LinkedPeers: A Distributed System for Interlinking Multidimensional Data

Athanasia Asiki, Dimitrios Tsoumakos, and Nectarios Koziris

School of Electrical and Computer Engineering
National Technical University of Athens, Greece
{aassiki,dtsouma,nkoziris}@cslab.ece.ntua.gr

Abstract. In this paper we present *LinkedPeers*, a distributed system designed for efficient distribution and processing of multidimensional hierarchical data over a Peer-to-Peer overlay. he system design aims at incorporating two important features, namely large-scale support for partially-structured data and high-performance, distributed query processing including multiple aggregates. To achieve that, LinkedPeers utilizes a conceptual chain of DHT rings that stores data in a hierarchy-preserving manner and is able to adjust both the granularity of indexing and the amount of pre-computation according to the incoming workload. Extensive experiments prove that our system is very efficient achieving over 85% precision in answering queries while minimizing communication cost and adapting its indexing to the incoming queries.

1 Introduction

Our era can be characterized by an astonishing explosion in the amount of produced data, at a rate even bigger than Moore's law [1]. Market globalization, business process automation, new regulations, the increasing use of sensors, all mandate even more data retention from companies and organizations as a brute force method to reduce risk and increase profits. In most applications, data are described by multiple characteristics (or *dimensions*) such as time, customer, location, etc. Dimensions can be further annotated at different levels of granularity through the use of *concept hierarchies* (e.g., `Year`→ `Quarter` → `Month`→ `Day`). Concept hierarchies are important because they allow the structuring of information into categories, thus enabling its search and reuse.

Besides the well-documented need for efficient analytics, web-scale data poses extra challenges: While size is the dominating factor, the lack of a centralized or strict schema is another important aspect: Data without rigid structures as those found in traditional database systems are provided by an increasing number of sources, for example data produced among different sources in the Web [2]. The distribution of data sources renders many centralized solutions useless in performing on-line processing. Consequently, any modern analytics platform is required to be able to perform efficient analytics tasks on distributed, multi-attribute structured data without strict schema.

In this paper, we present the *LinkedPeers* system that efficiently stores and processes data described with multiple dimensions, while each dimension is organized by a concept hierarchy. We choose a Distributed Hash Table (DHT) substrate to organize *any* number of commodity nodes participating in LinkedPeers. Data producers can individually insert and update data to the system described by a predefined group of concept

A. Hameurlain et al. (Eds.): DEXA 2011, Part II, LNCS 6861, pp. 527–543, 2011.
© Springer-Verlag Berlin Heidelberg 2011

hierarchies, while the number of dimensions may vary for each data item. Queries are processed in a fully distributed manner triggering adaptive, query-driven reindexing and materialization mechanisms to minimize communication costs.

The motivation behind the design of LinkedPeers is to provide a large-scale distributed infrastructure to accommodate collections of partially-structured data. In contrast to approaches where both data and their relationships are pre-defined by rigid schemas, we intend to support a higher degree of freedom: System objects are described by d dimensions, each of which is further annotated through a corresponding concept hierarchy. LinkedPeers does not require that each inserted fact be described by values for all dimensions. On the contrary, it attempts to fully support it and not restrict the ability to efficiently process it.

LinkedPeers manages to preserve all hierarchy-specific information for each dimension, using a tree-like data structure to store data and interlinking trees among different dimensions. A natural ordering of the dimensions that stems from their importance, query skew, etc, yields to a corresponding organization of the DHT layer: LinkedPeers comprises of multiple 'virtual' overlays, one for each dimension. This strategy results with each object being split into d parts and ending up in nodes of the *primary* and *secondary* rings. Trees at secondary rings maintain information towards related trees of the primary ring.

The purpose of this design is to couple the operational autonomy of the primary ring with a powerful meta-indexing structure integrated at the secondary rings, allowing our system to return fast aggregated results for the queried values by minimizing the communication cost. By allowing adaptive result caching and precomputation of related queries, this efficacy is further enhanced.

The proposed scheme enables the processing of complex aggregate queries for any level of any dimension, such as: "*Which `Cities` belong to `Country` 'Greece' ?*" or "*What is the `population` of `Country` 'Greece' ?*" or "*Which `Cities` of `Country` 'Greece' have `population` above 1 million in `Year` '2000'?*", considering that the `Location` and `Time` hierarchies describe a numerical fact for population. The enforced indexing allows to find the location of any value of any stored hierarchy without requiring any knowledge, while aggregation functions can be calculated on the nodes that a query ends up.

To summarize, this work presents the LinkedPeers system which offers the following innovative features:

- A complete storage, indexing and query processing system for data described by an arbitrary number of dimensions and annotated according to defined concept hierarchies. LinkedPeers is able to perform efficient and online incremental updates and maintain data in a fault-tolerant and fully distributed manner.
- A query-based materialization engine that pro-actively precomputes relevant views of a processed query for future reference.
- Query-based adaptation of the indexing granularity of its indexing according to incoming requests.

Finally , to support our analysis, we present a thorough performance evaluation in order to identify the behavior of our scheme under a large range of data and query loads.

2 *LinkedPeers* System Description

2.1 Notation and Definitions

Data items are described by tuples containing values from a data space domain D. These tuples are defined by a set of d dimensions $\{d_0,..,d_{d-1}\}$ and the actual fact(s). Each dimension d_i is associated with a concept hierarchy organized along L_i levels of aggregation ℓ_{ij}, where j ($j \in [0, L_{i-1}]$) represents the *j-th* level of the *i-th* dimension. We define that ℓ_{ik} lies *higher* (*lower*) than ℓ_{il} and denote it as $\ell_{ik} < \ell_{il}$ ($\ell_{ik} > \ell_{il}$) *iff* $k < l$ ($k > l$), i.e., if ℓ_{ik} corresponds to a less (more) detailed level than ℓ_{il} (e.g., $Month < Day$).Tuples are shown in the form:

$$\langle v_{0,0},\ldots,v_{0,L_0-1},\ldots,v_{d-1,0},\ldots,v_{d-1,L_{d-1}-1},f_0,\ldots\rangle$$

where $v_{i,j}$ represents the value of the j-th level of the i-th dimension. Note also that any *value-set* $(v_{i,0},\ldots,v_{i,L_i-1})$ for the i-th dimension may be absent from a tuple and that $fact_j$ may be of any type (e.g., numerical, text, vector, etc). Level ℓ_{i0} is called the *root level* for the i-th dimension and its hashed value v_{i0} is called *root key*. The values of the $\ell_{iL_{i-1}}$ are also referred to as *leaf values*.

The values of the hierarchy levels in each dimension are organized in tree structures, one per root key. Without loss of generality, we assume that each value of ℓ_{ij} has at most one parent in $\ell_{i(j-1)}$. To insert tuples in the multiple overlays, one level from each dimension hierarchy is chosen; its hashed value serves as its *key* in the underlying DHT overlay. We refer to this level as *pivot level* and to its hashed value as *pivot key*. The pivot key that corresponds to the primary dimension (or *primary ring*) is called the *primary key*. The highest and lowest pivot levels of each hierarchy for a specific root key are called *MinPivotLevel* and *MaxPivotLevel* respectively.

The value-set of a dimension along the aggregated fact are organized as nodes of a *tree structure*, which contributes to the preservation of semantic relations and search. Figure 1 describes our running example. The shown tuples adhere to a 3-dimensional schema. The primary dimension is described by a 4-level hierarchy, while the other two are described by a 3-level and a 2-level hierarchy respectively. Note that the last two tuples do not contain values in d_1 and d_2 respectively. The selected pivot level for the primary dimension is ℓ_{02} and thus all the shown tuples have the same pivot key in the primary dimension. All the value-sets in each dimension are organized in tree-structures with common root keys.

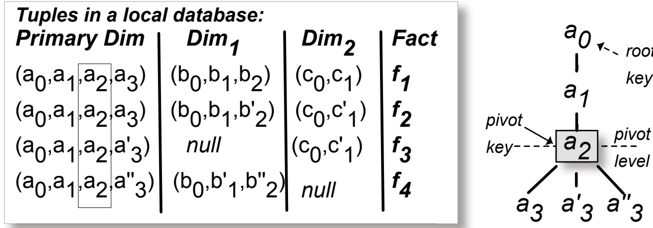
The basic type of query supported in LinkedPeers is of the form:

$$q = (q_{0k},\ldots,q_{ij},\ldots,q_{(d-1)m})$$

over the fact(s) using an appropriate aggregate function. By q_{ij} we denote the value for the *j-th* hierarchy level of the *i-th* dimension which can also be the special '*' (or *ALL*) value.

2.2 Data Insertion

Our system handles both bulk insertions and incremental updates in a unified manner. As our design implies one virtual overlay per dimension, one key (using the SHA1 hash function for instance) for a selected pivot value of each dimension is generated.


Tuples in a local database:

Primary Dim	Dim_1	Dim_2	Fact
(a_0,a_1,a_2,a_3)	(b_0,b_1,b_2)	(c_0,c_1)	f_1
(a_0,a_1,a_2,a_3)	(b_0,b_1,b'_2)	(c_0,c'_1)	f_2
(a_0,a_1,a_2,a'_3)	null	(c_0,c'_1)	f_3
(a_0,a_1,a_2,a''_3)	(b_0,b'_1,b''_2)	null	f_4

Fig. 1. A group of tuples with various value combinations among dimensions and the resulted tree structure for the primary dimension

During data insertions, the information about the pivot value is vital (only for initial insertions the pivot level can be selected according to the needs of the application). The design of LinkedPeers assumes if a value v_{ij} is selected as a pivot key during the insertion of a tuple, every other tuple that contains v_{ij} must also select it as its pivot key for dimension i. To comply with this assumption, a node should be aware of the existing pivot keys during the insertion of a new tuple. Thus, a fully decentralized catalogue storing information about root keys and their respective pivot keys in the network is implemented in LinkedPeers. Each root key is stored at the node with ID closest to its value. Every time that a new pivot key corresponding to this root key is inserted in the system, the root key node is informed about it and adds it in a list of known pivot keys. The root key node is also aware of the MaxPivotLevel used during the insertion of its values in the specific dimension.

The procedure for inserting the values of a tuple appropriately in all dimensions constitutes of the following basic steps:

- Inform each root key of every dimension about the corresponding value-set $(v_{i,0},\ldots,v_{i,L_i-1})$ of the tuple, so as to decide for the appropriate pivot level.
- Insert each value-set $(v_{i,0},\ldots,v_{i,L_i-1})$ to the corresponding i^{th}-ring.
- Create or update links among the trees of secondary dimensions towards the primary dimension.

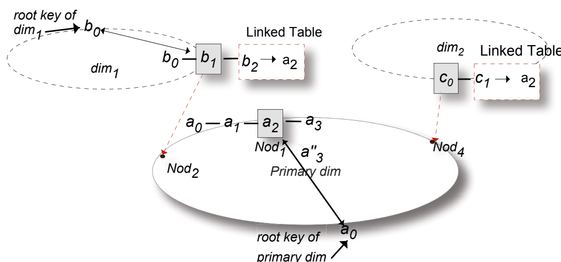


Fig. 2. The created data structures after the insertion of the first tuple of Figure 1

Initially, the initiator contacts the root key of the primary dimension's value-set. The root key of the primary dimension is informed about the new tuple and indicates the appropriate pivot level (if the same pivot key already exists, then its pivot level is used, otherwise the MaxPivotLevel). Afterwards, the DHT operation for the insertion of the tuple in the primary dimension starts and the tuple ends up to the node responsible for the decided pivot key. The node responsible for the pivot key of the primary dimension stores its value set in a tree structure and the whole tuple in a store defined as its *local database*. Moreover, it stores the result(s) of the aggregate function(s) over all these tuples (i.e., the results for a (pivot key, $*, \ldots,$*) query). Figure 2 demonstrates the insertion of the value-set (a_0, a_1, a_2, a_3) in the primary ring of an overlay consisting of nodes referred to as Nod_i. The root key a_0 does not exist in the overlay and ℓ_{02} is selected randomly as pivot level. The root index is created from a_0 towards a_2 and the tuple is inserted to the node Nod_1, which is responsible for the pivot key a_2 according to the DHT protocol. Nod_1 inserts all the values of the tuple in its local database as well.

he next step is to store the value-sets for the remaining dimensions in the corresponding ring. The node responsible for the primary key contacts each node responsible for the root keys and is informed about the appropriate pivot level in d_i. Since the pivot levels for the secondary dimensions are determined, the value-set of each dimension is stored in the node responsible for its pivot key. Again, the respective aggregates are also maintained in the nodes of the trees. The values of the secondary dimensions are associated to the primary dimension through the primary key. Each leaf value of a secondary tree structure maintains a list of the primary keys that is linked to. The structure storing the mappings among the leaf values and the primary keys is referred to as *Linked Table*. In case of an update taking place, the existing indices for any value of the tuple are also updated.

In Figure 2, the tree structures comprising of only one branch for the secondary dimensions are shown as well. During the insertion of value-set (b_0, b_1, b_2), the root index b_0 is created (the pivot level for c_0 is the root level and no further indexing is needed). Figure 3 shows the final placement of the values of the tuples of Figure 1 among the nodes of the overlay.

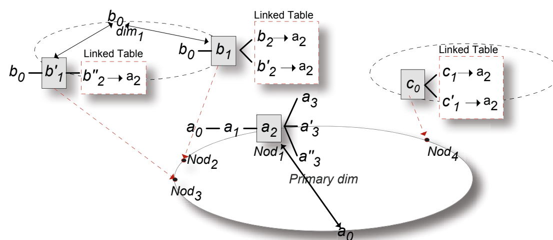


Fig. 3. Final placement and indexing of the tuples of Figure 1 in *LinkedPeers*

3 Query Processing

The queries posed to the system are expressed by conjunctions of multiple values. When a query includes a pivot value, then the node responsible for this value can be found with a simple DHT lookup. Otherwise, the native DHT mechanisms are not adequate to query the rest of the stored values. The proposed techniques can be further utilized to enable the search for any stored value.

The idea behind the approach followed for the insertion of tuples in the DHT overlay is the maintenance of the linking among the multiple dimensions, which can be searched either independently from each other or in conjunction with others. When the query does not define a specific value for a dimension (a '*'-value), then any possible value is acceptable for the query. A query is assumed to include up to *d-1* '*' for *d dimensions*.

LinkedPeers allows adaptive change of pivot levels according to the query skew. Therefore, query initiators are not aware if any of the queried values correspond to a pivot value, forcing them to issue consecutive lookups for any value contained in the query according to the dimension priority, until they receive a result. Initially, a *lookup* operation is initiated for the value of the dimension with the highest priority. If the node holding the queried value cannot be located by the DHT lookup, then a lookup for the next non-'*' value follows. If no results are returned for all the values in the query, then the query is flooded among the nodes of the overlay.

3.1 Exact Match Queries

Queries concerning a pivot value of any ring are called `exact match` queries and can be answered by the DHT lookup mechanism. The are two categories of an exact match query:

Category 1: Query is $q = (q_{0_{pivotlevel}}, \ldots)$, where a pivot value of the primary dimension is defined in the query. Other values may be included as well. The DHT lookup ends up at the node responsible for the pivot key of the primary dimension. If this is the only value asked, the corresponding tree structure is searched for the aggregate fact. Otherwise, the local database is scanned and the results are filtered according to the remaining values locally.

Category 2: Query is $q = (q_{0_j}, \ldots, q_{i_{pivotlevel}}, \ldots)$, where $j \neq pivotlevel$. In this case, a queried value in one of the secondary dimensions is a pivot value. The strategy followed to resolve this query is that consecutive queries are issued until the node responsible for $q_{i_{pivotlevel}}$ is reached. If the query contains no other values, then the tree structure of this node is adequate to answer it, otherwise the query is forwarded to all the nodes of the primary dimension that store tuples containing $q_{i_{pivotlevel}}$. These nodes query their local databases to retrieve the relative tuples and send back the results to the initiator. If more than one pivot values are present, then the query is resolved by the dimension with the highest priority. In the example of Figure 3, a query for value b_1 can be resolved by the aggregated fact stored in Nod_3. On the other hand, a query for the combination of values $(a_3, b_1, *)$ reaches Nod_2, which does not store adequate information to answer it and (using its Linked Table) forwards it to Nod_1, which queries its local database.

3.2 Flood Queries

Queries not containing any pivot value cannot be resolved by the native DHT lookup. The only alternative is to circulate the query among all nodes and process it individually. The hierarchical structure of data together with the imposed indexing scheme enable a controlled flooding strategy that significantly reduces the communication cost.

Initially, a flood query is forwarded from a node to its closest neighbour in the DHT substrate. Each visited node searches its tree structures for any of the values included in the query. The visited nodes without any relative data are avoided for future forwarding of the query during the rest of the procedure for the flood resolution.

Query forwarding continues until any of the queried values is found in the stored trees. This node becomes the *coordinator* of the flood procedure. If more than one of the queried values are found in the same node, then the query is resolved in the 'virtual' ring of the dimension with the highest priority.

The found value may belong to a level either below the pivot level or above the pivot level. In the first case, there are no other trees with the specific value. The node either sends the aggregated fact to the initiator of the query or forwards it to the nodes of the primary dimension following the same strategy described for the second category of the exact match queries. Otherwise, there may exist other trees with the same value. For example, if a flood message for value a_1 in Figure 3 reaches Nod_1, other nodes with the value a_1 and different pivot keys may also exist. Yet, it is certain that this value is not stored at a node corresponding to a different root key. Thus, the flood message is forwarded to the node with the corresponding root key, which becomes the coordinator of the procedure from now on. This node forwards the queries to the nodes whose pivot keys it knows of, with each of them either returning the aggregated fact (when the value belongs to the primary dimension or only a single dimension is queried) or a set of candidate nodes that are linked in the primary dimension.

3.3 Materialized Views

In many high-dimensional storage systems, it is a common practice to pre-compute different views (GROUP-BYs) to improve the response time. For a given data set R described by d (dimensions) annotated by single-level hierarchies, a view is constructed by an aggregation of R along a subset of the given attributes resulting in 2^d different possible views (i.e., exponential time and space complexity). The number of levels in each dimension adds to the exponent of the previous formula. In LinkedPeers, we consider a query-based approach to tackle the view selection problem: The selection of which views to pre-compute is query-driven, as we take advantage of the evaluation process to calculate parts of various views that we anticipate to need in the future.

Figure 4 depicts all the possible combinations of the query values (a_1, b_2, c_1), relative to Figure 3. The attributes participating correspond to levels $\{\ell_{01}, \ell_{12}, \ell_{31}\}$ respectively. Each combination (or *view identifier*) consists of a subset of attribute values in $\{d_0, d_1, d_2\}$ ordered according to the priorities of dimensions in decreasing order. Moreover, each view identifier in the *i*-th level of the tree structure in Figure 4 is deduced by its successor view identifier in *(i-1)*-th level by omitting the participation of one dimension each time. When a value of a dimension is omitted in a view identifier, then it is

considered that its value is a ‘*’-value. The identifiers that have already registered on the left-side of this tree are omitted.

Let $S_i \subset S$ be the subset of view identifiers that start with the attribute value defined in dimension d_i. We call the subset of the specific view identifiers as $Partition_{d_i}$ and the dimension that participates in all identifiers of the dimension as $Root_{d_i}$. In Figure 4, $Partition_0$ comprises of all view identifiers that contain a_1, which is the $Root_0$, while a_1 does not appear in any identifier of the remaining partitions.

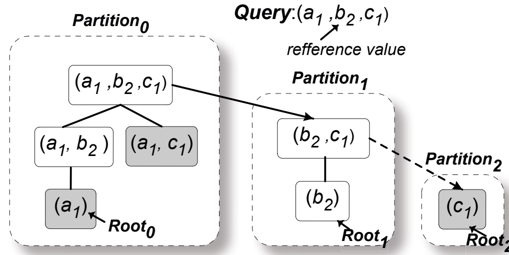


Fig. 4. All possible view identifiers for a query combining values in 3 dimensions

According to the strategy followed during flooding, all the nodes with trees containing the found value used for the resolution of the query (hence *reference value*) are definitely contacted. Thus, we conclude with certainty that there exist no extra nodes with tuples containing the reference value. This assumption is not valid for the rest of the values included in the query. This observation is significant for determining which views can be materialized and stored for future queries in a distributed manner:

Let S be the set of all the 2^d identifiers. We deduce that only a subset $S_{partial} \subset S$ of the view identifiers can be fully materialized, namely only the identifiers of the combinations including the reference value. In the example of Figure 4, let us assume that the flooded query for the combination (a_1, b_2, c_1) reaches Nod_2 and the reference value is b_2. The query will be forwarded to Nod_1 and it will be resolved. Nevertheless, it is not ensured that there are no other nodes storing tuples with a_1 or c_1. Thus, $S_{partial}$ comprises of the view identifiers in the *non-grey* boxes.

In more detail, the calculation of the views occurs among the nodes of LinkedPeers as follows: each peer that returns a found aggregated fact in a flooded query, also calculates the available view identifiers in $S_{partial}$ stored in its local database. Due to the flooding strategy, every peer with trees containing the reference value will be definitely contacted. According to this procedure, the following conclusions are made:

- The $S_{partial}$ may comprise only of identifiers belonging to $Partition_{d_0}, Partition_{d_1}, ..,$ $Partition_{d_{ref}}$, where the $Root_{d_{ref}}$ of $Partition_{d_{ref}}$ is the reference value used for the resolution of the flooded query.
- The upper bound of view identifiers that can be materialized is $2^d - 1$ (‘ALL’ is not materialized), if the query does not contain any ‘*’-value and without the restriction of the flood strategy. In case of ‘*’-values, the number of view identifiers is $2^{d-n} - 1$,

where n is the number of '*'-values. According to the type of the inserted dataset (number of dimensions, number of tuples), the type of the query workload (average number of '*'-values per query) and the specifications of the system, various policies can be defined to limit the number of calculated aggregated results.

Upon the reception of all the results, the coordinator merges the returned aggregated facts for each view identifier. Afterwards, it calculates the hash value of each $Root_{d_j}$ and inserts each $Partition_{d_j}$ $(j \in [0, d_{ref}])$ to the overlay. The node responsible for the $Root_{d_{ref}}$ also creates *indices* towards the locations of its tree structures to forward any query that cannot be resolved by the stored materialized views. The idea behind the splitting of the partitions is that the stored combinations need to be located with the minimum message cost, namely with the primitive DHT lookup. Since a query is dissembled in its elements and the queries are issued according to the priority of the dimensions, each identifier is stored to the dimension with the highest priority of its values.

Although any approach of existing relational schemas for storing views could be utilized to store the aggregated facts, we maintain simple 'linked-listed' structures. As shown in Figure 5, the view identifiers of Figure 4 are stored to the nodes responsible for the values appearing in the 'dark grey' boxes. All the queries arriving at the node responsible for $Root_0$ (namely a_1) should also include the $Root_{d_{ref}}$, which is b_2. The combination of value(s) that a query should include so as to be resolved by the existing view identifiers are marked with red boxes.

The created indices and views are soft-state and they expire after a predefined period of time, which is renewed each time that an existing index is used. The indices are bidirectional to ensure data consistency during re-indexing operations. Finally, we pose a limit to the maximum number of indices held by each node. Overall, the system tends to preserve the most "useful" indices towards the most frequently queried data items.

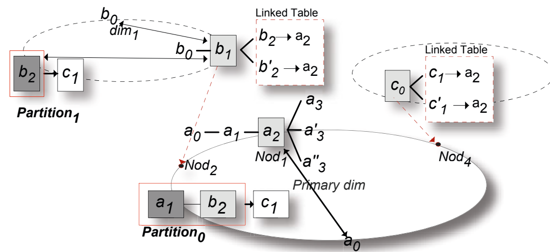


Fig. 5. Distribution of materialized view identifiers among the nodes of LinkedPeers

3.4 Indexed Queries

When a query reaches a node holding an index, then the stored views (if any) are searched for the combination of values included in the query. If the combination is found, the aggregated value is returned to the initiator. In the case that the combination does not exist, but the index is aware of the nodes with the pivot keys for the specific value, the query is forwarded to the respective pivot keys. If the query is simple or the

found value belongs to the primary dimension, then the aggregated facts for the query are returned. Otherwise, the reached nodes return the locations of the primary ring that are correlated with the indexed value. The query is forwarded to these nodes contacting their local database. After an indexed query which has not been resolved with the use of a stored view identifier, the procedure for materializing all the possible view identifiers described in the Section above is followed.

4 Adaptive Query-Driven Re-indexing

A significant feature of our system is that it dynamically adapts its indexing level on a per node basis to incoming queries. To achieve this, we introduce two re-indexing operations regarding the selection of pivot level: *Roll-up* towards more general levels of the hierarchy and *drill-down* to levels lower than the pivot level.

The idea behind individual re-indexing of stored tuples is based on the fact that each node has a global view of the queries regarding each level $\ell_{ij} < pivotlevel$, but only a partial view of the queries for each level $\ell_{ij} > pivotlevel$ of a tree. Therefore, it has sufficient information to decide if a drill-down will be favorable for the values of this tree. On the other hand, a node has to cooperate with other peers that store a value of a level $\ell_{ij} < pivotlevel$ in order to decide if this level is more appropriate. The decision for a possible re-indexing operation is made according to statistics collected by the incoming queries in the trees responsible for the specific value used during the resolution of a query. The goal is to increase the number of queries answered as exact matches in each dimension. The decision process for a possible re-index is triggered only after an indexed or flooded query only for the reference value, following the procedure described in our previous work [3]. Nevertheless, major enhancements have been made for the customization of the re-indexing operations in multiple dimensions due to the requirements arisen from the interconnection among the rings.

Roll-up: In general, if a node detects that the demand on a value above the pivot level relatively exceeds the demand for the other levels, it initiates the procedure to decide if a roll-up towards this level would be beneficial (communicating with the other interested nodes). A positive decision leads to the re-insertion of all trees containing the specific value with the new hash value in the overlay and the trees with the old pivot value are deleted. During a roll-up, one or more nodes re-insert their trees (or the whole tuples in the case of the primary dimension), which end up in one node responsible for the new pivot key. Each node also informs the root key about the new location and the new pivot key and erases all the soft-state indices towards any value of the re-indexed trees. The views containing any of these values in other rings are not affected, since the relocation of the trees does not affect the stored aggregated facts. The final step is the update of the links among the primary and the secondary rings: Each participating node signals the nodes that is linked to so as to replace the old pivot levels of the secondary ring with the new ones in their local databases (roll-up is performed in a secondary ring) or the links in all trees of the secondary rings related to the rolled-up trees, since the links need to be valid for the resolution of future queries.

Drill-down: The drill-down procedure is less complex, due to the fact that only one node holds the unique tree with values for this level. Thus, the node can locally decide

if the drill-down is needed. In this case, it splits the tree to tuples grouped by the new pivot key and re-inserts them in LinkedPeers. The root key is also informed about the new situation and all existing indices towards these trees are erased. Finally, the node that decided the drill-down updates the links among itself and the rest of the rings as described for the roll-up procedure.

5 Experimental Results

5.1 Simulation Setup

We now present a comprehensive evaluation of LinkedPeers. Our performance results are based on a heavily modified version of the FreePastry [4] using its simulator for the network overlay, although any DHT implementation could be used as a substrate. The network size is 256 nodes, all of which are randomly chosen to initiate queries.

Our synthetic data are trees (one per dimension) with each value having a single parent and a constant number of *mul* children. The tuples of the fact table to be stored are created from combinations of the leaf values of each dimension tree plus a randomly generated numerical fact. By default, our data comprise of 1M tuples, organized in a 4-dimensional, 3-level hierarchy. The number of distinct values of the top level is base $= 100$ with *mul*=10. The level of insertion is, by default, ℓ_1 in all dimensions. For the query workloads, a 3-step approach is followed: We first identify which part of the initial database (i.e., tuple) the query will target (*TupleDist*). Next, the probability of a dimension *d not* being included (i.e., a '*' in the respective query) is P_{d*}. Finally, for included dimensions, we choose the level that the query will target according to the *levelDist* distribution. In our experiments, we express a different bias using the uniform, 80/20 and 90/10 distributions for *TupleDist* and *levelDist*, while P_{d*} increases gradually from 0.1 for the primary dimension to 0.8 for the last utilized dimension. Generated queries arrive at an average rate of $1\frac{query}{time_unit}$, in a 50k time units total simulation time.

In this section, we intend to demonstrate the performance of our system for different types of inserted data and query workloads. The experimental results focus on the achieved *precision* (i.e., the percentage of queries which are answered without being flooded) and cost in terms of messages per query.

5.2 Performance under Different Number of Dimensions and Levels

In the first set of experiments, we identify the behavior of our system under a variety of data workloads for different number of dimensions and different number of levels. The queries target uniformly any tuple of the dataset and the levels of the hierarchies in each dimension. In the first set of the experiments, we vary the number of dimensions, while each dimension is described by a 3-level concept hierarchy. Figure 6 demonstrates the percentage of the queries of the query workload that include at least one pivot value (`Pivot Level Queries`), the percentage of the queries resolved as exact match queries in LinkedPeers (`Exact Match`) and the achieved precision. The precision for non-flooded queries remains above 85% for all types of datasets, despite the number of dimensions. Queries that are not directed towards the pivot level are answered with

the use of an index or a materialized view assuring that the precision remains high. The difference among the exact matches and the pivot level queries is due to the utilized strategy that it is preferred for a query to be resolved as an indexed query in a dimension with higher priority than as an exact match to a dimension with a lower priority.

In Figure 7, the results for *4-dimensional* workloads with varying number of level in the hierarchies are demonstrated. The decrease in the precision (from 99% to about 70%) is due to the fact that the increase of levels results to the decrease in the probability of querying a value that it is already indexed. Since the probability of utilizing an index decreases, all the queries targeting the initial pivot level (ℓ_1) even in the secondary rings are resolved as exact matches.

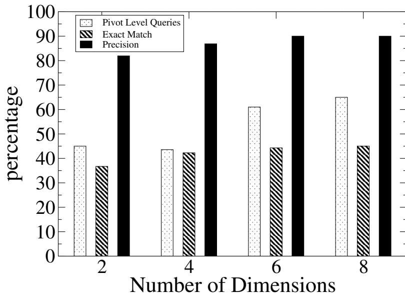


Fig. 6. Percentage of queries for different number of dimensions with *3-level* hierarchies

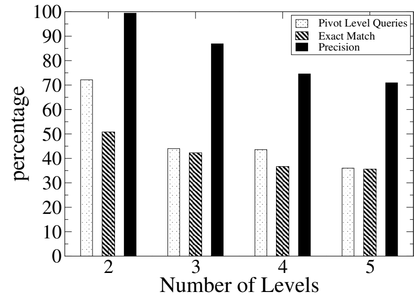


Fig. 7. Precision for different number of levels in *4-dimensional* datasets

5.3 Query Resolution for Different Types of Datasets

In this experiment, the achieved precision of LinkedPeers for various types of datasets is demonstrated in Figure 8. The number of distinct values in the top level `base` and the number of children `mul` are altered changing the density of the dataset. `Base` and `mul` influence the connections among primary and secondary rings and the number of distinct values in each level. As shown in Figure 8, there is a small decrease in the precision of non-flooded queries, while the `base` and the `mul` increases (this decreases the dataset density). Nevertheless, LinkedPeers achieves to resolve the majority of queries without flooding. The percentage of exact match queries in the primary dimension (`Exact_PR`) remains stable for all datasets as shown in Figure 9, since it depends on the query workload. Nevertheless, the exact matches in the secondary rings (`Exact_SR`) increase as the indexed queries decrease, since the indices of the primary dimension are used less, and more queries are resolved by the secondary rings.

5.4 Precision for Skewed Workloads

The adaptive behavior of LinkedPeers is identified in this set of experiments by testing the system under a variety of query loads. Specifically, we vary the *TupleDist* and the number of queries directed to each level by increasing the bias of *levelDist*. In Figure 10, data are skewed towards the higher levels of the hierarchy. The percentage of queries including at least one value in ℓ_0 or ℓ_1 are denoted as `Queries_L0` and `Queries_L1`

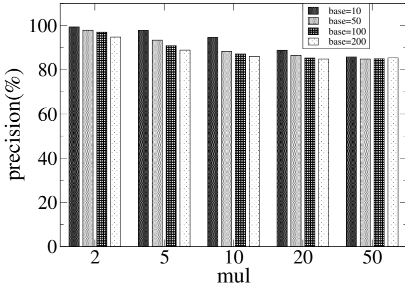


Fig. 8. Impact of `mul` and `base` in the achieved precision

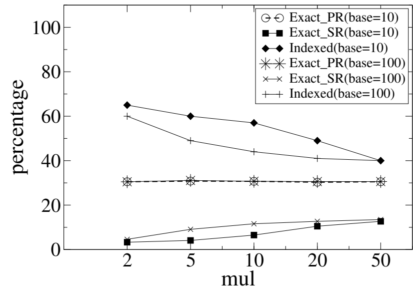


Fig. 9. Percentage of each query category for different data workloads

respectively. By making more bieased the *levelDist*, we observe remarkably high precision rates (close to 100%). Despite the fact that the percentage of queries towards ℓ_1 (`Queries_L1`) decreases significantly as the *levelDist* becomes more biased, the reindexing operations that take place ensure that the majority of queries are resolved as exact match queries by adjusting appropriately the pivot levels in each dimension.

Figure 11 depicts the results, when the query workload favors the lower levels of the hierarchies.The decrease to the precision for more biased *levelDist* is due to the fact that lower levels of the hierarchy have a considerably larger number of values. By increasing the number of queries towards these values, we increase the probability of queries targeting non-indexed values until the re-indexing mechanisms adapt the pivot levels of the popular trees appropriately.

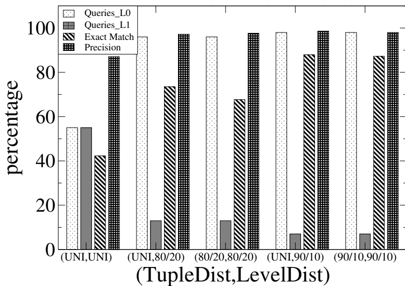


Fig. 10. Precision and exact match queries for skew towards higher levels and various *(TupleDist,levelDist)* combinations

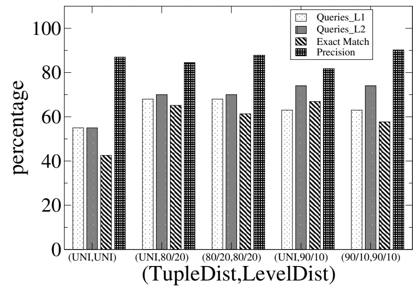


Fig. 11. Precision and exact match queries for skew towards lower levels and various *(TupleDist,levelDist)* combinations

5.5 Testing against the Use of Materialized Views

Apart from the re-indexing operations, the materialized views can be also utilized to minimize the query cost. In the next experiment, we test our method against query workloads targeting the dataset either uniformly or biased (90/10) (`TupleDist`) with uniform and biased (90/10) skew (*levelDist*) towards the higher levels (denoted as `UP`)

and towards the lower levels (DOWN). As shown in Figure 12, the queries resolved with the utilization of materialized views (ViewQ) increase in the query workloads targeting a part of the dataset at most, since the probability also increases for querying a materialized combination. More existing views are utilized, when the queries target uniformly all the levels of the hierarchies in all dimensions. In this case, the reindexing mechanisms cannot adjust the pivot levels to all the incoming queries and nearly the majority of indexed queries (Indexed) are resolved with the use of a materialized view.

5.6 Cost of the Various Types of Query Resolution

The cost of a query is considered as the messages that need to be issued for its resolution. A query resolved as exact match in the primary dimension utilizes only the DHT lookup mechanism. Figure 15 depicts the average number of messages only for exact queries resolved by secondary dimensions (Exact_SR), which number is significantly smaller (less than 20% of all queries in all cases) and indexed queries (Indexed). The average number of messages for Exact_SR depends on the type of dataset, namely the number of links among secondary pivot keys and primary pivot keys. When the query workload is skewed towards the higher levels (UP), then the messages decrease due to the fact that popular trees roll-up towards ℓ_0. Thus, the secondary keys are connected to a smaller number of primary keys. The opposite observation is valid for the (DOWN) query workloads. It is important to notice, that the majority of the indexed queries (Indexed) in the workloads with higher cost for the indexed queries are resolved with views (see Figure 12), thus avoiding this cost.

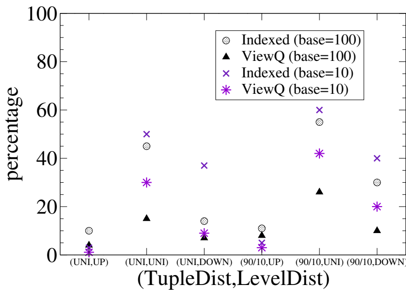


Fig. 12. Utilization of materialized views compared to queries resolved as indexed

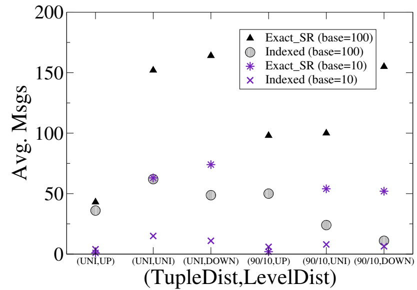


Fig. 13. Average number of messages for exact matches in secondary rings and indexed queries

5.7 Performance for Dataset of the APB Benchmark

The adaptiveness of the system is also tested using some realistic data. For this reason, we generated query sets by the APB-1 benchmark [5]. APB-1 creates a database structure with multiple dimensions and generates a set of business operations reflecting basic functionality of OLAP applications. The generated data are described by 4-dimensions. The customer dimension (C) is 100 times the number of members in the channel dimension and comprises of 2 levels. The channel dimension (Ch) has one level and 10

members. The product (P) dimension is a steep hierarchy with 6 levels and 10.000 members. Finally, the time dimension (T) is described by a 3-level dimension and made up of two years. The dataset is sparse (0.1 density) and comprises of 1.3M tuples.

Figure 14 shows the percentage of exact match queries resolved in primary and secondary rings compared to all exact match queries of a 25K query workload and for different combinations of ordering of dimensions. For all orderings, the precision of non-flooded queries is over 98%. The selection of the primary dimension influences the number of exact match queries in the primary ring. Figure 15 presents the average number of messages for exact matches resolved by a secondary ring and indexed queries, since only a DHT lookup is performed for exact match queries in the primary ring. The average number of messages is small for both exact and indexed queries, apart from the case that the customer dimension has been selected as a primary dimension. In the rest of the cases, the resolution of the queries occurs with a very low cost in terms of additional nodes to visit, even though the majority of the exact queries are resolved by a secondary dimension, as shown in Figure 14. The increase of messages for the CPChT dataset is due to the large number of distinct values used as pivot keys and thus each node responsible for a pivot key stores smaller portion of the total dataset in its local database. For all combinations of datasets, the overhead of the additional indexing structures needed by LinkedPeers such as tree structures, root indices, links and indices and statistical information is up to 1%. Thus, LinkedPeers can be considered as a lightweight solution for indexing multidimensional hierarchical data.

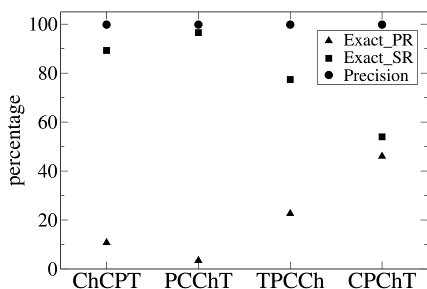


Fig. 14. Precision for APB query workload in LinkedPeers

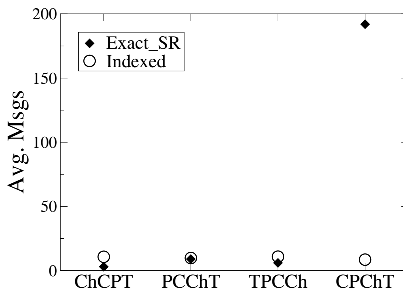


Fig. 15. Average number of messages for exact match and indexed queries

6 Related Work

P2P systems based on Distributed Hash Tables (DHTs), for example [6], appear greatly effective for storing and locating *key* – *value* pairs. Nevertheless, complex queries cannot be supported without the implementation of additional indexing mechanisms. An approach to enable advanced search facilities in DHTs is the replacement of the hash function and the respective modification of the structure and behavior of the overlay to serve multi-attribute queries. Space Filling Curves [7], [8] are usually used as a replacement of the cryptographic hash function of the DHT protocols to produce a locality preserving mapping of multiple attribute values to a single key. In [8], an SFC

hash function is utilized over hierarchical attributes. Nevertheless, it is assumed that the full path from the root level towards the searched level is known and the values for all attributes in a query are given. Afterwards, the queries are transformed to range queries resolved by consecutive DHT lookups, usually resulting in flooding among all nodes. Our implementation does not pose any requirement for querying all the dimensions and allows the querying of any level of the hierarchy separately. There has been also significant work in the area of databases over P2P networks. PIER [9] proposes a distributed architecture for relational databases supporting operators such as join and aggregation of stored tuples. The Chatty Web [10] considers P2P systems that share (semi)-structured information but deals with the degradation, in terms of syntax and semantics, of a query propagated along a network path. In GrouPeer [11], SPJ queries are sent over an unstructured overlay in order to discover peers with similar schemas. Peers are gradually clustered according to their schema similarity. PeerDB [12] also features relational data sharing without schema knowledge. All these approaches offer significant and efficient solutions to the problem of sharing structured and heterogeneous data over P2P networks. Nevertheless, they do not deal with the special case of hierarchies over multidimensional datasets.

7 Conclusions

In this work, we described *LinkedPeers*, a distributed infrastructure for storing and processing multi-dimensional hierarchical data. Our scheme distributes large amount of partially-structured data over a DHT overlay in a way that hierarchy semantics and correlations among dimensions are preserved. Each data item can be described by an arbitrary number of dimensions and aggregate queries are resolved in a fully distributed manner. Re-indexing and pre-computation mechanisms are triggered dynamically during the resolution of queries. Our experimental evaluation over multiple and challenging workloads confirmed our premise: Our system manages to efficiently answer the large majority of queries using very few messages. It adds small overhead in storing hierarchical data and provides a lightweight indexing scheme, resolves efficiently aggregated queries and adapts to sudden shifts in skew by enabling re-indexing operations.

References

1. MacManus, R.: The coming data explosion (2010), `http://www.readwriteweb.com/archives/`
2. Linked Data - Connect Distributed Data across the Web, `http://linkeddata.org/`
3. Asiki, A., Tsoumakos, D., Koziris, N.: Distributing and searching concept hierarchies: an adaptive dht-based system. Cluster Computing 13, 257–276 (2010)
4. FreePastry, `http://freepastry.rice.edu/FreePastry`
5. OLAP Council APB-1 OLAP Benchmark, `http://www.olapcouncil.org/research/resrchly.htm`
6. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In: Proc. of the ACM SIGCOMM (2001)

7. Schmidt, C., Parashar, M.: Squid: Enabling search in dht-based systems. Journal of Parallel and Distributed Computing 68(7), 962–975 (2008)
8. Lee, J., Lee, H., Kang, S., Kim, S.M., Song, J.: CISS: An efficient object clustering framework for DHT-based peer-to-peer applications. Computer Networks 51(4)
9. Huebsch, R., Hellerstein, J., Boon, N.L., Loo, T., Shenker, S., Stoica, I.: Querying the Internet with PIER. In: VLDB (2003)
10. Aberer, K., Cudre-Mauroux, P., Hauswirth, M.: The Chatty Web: Emergent Semantics Through Gossiping. In: WWW Conference (2003)
11. Kantere, V., Tsoumakos, D., Sellis, T., Roussopoulos, N.: GrouPeer: Dynamic clustering of P2P databases. Inf. Syst. 34(1), 62–86 (2009)
12. Ooi, B., Shu, Y., Tan, K., Zhou, A.: PeerDB: A P2P-based System for Distributed Data Sharing. In: ICDE (2003)