A DHT-based System for the Management of Loosely Structured, 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 DHT-based system designed for efficient distribution and processing of multidimensional, loosely structured data over a Peer-to-Peer overlay. Each dimension is further annotated with the use of concept hierarchies. The 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 enable the efficient resolution of such queries, *LinkedPeers* utilizes a conceptual chain of DHT rings that stores data in a hierarchy-preserving manner. Moreover, adaptive mechanisms detect dynamic changes in the query workloads and adjust the granularity of the indexing on a per node basis. The pre-computation of possible future queries is also performed during the resolution of an incoming query. 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 is characterized by an astonishing explosion in the amount of produced data forming a new reality in the digital world. This tremendous increase of content is a global phenomenon, affecting a variety of applications and making it one of the biggest challenges in the area of Information Technologies. Market globalization, business process automation, web applications, 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 [1]. 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 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. The effectiveness of the techniques applied in *LinkedPeers* is also studied for the use case of hosting Web data published and classified in RDF format.

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. It is defined 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:

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

where $v_{i,j}$ represents the value of the j-th level of the i-th dimension. Note also that any *value-set* ($v_{i,0}, \dots, v_{i,L_i-1}$) for the i-th dimension may be absent from a tuple, but one dimension has to be considered as primary. The *fact* (e.g., f_0) 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 lowest level of a hierarchy ($v_{i,(L_i-1)}$) are also referred to as *leaf values*.

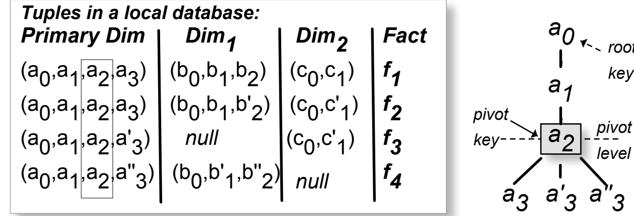
The values of the hierarchy levels in each dimension are organized in tree structures, one per root key. Without loss of generality, it is assumed that each value of ℓ_{ij} has at most one parent in $\ell_{i(j-1)}$. To insert tuples in the multiple rings, one level from each dimension hierarchy is chosen; its hashed value serves as its *key* in the underlying DHT overlay. Any reference to this level is noted 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 the 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:

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

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


Tuples in a local database:

Primary Dim	Dim1	Dim2	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.

2.2 Data Insertion

The proposed 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.

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 the i-th dimension. 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.

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

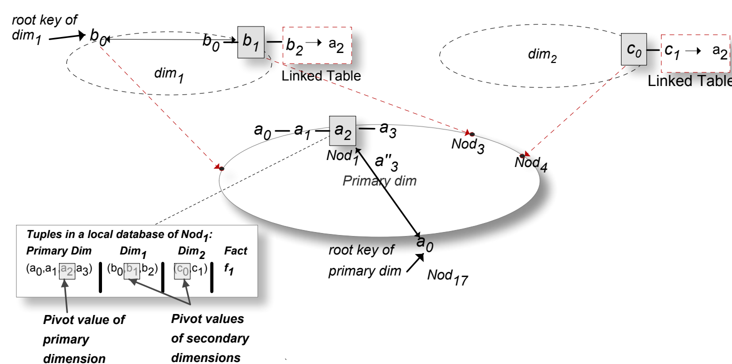


Fig. 2. The created data structures after the insertion of the first tuple of Figure 1

the appropriate pivot level: if the same pivot key already exists, then its pivot level is used, otherwise the MaxPivotLevel. In case that the root key does not already exist, then it is stored in the node responsible for it and the pivot level is chosen either randomly or according to a predefined pivot level for the whole system. 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 that have the same value in each level (i.e., the results for $(v_{0j}, *, \ldots, *)$ queries, where $j \in [0, L_i - 1]$). Figure 2 demonstrates the insertion of the value-set $(\mathrm{a}_0, \mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3)$ 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 of the value a_2 according to the DHT protocol. Nod_1 inserts all the values of the tuple in its local database as well.

The 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*. The node holding the primary key also stores the pivot levels of the value-sets of the secondary dimensions in its local database along with the whole tuple. Another remark is that if the insertion of tuples does not take place during the initial loading of data in the system and the root key already exists, then any existing soft-state indices should also be updated according to the procedure described in our previous work for updating hierarchical data [2]. In this case, since soft-indices may store the aggregated

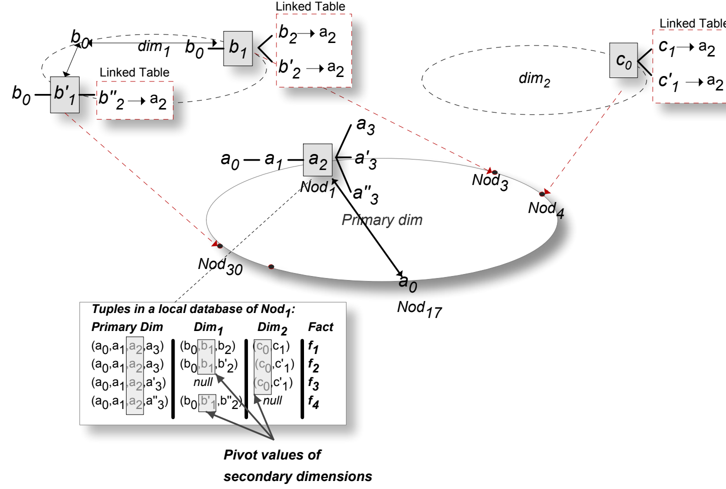


Fig. 3. Final placement and indexing of the tuples of Figure 1 in *LinkedPeers*

facts for the indexed value, the soft-state indices should not only get informed about the locations about the new trees, but also about the new facts. In case that the tree already existed, the marked values as indexed should also learn about the new tuple.

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. When the second tuple is inserted in the overlay, the root index for a_0 indicates that the value a_2 exists already as pivot key and thus this tuple needs to be stored in Nod_1. A new branch below the pivot level is inserted in the existing tree. The values of dim_1 do not exist in the third tuple, but this fact does not affect the procedure of the insertion. The insertion of the values for dim_1 results also in the construction of a new tree for the pivot value b'_1. Since the primary key of all tuples is the same, the local database of Nod_1 contains all the rows shown in Figure 1.

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 search 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 $q = (q_{0pivotlevel}, \ldots)$, where a pivot value of the primary dimension is defined in the query. Any other values may be included for other dimensions 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 $q = (q_{0j}, \ldots, q_{ipivotlevel}, \ldots)$, where q_{0j} does not correspond to a pivot value. 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 $q_{ipivotlevel}$ 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 $q_{ipivotlevel}$. 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 in the query, 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_3, 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. In case that the query contains a single value, then the tree structures of each node are searched. Otherwise, a node searches its local database for the queried values and sends the found results back to the initiator.

To minimize the communication and processing costs, extra steps are taken for the resolution of a flood query. Both the DHT mechanisms and the properties of the data structure are utilized to avoid visiting the same node multiple times and impose an order

in the way that the nodes are visited, instead of flooding the query in an uncontrolled manner. The hierarchical structure of data along 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. It also searches its local database for any of the queried values and the combination of values included in the query. If nothing is found in the reached node, then the current node registers the *key* range(s) under its responsibility in the flood message and forwards the query to its closest neighbour. This strategy is enforced so as to avoid visiting the specific node again during the rest of the procedure for the flood resolution. The reasoning behind this strategy is that if a node has been already queried and does not store any relative tuples to the query, then there is no benefit of searching the same node again, even if it is indicated as a candidate node for holding tuples that answer the query.

If any relative information to the query is found in a reached node, then the query forwarding stops. In case that the queried value is found in a tree structure of the node, then 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 possible cases of a found flooded value are two: either to belong to a level above the pivot level or to a level below the pivot level. The refereed node does not become the coordinator, when a value is found in one or more tuples of the local database. Nevertheless, in this case there is enough information in the stored tuple to find out if the found value is located above the pivot level or below the pivot level, so as to forward the flood query either to the root key of this value or its pivot key respectively. Any found tuples answering the specific query (or aggregated facts) are also included to the flood message during the forwarding of the query. Apart from this additional step, the procedure for resolving the query continues as described below without any other changes.

Assuming that the found value is located below the pivot level of a tree structure, then there are no other trees with the specific value. The node either sends the result to the initiator of the query (if the query involves only a single value or the found value belongs to the primary dimension) or forwards the query along the links to the nodes of the primary dimension excluding the ones that have been already visited. The same strategy described for the second category of the exact match queries is followed. The nodes with the primary keys respond with the relative tuples or the aggregated fact. These results are collected by the coordinator and are sent back to the node that initiated the query.

In case that the found value belongs to a level above the pivot level, there may exist other trees with the same value, even if one has been already found. 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 tree having 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 flood query to the nodes whose pivot keys is aware of, excluding the nodes that have been already visited. If the found value belongs to the primary

dimension or the query does not involve any other values, then nodes respond with the relative tuples or the aggregated fact respectively. Otherwise, each node includes in its response any relative facts that may have found in its local database and a set of candidate nodes that the pivot key(s) of the found value is(are) linked in the primary dimension. Upon receiving all the results, the coordinator merges the links and excludes from querying the nodes that have been already visited. Finally, the local databases of the remaining nodes are queried and the returned results are merged with the already found ones and returned back to the initiator.

3.3 A Query-driven Approach for Partial Materialization

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*, a query-based approach is considered to tackle the view selection problem and : The selection of which "views" to pre-compute is query-driven, as it is taken advantage of the evaluation process to calculate parts of various views that are expected to be needed in the future and maintain "partial materialized views" in a distributed manner.

Figure 4 depicts all the possible combinations of the values of the query (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 of values consists of a subset of attribute values in $\{d_0, d_1, d_2\}$ ordered according to the priorities of dimensions in decreasing order. A possible combination of values that can be queried is mapped to a "view identifier" comprising of the respective values. When a view identifier (or combination respectively) is "materialized", then the result for this combination of queried values is computed and stored for future use. For example, the view identifier (a_1, c_1) in Figure 4 stores the results of the query $(a_1, *, c_1)$. 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. The subset of the specific view identifiers is called $Partition_{d_i}$ 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.

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, it can be concluded 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 combinations can be materialized and stored for future queries in a distributed manner:

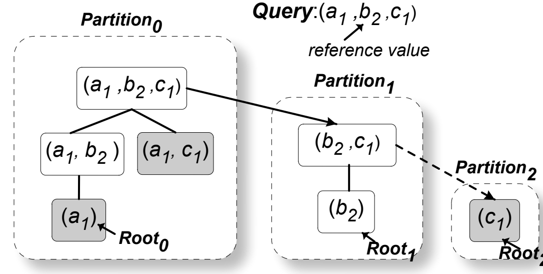


Fig. 4. All possible view identifiers for a query combining values in 3 dimensions.

Let S be the set of all the 2^d identifiers. It can be deduced 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_3 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, which can be materialized.

In more detail, the calculation of the partial 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.
- If the query is flooded in all the nodes of the network, then all the combinations of the queried values can be calculated resulting in $2^d - 1$ combinations ('ALL' is not materialized), if the query does not contain any '*'-value. In case of '*'-values, the number of view identifiers is $2^{d-n} - 1$, where n is the number of '*'-values. If the described strategy for minimizing the visited nodes during the flood of a query is enforced, then only the combinations that contain the reference value can be calculated. Nevertheless, taking into account 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 (i.e., bandwidth consumption, storage capacity) 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, simple 'linked-listed' structures are maintained in *LinkedPeers* storing the different view identifiers, along with the corresponding facts. As shown in Figure 5, the materialized 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 at least include so as to be resolved by one view identifier of such a group is marked with red boxes. It may also include the value(s) contained in the white boxes. For instance, the queries $(a_1, b_2, *)$ and (a_1, b_2, c_1) can be directly answered by the calculated results stored in the node

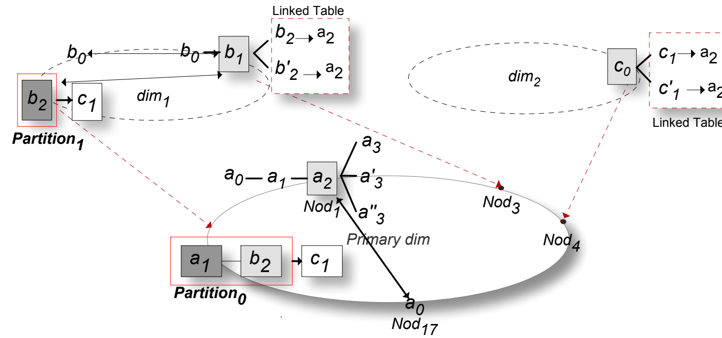


Fig. 5. Distribution of materialized view identifiers among the nodes of LinkedPeers

responsible for the $Partition_0$, as shown in Figure 4. The key used by the DHT for assigning these two view identifiers to the appropriate node is the hashed value of a_1.

The created indices and views are soft-state in order to minimize the redundant information. This means that they expire after a predefined period of time (Time-to-Live or *TTL*). Each time that an existing index is used, its TTL is renewed. This constraint ensures that changes in the system (e.g., data location, node departures, etc) will not result in stale indices, affecting the validity of the lookup mechanism. 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.

During the update procedure, there are different alternatives that can be considered for updating the materialized combinations, since their aggregated facts may change. On one hand, it can be assumed that since the partial materialized views are soft-state, it can be avoided to explicitly update their values and rely on the fact that after their expiration, the new values will be taken into account. On the other hand, if the returned results need to be accurate and there are strict constraints on this, then a strategy can be

enforced that lookups about the values of the new tuple(s) and updates any materialized combinations of values appropriately.

3.4 Indexed Queries

When a query reaches a node holding an index, then the stored view identifiers (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.

The nodes with actual tuples of the indexed value need to know the existence of an index. The bidirectionallity of the indices is introduced only to ensure data consistency, despite of them being soft-state. During re-indexing operations, the locations of stored tuples change and indices correlated to these tuples need either to be updated or erased, preventing the existence of stale indices. It has been chosen to erase them, so as to avoid increasing the complexity of the system. Detailed information for an existing index is not essential for the node, where the tuples are stored. A simple mark for each indexed value is adequate in order to erase its index, if needed. In this case, some redundant operations for erasing expired indices may occur. If there are no memory restrictions and local processing is preferable to bandwidth consumption, indexed values can be marked with a time-stamp. Every lookup for an indexed value renews the TTL in both sides of the index and only valid indices are erased during re-indexing operations. The created views holding the data for the calculated combinations are not aware of the locations of the trees and for this reason views are only soft-stated.

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, two re-indexing operations regarding the selection of pivot level are introduced: *Roll-up* towards more general levels of the hierarchy and *drill-down* to levels lower than the pivot level.

The idea behind the decision procedure is based on the fact that a node is more capable of detecting if the values for a level $\ell_{ij} > pivotlevel$ (where $pivotlevel$ denotes the pivot level of a specific hierarchy in the *i*-th dimension) of a tree are queried at most and thus to proceed to re-indexing operation. On the contrary, it has to cooperate with the rest of the nodes storing a value for a level $\ell_{ij} < pivotlevel$ to obtain a global view and decide if a re-indexing operation towards this level for the involved trees would be beneficial. Therefore, the node has sufficient information to decide if a drill-down will be favorable for the values of this tree. A roll-up towards a level $\ell_{ij} < pivotlevel$ is

decided by all the involved nodes storing trees with the specific value in this level. 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. Since some queries are resolved by the node holding an index and are not further forwarded to the node(s) with the actual tree structure(s), some statistical information is maintained in these nodes. This information is pushed to the referred nodes and merged with their maintained records, as soon as another query needs to be forwarded to these nodes by the index. The queries answered with the use of view identifiers are not counted in the decisions for re-indexing, since they are resolved directly by the node holding them and the query processing is not encumbered. The goal is to increase the number of queries answered as exact matches in each dimension.

Each time that a value of a tree structure is looked up, then the maintained statistical information is updated. The decision process for a possible re-indexing operation can be triggered after an indexed query resolved without using a materialized view or a flooded query and only for the reference value. As far as the indexed queries are concerned, a node holding the tree with the specific value checks for a re-indexing operation after a number of indexed queries has been received by the specific node. When a tree structure is searched during the resolution of a flood or indexed query, then the respective statistical information is checked to find out if a re-indexing operation is indicated. If a drill-down is decided or a roll-up seems probable, then this alarm is included to the answer of the node. Since only the tree structures for the reference value are definitely visited, a re-indexing decision is examined only for this value and not all the values contained in the query. The procedure for deciding if a re-indexing operation is advisable is performed according to the algorithms proposed in [2]. Nevertheless, major enhancements have been made for the customization of the re-indexing operations in multiple dimensions due to the requirements arisen from the existing links among the rings. If a re-indexing operation is not needed after a flood query, then no action is taken other than the creation of the soft-state indices and materialization of the view identifiers.

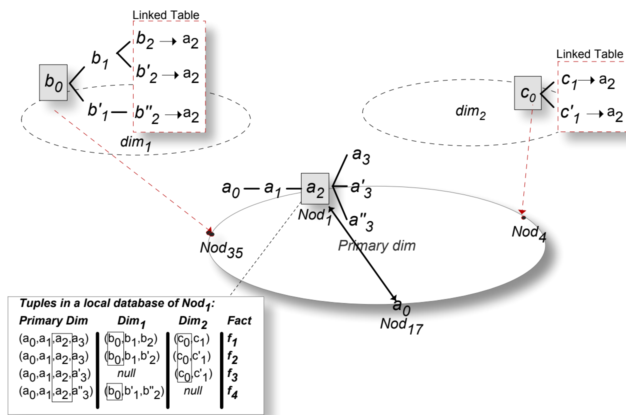


Fig. 6. Re-organization of the tree structures of dim_1 after a roll-up operation towards ℓ_0

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 (by communicating with the other nodes holding this value). For this reason, it sends a remark to the node collecting the results that a possible roll-up operation towards the queried level needs to be examined. When the coordinator (if a flood operation occurs) or the node that holds the index is informed about the requisite of a possible roll-up towards the queried level, it starts the collection of statistical information from all trees containing the queried value. Afterwards, it decides if a roll-up operation is needed and 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, which end up in one node responsible for the new pivot key. If the roll-up value belongs to a primary dimension, then all the relative tuples in the local database are transferred to the new node. Each node also informs the root key about the pivot key(s) to be erased and the new pivot key to replace it (them) and erases all the soft-state indices towards any value of the re-indexed trees. The root key waits to receive the messages for updating its list of pivot keys from all the nodes participating in the roll-up operation and afterwards replaces the old pivot keys and nodes with the new ones. In the meantime, queries concerning any value of the trees participating in a roll-up operation are answered by the nodes responsible for the old pivot keys. The stored view identifiers containing any of these values in other rings are not affected, since the relocation of the trees does not influence the stored, aggregated facts. The final step is the update of the links among the primary and the secondary rings, since the links need to be valid for the resolution of future queries. If the roll-up operation is performed in the primary ring, then the entries in the *Linked Tables* containing the old pivot keys need to be updated. Each tuple in the local databases stores also the pivot levels of the secondary dimensions. Thus, the node responsible for the new pivot key finds the pivot keys of the secondary dimensions from its tuples along with their leaf values and informs them about its new pivot key, so as to update their links towards the primary dimension. When the roll-up is performed in a secondary ring, then the pivot levels stored in the tuples containing the new pivot key need to be updated as well. For this reason, the node responsible for the new tree, sends its pivot level along all of its different links towards the primary dimension.

In Figure 6, the outcome of a roll-up operation towards ℓ_0 in the secondary ring of dim_1 is shown. The involved tree structures storing the value b_0 before the roll-up operation are shown in Figure 3. It can be assumed that an indexed query for the value b_0 triggered a roll-up operation resulting in the re-insertion of the trees with pivot keys b_1 and b'_1 respectively. The node responsible for the new pivot key b_0 informs also the node responsible for the primary key a_2 that the new pivot level for all the tuples containing the value b_0 in dim_1 is ℓ_0.

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 answering the query can locally decide if the drill-down is needed and proceed to the required actions. Afterwards, it splits the tree to tuples grouped by the new pivot key and re-inserts them in *LinkedPeers*. In case that the queried value belongs to the primary dimension, the

tuples of the local database are transfered to the nodes responsible for the new pivot keys as well. After the re-insertion of the relative trees is completed, the node responsible for the old pivot key informs also the root key about the new pivot keys and the new locations of the trees and all existing indices towards the values of the old tree 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.

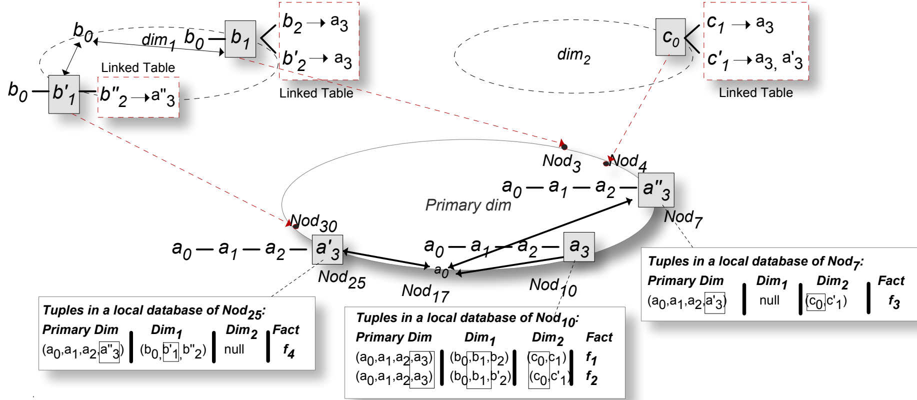


Fig. 7. Re-organization of the tree structures and the local databases in the primary dimension after a drill-down operation towards ℓ_3

Figure 7 exhibits the placement of the tuples in the primary dimension after a drill-down towards ℓ_3 of the tree with pivot key a_2 shown in Figure 3. A flood query for the value a_3 will end up in the node responsible the pivot key a_2 and will trigger the procedure for deciding if a drill-down is needed towards ℓ_3. If this is the case, the node re-inserts the tuples of the specific tree with the new pivot keys a_3, a_3' and a_3'' respectively. The specific node informs also the root key a_0 about the new pivot keys and the pivot keys of the secondary dimensions that is linked to. For example, the entry $b_2 \rightarrow a_2$ in the Linked Table of the pivot key b_1 is now replaced with the entry $b_2 \rightarrow a_3$.

Group-Drill-down: When a roll-up is examined towards a queried level above the pivot level, it is also examined if a drill-down of the involved trees to a level $l_{ij} \geq MaxPivotLevel$ is needed. It is possible to find a level $\ell_{ij} \geq MaxPivotLevel$ to be the most popular but this tendency not to appear in the partial views of the involved nodes and for this reason the trees have not already performed a drill-down towards this level. In this case, the coordinator informs the involved nodes that a drill-down to this level is needed. This procedure is called `Group-Drill-down`, since more than one nodes participate in the drill-down. All the trees with the queried value drill-down to the new pivot level. If the new pivot level is equal to the MaxPivotLevel, the trees already in the MaxPivotLevel do not perform any action.

5 Use Case: Indexing and Searching Web Data

Recently, most of the web sources have adopted general standards to publish their data enabling the integration and combination of different content. A well-established effort is the "Linked Data Principles" paradigm [3] defining a group of rules for publishing data on the Web and linking content among different resources targeting the creation of a signle global data space. These efforts mainly follow the Resource Description Framework (RDF) [4], which has been widely introduced for the representation and exchange of such information, since it provides a flexible way to describe things in the world (e.g., people, locations or abstract concepts) and their relationships. RDF data is, in essence, a collection of statements represented as triples: ⟨*subject, property, object*⟩. Each property in the triple states the relation between the subject and the object. Moreover, the current trend for sources publishing data of the described form is to make them available through SPARQL endpoints [5] responsible for the evaluation of the queries posed by the users.

It has been observed that many of these sources exploit standards such as the RDFS and the Web Ontology Language OWL to represent the semantic knowledge, i.e., to express entities, facts, relations between facts and properties of relations. Moreover, the instances (or individuals) are usually described by concepts (e.g., countries, cities, organizations, people). These concepts can be arranged in a hierarchical manner through the use of taxonomies or category hierarchies. For instance, in the DBpedia project [6], a cross-domain manually created ontology extracted from the infoboxes within Wikipedia is utilized to describe all individuals (or instances). We refer to both individuals and concepts as entities in the rest of this discussion. It is considered that each entity can appear as a subject in a triple and as an object in another triple. The classes of these ontologies can be used to build trees according to the `rdfs:subClassOf` relation, which is used to state that one class is a subclass of another and this information is utilized for the ordering of concepts among the levels of the trees.

The proposed mechanisms for interlinking multidimensional data in *LinkedPeers* can be customized so as to provide a P2P infrastructure for storing and indexing data published in such forms. In this platform, new entities are inserted and end up in the nodes responsible of the pivot value of their trees. The distribution of data among the nodes of the overlay occurs in a manner that preserves the ontology-specific information of each entity, while it also interlinks different entities according to their properties. The queries concerning entities at any level can be performed in a unified manner due to the organization of data in hierarchical structures. Moreover, the adaptation of the indexing granularity among different levels of entities and classes is performed according to the incoming queries.

To serve the needs of the investigated application, two virtual overlays are constructed according to the logic of the *LinkedPeers* architecture: one primary ring storing the entity appearing as subject in the triple and a secondary ring indexing the entities appearing as objects. As a result of constructing these two rings, only the queries about entities can be resolved by the developed mechanisms. If a query is about a specific property, then it can be only resolved through flooding and evaluation among all the nodes of the system. The creation of indices for the properties is disabled due to the fact that properties usually do not adhere to hierarchical relationships. As described in [7],

it can be expected that queries may contain at least the type of the subject or the object. If this is the case, the query resolution starts from this value and follows the links maintained in the system until it is resolved. Nevertheless, if the system is required to serve mostly queries about properties, then an additional secondary ring can be added as the one for indexing the objects without requiring further modifications.

During data insertion, the following assumption is made: when two new entities and their among relationships are inserted, then the all relevant information regarding the "*full path*" of the hierarchy of classes describing the specific entities is provided. For example, the triple <*NTUA, campus, Athens*> states that the entities NTUA and Athens are linked with the relationship campus. When this triple containing the above entities is inserted, all the triples describing the semantic information about these entities need to be provided, i.e., all the classes (defining the corresponding concepts) related to the specific entities. In the described example, all the triples provided during the insertion form the following tuple:

⟨Organization, Institution, University, **NTUA**,
Place, City,Administrative Region, **Athens,campus**⟩

In this tuple, the emphasized values denote the actual entities appearing in the discussed triple and their relationship. The rest of the values correspond to the instances of the classes that the entities belong to and are ordered according to the utilized categorical ontology.

The insertion of these tuples occurs as described in Section 2.2. Nevertheless, the local databases may store RDF triples representing the information contained in the above assumed tuple, thus enabling the reasoning and other ontology related operations. The only prerequisite related to the creation of the tree structures is that all the branches of the same root value have the same number of levels so as the re-indexing operations to take place. Since, in a real-world ontology, a different number of levels below each node may exist, we fill the missing levels above the leaf value with "pseudo" values.

The queries are resolved according to the described strategies which locate the stored entities of any level. Since the same entity may appear in the primary and the secondary dimension at the same time, this fact is taken into account during the resolution of a query. If the queried entity is searched as a subject, then the query is handled as described in the cases of the primary dimension; otherwise the appropriate procedures of forwarding the query along the maintained links are performed. The same approach is followed during the re-indexing operations: The update of the links among the primary dimension and the secondary dimension occurs in both direction for the same tree, namely both the pivot levels for the secondary dimension in the tuples of the database and the records of the Linked Tables are updated.

6 Experimental Results

6.1 Simulation Setup

A comprehensive evaluation of *LinkedPeers* is presented. The performance results are based on a heavily modified version of the FreePastry [8] 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.

The 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: At first, the part of the initial database (i.e., tuple) the query will target (*TupleDist*) is identified. Next, the probability of a dimension *d not* being included (i.e., a '*' in the respective query) is P_{d*}. Finally, for included dimensions, the level is chosen that the query will target according to the *levelDist* distribution. In the presented experiments, a different bias is expressed 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.

This section is intended to demonstrate the performance of the 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.

6.2 Performance Under Different Number of Dimensions and Levels

In these experiments, the behavior of the system under data workloads containing tuples with various number of dimensions or tuples with variable number of levels per dimension is identified. The queries target uniformly any tuple of the dataset and any level of the hierarchies in each dimension.

In the first set of the experiments, the number of dimensions varies, while each dimension is further annotated by a 3-level concept hierarchy. Figure 8 demonstrates the percentage of the queries in the workload including at least one pivot value (denoted as `Pivot Level Queries`), the percentage of the queries resolved as exact match queries in *LinkedPeers* (denoted as `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 combination assuring that the precision remains high. The difference among the exact matches and the pivot level queries is due to the fact that in the utilized strategy followed during the resolution of queries, it is preferred to use an index of a higher dimension than continue looking up for a pivot value in the dimensions with lower priorities.

In Figure 9, the results for *4-dimensional* workloads with varying number of levels 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 has a negative impact on the probability of a query to include one pivot value for at least one of the dimensions. Thus, the percentage of the `Pivot Level Queries` decreases and consequently the same happens to the percentage of the `Exact Match` queries, as shown in the first pairs

of columns in Figure 9. The increase of levels also influences the querying of a value that it is already indexed. For this reason, the deviation between the `Pivot Level Queries` and the `Exact Match` is bigger for two levels, where all queries including a value in the primary dimension are resolved with the use of the pivot key or the root index according to the proposed strategy for the query processing.

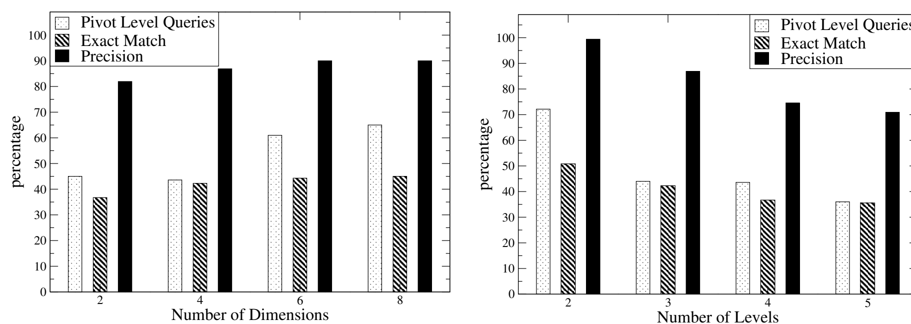


Fig. 8. Resolution of queries for data workloads with different number of dimensions, while each dimension is annotated with a *3-level* hierarchy

Fig. 9. Resolution of queries for data workloads with different number of dimensions for *4-dimensional* datasets

6.3 Query Resolution for Different Types of Datasets

In this experiment, the achieved precision of *LinkedPeers* for various types of datasets is demonstrated in Figure 10. The number of distinct values in the top level `base` and the number of children `mul` varies resulting in the change of the density for the dataset. `Base` and `mul` influence the connections among primary and secondary rings, the number of distinct values in each level and in general the dataset density. As shown in Figure 10, as `mul` increases, a decrease in the precision is observed. The same trend is also shown for workloads generated with the same value for `mul` parameter, but with different values for the `base`. 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 11, 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.

6.4 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. In more detail, the first set of experiments is about query loads biased towards the higher levels of the hierarchies, while different

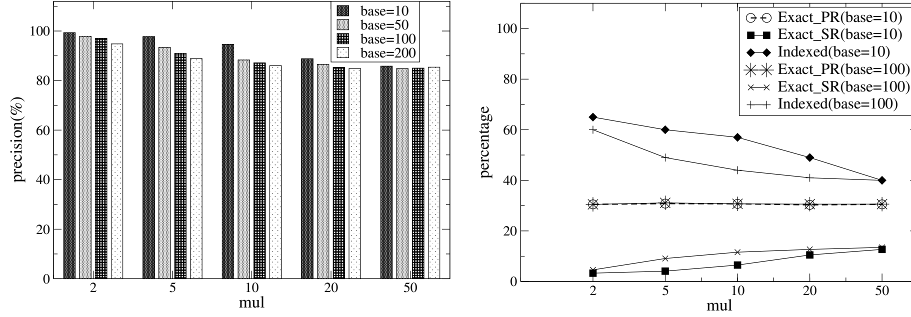


Fig. 10. Impact of `mul` and `base` in the achieved precision

Fig. 11. Percentage of each query category for different data workloads

values of *TupleDist* are utilized for their generation. The number of queries directed to each level depends on the value of *levelDist*. The results of these experiments are shown in Figure 12 focusing on the achieved precision and the percentage of `Exact Match` queries. In this Figure, it is also depicted the percentage of queries including at least one value belonging to ℓ_0 and at least one value belonging to ℓ_1, which are denoted as `Queries_L0` and `Queries_L1` respectively. As shown in the Figure, the more biased the query load towards the higher levels, the higher the precision becomes and remarkably results (close to 100%) are observed. In the biased loads, the percentage of `Queries_L0` is significant bigger that the one of `Queries_L1` and it is easier for the system to decide the required roll-up operations towards ℓ_0 in each dimension. Thus, even if the selection of ℓ_1 is not appropriate for the resolution of queries without flooding, the system manages to adjust the pivot levels of the queried hierarchies and resolve a large portion of the queries as exact matches (denoted as `Exact Match`), proving that the re-indexing mechanisms are highly effective.

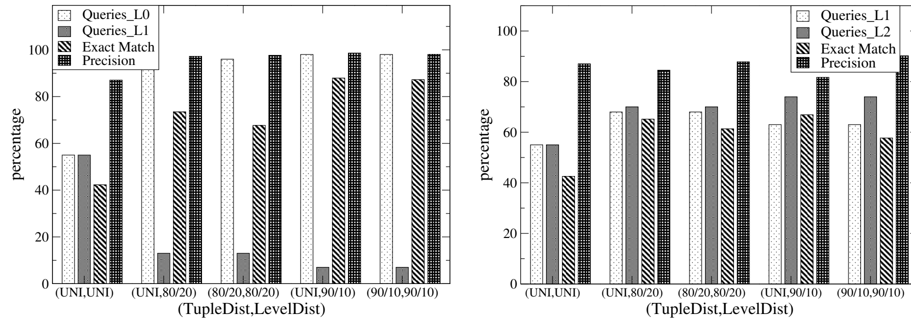


Fig. 12. Precision and exact match queries for skew towards higher levels and various *(TupleDist,levelDist)* combinations

Fig. 13. Precision and exact match queries for skew towards lower levels and various *(TupleDist,levelDist)* combinations

Figure 13 depicts the respective measurements, when the query loads favour the lower levels of the hierarchies. A decrease is noticed in the precision of loads, where the *levelDist* becomes more biased for the same *TupleDist*. This is due to the fact that lower levels of the hierarchy have a considerably larger number of values. As the number of queries targeting the lower levels increases, the probability of queries targeting non-indexed values is higher until the re-indexing mechanisms adapt the pivot levels of the popular trees appropriately. Since the percentage of `Queries_L1` and `Queries_L2` does not indicate clearly which level is a a more appropriate selection as pivot level,the percentage of exact matches is lower in the biased loads compared to the achieved one in the first set of experiments (see Figure 12).

6.5 Testing against Partial Materialization

Apart from the re-indexing operations, the materialized combinations can be also utilized to minimize the query cost. In the next experiment, the method is tested 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`).

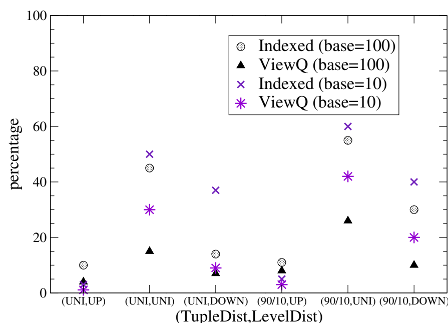


Fig. 14. Utilization of materialized combinations compared to queries resolved as indexed

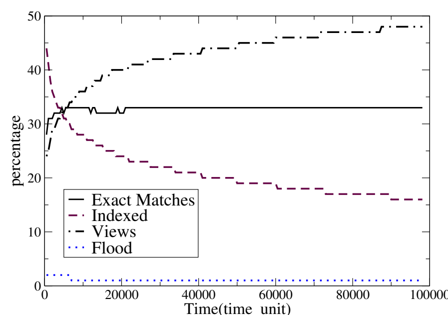


Fig. 15. Utilization of identifiers over time compared to other queries for the UNI query load

As shown in Figure 14, the percentage of queries resolved with the utilization of a precomputed combination (`ViewQ`) increase in the query loads with 90/10 as *TupleDist* compared to the percentage of the corresponding loads with UNI *TupleDist* for the same values of *levelDist*. This happens due to the following fact: if a part of the dataset is queried at most, then the probability of asking a calculated combination of values increases as well. Thus, more queries are resolved with the use of combinations. Moreover, the percentage of retrieving an answer from a stored combination is higher for uniform (`UNI`) *levelDist*. In this case, the re-indexing mechanisms cannot adjust the pivot levels to all the incoming queries and the indexed queries are more often resulting in the utilization of more identifiers.

Figure 15 depicts how queries are resolved during the simulation of the query load that targets mostly a specific part of the dataset and uniformly all the levels of the hierarchy in Figure 14. The total number of queries has increased to 100k and the percentage of queries answered with the use of a materialized combination is not included in the percentage of the indexed queries. It can be observed that the utilization of view identifiers increases over time and less queries needs to be forwarded across the indices.

6.6 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 16 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.

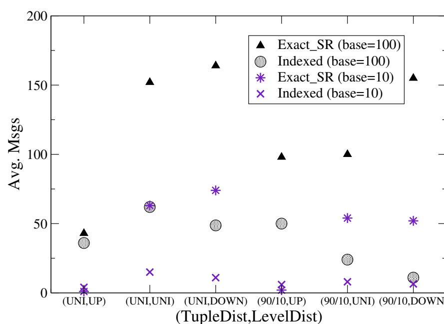


Fig. 16. Average number of messages for exact matches in secondary rings and indexed queries

6.7 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 with the APB-1 benchmark [9]. 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 17 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 18 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 17. 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.

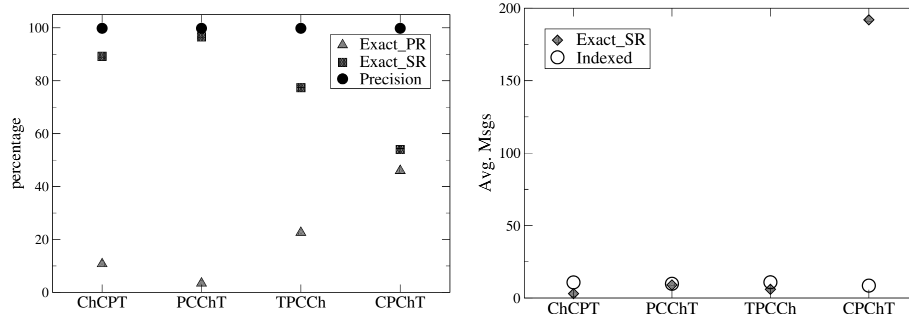


Fig. 17. Precision for APB query workload in *LinkedPeers*

Fig. 18. Average number of messages for exact match and indexed queries

6.8 Performance Evaluation for Web Data

In this section, we exhibit the performance of the proposed setup for hosting hierarchical data coming from Web sources following the discussion of Section 5. The presented response times are measured in a real testbed consisting of 16 physical machines with a Xeon processor at 2GHz with 8GB of memory running a 64-bit Debian Linux kernel. The nodes communicate through FreePastry sockets using Gigabit Ethernet.

The local databases for each node are setup with SQLite [10]. The schema used is a single table; a more sophisticated schema may also be used to improve the retrieval performance. Yet, the purpose is to showcase the advantages of a unified distributed indexing scheme for multiple RDF repositories. Further performance improvements in the storage of such data in the local databases is outside the scope of our evaluation. Moreover, the functionality of pre-computing aggregated facts of various combinations for future use has been disabled for the specific experiment.

The proposed approach is compared to a setup consisting of a central repository (denoted as `VirtStore` in the Figures) built with the use of *Virtuoso Open-Source Edition version 6.1* [11], which is a centralized triple store for RDF data. The specific store is a popular open-source solution for storing RDF data. The default configuration provided during the Debian repository installation is utilized. Queries to the store are posed from a client hosted on a different machine. Queries are executed using the Virtuoso Jena provider [12], a fully operational Native Graph Model Storage Provider enabling Semantic Web applications to directly query the Virtuoso RDF store. Before the execution of a set of queries, the system is forced to drop all filesystem caches; the database is also rebooted to clear any internal caches. During the execution of a queryset, the database creates and uses its internal caches.

The comparison dataset in the presented experiments is created using the Lehigh University benchmark (LUBM) [13]. The specific benchmark generates synthetic datasets of any size that commit to a single realistic ontology and models information encountered in the academic domain. The LUBM ontology, while not a large one, has the distinctive property of having a small number of different values in the most generic level while being extremely wide in the level of the leaves. For the experimental evaluation, a dataset featuring 100 universities with 18 different predicates has been generated resulting in a dataset of 13.5M triples. By default, ℓ_2 is chosen as the pivot level.

The benchmark also includes a set of different query categories with various levels of complexity and selectivity. A significant characteristic of these queries is that some of the categories assume the `rdfs:subClassOf` relationship among the concepts. This property is handled transparently by the proposed platform. A set of nine meaningful queries is included in order to provide direct comparison between the proposed scheme and the centralized store. These queries do not particularly favor any specific storage scheme and require no complex execution plan. Each category of query is now described in detail:

Query 1 (LQ1): This query aims to find any person whose type is `Graduate-Student` and is related to a specific course (e.g., `GraduateCourse0`) according to the relationship `takesCourse`. The resolution of this query starts with the lookup of the value `GraduateStudent`. If no results are found, then the value of the course is looked up and if it is not a pivot key or indexed value, then the query is flooded.

Query 2 (LQ2): This query refers to all publications related through the `publica-tionAuthor` property to a specific professor (e.g., `AssociateProfessor0`). The difference with LQ1 is that the class `Publication` has a wide hierarchy.

Query 3 (LQ3): This query targets the retrieval of all the information related to a professor that `worksFor` for a specific university's department (i.e. `Department0`). It also queries about multiple properties of a single class and most of these proper-

ties do not link the subject with other entities: For this reason, they are only stored locally in the node of the specific professor (i.e., the property about the `name` and the `emailAddress`).

Query 4 (LQ4): This query searches for all `Professors` that work for any department of a specific university (e.g., `University0`)) and selects someone only if she is `Chair` of the department.

Query 5 (LQ5): This query is about all the persons that are `memberOf` of a specific department of a university (e.g., `Department0`). The execution of this query can be done in parallel if splitted into two queries corresponding to the subClasses of `Person`. In fact, it has been eliminated the `Person` from the root level. The majority of entities included in the LUBM dataset inherit the `Person` class and thus the purpose was to avoid the creation of a very steep hierarchy. The results for each query are returned back to the node that initiated the query and their union is presented to the user.

Query 6 (LQ6): This query searches for all the members of the class `Student` or the members of one of its subclasses, for example the members of `Undergraduate-Student`.

Query 7 (LQ7): The purpose of this query is to find all `Students` that are related through `takesCourse` to the courses taught by a specific professor (e.g., `Associate-Professor0`). In this query, `Course` always appears as an object in the subqueries.

Query 8 (LQ8): This query resembles the previous category but it is characterized by increased complexity. It searches for all the `Students` that are members of all the `Departments`, which in their turn are `subOrganization` of a specific university. In this complex query, the department appears both as a subject and an object in the relative triples stored and indexed in the system.

Query 9 (LQ9): This query searches for all the students that `takesCourse` a specific course (e.g., `GraduateCourse0`).

At first, we present the query response times in Figures 19 and 20 for all categories of the defined queries for the LUBM dataset. In the generated queries, we vary the values of the constants (e.g., `GraduateCourse0`, `AssociateProfessor0`, etc) choosing uniformly among the children that adhere to the type defined by the query and average the response times over 1,000 iterations. A single client poses queries to both systems, waiting until the answer(s) are received before posing a new query. For VirtStore, we also register the maximum response time per query: this corresponds to the time required for the first answer, since the system performs some main memory result caching to respond faster to the consequent queries.

Figure 19 depicts the response time for the categories of queries that are mainly of the *(?s, o, p)* format, where the subject is required to be of a specific type. Moreover, these categories of queries have small input and high selectivity. In Figure 20, the categories of queries become more complex and return a large number of rows. The average number of returned triples per query is shown in the respective tables above the figures. The results show that LinkedPeers achieves better response times for all categories of queries.

Even as the complexity of the queries increases, our system manages to resolve the queries efficiently as shown for queries LQ7 and LQ8, which are path queries and involve more joins. Moreover, each node holds a smaller portion of the whole dataset con-

tributing to its faster processing. Moreover, due to the fact that our system incorporates the whole hierarchy information, we perform less lookups to discover if a found entity is of the requested type. These queries are resolved 2.7 and 1.8 times faster compared to the VirtStore. LQ5 is another category which is strongly favoured by our distributed approach, since the query is split into two different subqueries executed in parallel, resulting in a faster response time of the order of 15 times. A significant speedup is achieved for LQ6 as well, which is the simplest query that can be posed to our system: It can be directly resolved by the maintained tree structures.

Query	LQ1	LQ2	LQ3	LQ4
Avg. # of rows	4	14	29	20

Query	LQ5	LQ6	LQ7	LQ8	LQ9
Avg. # of rows	625	956711	44	11532	20

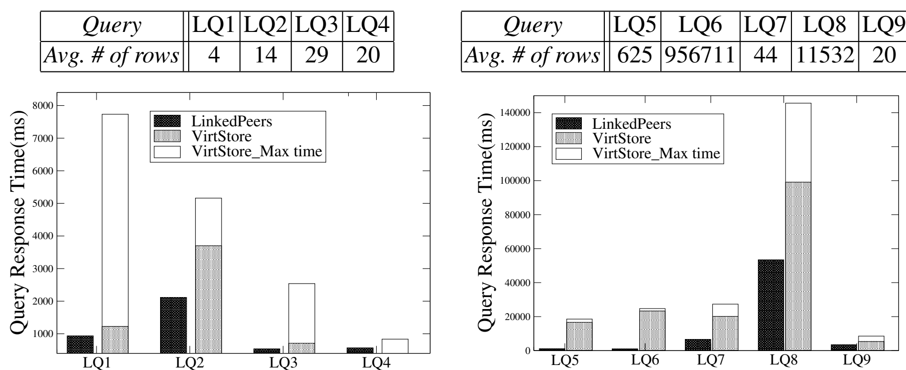


Fig. 19. Performance comparison of the response time(ms) for query categories 1–4 of the LUBM benchmark

Fig. 20. Performance comparison of the average response time(ms) for query categories 5–9 of the LUBM benchmark

We also test the performance of both systems under increased load posed from concurrent users, as is the case in most web applications hosting linked data. To achieve this goal, we generate a query workload of 2500 randomly chosen queries from all LUBM categories. First, we send the queries to the systems using a single client. The average response times are shown in Figure 21. Our system outperforms the centralized approach by answering the queries almost 5 times faster on average, which is a consistent result to the achieved response times of each category. In Figure 22, the average response times are registered when the same workload is applied by 5 concurrent clients. Our system manages to maintain its faster response rates at almost the same levels observed in Figure 21. The distribution of data among multiple nodes in our system contributes to this fact due to the parallel processing of multiple queries from different nodes concurrently (even though in our implementation each node processes a single query per time). Moreover, each node holds a smaller portion of data which can be processed in less time.

7 Related Work

Various research works investigate how the effectiveness of P2P systems to handle large volumes of data can be further exploited for the creation of scalable platforms providing

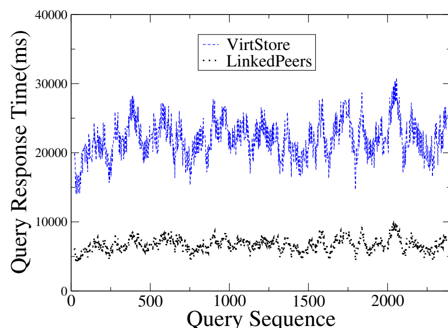


Fig. 21. Response times for each query of a workload including randomly chosen queries of the LUBM categories

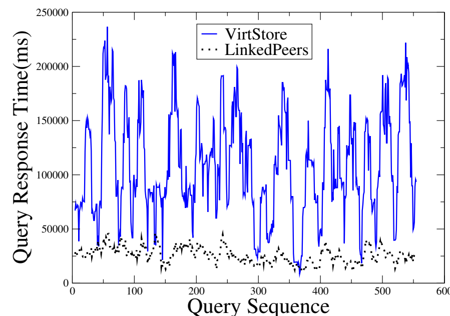


Fig. 22. Response times for a workload including randomly chosen queries of the LUBM categories and with 5 clients

advanced indexing and search functionalities. Distributed Hash Tables (hence DHTs) are significant candidates for the design of distributed systems for data-intensive applications, since DHT-based overlays present logarithmic search path lengths compared to the network size in most cases. Nevertheless, the imposed indexing does not support complex queries and more advanced indexing schemes according to the structure of the data and the types of the queries are needed.

A major category under consideration is the one of Peer Database Management Systems (PDMSs) emphasizing on storing relational data and supporting operators such as selection, projection, union, etc. PIER [14] falls into this category and utilized a DHT-based overlay for the insertion of self-describing tuples. The query resolution mainly relies on an additional tree like structure implemented on top of the DHT-based overlay. Piazza [15] is another effort focusing on XML data and achieves better search performance by reformulating queries and pre-computing semantic paths. In PeerDB [16] system, the limitation of an existing predefined schema is surpassed and agents are responsible for the processing of queries. In all these cases, the systems are based on a global indexing for all stored tuples and this act results in a costly procedure in terms of bandwidth consumption and time as the volume of data and the number of schemas and attributes increases, especially when high update and churn rates are observed. A more dynamic solution for the creation of a peer-based data management system that indexes only a portion of the stored tuples is PISCES [17]. The partial index is built upon approximate information gathered by a histogram based approach about the total number of nodes, the total query number, the query distribution and nodes' arrivals and departures. Nevertheless, this approach targets mainly to support queries regarding relational data and does not handle the special case of hierarchies over multidimensional dataset and the computation of aggregate queries. As far as the latest aspect is concerned, the online processing of aggregate queries is described is the Distributed Online Aggregation (DOA) Scheme [18]. In this case, the calculation of results is approximate and multiple iterations take place: in each iteration a small set of random samples are retrieved from data sites and assigned to available nodes for processing. The random sampling among the distributed nodes may bias the calculated result and

premises that each node must provide an interface for picking random sample from its database. Apart from this fact, all data need to be mapped to a global schema, which many times is not feasible and introduces complexity to the system.

Another category of approaches coping with the indexing of data items described by multiple items in P2P systems introduces the replacement of the hash function of DHTs overlays with functions taking into account the values of the attributes during the assignment of the items among the nodes. For this reason, Space Filling Curves (hence SFCs) are used for the generation of keys, since it is highly probable to produce locality preserving mappings of the values of multiple attributes to a single key. Towards this direction, various P2P systems can be found in the literature, such as Squid [19], CISS [20], SCRAP [21]. In these systems, the resolution of range queries is mainly studied by mapping the ranges in the values of attributes to ranges of keys and retrieving them from the DHT. In CISS, the problem of hierarchical attributes is also considered. Nevertheless, it is assumed that the full path from the most general level towards a detailed level is known and the values for all attributes are defined in the query, so as a small number of resultant key ranges to retrieved by issuing consecutive lookups. The type of queries studied in this work does not meet this requirement and such an approach would result in flooding at most cases.

In the case of semi-structured data published according to the RDF model, many approaches combine techniques from the relational databases to build large centralized repositories that index and query such data, as it has been discussed in [22]. The most representative categories are the triple stores and vertically partitioned tables. In most cases, the triple stores such as Virtuoso [11], 3store [23] and RDF-3X [24] store RDF triples in a simple relational table. A followed methodology is to collect large dumps of data (possible through crawling), to preprocess and to load these dumps in a centralized resource so as to enable querying of the merged data locally. Although centralized solutions are advantageous during query processing by providing access to the whole dataset, these solutions are vulnerable to the growth of the data size ([25], [26]) and the synchronization of local copies in the centralized repositories, especially when the data sources change frequently or the RDF instances are created on-the-fly. As the growing volume of data cannot be handled efficiently in centralized solutions, various distributed approaches have been proposed in the literature. As far as structured P2P systems are concerned, the majority of efforts focuses on distributing RDF data among multiple peers. RDFPeers [27] was one of the first efforts to store triples on top of a MAAN overlay [28] by hashing the subject, the object and the property and insert the same triple with three different keys in the overlay. ATLAS [29] uses also RDFpeers for querying RDF data. GridVine [30] follows a similar approach to RDFPeers for inserting RDF triples in a P-Grid overlay and storing them in the local database of the node. All these efforts may easily lead to overloaded nodes with poor performance for popular triples (i.e. the node responsible for the key of rdf:type). Also, they cannot exploit the semantic information included in this kind of data and they are forced to build additional semantic layers or interfere to the organization of the overlay according to the semantics of (e.g. [31], [32]) or extend the RDF model. These modifications add more complexity and increase the cost for maintenance of the external structures and during update procedures. Also, all these approaches encounter all the problems of distributing

query processing, where large volumes of data need to be fetched in the query initiator and joined before they proceed to the evaluation of the rest of the parts of the query.

8 Conclusions

In this work, we described the system called *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. The objective of efficient management of such data is analyzed for the use case of RDF data published by web sources and the experimental results also verify that the resultant system appears as a viable solution for storing and organizing such data compared to centralized approaches.

References

1. L. Data, “Connect Distributed Data across the Web,” http://linkeddata.org/.
2. A. Asiki, D. Tsoumakos, and N. Koziris, “Distributing and searching concept hierarchies: an adaptive dht-based system,” *Cluster Computing*, vol. 13, pp. 257–276, 2010.
3. C. Bizer, T. Heath, and T. Berners-Lee, “Linked Data - The Story So Far,” *Int. Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.
4. RDF, “Resource Description Framework(RDF),” http://www.w3.org/RDF/.
5. SPARQL, “SPARQL Query Language for RDF,” http://www.w3.org/TR/rdf-sparql-query/.
6. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, “Dbpedia - a crystallization point for the web of data,” *Web Semant.*, vol. 7, pp. 154–165, September 2009.
7. H. Halpin, “A query-driven characterization of linked data,” in *LDOW*, 2009.
8. FreePastry, http://freepastry.rice.edu/FreePastry.
9. apb, *OLAP Council APB-1 OLAP Benchmark*, http://www.olapcouncil.org/research/resrchly.htm.
10. SQLite, “http://www.sqlite.org/.”
11. O.-S. E. Virtuoso, “Version 6.1,” http://www.openlinksw.com/wiki/main/Main.
12. JenaProvider, “Virtuoso jena provider,” http://www.openlinksw.com/dataspace/dav/wiki/Main/VirtJenaProvider.
13. Y. Guo, Z. Pan, and J. Heflin, “An evaluation of knowledge base systems for large owl datasets,” in *International Semantic Web Conference*, 2004, pp. 274–288.
14. R. Huebsch, J. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica, “Querying the Internet with PIER,” in *VLDB*, 2003.
15. I. Tatarinov and A. Halevy, “Efficient query reformulation in peer data management systems,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, ser. SIGMOD ’04. New York, NY, USA: ACM, 2004, pp. 539–550.
16. B. C. Ooi, K.-L. Tan, A. Zhou, C. H. Goh, Y. Li, C. Y. Liau, B. Ling, W. S. Ng, Y. Shu, X. Wang, and M. Zhang, “Peerdb: Peering into personal databases,” in *SIGMOD Conference*, 2003, p. 659.
17. S. Wu, J. Li, B. C. Ooi, and K.-L. Tan, “Just-in-time query retrieval over partially indexed data on structured p2p overlays,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD ’08. New York, NY, USA: ACM, 2008, pp. 279–290.

18. S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan, "Distributed online aggregations," *Proc. VLDB Endow.*, vol. 2, pp. 443–454, August 2009.
19. C. Schmidt and M. Parashar, "Enabling flexible queries with guarantees in p2p systems," *IEEE Internet Computing*, vol. 8, pp. 19–26, May 2004.
20. J. Lee, H. Lee, S. Kang, S. M. Kim, and J. Song, "CISS: An efficient object clustering framework for DHT-based peer-to-peer applications," *Computer Networks*, vol. 51, no. 4, pp. 1072–1094, 2007.
21. P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: multi-dimensional queries in p2p systems," in *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, ser. WebDB '04. New York, NY, USA: ACM, 2004, pp. 19–24.
22. K. Hose, R. Schenkel, M. Theobald, and G. Weikum, "Database foundations for scalable rdf processing." in *Reasoning Web*, ser. Lecture Notes in Computer Science, A. Polleres, C. d'Amato, M. Arenas, S. Handschuh, P. Kroner, S. Ossowski, and P. F. Patel-Schneider, Eds., vol. 6848. Springer, 2011, pp. 202–249.
23. S. Harris and N. Gibbins, "3store: Efficient bulk RDF storage," in *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*. Citeseer, 2003, pp. 1–20.
24. T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, pp. 91–113, February 2010.
25. P. Haase, T. Mathäß, and M. Ziller, "An evaluation of approaches to federated query processing over linked data," in *Proceedings of the 6th International Conference on Semantic Systems*, ser. I-SEMANTICS '10. New York, NY, USA: ACM, 2010, pp. 5:1–5:9.
26. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich, "Data summaries for on-demand queries over linked data," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 411–420.
27. M. Cai and M. Frank, "Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network," in *Proceedings of the 13th international conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 650–657.
28. M. Cai, M. Frank, J. Chen, and P. Szekely, "Maan: A multi-attribute addressable network for grid information services," *Journal of Grid Computing*, vol. 2, pp. 3–14, 2004, 10.1007/s10723-004-1184-y.
29. Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi, "Atlas: Storing, updating and querying rdf(s) data on top of dhts," *Web Semant.*, vol. 8, pp. 271–277, November 2010.
30. K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. V. Pelt, "Gridvine: Building internet-scale semantic overlay networks," in *Int. Semantic Web Conference*, 2004, pp. 107–121.
31. M. Karnstedt, K.-U. Sattler, M. Hauswirth, and R. Schmidt, "A dht-based infrastructure for ad-hoc integration and querying of semantic data," in *Proceedings of the 2008 International Symposium on Database Engineering and Applications*, ser. IDEAS '08. New York, NY, USA: ACM, 2008, pp. 19–28.
32. J. Zhou, W. Hall, and D. De Roure, "Building a distributed infrastructure for scalable triple stores," *Journal of Computer Science and Technology*, vol. 24, pp. 447–462, 2009, 10.1007/s11390-009-9236-1.