} = \Phi$, *p* generates a key $k=h(\alpha)$ and sets q_0 to be the peer that is

responsible for *k* in the DHT (*i.e.,* q_0 is set to the first index maintainer). After determining q_0, the peer *p* sends a message to it and asks it about the sub-domain to which *v* belongs, say *sd(v)*. If q_0 is still the responsible for *sd(v)*, it returns the address of itself to *p*. Otherwise, i.e. if q_0 has partitioned its sub-domain, it forwards *p*'s request to the peer, say q_1, which q_0 knows as the responsible for *sd(v)*. Similarly, if q_1 has partitioned its sub-domain and the value *v* does not belong to the sub-domain maintained by q_1, it forwards the request to the peer that it knows as the responsible for the requested sub-domain. This forwarding continues until the *p*'s request reaches to the peer q_u that is responsible for the sub-domain to which *v* belongs. The peers, which are over the path from q_0 to q_u, attach their information about sub-domains (involving the sub-domains and the peers which are responsible for them) to the message that is forwarded to q_u. The peer q_u returns this information to *p* as well as the address itself. The sent information is exploited by *p* in order to improve $S_{p,\alpha}$.

Assuming equal capacity for peers, the following lemma shows an upper bound on the number of hops for accessing a value in the distributed index.

Lemma 1. *Assume equal capacity for all peers and a uniform sub-domain partitioning, then the number of hops for accessing an attribute value in the distributed index is O(Log(n/c)) where c is the capacity of each peer and n is the total number of (distinct) attribute values.*

Proof. Let $|s|$ be average space needed for indexing an attribute value, the total space needed for indexing all values is $n*|s|$. The capacity of each peer is *c*, thus a peer *p* partitions its sub-domain only when the size of index data at *p* is higher than *c*. Since each sub-domain is partitioned uniformly into *t* new sub-domains, the minimum amount of the index data which belongs to each created sub-domain is *c/t*. Therefore, the total number of created sub-domains for the attribute is less than or equal to $m = (n*|s|) / (c/t)$. For accessing a value in the distributed index, we need to find the sub-domain of the value (and the peer which is responsible for the sub-domain). In the worst case, we must search the requested sub-domain in a search space including *m* sub-domains (and *m* peers). In the worst case, the search starts at the first indexer maintainer. The requester peer has no information about the partitions. Implied by uniform partitioning, at each hop the search space is divided by *t*. Thus, at most after $(Log_t (m))$ hops, the size of search space becomes equal to 1. In other words, the algorithm of accessing a value in the distributed index is done in $O(Log_t ((n*|s|*t) / (c)))$. Assuming $|s|$ to be a constant, the algorithm is done in $O(Log_t (n/c)$ which is $O(Log (n/c))$. □

Without assuming equal capacity for peers, the following theorem shows that an upper bound on the on the number of hops to access a value in the distributed index.

Theorem 2. *Under uniform sub-domain partitioning scheme, the number of hops for accessing a value in the distributed index is* $O(Log(n/c_{min}))$ *where* c_{min} *is the minimum capacity of peers and n is the total number of attribute values.*

Proof. Let $|s|$ be average space needed for indexing an attribute value, then it can be easily shown that the maximum number of created sub-domains for the attribute is $(n*|s|) / (c_{min} /t)$. Thus, In a similar way as in the proof of Lemma 1, we can show that the algorithm of accessing a value in the distributed index is done in $O(Log(n/c_{min}))$.

7. PERFORMANCE EVALUATION

7.1 Experimental Setting

We employ the p2psim[6] discrete event simulator to simulate P2P networks with up to 4000 peers. We generate a two-dimensional grid space of length 100 milliseconds (In this experiment, simulation time is used, without affecting the validity of the results) at each side, and distribute peers uniformly within the space. The data transferring latency between peers is estimated as the Euclidean-distance between the corresponding coordinates within the space.

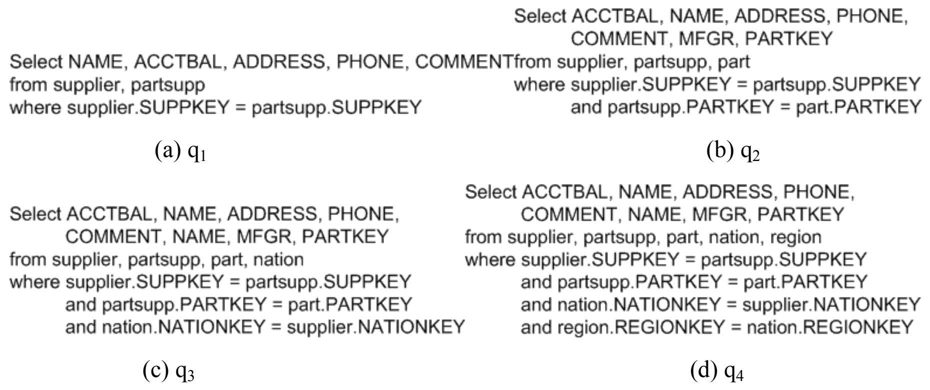
We use TPC-H benchmark[7] to generate data load since it supplies multi-join queries. We run the benchmark data generator (*i.e.,* dbgen) with scale factor set to 1, creating a database of eight relational tables. Given a join query, each peer randomly chooses a table from the database in the following way: (1) the schema of the chosen table includes the join attributes of the query; (2) each relational table *T* in the database is chosen randomly with a weighted probability proportional to the number (denoted by $|T|$) of tuples in *T*; (3) once a table (*e.g., T*) is chosen, the number of tuples to be obtained by each peer equals $min\{150, 0.1* |T|\}$; with respect to a network of 4000 peers, the overall number of tuples is 599,701 (With TPC-H, the total data volume in this simulation is over 76 MB, which is relatively small but does not affect the validity of the evaluation because each peer stores sufficient tuples and tuple size decides the overall data volume), which is sufficient to simulate the join query processing of various approaches; and (4) the obtained tuples by each peer follow the data distribution mechanisms to be described shortly. Although each peer hosts a horizontal fragment of a single table in this simulation, our approach can be easily extended to allow each peer managing data from multiple tables.

To obtain tuples from a table, we consider both uniform random distribution and skewed distribution. Under the former distribution, each peer simply chooses tuples from the corresponding table by following uniform random distribution. However, in real applications such as location-aware distributed services, geographically proximate data are often clustered on peers; thus we also consider skewed data distribution: each peer chooses an initial tuple (uniform) randomly from its chosen table and obtains a specified number of nearest neighbors of the initial tuple (with respect to the join attributes) from the same table.

We evaluate four join queries with up to four join attributes, as shown in Figure 7. The two-way and five-way join queries (*i.e.,* Figure 7(b)(d)) are transcribed from the queries supplied by the TPC-H benchmark, while the other three-way and four-way join queries (*i.e.,* Figure 7(a)(c)) are manually created since they are not provided by the benchmark. We only consider low dimensionality in this work (*e.g.,* $d <= 4$), which is sufficient for many practical applications. For example, location-aware services in mobile computing systems may require only a few join attributes (*e.g.,* the longitude and latitude corresponding to locations). For higher dimensionality (*e.g.,* tens or hundreds of features as in images and video), a feasible solution may be to use dimension-reduction techniques [17].

[6] http://pdos.csail.mit.edu/p2psim/

[7] http://www.tpc.org/tpch/



```
Select NAME, ACCTBAL, ADDRESS, PHONE, COMMENT
from supplier, partsupp
where supplier.SUPPKEY = partsupp.SUPPKEY
```

(a) q_1

```
Select ACCTBAL, NAME, ADDRESS, PHONE,
          COMMENT, MFGR, PARTKEY
from supplier, partsupp, part
where supplier.SUPPKEY = partsupp.SUPPKEY
          and partsupp.PARTKEY = part.PARTKEY
```

(b) q_2

```
Select ACCTBAL, NAME, ADDRESS, PHONE,
          COMMENT, NAME, MFGR, PARTKEY
from supplier, partsupp, part, nation
where supplier.SUPPKEY = partsupp.SUPPKEY
          and partsupp.PARTKEY = part.PARTKEY
          and nation.NATIONKEY = supplier.NATIONKEY
```

(c) q_3

```
Select ACCTBAL, NAME, ADDRESS, PHONE,
          COMMENT, NAME, MFGR, PARTKEY
from supplier, partsupp, part, nation, region
where supplier.SUPPKEY = partsupp.SUPPKEY
          and partsupp.PARTKEY = part.PARTKEY
          and nation.NATIONKEY = supplier.NATIONKEY
          and region.REGIONKEY = nation.REGIONKEY
```

(d) q_4

Figure 7. Join Queries

For comparison purposes, as the baseline approaches we re-implemented the distributed hash join approach (proposed in the context of the PIER system [11]) and its variants including symmetric semi-join approach, and Bloom-filter-based semi-join approach. The design principles of these approaches are briefly described below. Briefly, regarding a specific join attribute, suppose that the number of distinct join attribute value equals to *m*. Since DHT employs perfect hashing functions such as SHA-1[8], each tuple is expected to correspond to a unique random id in the hash function space. Thus in the re-hashing process that is enforced by the hash join mechanism, each tuple with a distinct projected join attribute value is expected to consume a distinct DHT routing process that costs *O(logN)* messages, where *N* denotes network size. Thus on average, the number of messages of the re-hashing process is bounded by *O(mlogN)*, linearly depending on data load size (*m*). Since in P2P networks, each peer is expected to share data (*e.g.,* stimulated by protocols such as tit-for-tat strategies[9]) while P2P networks usually contain a large number of peers, the data volume tends to be huge, which may incur scalability problem regarding bandwidth cost (proportional to the number of messages) when hash-join approach is applied. More specifically, the hash-join approach and its variants are reviewed below.

Hash-Join Approach. A join query *Q* is initially populated on all peers. Each peer *p* scans local tuples. If a local tuple *t* contains join attributes (*e.g.,* $\alpha_1, \alpha_2, ...,$ and α_n), *p* will ship *t* via DHT to a (remote) peer p_0 that is responsible for the hash id $h(Concat(\alpha_1, v_1, \alpha_2, v_2, ..., \alpha_n, v_n))$, where v_i is the value of *t* over attribute α_i ($i \in [1, n]$), *"Concat"* is a function that concatenates the components in sequence (*i.e.,* $\alpha_1, v_1, \alpha_2, v_2, ..., \alpha_n, v_n$), and *h* is the DHT hash function. The ordering of join attributes for concatenation is consistently predefined among all peers. Peer p_0 then builds a hash table for each partner table over the join attributes. Thus hash join operation is conducted over all p_0 peers independently and concurrently.

Symmetric Semi-join Approach. This approach manages to reduce bandwidth consumption by applying semi-join approach. Each peer *p* initially conducts a projection operation of their tuples over join attributes. Then the projected values are re-hashed via the hash-join approach described above. Then only those tuples that correspond to semi-join results are retrieved for producing final query results. This approach is close to the approach developed in this work. However, semi-join requires extra communication phases to complete the query processing, potentially incurring higher query processing latency.

Bloom-Join Approach. This approach applies Bloom filter technique [8]. Each peer *p* initially sends a Bloom filter bf over all its tuples (involving join attribute values) and re-hashes bf to a (remote) peer, where all Bloom filters are aggregated for each involved relational table. The aggregated Bloom filter is multicast among the peers that host the tuples of the partner table, such that those tuples that never produce query results are pruned through Bloom filtering locally at each peer. Each peer then re-hashes the remaining tuples through the hash-join approach that is addressed above and the final results are returned to query issuer. The Bloom filtering may help eliminate unnecessary data shipping. However, the propagation of the Bloom filters itself may increase the bandwidth consumption. Thus the overall bandwidth cost may not necessarily be saved in practice, as demonstrated in the subsequent performance evaluation.

We choose Chord protocol [21] as the underlying DHT-based routing mechanism, which is supplied by p2psim. Due to lack of data distribution information, the order of join attributes is randomly chosen and all join query operators are conducted in a non-blocking fashion.

7.2 Query Processing Performance

We evaluate bandwidth consumption and query execution latency (including both routing and local processing latency) under the uniform random and skewed data distribution schemes. Without loss of generality, we set the partition number over each join attribute dimension to be 3. This leads to 3 partitions for q_1 and 81 partitions for q_4, which is sufficient to test the behavior of our approach under a number of

[8] http://www.itl.nist.gov/fipspubs/fip180-1.htm

[9] http://www.bittorrent.org/bittorrentecon.pdf

data space partitions. The sensitivity of the query processing performance over other numbers of partitions will be addressed shortly. For presentation, in the following figures, we name our approach as ``index-based", the Bloom-filter-based semi-join approach as ``BF-semi", the semi-join approach as ``semi-join", and the distributed hash join approach as ``plain".

The bandwidth consumption of join query processing is shown in Figure 8(a), which illustrates that our approach is more efficient than the baseline approaches. Figure 8(b) demonstrates the latency of the query processing including local processing cost. The results show that our approach take less time to complete due to the exploitation of parallelism and in-advance pruning of irrelevant peers from query processing. Without loss of generality, the latency of processing each tuple is assumed to be 1 millisecond and each peer processes all tuples sequentially. Experiments with other processing latency (*e.g.,* 10 milliseconds per tuple) show similar results and we omit the details here for brevity.

Similar experimental results are obtained under the skewed data distribution, as shown in Figure 8(c)(d). In comparison to the uniform random data distribution setting, both bandwidth and query processing latency of our approach are much lower because the data that are proximate over join attributes are clustered on a subset of peers in the network such that both the number of groups and the group cardinalities may be much smaller, potentially lowering query processing cost.

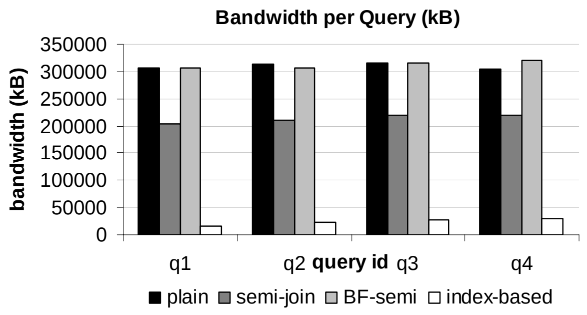
7.3 Sensitivity Test

When a different number of sub-domains are employed, the peer responsible for the corresponding sub-domain may execute different workloads. The query processing cost (*i.e.,* bandwidth and latency) with 5000 peers with different sub-domain numbers under different data distributions is demonstrated in Figure 9.

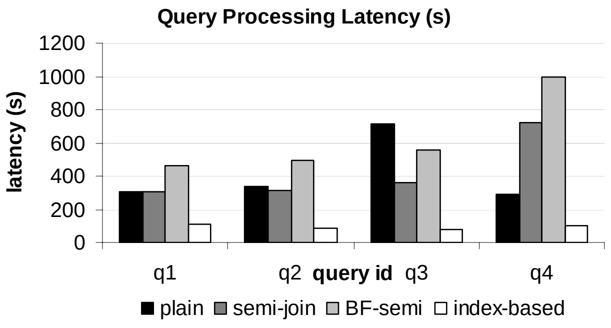
Specifically, the bandwidth costs are shown in Figure 9 (a)(c), which do not increase significantly with the growth of sub-domain number because the volume of data involved in the query processing does not change. The slight increase of the bandwidth is due to the routing overhead: when there are more sub-domains, peers are expected to contact with more indexing peers for attribute-value storage, potentially increasing bandwidth cost.

Without loss of generality, we assume that the local processing cost is linearly proportional to the number of tuples that are involved in the local join query processing. The processing latency when the processing of each tuple takes 1 millisecond is shown in Figure 9(b)(d) under different data distributions. As shown in Figure 9(c)(e), similar results are obtained for the setting when the processing of each tuple takes 10 milliseconds. Each peer processes all tuples sequentially so that the local processing latency is proportional to the number of tuples being processed.

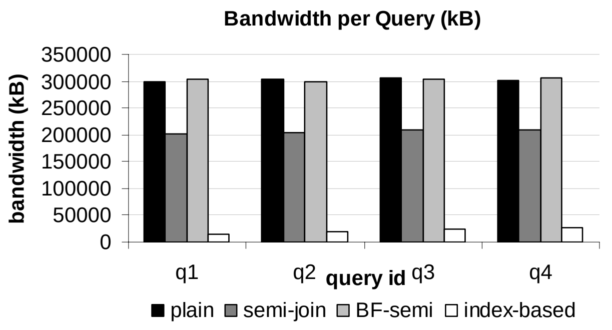
It is obvious that, under the uniform data distribution, the query processing latency tends to decrease with a larger number of sub-domains. The tradeoff is that, when the number of sub-domains increases, the maintenance overhead may increase correspondingly. However, under skewed data distribution, query processing latency increases when the sub-domain number grows. This is because, the latency consumed by the attribute-value storage process may increase when there are more sub-domains (each peer needs to ship tuples sequentially to more indexing peers), and the reduction of the query processing latency itself under skewed data distribution does not cross off the increase of the latency incurred by the attribute-value storage.



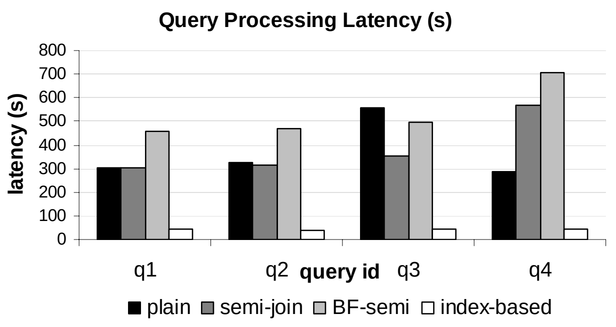
(a) Bandwidth under uniform data distribution



(b) Latency under uniform data distribution

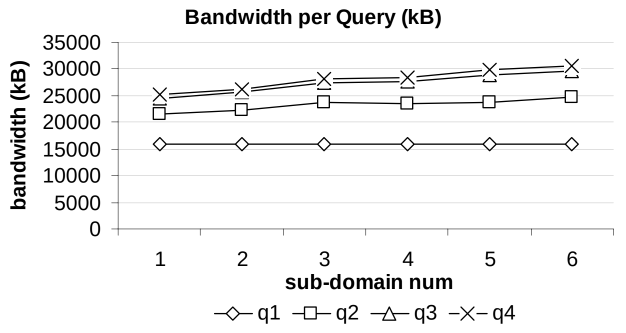


(c) Bandwidth under skewed data distribution

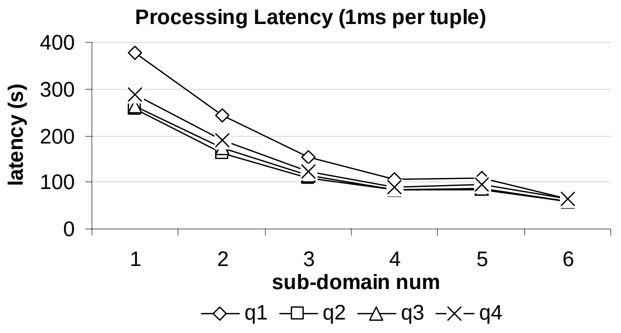


(d) Latency under skewed data distribution

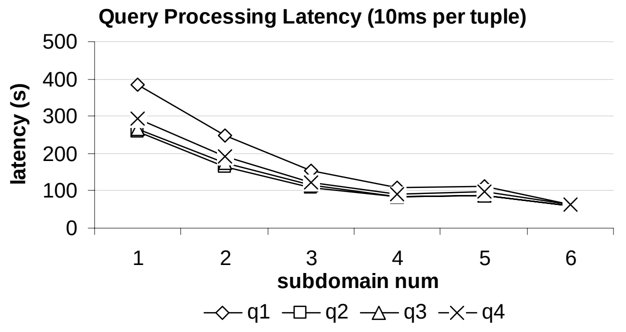
Figure 8. Performance Comparison



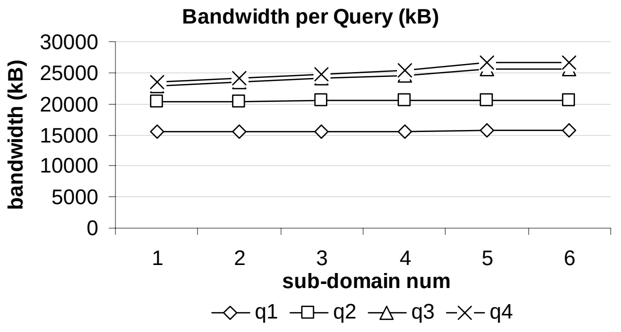
(a) Bandwidth under uniform data distribution



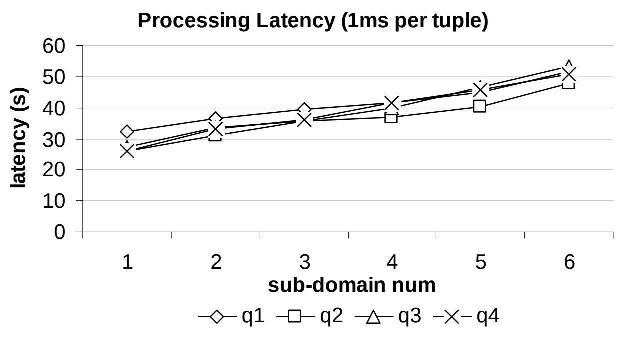
(b) Latency under uniform data distribution (1ms)



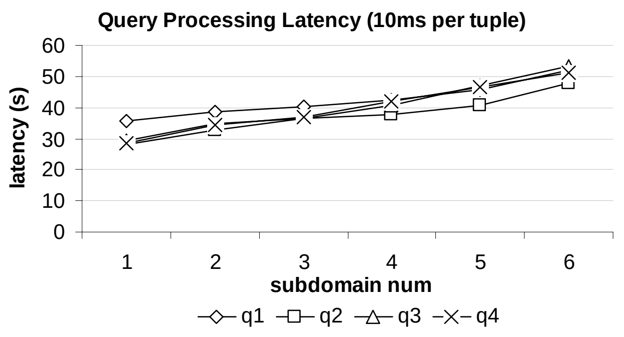
(c) Latency under uniform data distribution (10ms)



(d) Bandwidth under skewed data distribution



(e) Latency under skewed data distribution (1ms)



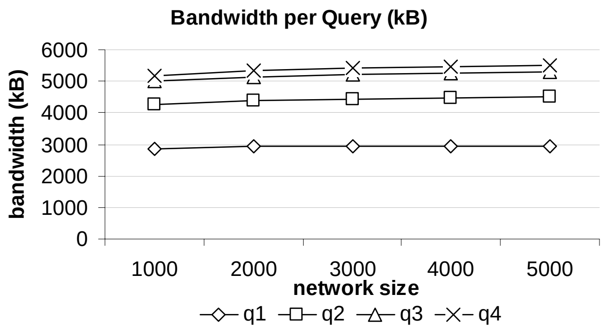
(f) Latency under skewed data distribution (10ms)

Figure 9. Sensitivity Test

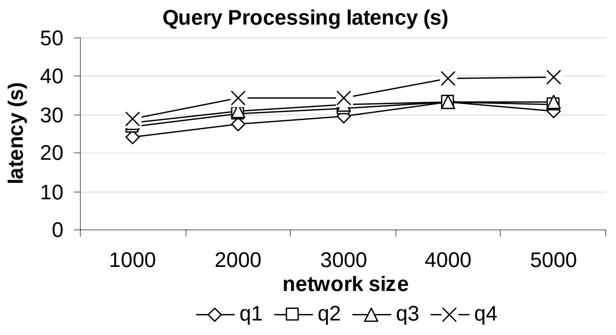
7.4 Scalability Test

We evaluate the bandwidth and processing latency per query with respect to increasing network size. Each peer in the network hosts data based on uniform data distribution and skewed data distribution respectively.

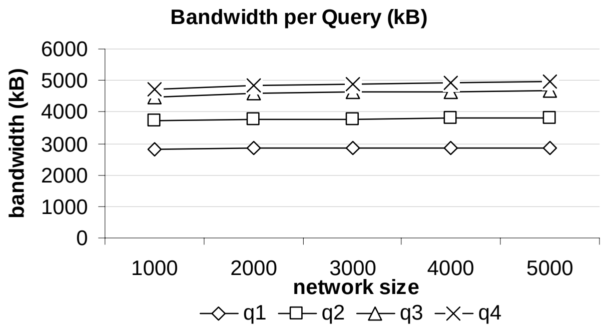
The results (in Figure 10) show that, both the bandwidth and query processing latency do not increase dramatically when network size increases, which demonstrate the scalability of the proposed approach. It is noted that, under the skewed data distribution, the query processing latency of some queries (*e.g., q_2*) may decrease when network increases. Since query processing latency may consist of both DHT routing latency and local computation latency, when peers that host indices are different, the DHT routing latency varies.



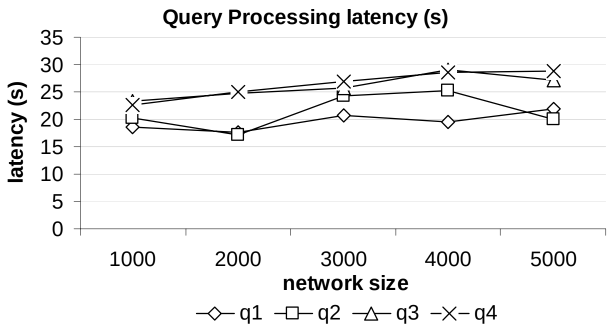
(a) Bandwidth under uniform data distribution



(b) Latency under uniform data distribution



(c) Bandwidth under skewed data distribution



(d) Latency under skewed data distribution

Figure 10. Scalability Test

8. CONCLUSION

In massively distributed P2P networks, the integration of multiple data sources supports various applications. In this paper, we addressed the problem of join query processing in DHTs, and developed an effective approach to employ join index information to resolve join query processing in a purely decentralized fashion.

Through attribute-value storage approach, we deploy join index information over DHT, facilitating join query processing with reduced bandwidth consumption. Join index information is maintained across multiple peers via a dynamic partitioning scheme, which is guaranteed to be scalable. Moreover, the approach adjusts the number of indexing peers based on peer capacity, alleviating load-balancing and enhancing the adaptivity of the approach. The correctness of our approach regarding the join query results is proved and extensive simulation demonstrates the effectiveness and efficiency of our approach. Simulations regarding TPC-H benchmark join queries show that, our index-based approach outperforms the existing hash-join-based approaches [11] by factors of 4 and 2.5 in terms of bandwidth cost and query processing latency respectively.

As mentioned, join index may also capture other information such as the correlation among join attribute values or functional dependency. Query optimization strategies that exploit such information may improve the join query processing performance even further, which will be explored in the future. Moreover, we intend to develop efficient index-based solutions for join query processing in unstructured P2P systems.

REFERENCES

[1] I. Aekaterinidis and P. Triantafillou. Substring matching in P2P publish/subscribe data management networks. In *Proc. Int. Conf. on Data Engineering*, pages 1390-1394, 2007.

[2] R. Akbarinia, E. Pacitti, and P. Valduriez. Processing top-k queries in distributed hash tables. In *Parallel Processing, 13th International Euro-Par Conference*, pages 489-502, 2007.

[3] E. Anceaume, M. Gradinariu, A. K. Datta, G. Simon, and A. Virgillito. A semantic overlay for self-* peer-to-peer publish/subscribe. In *Proc. 26th Int. Conf. on Distributed Computing Systems*, page 22, 2006.

[4] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion project: from data integration to data coordination. *ACM SIGMOD Record, 32(3)*:53-58, 2003.

[5] E. Babb, Implementing a relational database by means of specialized hardware," *ACM Trans. Database Syst., 4(1)*, pp. 1-29, 1979.

[6] W. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top-k retrieval in peer-to-peer networks. In Proc. *Int. Conf. on Data Engineering*, pages 174-185, 2005.

[7] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Query processing in a system for distributed databases, *ACM Trans. Database Syst., 6(4)*, pp. 602-625, 1981.

[8] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422-426, 1970

[9] D. J. DeWitt and J. Gray, Parallel database systems: The future of high performance database processing, in *Communications of the ACM (CACM)*, 1992.

[10] W. Fontijn and P. A. Boncz, AmbientDB: P2P data management middleware for ambient intelligence, in *Proc. Workshop on Middleware Support for Pervasive Computing (PerWare)*, 2004, pp. 203-207.

[11] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, Querying the internet with PIER, in *Proc. 29th Int. Conf. on Very Large Data Bases*, 2003, pp. 321-332.

[12] S. Kashyap, S. Deb, K. Naidu, R. Rastogi, and A. Srinivasan, Efficient gossip-based aggregate computation, in *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* 2006, pp. 308-317.

[13] B. Liu, W.-C. Lee, and D. L. Lee, Supporting complex multi-dimensional queries in P2P systems, in *Proc. 25th Int. Conf. on Distributed Computing Systems*, 2005, pp. 155-164.

[14] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implmentations*. Prentice-Hall, 2000.

[15] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, Second Edition. Prentice-Hall, 1999.

[16] V. Papadimos, D. Maier, and K. Tufte, Distributed query processing and catalogs for peer-to-peer systems, in *First Biennial Conference on Innovative Data Systems Research*, 2003.

[17] M. Perpinan, *A review of dimension reduction techniques*, University of Sheffield, Tech. Rep. CS-96-09, 1997.

[18] T. Pitoura and P. Triantafillou, Self-join size estimation in P2P data systems, in *Proc. Int. Conf. on Data Engineering*, 2008.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161-172, 2001.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329-350, 2001.

[21] 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. In *Proc. ACM SIGCOMM*, pages 149-160, 2001.

[22] D. Stutzbach and R. Rejaie: Improving Lookup Performance Over a Widely-Deployed DHT. In *Proc. INFOCOM* 2006

[23] P. Triantafillou and T. Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In Proc. *First International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, pages 169-183, 2003.

[24] P. Valduriez and G. Gardarin, Join and semijoin algorithms for a multiprocessor database machine, *ACM Trans. Database Syst., 9(1)*, pp. 133-161, 1984.

[25] M. Zaharia and S. Keshav, Gossip-based search selection in hybrid peer-to-peer networks, in *Peer-to-Peer Systems, First International Workshop*, 2006.

[26] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. *Tapestry: An infrastructure for fault-tolerant wide area location and routing.* Technical report, University of California, Berkeley, 2001.