Cluster Comput (2010) 13: 257–276
DOI 10.1007/s10586-010-0136-5

Distributing and searching concept hierarchies: an adaptive DHT-based system

Athanasia Asiki · Dimitrios Tsoumakos · Nectarios Koziris

Received: 10 November 2009 / Accepted: 18 March 2010 / Published online: 8 April 2010

Abstract Concept hierarchies greatly help in the organization and reuse of information and are widely used in a variety of information systems applications. In this paper, we describe a method for efficiently storing and querying data organized into concept hierarchies and dispersed over a DHT. In our method, peers individually decide on the level of indexing according to the granularity of the incoming queries. Roll-up and drill-down operations are performed on a per-node basis in order to minimize the required bandwidth for answering queries on variable aggregation levels. We motivate our approach by applying it on a large-scale Grid system: Specifically, we apply our fully decentralized scheme that creates, queries and updates large volumes of hierarchical data on-line and replace the traditional centralized and strictly indexed information systems. Our extensive experimental results support this argument on many diverse configurations: Our system proves very efficient in skewed workloads, both over single and multiple hierarchy levels at the same time. It adapts to sudden changes in popularity and effectively stores and updates large amounts of data at very low cost.

This work was partly supported by the European Commission in terms of the GREDIA FP6 IST Project (FP6-34363).

A. Asiki (✉) · D. Tsoumakos · N. Koziris
Computing Systems Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece
e-mail: nasia@cslab.ece.ntua.gr

D. Tsoumakos
e-mail: dtsouma@cslab.ece.ntua.gr

N. Koziris
e-mail: nkoziris@cslab.ece.ntua.gr

Keywords Distributed hash table · Concept hierarchies · Adaptive indexing · Grid information system

1 Introduction

A concept hierarchy (or *taxonomy*) defines a sequence of mappings from more general to lower-level concepts. For example, Fig. 1 shows a simple hierarchy for the `Virtual Organization` concept, where

$$\texttt{VO} < \texttt{Category} < \texttt{Region} < \texttt{Site}$$

and one for `time` where a *partial* order is defined. Concept hierarchies are important because they allow the structuring of information into categories, thus enabling its search and reuse. Specifically, users may view data at different levels of a dimension hierarchy: With the *roll-up* operation we climb up to a more summarized level of the hierarchy, while a *drill-down* defines the opposite operation (i.e., navigating to lower levels of the hierarchy with increased detail). The drilling paths are usually defined by the hierarchies within the dimensions. The mappings of a concept hierarchy are usually provided by application or domain experts.

Works in the field of data-warehousing (e.g., [10, 18], etc.) utilize hierarchies across the dimensions of a data cube but these present strictly centralized solutions. In the area of distributed computing, while there has been considerable work in sharing simple relational data using both structured and unstructured overlays (e.g., [13, 14, 17]), no special consideration has been given to data supporting hierarchies. We investigate the problem of indexing and querying such data in a way that preserves the semantics of the hierarchies and is efficient in retrieving the requested values in a fully distributed environment.

To motivate our approach, we describe how it can be applied in order to function as a distributed and efficiently

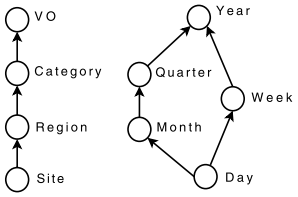

Fig. 1 A concept hierarchy for the VO and Time (lattice) dimensions

operational grid *information system*. Grid computing allows for coordinated resource sharing and problem solving in dynamic virtual organizations (*VOs*). A VO is a group of users from multiple institutions who collaborate to achieve a specific goal. The goal of grid computing is to provide a network of systems which, acting like a single supercomputer, offers resources that are easily accessible. In order for jobs to be adequately served by the most appropriate resources, the information system stores all needed information about the characteristics of the available resources over time.

There exist a number of systems that accomplish the tasks of an information system (e.g., [3, 5], etc.). Nevertheless, they either feature a central information repository or a hierarchy of aggregation sites that introduce both scalability (single points of failure) and performance (processing burden on a single site) issues. Work in the area of distributed databases offers a variety of systems which can be used to disseminate and query this information. Nevertheless, these schemes cannot be used to maintain the semantics of the hierarchy and efficiently retrieve views of the data at different granularities. This is very important for applications such as a large scale information system, as queries naturally target different levels of detail: Historic queries usually require grouping by the highest hierarchy levels (e.g., group-by `VO` or group-by `Year`), whereas online queries are naturally directed towards more detailed levels (e.g., group-by `Site` or by `Day`).

Let us assume that the system's database contains a location dimension that relates to the `VO` location information (see Fig. 1). Monitoring information is described by attributes (*facts*) such as the number of running jobs, number of waiting jobs, available storage space, total storage space, etc. Common accounting queries could be "*Give me the average CPU time*" or "*Give me the minimum available space in Gbytes*", presented to the user grouped by a VO value.

In this paper, we present a system that efficiently stores, queries and updates data organized in concept hierarchies. We choose the DHT as a reliable substrate over which we store, index and query our data, thus eliminating all possible bottlenecks created by the hierarchical or centralized approaches mentioned before. Data producers individually insert data to the distributed information system. Queries are still answered, while incremental updates are efficiently processed. Our solution takes the query granularities into account, adjusting the indexing structure to favor performance. Second, we intend to provide a system that will preserve all hierarchy-specific information. In our technique, a tree-like data structure is used to store data and maintain indices to related keys, enabling us to respond to more complex, hierarchy-based queries such as: "*Which* `sites` *correspond to* `VO` *'Biomed'* " or "*What* `category` *does* `region` *'AsiaPacific' belong to*". We can summarize the contributions of this paper in the following points:

- We present a complete storage, indexing and query processing system for hierarchical data. This system has many desirable properties: It adapts the granularity of its indexing according to incoming requests; Performs efficient and online incremental updates; Maintains data in a fault-tolerant and fully distributed environment.
- We motivate the usefulness of this scheme by customizing it to serve as a high-performance information system. We show how our method outperforms traditional approaches by eliminating offline processing and other performance bottlenecks.
- We present a thorough performance analysis in order to identify the behavior of our scheme under a large range of work and data loads.

The rest of this paper is organized as follows: The next Section summarizes related work both in exploiting hierarchies as well as existing information systems. Section 3 defines the problem and presents our solution in detail, while Sect. 4 refers to the case study of the information system. Section 5 analyzes some aspects considering the functionality and optimization of our system. In Sect. 6 the experimental setup is described and the system is evaluated based on the collected results, while we conclude our work in Sect. 7.

2 Related work

There has been significant work in the area of databases over P2P networks. PIER [13] proposes a distributed architecture for relational databases supporting operators such as join and aggregation of stored tuples. A DHT-based overlay is used for query routing. The Chatty Web [7] considers P2P systems that share (semi)-structured information but deals with the degradation, in terms of syntax and semantics, of a query propagated along a network path. In [20], the authors propose optimization techniques for query reformulation in P2P database systems.

In GrouPeer [14], SPJ queries are sent over an unstructured overlay in order to discover peers with similar schemas. Peers are gradually clustered according to their schema similarity. PeerDB [17] also features relational data sharing without schema knowledge. Query matching and rewriting is based on keywords provided by the users. GridVine [8], and pSearch [19] are based on structured P2P overlays. GridVine hashes and indexes RDF data and schemas,

and pSearch represents documents as well as queries as semantic vectors. All these approaches offer significant and efficient solutions to the problem of sharing structured and heterogeneous data over P2P networks. Nevertheless, they do not deal with the special case of hierarchies over multi-dimensional datasets.

An interesting method for representing hierarchical data is presented in [15]. The method is applied on unstructured networks containing XML documents in order to favor the routing of path queries. Each XML document is represented by an unordered label tree and bloom filters are used to summarize it.

Several indexing schemes have been presented for storing data cubes (e.g., [16, 21]). However, only few support both aggregate queries and hierarchies. In [18], hierarchies are exploited to enable faster computation of the possible views and a more compact representation of the data cube. The *Hierarchical Dwarf* contains views of the data cube corresponding to a combination of the hierarchy levels. The other approach is the DC-Tree [10]. In this work, the attributes of a dimension are partially ordered with respect to the valid hierarchy schema for each dimension. The DC-tree stores one concept hierarchy per dimension and assigns an ID to every attribute value of a data record that is inserted. These approaches are very efficient in answering both point and aggregate queries over various data granularities but do so in a strictly centralized and controlled environment.

There exist multiple systems and architectures proposed to implement the information system component. The most common is the *Globus Monitoring and Discovery Service (MDS)* [3]. A *Grid Index Information Service (GIIS)* provides an aggregate directory of lower level data stored at multiple *Grid Resource Information Services (GRISs)*. The hierarchical structure that can be composed between GIISs enables complete information retrieval by querying the top level GIIS. However, MDS has shown not to be a solution for large-scale production because it does not scale: Multiple client requests quickly lead to an overload of the top level GIIS [22]. Another schema used especially for accounting and publication of user-level information is the *Relational Grid Monitoring and Discovery Service (R-GMA)* [5], which supports complex type of queries allowed by relational databases. R-GMA presents information as a single virtual database containing a set of virtual tables, nevertheless the bulk of data need be transferred offline to a centralized database after a period of time, with all the performance drawbacks that this entails. The major drawback in all these methods is the fact that none of these architectures scale as the number of data collectors increase [22]. Moreover, they all assume an offline (or periodic at best) data migration phase to a more central location where global information can be available. In contrast, we propose a completely decentralized system where all data are continuously available and indexed at the requested granularities for fast retrieval.

3 An adaptive indexing scheme to support concept hierarchies

In this paper, we describe a system for processing bulk hierarchical data in a DHT-based overlay. Our goal is to enable efficient querying while preserving the hierarchy semantics. In addition to the fact that the DHT substrate can transparently handle node churn, replication, reliable distributed storage, etc., our technique offers adaptive indexing according to the granularity of the incoming queries and online updating with low cost and no downtime.

3.1 Notation

Let the data items stored in the system be in the form of *tuples* containing values for all levels ℓ_i of a concept hierarchy with L levels. They also contain a numerical fact of interest (e.g., `CPU Time, Available Memory`, etc.) or the location of actual data. We call the uppermost level of the hierarchy (ℓ_0) *root level* and its value *root key*. We define that $\ell_a < \ell_b$, where $(a, b \in [0, L-1])$, if and only if ℓ_a is higher in the concept hierarchy (namely closer to ℓ_0) than ℓ_b. The values of the hierarchy levels are organized in tree structures, one per root key. Without loss of generality, we assume that each value of ℓ_i has at most one parent in ℓ_{i-1}. During the insertion of a tuple, a level of its hierarchy is chosen and its hashed value serves as its *key* in the underlying DHT overlay. We refer to this level as *pivot level* and to its value as *pivot key*. Finally, the highest and the lowest pivot levels of the hierarchy for a specific root key are called *MinPivotLevel* and *MaxPivotLevel* respectively.

3.2 Data insertion

The key for each tuple is the result of a hash function applied on the value of the selected pivot level. Tuples are assigned to the node with ID numerically closest to the generated keys, according to the standard DHT operations.

In our system, both initial insertions as well as incremental updates are handled in a unified manner. We introduce a completely distributed catalogue containing all the root keys and their corresponding pivot keys. Each root key is stored in the node responsible for it along with the list of pivot keys that have been already inserted. The root key is also aware of the MaxPivotLevel used during tuple insertion containing its value.

The procedure followed during the tuple insertion is as follows: The root key for this tuple is generated and a lookup for this key takes place in the DHT overlay. If the root key exists, the tuple ends up in the node responsible for it. We consider an *insertion* to be the procedure followed in case that the root key does not already exist in the overlay. Otherwise, the *update* procedure is followed (see Sect. 3.5).

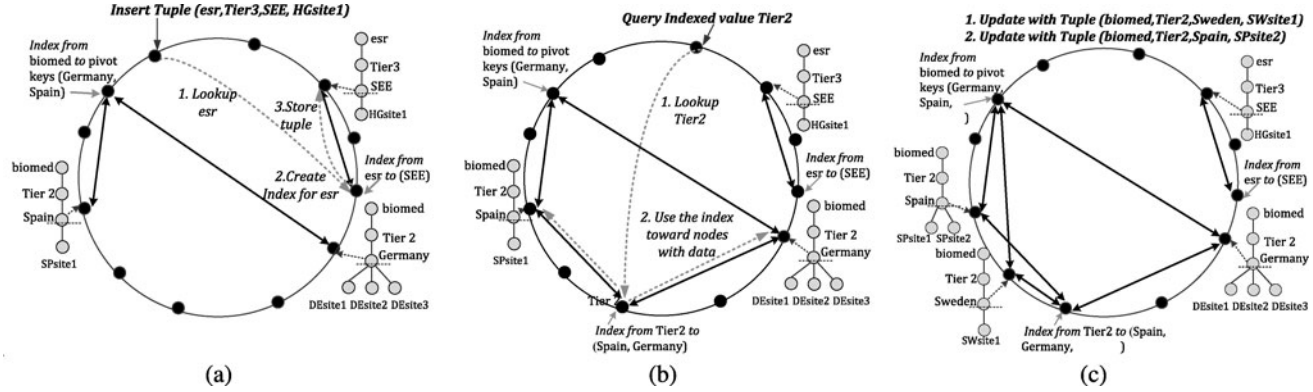

Fig. 2 (**a**) Insertion of a new tuple with its root key not already stored in the overlay. (**b**) Lookup using soft-state indices for a value belonging to a non-pivot level. (**c**) Updates for an already existing and a non-existing pivot key

In case of an insertion, the tuple or the group of tuples with the same root key arrive at the node responsible for it. The node selects a pivot level (either a random or a predefined one) and the pivot key(s) of the tuple(s) is (are) calculated for the appropriate pivot level. Each new pivot key is added to the list of pivot keys. Finally, each tuple is stored in the node with the ID closest to its pivot key.

Each peer organizes the tuples in trees that preserve their hierarchical nature. As a consequence, each distinct value of the pivot level corresponds to a tree that reveals part of the hierarchy. When a new tuple arrives at the node responsible for it, the node searches its keys. If no tuples for this pivot key have been stored, a new tree with a single branch is created. In the opposite case, a new branch is added below the value of the pivot level with the new values of the remaining levels.

An example of insertion is shown in Fig. 2(a). A graphic convention is that solid lines represent existing indices, while dotted lines correspond to logical steps followed during the described procedure. Let us assume that tuples follow the VO hierarchy depicted in Fig. 1 with ℓ_2 (Region) as the globally defined pivot level for initial insertions. A tuple with root key '*esr*' is inserted in the overlay. Since the specific root key does not already exist, a new *index* is created in the corresponding node. Afterwards, Region is selected as pivot level and the tuple is forwarded to the node responsible for this key, which creates a new tree with only one branch for this tuple.

3.3 Data lookup and soft-state indices

Queries concerning the pivot level are exact match queries and can be answered by the DHT lookup operation. Queries for any other level cannot be resolved unless flooded across the DHT. Towards the exploitation of the knowledge acquired by flooded queries, we introduce *soft-state bidirectional indices* to our scheme. When a node answers a flooded query, it checks whether a roll-up or drill-down is necessary. If this is not the case, the query initiator starts the procedure of creating an index, as soon as it receives the complete answer. It inserts the result of hashing the requested value in the DHT along with IDs of nodes having the actual tuples. The tuple holders also mark the specific value as *indexed*. The next time that a query for this key is initiated, the lookup operation locates the node holding the index, finds out the IDs of nodes with relevant tuples and retrieves them.

The created indices 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. Moreover, after the number of indices has reached a limit I_{max}, the creation of a new index results in the deletion of the oldest one(s). Overall, the system tends to preserve the most "useful" indices, namely the ones directed towards the most frequently queried data items.

The nodes with actual tuples of the indexed value need to know the existence of an index, in order to erase it after a re-indexing operation. The bidirectionality 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. Our choice is 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 it is 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 timestamp. 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.

In the example of Fig. 2(b), a query for '*SEE*' is resolved directly by the lookup operation of the DHT protocol. Lookups for values of the root level are processed utilizing the indices created during insertions. Nevertheless, any query for any level other than the pivot level or the root level ends up with no results. For example, a query for data items described by '*Tier2*' does not contact the two nodes with the corresponding trees before the creation of the index. The next step is the flooding of the query and the nodes storing the keys of '*Spain*' and of '*Germany*' are reached. The query initiator, which is now aware of the existing pivot keys, creates indices storing the information about these pivot keys to the node responsible for the value '*Tier2*'. This node now has an index pointing to the node '*Spain*' and another to node '*Germany*'. In the future, queries for '*Tier2*' will be answered without flooding, utilizing the created soft-state indices. As shown analytically in Fig. 2(b), a query for the `Category` '*Tier2*', after the creation of the indices, reaches the node responsible for this key, which in turn forwards the query to all relative nodes directly. The nodes storing the trees with the queried value return only the relevant tuple(s).

Since root indices are created during the insertion of tuples, an optimization for the flooding scheme can be applied to reduce the number of messages required for flooding, taking advantage of their existence. When a flooded query is forwarded from a node to its subsequent neighbor in the overlay, the covered range of IDs of the visited node (hence *CoveredIdRange*) is registered. According to the described approach of storing the data, if the query regards a value belonging to a level below the pivot level, the corresponding tuples can be retrieved by only one node. Therefore, the forwarding of the flooded query terminates, as soon as the node with the queried value is contacted. In case of a query for a value above its pivot level, the forwarding of the query cannot be terminated when the first contacted node responsible with a tree containing this value is found. Nevertheless, this node can answer to the initial node with its corresponding tuples and it can forward the query to the node responsible for the root key. Upon the delivery of the flooded query to the node with the root index, the node examines the CoveredIdRange of all visited nodes and sends the query in parallel only to the nodes storing the pivot keys, that are candidates to answer the flooded query and are not included in the CoveredIdRange. According to this strategy, the information stored in the root key is utilized and the visiting of all nodes may be avoided during flooding. For example, let us suppose data tuples organized as shown in Fig. 2(a). A query for '*Tier2*' is flooded and thus it can be forwarded from a node to its subsequent neighbor clockwise. In this case, the node with the pivot key '*Germany*' is reached at first. The specific node forwards the query to the node responsible for the root index as implied by the described optimization. Eventually, the node with the root index sends the flooding query only to the node responsible for the non-visited pivot key '*Spain*'.

3.4 Re-indexing operation

Our goal is the described system to dynamically adapt on a per node basis to online queries, so as to increase the ratio of the non-flooded queries. In order to achieve this goal, we introduce two re-indexing operations regarding the selection of pivot level: *roll-up* towards more general levels of the concept hierarchy and *drill-down* to levels lower than the pivot level.

The idea behind individual re-indexing of stored tuples is based on the fact that each node has a global view of the queries regarding each level $\ell_i < pivotlevel$, but only a partial view of the queries for each level $\ell_i > pivotlevel$. Therefore, it has sufficient information to decide if a drill-down will favor the increase of the exact-match queries for its values. On the other hand, a node has to cooperate with other peers that store a value of a level $\ell_i < pivotlevel$ in order to decide if this level is more appropriate.

The re-indexing of the data tuples (through a choice of a different pivot level) is performed on a per-tree basis, requiring no global coordination. Each node collects information using the incoming queries and finds out if the pivot level of a tree remains its most popular level. Otherwise, the node proceeds with the re-indexing of the tuples of this tree. The popularity of the levels of a tree is estimated based on their average rates of incoming queries (hence *InQ*) over a time period. A node maintains one record per tree with these rates during a restricted time-frame W. This parameter should be properly selected to perceive variations of query distributions and, at the same time, stay immune to instant surges in load.

In more detail, the mechanism works as follows: A node may check if a re-indexing is required based on the objective to achieve. The implemented strategy implies that a node decides whether a roll-up or drill-down is required when it answers a flooded query or when a number of queries for indexed values have been received. While the main objective is the increase of the queries answered without flooding, this strategy targets to the increase of exact match queries as well.

The number of queries for indexed values triggering a node to examine a possible re-index may vary and has an impact to the adaptiveness of the system. A small value indicates that the potential of re-indexing is examined more often and thus more re-indexing operations may take place. Nevertheless, if a decision has erroneously been made, it can be easily corrected. However, during re-indexing operations, existing indices are deleted and this may have a negative impact on the system. In the opposite case, the system tends to

depend more on the effectiveness of the indices. We have observed that re-indexing operations are necessary, when popular values belong to levels with several distinct values. Indices perform better for higher levels with less values, since there is a high probability for repeated utilization of an index.

A node decides if a re-indexing operation will favor the increase of non-flooded queries based on the '*popularity*' of each level according to the procedure described in basic steps in Algorithm 1. A *thr* parameter is used to indicate if a re-indexing operation is required. The following criterion defines if a re-indexing to a level ℓ_q is allowed:

$$InQ_{\ell_q} > thr \times \sum_{i=0}^{i=L-1} InQ_{\ell_i}, \quad \ell_q \neq pivotlevel, thr \in [0,1].$$

In the described algorithm, two possible cases are considered to indicate the necessity of a re-indexing operation: **The queried level ℓ_q lies lower in the hierarchy than the pivot level of the tree ($\ell_q > pivotlevel$).** Only one tree stores the values of a level below the pivot level. Therefore, the specific node is aware of the exact popularity of these values and feels 'confident' to decide if a drill-down is needed. If the most popular level ℓ_{pop} of the tree lies below the pivot level and the defined criterion is valid for its InQ, then a drill-down to this level is decided. After the decision for drill-down is made, the node finds all the distinct values of the new pivot level and hashes them one by one, sending the new groups of tuples to the corresponding nodes. The already gathered statistic information is sent along with one randomly selected group, in order to maintain information about the query distribution for the values contained in the drilled-down tree within W. Any existing indices for any value of this tree are removed. If a drill-down is not needed, the node includes in its answer to the initiator the fact that the queried level is lower than the pivot level, hence it can carry on with the creation of the soft-state index and expedite the process.

The queried level ℓ_q lies higher that the pivot level of the tree ($\ell_q < pivotlevel$). In this case, there are more than one trees with this value needed to participate in a possible roll-up to this level. Otherwise, lookups for this value will not return complete results. If the threshold criterion is satisfied for the ℓ_q, then the node is positive to the potential of adopting this level as pivot level for this tree. This step is indicative of an imbalance and the query initiator is informed about this. The query initiator decides for a re-indexing operation according to the procedure described in Algorithm 2. If the query initiator is aware of at least one node willing to

Algorithm 1 Decision Algorithm in the node answering a query

pivotlevel: current pivot level
ℓ_q: the queried level of the flooded or indexed value
NotPivotKey: the flooded or indexed value
InQ_{tot}: rate of incoming queries for the tree with *NotPivotKey*
InQ_{ini}: initial minimum rate to allow re-index operations
$InQ_{l_{pop}}$: rate of incoming queries for the most popular level
action: the decided action
$\ell_{pop} \Leftarrow FindMostPopularLevel$
if $\ell_q > pivotlevel$ **then**
  if $(InQ_{tot} > InQ_{ini})$ AND $(\ell_{pop} > pivotlevel)$ AND $(InQ_{\ell_{pop}} > thr \times InQ_{tot})$ **then**
    Drill-down to l_{pop}
    $action \Leftarrow NoAction$
  else if *NotPivotKey* is NOT indexed **then**
    $action \Leftarrow CreateIndex$
  else
    $action \Leftarrow NoAction$
  end if
else if $\ell_q < pivotlevel$ **then**
  if $(InQ_{tot} > InQ_{ini})$ AND $(\ell_q = \ell_{pop})$ AND $(InQ_{\ell_q} > thr \times InQ_{tot})$ **then**
    $action \Leftarrow PositiveToRollup$
  else if *NotPivotKey* is NOT indexed **then**
    $action \Leftarrow CreateIndex$
  else
    $action \Leftarrow NoAction$
  end if
end if

Algorithm 2 Decision Algorithm in the querying node

l_q: the queried level of the flooded or indexed value
NotPivotKey: the flooded or indexed value
action: the required action by involved nodes {$action = PositiveToRollup$ if at least one node is positive to roll-up}
if $action = PositiveToRollup$ **then**
  Gather statistic information
  Calculate *InQ* for each level
  $\ell_{pop} \Leftarrow FindMostPopularLevel$
  $MaxPivotLevel \Leftarrow FindMaxPivotLevel$
  if $(\ell_q = \ell_{pop})$ AND $(InQ_{\ell_{pop}} > thr \times InQ_{tot})$ **then**
    Roll-up to l_{pop}
  else if $(\ell_{pop} \geq MaxPivotLevel)$ AND $(InQ_{\ell_{pop}} > thr \times InQ_{tot})$ **then**
    Group-Drill-down to ℓ_{pop}
  else if *NotPivotKey* is NOT indexed **then**
    Create Index for *NotPivotKey*
  end if
else if $action = CreateIndex$ **then**
  Create Index for *NotPivotKey*
end if

roll-up to this level, it starts a procedure to confirm the local intuition by using statistic information provided by all the nodes having answered the query. After receiving the tuples containing the number of InQ per level, it calculates the total value of InQ per level.

The calculation of the total rate of InQ per level is not straightforward. Queries concerning an ℓ_i for any $i \geq$ *pivotlevel* end up only in one node and are thus counted once for statistic purposes. The same property is not valid for queries requiring values of higher levels than the pivot level. These queries reach more than one node and are counted in all of them. During the gathering of statistic information for roll-up decisions, the problem of multiple counting of such queries in the calculation of the rate for each level needs to be solved. The complexity in the calculation of the overall rate of InQ increases since more than one pivot levels may exist for the involved trees in the re-indexing procedure. For example, let us assume that the state of trees with the '*biomed*' as their root key is the one shown in Fig. 3(b). In this case, the value of InQ for '*Tier2*' is sent twice by the nodes with the trees of '*Tier2*'. To avoid this situation, a path containing the values for all levels in $[0, pivotlevel]$ is sent along the statistic information to the querying node, so as to make the correct decision. Through this procedure, the querying node is also informed for the MinPivotLevel and MaxPivotLevel of all existing trees containing the queried value (hence *NotPivotKey*).

If the InQ of ℓ_q is more than *thr* of the total number of InQ, then the initiator messages the involved nodes to roll-up the corresponding trees to this level by re-inserting their tuples. If the re-indexing criterion for l_q is not fulfilled and since statistic information has been collected, the querying node examines if a drill-down to a level $\ell_i \geq MaxPivotLevel$ (the equality is for the case that all the involved trees do not have the same pivot level but some of them are already in the MaxPivotLevel) is dictated by the collected statistics. Our intention is to take advantage of the fact that the querying node has now a more global view of the InQ per level. It is possible to find a level $\ell_i \geq MaxPivotLevel$ to be the most popular but this tendency not to appear in the partial views of the involved nodes. In this case, the query initiator informs the involved nodes that a drill-down to this level is needed. We call this procedure `Group-Drill-down`, since more than one nodes participate in the drill-down. All the trees with the queried value in ℓ_q 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. If a re-indexing operation is not needed, no action is taken other than the creation of a soft-state index for this value.

Lock mechanisms are activated during the time that a re-indexing decision is being made. The purpose of this locking is to avoid examining the same re-indexing possibility multiple times for concurrent lookups on specific trees. Locks

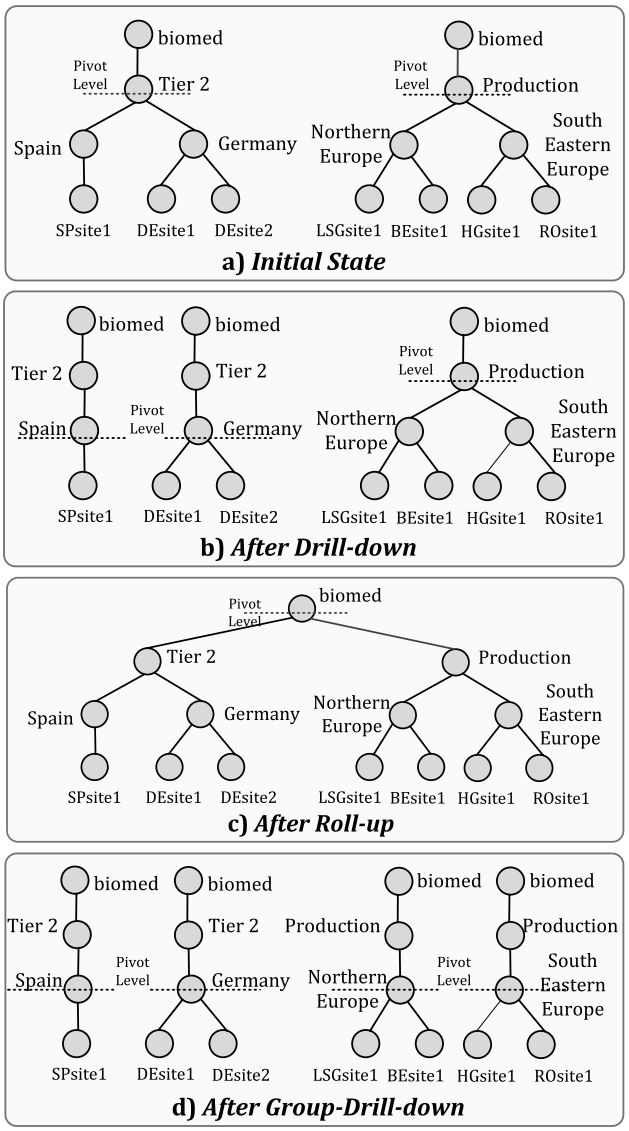

Fig. 3 Examples of drill-down and roll-up operations

are revoked after the completion of an ongoing procedure or after a short period of time. The steps described in Algorithms 1 and 2 are performed, only if the corresponding locks are inactive. Otherwise the described procedures are not performed and only the query is answered.

Examples of the described re-indexing operations are applied in the trees of Fig. 3 with root value '*biomed*'. The two trees of Fig. 3(a) are stored in different nodes of the DHT overlay and are considered the initial state before any re-indexing operation. We suppose that a query for '*Spain*' triggers a drill-down operation. The result is shown in Fig. 3(b). On the other hand, a query for '*biomed*' may result in a roll-up to the root level depicted in Fig. 3(c) or a Group-Drill-down to the `Region` level depending on the total InQ per level. The Group-Drill-down results to the trees of Fig. 3(d)

and differs from the simple drill-down, since all the trees drill-down to ℓ_2.

3.5 Updates

An *update* is the procedure followed during the insertion of a tuple, when its root key already exists in the overlay. The update procedure comprises of two consecutive phases: the insertion of the tuple and the updating of any existing indices for the values of the tuple. The insertion phase presents minor differences compared to the insert procedure.

The updates of the existing datasets is a more complicated procedure. During the insertion of new tuples, it is critical to select the correct pivot level so as to ensure the correctness of the lookup operations. The selection of the pivot level is not simple, since the pivot levels of the stored trees of a specific root key may vary due to performed re-indexing operations.

The following assumptions are made:

- If a new tuple contains a pivot key, then this key should be used during insertion. Otherwise, lookups for this value will return only the tuples, that existed before this operation.
- Even if none of the values belonging to the specific tuple have been used as pivot keys, they may have already been stored in the network. The selection of such a value as pivot key would result in the discovery of the new tuple only in a later search. Therefore, we consider that the pivot level be equal to the MaxPivotLevel in this case. Re-indexing operations would take over to find the most appropriate pivot level of this tuple.

The difficulty of updates increases because the information about the stored pivot values and the MaxPivotLevel is distributed over the overlay. We have implemented a distributed catalogue by creating an index among the node responsible for the root key and the nodes with the pivot keys, namely a record with the root key and the corresponding pivot keys. This enhancement allows online updates with the system continuing to efficiently serve requests of the users.

During a new tuple insertion, a lookup operation for the root key is performed. The responsible node is contacted and it finds out if any value of the tuple corresponds to a pivot key. In this case, the tuple is stored to the responsible node for the pivot key and its new values below the pivot level are added as a new branch to the existing tree. In the opposite case, the hashed value of the MaxPivotLevel is considered as the pivot key of this tuple during its insertion in the overlay. The existence of trees with equal values above the pivot level is not excluded by this assumption and neither is the existence of corresponding indices. These indices should be updated so as indexed lookups to return complete answers. The node storing the tuple initiates lookups for each ℓ_i, where $0 < \ell_i < pivotlevel$ and the corresponding indices are informed about the new tuple.

Examples for the possible cases during an update are depicted in Fig. 2(c). The node holding the index for the root key '*biomed*' concludes that the none value of the tuple '*(biomed, Tier2, Sweden, SWsite1)*' corresponds to an existing pivot key. Moreover, it is aware that the MaxPivotLevel for its trees is the `Region` and thus the tuple is inserted with '*Sweden*' as its pivot key. The new pivot key is also added in the list of pivot keys for this root key. However, the value '*Tier2*' is already indexed. During lookup for the value '*Tier2*' according the update procedure, the responsible node is discovered and a new index among this node and the node with actual data is created. During the update for tuple '*(biomed, Tier2, Spain, SPsite2)*', the node with '*biomed*' index proceeds in the insertion of tuple with '*Spain*' as pivot key, resulting to the creation of a new branch in the existing tree. The indexed value '*Tier2*' is not affected and no further action is taken.

4 Case study: grid information services

A motivating scenario for the usefulness of the proposed system can be found in the collection of information produced by information services in Grid environments. Grid computing resources and services advertise a large amount of data, which are used by various users across multiple administrative domains. This information is not intended only for event handling, as in traditional monitoring systems for networks and cluster computing. The produced information by various mechanisms such as cluster monitors (Ganglia [2], Hawkey [4]), services (GRAM, RLS [6]), queuing systems, etc., is organized and provided to various applications, such as accounting systems, schedulers, portals, etc. For this reason, the key to the design of Grid Information Services is to identify the information that is required and to determine how to best make it available.

In todays Grid Monitoring Architectures (MDS, R-GMA), data concerning the state of the infrastructure are collected by a combination of various monitoring systems on a resource base and organized by *information producers* (or *providers*). In the previous generation of monitoring architectures, the information producers were organized in a hierarchical structure and published their data, which finally were collected and stored in a central LDAP-based database. In more modern approaches, the information producers are known to the system by subscribing themselves to an *Index service* (MDS) or a *Registry* (R-GMA). Information consumers ask this structure for the location of the producers and contact all of them in order to acquire the needed data. Moreover, various aggregator services exist

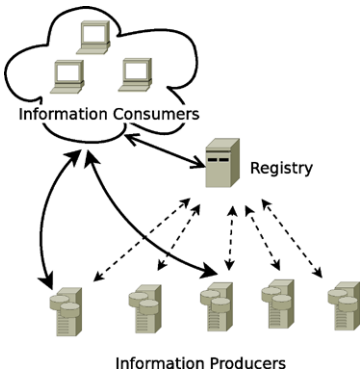

Fig. 4 An information system according to the distributedCP model

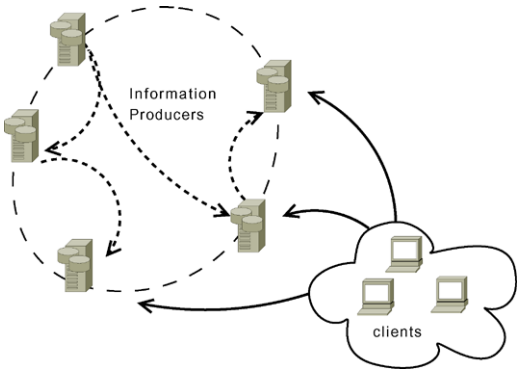

Fig. 5 The proposed architecture for the information system

that collect information (via subscription, polling or execution) from information producers using a common configuration mechanism to specify the type of data and the collected information. For example, VOs maintain such services to collect VO-wide resource information by collecting data from the Information servers running at many sites. Due to the large volume of data and their usefulness to various services, we strongly believe that a solution for efficient storage and indexing of the produced information, which dynamically adapts to requests of users or services, contributes to the operability and the performance of a Grid infrastructure.

The architectures of such information systems can be divided into the following structural categories according to the existing technologies and real paradigms implemented in existing infrastructures.

Centralized solution: The produced information by various monitoring tools is published in a central database. The users query this database in order to retrieve the information of their interest. A usual technique is to replicate the database or maintain such a database per VO, so as to lighten the heavy load in flash crowd situations and to avoid single point of failures.

Distributed solution: A producer–consumer model is implemented in a distributed manner resulting in the implementation of a virtual database. The producers register themselves within the Registry and describe the type and structure of the provided information. The consumers contact the Registry to find the locations of the producers and afterwards contact them directly to obtain the relevant data, as shown in Fig. 4 (*distributedCP* model).

Our system is a complete solution for the organization and storage of such information. The proposed scheme can be integrated in a Grid environment providing a fully decentralized solution. In this architecture, the nodes hosting the information producers and used for information-related purposes are organized in a DHT-based overlay, as shown in Fig. 5. The routing of messages among these resources is performed according to the DHT protocol and no centralized structures are required. The P2P overlay introduces a scalable solution, while data and query load balancing is achieved. A concept hierarchy is defined for the various levels of aggregation that characterize the produced data. Each numerical fact is described by the corresponding concept hierarchy. No off-line collection and processing of data is required, since online updates are supported. The re-indexing mechanism enables the summarization of data according to the incoming queries. Moreover, the Registry or Index service with the locations of service instances is implemented in a distributed manner. While in existing distributed approaches a consumer queries a central point to find all the relative producers, this procedure is eliminated in our self-organized system. The data are dynamically aggregated and stored in a distributed manner according to the preferences of the users achieving reduction of processing and communication cost.

An example for the usage of the described integration can be considered for the service providing information for accounting purposes of the EGEE Accounting Portal [1]. This service uses APEL [11], which is a log processing application to filter data produced in each site. Afterwards, R-GMA producers collect data from sites and streams them to a centralized database. The central database collects millions of records per grid job and stores them offline. Due to the enormous volume of data, only summarized views of the various metrics such as Number of jobs, Normalized CPU usage, SumCPU, CPU efficiency, etc., are computed offline and become available to the users. The right balance between quantity and timeliness of information on one hand and associated costs on the other should be considered for this centralized solution. A simply hierarchy describing this kind of data could be the VO hierarchy used in our examples. In Sect. 6, a more detailed concept hierarchy in the experiment for the specific use case is considered and the corresponding data are distributed among the nodes of a P2P overlay.

5 Discussion

In this section, we will briefly refer to some aspects of our system that relate to its parameters as well as optimization issues.

Memory requirements: The architecture of the proposed system requires the storage of additional information for the implementation of the proposed algorithms. In more detail, each node in the overlay may, relative to the hashing function as well as the query workload, store the following:

Data The inserted data are considered as tuples containing a value for each level of the hierarchy and the corresponding described fact value(s). When a tuple arrives to the node responsible for its pivot key, it is inserted in a tree-structure containing the specific pivot key, so as to avoid the storage of duplicate values for levels above the *pivotlevel* and facilitate the search procedures during lookups. For each stored tree, at least one tuple with L numerical values for statistic purposes is maintained, one per level of the hierarchy.

Root Indices Each time that a different root key is inserted in the overlay, indices are created towards its pivot keys. Moreover, when a new key is inserted for an existing root key, the corresponding index is added.

Soft-state indices The flooded queries result in the creation of soft-state indices, which are utilized for faster resolution of future queries. The maximum number of allowed indices is defined for each node through the I_{max} parameter. The required space for soft-state indices is $O(I_{max})$.

Parameter selection: The introduction of various parameters in our system influences its performance relative to their values. The threshold value (*thr*) plays an important role to the number of the performed reindexing operations. On one hand, a large threshold allows less reindexing operations and therefore increases the utilization of the soft-state indices. This case favors query workloads where the upper levels of the hierarchy are at most queried while the number of different values is limited. Thus, the probability of duplicate queries being issued is larger. On the other hand, a smaller *thr* value results in more frequent reindexing operations depicting the changes in the query trends. In general, the drawback of more roll-up and drill-down operations is the bandwidth consumption for the movement of the involved tuples and the invalidation of the existing soft-state indices. However, these operations are highly effective, when the values of the lower levels of the hierarchy are more popular. In this case, the contribution of indices to the performance of the system is not adequate. Therefore, lower values of the *thr* parameter are required for query workloads directed mainly towards lower levels of the hierarchy and targeting numerous different values.

Soft-state indices help the system improve its performance. Nevertheless, maintenance of index consistency may be problematic in a dynamic system that adapts according to user preferences. For this reason, we introduce the TTL parameter: Indices which have not been used for a period larger than TTL, are not considered as useful and removed. 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. The amount of indices can be also calibrated according to the system capabilities. While memory becomes a cheaper commodity by the day, the plain size of data discourages an "infinite" memory allocation for indices. Therefore, after the number of indices has reached a limit I_{max}, the creation of a new index results in the deletion of the oldest one. If an indexed value occupies more than one index, then all of them are erased for consistency reasons. Calibrating I_{max} for performance without increasing it uncontrollably entails knowledge of our data (e.g., how skewed each hierarchy is). Overall, the system tends to preserve the most useful indices, namely the ones directed towards the most frequently queried data items.

The window parameter W represents the number of previous statistics maintained by each node for the calculation of average rates of incoming queries. A large value of W will fail to perceive load variations, whereas a very small will have a negative impact, since the average rate of incoming queries will not be adequate to lead to re-indexing decisions or may lead to erroneous ones. In order to estimate its value, we set $W = O(1/\lambda)$, i.e., we connect the size of the window with the query interval. The more frequent the requests, the smaller W can be and vice-versa.

Consistency: Another aspect to be considered is the various strategies to follow, so as to ensure data consistency during the various operations. In a highly dynamic environment, as the one described, where updates and re-indexing operations occur online, special care should be given to ensure the consistency of data during these operations, against some additional communication and maintenance cost. A remarkable point regarding consistency issues is the validity of root indices for the correct execution of updates. The root keys should be constantly aware of all the existing pivot keys and confirm their existence after a period. Therefore, it is advisable that root keys issue periodical messages to verify the validity of the information about their pivot keys.

Moreover, it is also important to ensure the participation of all relative trees in the roll-up and drill-down operations and the effectiveness of the locking mechanisms during reindexing, so as consequent lookups on the re-indexed data to return complete results. During these operations, it can be observed that lookups fail to return complete results due to the movement of data. In order to face this miss, the data are cached at the initial nodes as well, until the re-indexing procedure is completed. Once the tuples are re-inserted and the root key completes the updates of its indices towards the

new pivot keys, the old trees are removed from their predecessor nodes after the expiration of a TTL. Lookups for values being re-indexed are also delayed to a node that receives the re-indexed data until the completion of the transaction. The root key aware of the ongoing operations on its data, is also responsible to forward a non-exact match query either to the nodes initially responsible for the data or the newly-responsible ones. The creation of soft-state indices for any value included in the involved trees is not allowed during the re-indexing period.

Finally, special care is taken during the creation of the soft-state indices to avoid stale indices. Apart from the TTL restriction, indices encountering any problem during their creation are removed.

6 Experimental results

6.1 Simulation setup

We now present a comprehensive simulation-based evaluation of our scheme. Our performance results are based on a heavily modified version of the FreePastry simulator [12], although any DHT implementation could be used as a substrate. By default, we assume a network size of 256 nodes, all of which are randomly chosen to insert tuples and to initiate queries.

In the most part of our simulations, we use synthetically generated data. Our data is a tree with each value having at most one parent. Each distinct value of ℓ_i has a constant number of children in ℓ_{i+1}. By default, our data comprise of 100k tuples, organized in a 4-level hierarchy (see Fig. 1(a)) with one numerical fact (e.g., `CPU_time`). The number of distinct values per level are $|\ell_0 = 100|$, $|\ell_1 = 1000|$, $|\ell_2 = 10000|$ and $|\ell_3 = 100000|$. The level of insertion is, by default, ℓ_1, unless stated otherwise.

For our query workloads, we consider a two-stage approach: we first identify which level our query will target according to the *levelDist* distribution; the requested value is then chosen from that level following the *valueDist* distribution. In our experiments, we use the Zipfian ($p_i \sim 1/i^{\theta}$) distribution for *levelDist*, while we express a bias inside each level using the uniform, 80/20, 90/10 and 99/1 distributions for *valueDist*.

Generated queries are issued at an average rate of $1\frac{query}{time\_unit}$, in almost 50k time units total simulation time. We present results for queries on a single dimension with multiple levels of hierarchy. Our default *thr* value is set to 0.3, which is a large enough value to avoid very frequent re-indexing attempts. Simulations with different values of *thr* around this default show small qualitative difference. The default value of *W*, which controls how quickly the system can adapt to changes, is set to 1000 time units. Finally, we assume a practically infinite value of *TTL* (indices never expire).

In this section, we mainly intend to demonstrate the performance and adaptability of our system under various conditions. Our goal is to show that we prove highly efficient under a variety of data and load distributions and can quickly adapt to sudden changes in skew without any modification to the default parameters. Specifically, we measure the percentage of queries which are answered without flooding (*precision*).

6.2 Performance under different levels of skew

In the first set of experiments we identify the behavior of our system under a variety of query loads. Specifically, we vary the number of queries directed to each level by increasing the θ parameter in the *levelDist* distribution. For each value of θ, we also choose values inside each level using four different distributions.

In Fig. 6, queries are skewed towards ℓ_0. As θ increases for *levelDist*, the performance of our method improves: Re-indexing is performed sooner as more queries take place and the exact matches due to the chosen pivot level increase. By increasing the skew for *valueDist*, we observe remarkably high precision rates (close to 100%), because both the ratio of popular queries and the density of queries for certain tuples increase. Another point that plays a big role is the limited number of distinct values of ℓ_0. Obviously this is quite small compared to the last level, thus enabling soft-indexing and faster re-indexing. For a set θ value, the method performs justifiably better as the distribution becomes more skewed: More queries exist for fewer distinct values. Finally, we notice that the larger the θ value, the smaller the difference in precision among the different *valueDist* distributions.

Figure 7 shows results where our workload favors ℓ_3. Again, we notice a similar trend in performance as *valueDist* becomes more biased and our method shows high precision values, albeit reduced compared to the previous case. We

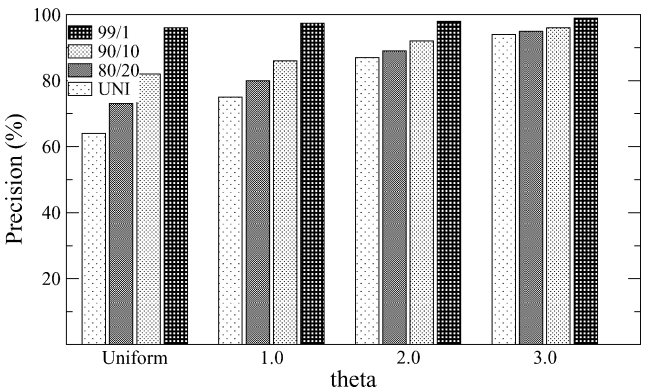

Fig. 6 Precision when skew directed towards ℓ_0

notice that the precision for the same *valueDist* distribution decreases as θ increases: This is due to the fact that ℓ_3 has a considerably larger number of values. By increasing the number of queries for those values, we increase (relative to the choice of *valueDist* of course) the probability of queries to non-indexed values. Nevertheless, the decrease is smaller as *valueDist* becomes more skewed.

6.3 Testing against multiple bias points

In the next experiment, we test our method against a more challenging type of workload (*MULTI*): While different levels receive an equal number of queries, nevertheless we target a *different* part of a data tree from each level. Specifically, we divide all levels in quarters and target (using different values of *valueDist*) one quarter per level so that no quarter is related with any other in the data tree. This is a very challenging workload, as it forces the method to store different data at different levels of granularity. Table 1 summarizes the results, where besides the precision we document the cost in number of re-indexed tuples as well as the number of total roll-up and drill-down operations.

Our technique proves extremely efficient in all four workloads, achieving very high precision (between 92% and 100%) at low cost: The largest number of operations occur when we uniformly query the different values in which case 75% of the tuples are re-hashed and re-inserted from re-indexing operations. As the *valueDist* becomes more biased, the number of re-indexing calls decreases. This clearly demonstrates that our method adjusts its operation according to the need: The number of trees being re-indexed is proportional to the number of unique trees that are popular. This is a highly desirable property since for most applications we anticipate both dynamic and highly skewed loads.

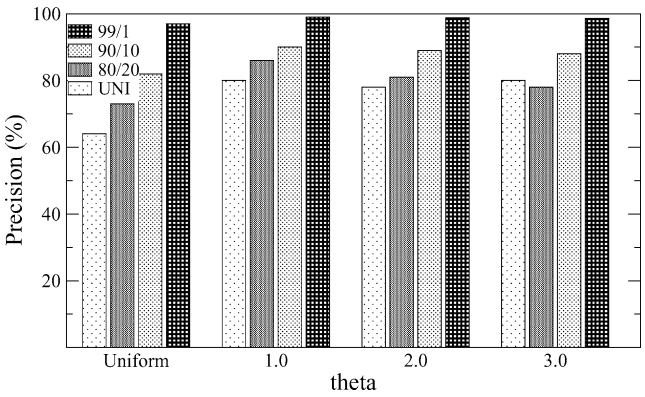

Fig. 7 Precision when skew directed towards ℓ_3

6.4 Performance in dynamic environments

The adaptiveness and performance of the proposed system in a dynamic environment is examined by this set of experiments. The query distribution encloses a sudden change in skewness from level ℓ_0 towards ℓ_3 and vice versa in the middle of the simulated queries.

Figure 8 demonstrates the behavior of the system when the query load shifts from ℓ_0 towards ℓ_3. The results show that, in all cases, our system increases its precision due to the combination of re-indexing operations and soft-state indices and the majority of questions are answered by exact match lookups. The precision reaches over 90% for $\theta = 2.0$ and over 80% for $\theta = 1.0$ before the change in skew. In the transitional stage, the flooding of the queries increases but the system rapidly manages to recover and regain its performance characteristics (after at most 5% of the queries). The steep decrease in precision happens at the exact time of the shift in the workload: A much larger number of distinct values belong to ℓ_3, thus the existence of useful indices is less probable. The contribution of soft-state indices is not sufficient to handle the query load until drill-down operations take place. In this stage, the larger the value of θ, the larger the decrease in precision and the faster the recovery: As we show in Fig. 9, where the query loads for *valueDist* 90/10 are considered, both exact match and indexed lookups are fewer for $\theta = 2.0$. This happens because queries are more skewed towards ℓ_3 and benefit even less from the already rolled-up trees. However, as θ increases, drill-down decisions are taken faster, favoring the increase of the exact match queries that answer the majority of the requests.

The precision of the algorithm is tested against a sudden shift from ℓ_3 to ℓ_0 for various workloads and displayed in Fig. 10. During the steady stages of the simulation, similar trends to the ones of one directional skew are observed and the system presents high performance over 80% for all workloads. As the valueDist becomes more biased, higher

Table 1 Performance comparison for the MULTI workload over different values of valueDist

valueDist	precision (%)	#roll-ups	#rolled-up trees	#drill-downs	re-inserts (%)
uniform	92.0	25	250	500	75
80/20	94.3	25	250	171	42
90/10	95.2	25	250	51	30
99/1	99.5	1	10	6	1.6

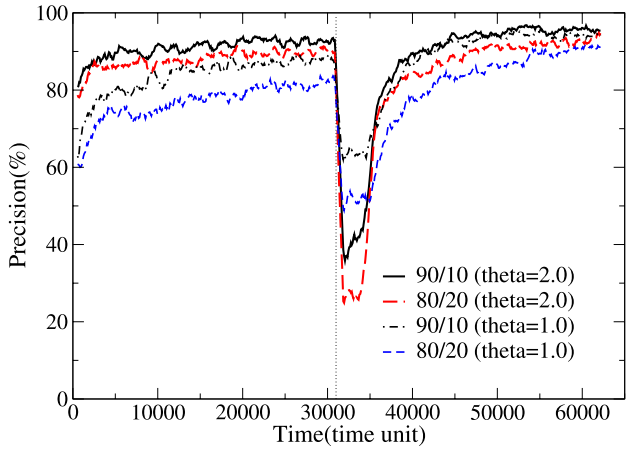

Fig. 8 Precision over time for various workloads, when skewness changes from ℓ_0 to ℓ_3

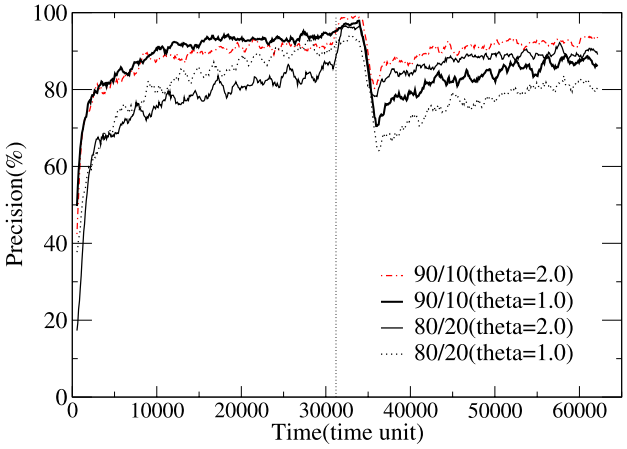

Fig. 10 Precision over time for various workloads when the skewness of workload changes from ℓ_3 to ℓ_0

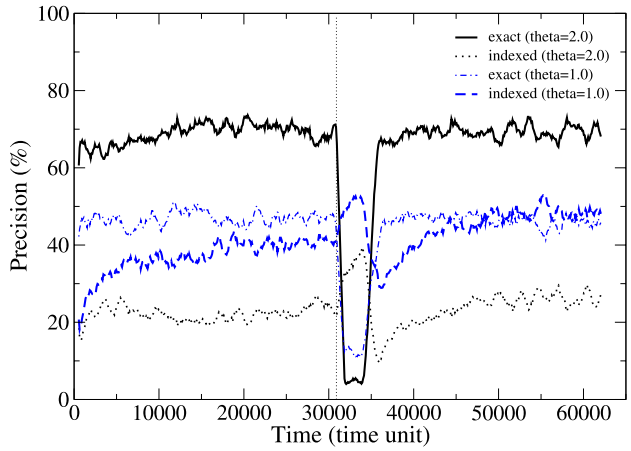

Fig. 9 Precision over time of non-flooded queries for valueDist 90/10, when skew is directed from ℓ_0 to ℓ_3

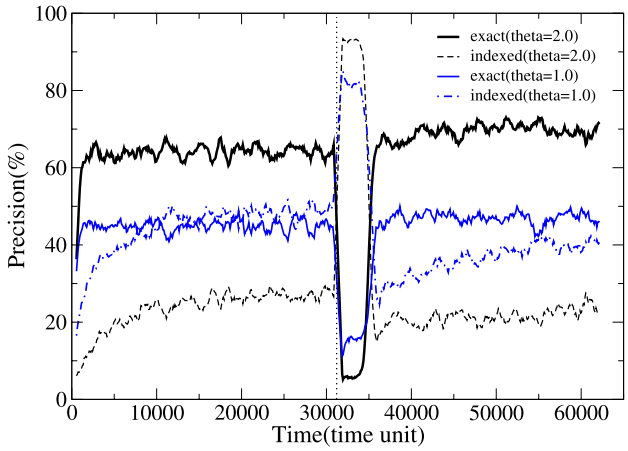

Fig. 11 Precision over time of non-flooded queries for valueDist 90/10, when skew is directed from ℓ_3 to ℓ_0

precision is accomplished, since the number of popular values shrinks and drill-down operations are performed faster increasing the adaptiveness of the system. After the change in the direction of skew, less queries are flooded for the $\theta = 2.0$ workloads (behavior that contrasts to the previous experiment). Figure 11 demonstrates a more comprehensive view of the system after the change in skew. Indices take over to serve lookups immediately. Due to the smaller number of distinct values in higher levels, indices perform well. However, the consecutive roll-ups destroy the existing indices and the performance of the system is influenced negatively. The system regains its performance by the rapid increase in the exact lookups.

The comparison of results among the two shifts of the workload reveals that the soft-state indices are capable to preserve the high precision of the system in case of a skew towards higher levels due to the limited number of different values. On the contrary, the adaptiveness of the system significantly depends on the re-indexing operations, when lower levels of the hierarchy are the most popular. Nevertheless, in both cases, the system needs bounded time to reorganize its indexing mechanism and achieve high performance.

6.5 Storage load for different number of nodes

The inserted tuples in the system are stored in tree structures with the same pivot key to avoid the storage of redundant information for values above the pivot level. However, these values may exist in more than one trees stored in different nodes of the overlay depending on the selected pivot level. The total number of tree nodes receives its minimum value, when the root level is chosen as *pivotlevel* and the maximum number, when the lowest level of the hierarchy is the *pivotlevel* of all trees (≈111k and 400k nodes respectively for our dataset). Each node of the tree (hence *tree node*) represents the value of a tuple for this level. The total number of

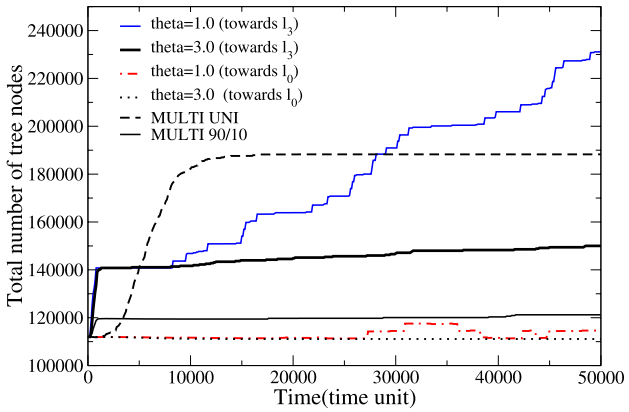

Fig. 12 Total number of tree nodes over time for workloads skewed towards l_0 and l_3 and *valueDist* 90/10 and *MULTI* workloads with uniform and 90/10 *valueDist*

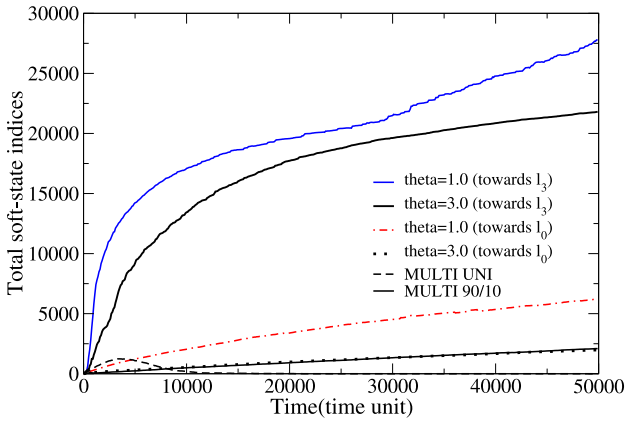

Fig. 13 Total number of soft-state indices over time for workloads skewed towards l_0 and l_3 and *valueDist* 90/10 and *MULTI* workloads with uniform and 90/10 *valueDist*

tree nodes during the simulation time is presented in Fig. 12. The increase in the number of tree nodes indicates that drill-down operations take place, while a decrease depicts the effects of roll-up operations. Thus, the total number of tree nodes over time demonstrates the evolution of executed re-indexing operations. Queries directed towards l_3 for $\theta = 1.0$ lead the system to reinsert its tuples slowly to lower levels of the hierarchy, which tend to be the most popular. Since the difference in the popularity amongst the levels is not excessive, the drill-down operations happen during the whole time of the simulation. In case of the respective workload for $\theta = 3.0$, the increase of tree nodes is steeper and it is completed during a short period at the initial stages of the simulation. The most popular trees re-insert their tuples to lower levels quickly and no major variations are noticed during the rest of the simulation time. The workloads directed towards l_0 cause no significant variations in the total number of tree nodes. In both cases of *levelDist* for these workloads, the number of tree nodes is preserved close to its initial value, since the total tree nodes with l_1 as pivot level does not differ notably compared to the tree nodes with l_0 as pivot level. Finally, we comment on the *MULTI* workloads, where roll-up and drill-down operations coexist. When the queries target the values uniformly inside the levels, the re-insertion of tuples continues for a longer period and for a larger number of tuples than in the respective workload with *valueDist* 90/10. The popularity of levels changes for different group of trees in the *MULTI* workloads and thus the insertion of tuples to the most appropriate level occurs in a more concurrent way than in the workloads skewed towards l_3 for $\theta = 1.0$.

The total number of soft-state indices in the system appears in Fig. 13. The creation of soft-state indices is influenced by the evolution of re-indexing operations. According to the described strategy, a soft-state index follows the flooding of a query as a supplemental solution to the adaptiveness of the pivot level. Therefore, the steep inclination

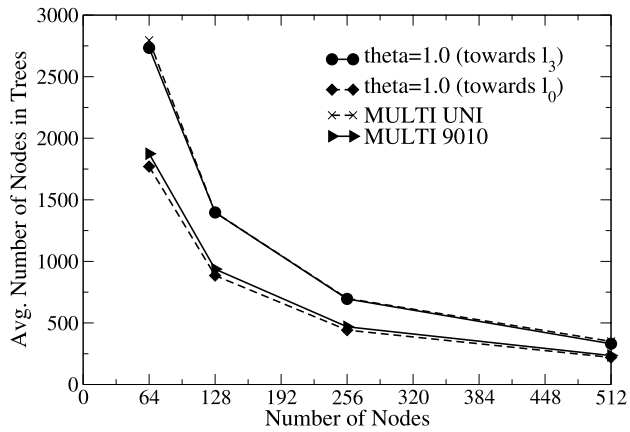

Fig. 14 Average number of tree nodes per node

in the curves of the workloads directed towards l_3 is justified considering the fact that the number of different values in the lower levels is larger than in the upper levels. In the workloads with higher levels being more popular, the increase of indices occurs at a lower rate. The efficiency of the re-indexing operations becomes more evident in the case of the *MULTI* workload following a uniform distribution inside the level. New indices are temporarily created, which are removed as soon as roll-up and drill-down operations take over and the system adapts to the trends of the incoming queries.

The different objects stored in the nodes of the overlay can be divided in the following categories: the tree structures storing the values of the selected hierarchy, the root indices which are created during the insertion of a new root value and are aware of their pivot keys, the soft-state indices and the statistical information maintained for the re-indexing decisions. The experimental results of Figs. 14–16 demonstrate the average number of these objects per node or the items that determine their volume. These values are the average results during the whole simulation time, derived

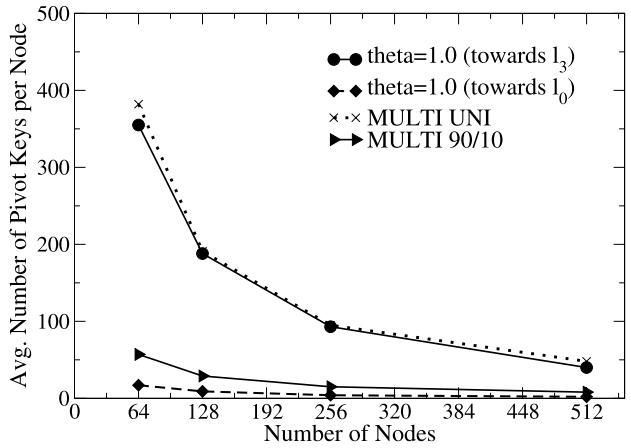

Fig. 15 Average number of pivot keys per node

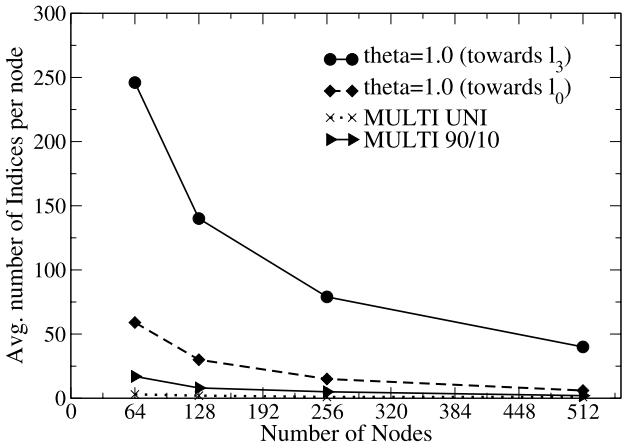

Fig. 16 Average number of soft-state indices per node

by the measurements in regular periods. Thus, they incorporate the various changes caused by the roll-up and drill-down operations. The *MULTI* workload with uniform distribution and the skewed workload towards l_3 with $\theta = 1.0$ have the maximum number of tree nodes, which conforms to the results of Fig. 12, where the curves of these workloads reach the highest values. It should be noticed that the presented values for the *MULTI* workload is the maximum observed values for this type of workload, since the uniform *valueDist* is studied. Therefore, all the trees of the dataset re-insert their tuples to the appropriate level. If the distribution of queries alters to 90/10, then the average number of tree nodes decreases significantly. The average number of tree nodes for the workload skewed towards l_0 remains low. The average number of pivot keys (see Fig. 15) behaves in analogous manner to the tree nodes for all the workloads. The number of pivot keys is correlated to the stored statistic information and the number of root indices. For each pivot key, a tuple with L levels is maintained. Moreover, if the average rate of incoming queries is calculated with the method of a sliding window with n slots, the number of maintained tuples should be multiplied with this factor. The number of existing keys also equals to the number of root indices, since a separate index is maintained for each pivot key.

The evaluation of the average number of indices is shown in Fig. 16 and verifies the fact that the more effective the re-indexing operations are, less indices are created. The *MULTI* workload requires the less indices, since the trees adapt their pivot levels appropriately. The average number of soft-state indices is larger for workloads skewed towards l_3, and this is a result of various factors. Many trees perform drill-down operations to lower levels of the hierarchy and the tuples are more dispersed among the nodes of the overlay, and thus an indexed value occupy more indices. Moreover, the number of distinct values is larger and increases the probability of a non-indexed value to be queried followed by the creation of a new index. The opposite conclusions are valid for the workloads directed towards l_0.

The precision of the system is stable independently of the number of nodes participating in the overlay, with an average 1% deviation. Moreover, the number of nodes does not influence the total number of pivot keys and tree nodes, since these values depend mainly on the distribution of the query workload. The number of soft-state indices is affected by the number of nodes in the overlay: less nodes increase the probability of more than one tree containing the same indexed value to be located in the same node and thus only one index to be created. The distribution of these objects among the nodes is better as the number of nodes increases. According to the exposed results, when more nodes participate in the overlay, each node is responsible for a smaller average number of items.

6.6 Effect of the I_{max} parameter

The I_{max} parameter is configured on a per node basis and defines the maximum number of soft-state indices that a node may store. The creation of a new index results in the removal of '*oldest*' ones, namely the ones not utilized for the longest time, if I_{max} is exceeded. The idea behind this strategy: if the indices to be removed were useful, then they would have been renewed. Otherwise, their removal would not have a negative impact in the performance of the system.

The I_{max} value has a varying impact on the performance of the system depending on the type of the workloads. The utilization of the indices increases when the query workload does not indicate to the system an evident direction for a re-indexing operation, namely in case that more than one levels are almost equally popular. The more uniform the distribution of queries, the more indices are created.

Figure 17 shows the achieved precision for various θ, when the skew is directed towards l_0 and l_3 respectively and the queries target uniformly the values inside a level. The

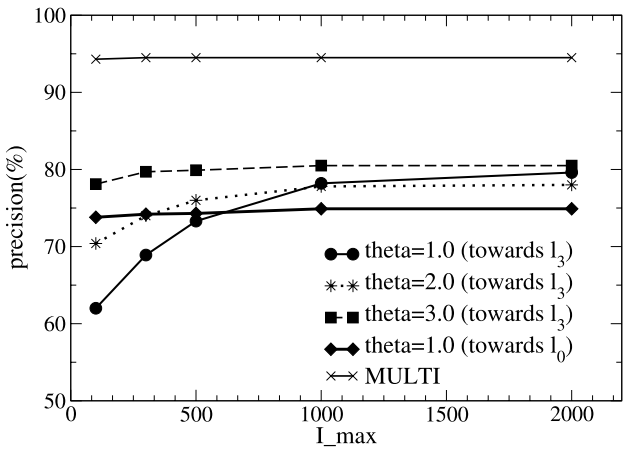

Fig. 17 Precision for various UNI workloads

workload directed towards the root level does not seem to be affected by the I_{max} variation, even though the *valueDist* is uniform, which is the most demanding case for indices. This is explained by the permanent existence of the root indices, which are not taken into account during the calculation of the I_{max} value. Moreover, the possible values to be queried become fewer in the higher levels of the hierarchy and therefore less storage space is required for the creation of soft-state indices. In the opposite case of query workloads directed towards l_3, the increase of the I_{max} value may influence the precision by almost 10%, especially in the case of $\theta = 1.0$, where values from almost all levels are queried and the re-indexing operations are not adequate to cover the demand of values in all levels. The *MULTI* workload is a test case for the system, where the proposed re-indexing algorithms favor its performance at most. The re-indexing decisions depict clearly the query trends and the precision is based on the correction of the pivot level. Therefore, the variation in the value of the I_{max} does not affect the achieved precision.

6.7 Performance for hierarchies with different number of levels

The number of levels may influence the precision of the system as well. The reason is twofold. More levels lead to more possible candidates for a re-indexing decision and increase the probability that queries are resolved with the use of soft-state indices or flooding. In this experiment, 4-level, 6-level and 8-level hierarchies were used. The number of inserted tuples remained 100k and the default insertion level is ℓ_1. In order to maintain the number of total tuples stable, the widths of the trees in each level were narrowed down. The distribution of the query workload, which includes around 50k queries, is generated using $\theta = 1.0$ for *levelDist* and a biased 90/10 distribution for *valueDist*. The presented results refer to query workloads skewed towards the highest and the lowest levels of the hierarchy.

Table 2 Precision for different number of hierarchy levels

Levels	Towards l_0 (%)	Towards l_3 (%)
4	86	91
6	80	77
8	78	77

Table 2 depicts the achieved precision for each one of the described workloads. The increase in the number of levels results in a decrease in the precision for both type of workloads, as expected. A significant reduction in the number of roll-up operations can be noticed for the experimental results in the query workloads directed towards l_0, as well as a decrease in the drill-down decisions for the query workloads directed towards l_3. Moreover, in both cases a trend of increased re-indexing operations in the opposite direction of the skewness of the workloads appeared. This behavior is justified by the fact that more levels exist and the re-indexing decisions may be more ambiguous requiring corrective re-indexing operations, so as the system to adapt in the incoming queries. It has been also observed that the number of queries answered with soft-state indices increase, especially for workloads directed towards l_0. Moreover, the difference in the achieved precision is more remarkable for the opposite direction of skew. The number of limited distinct values in the upper levels favor the re-usability of indices. As shown in the results of Table 2, the decrease in the precision is less, when the skewness is towards l_0.

6.8 Simulations for real datasets

The proposed system has been tested for a use case scenario that can be applied to a Grid Information system and tested against existing solutions. In the described experiment, we have considered the organization of data and the provided functionality of the EGEE Accounting Portal developed by CESGA [1] as a reference point, using real queries and data in our simulations. The portal gathers the accounting data of all sites participating in the EGEE and WLCG infrastructures as well as from other sites belonging to other Grid Organizations collaborating with the EGEE infrastructure. The data are further analyzed to generate statistical summaries that are available to the users. The numerical facts of a user's interest are various metrics such as Normalized CPU time, Number of Jobs, etc., being collected by tools for monitoring data of grid resources. In the case of the specific portal, the collected data are further processed and inserted in an offline database. The large volume of the collected values restricts their availability to the users. Aggregated views serving accounting purposes are generated and stored in a central

database for presentation. In the described approach of this paper, a similar approach has been adopted. The monitoring of resources and the collection of values is not investigated, supposed that the "traditional" grid tools (e.g. APEL) are being utilized. Therefore, we assume that the numerical facts of various metrics are published and processed by a local service on a per site basis. Afterwards, the published information can be stored in the corresponding Information Servers that may exist in each site. The goal of the specific experiment is to show that the collected information regarding the grid resources can be organized dynamically so as to adapt to the query workload favoring the distribution of load among the nodes. The query load per Information Server is compared to the centralized and distributedCP solution.

The description of the numerical values of the metrics follows a hierarchy of 7 levels based on the presentation of data in the aforementioned application. The hierarchy levels are:

```
VODiscipline < VO < Category <
SubCategory < Region < SubRegion
< SiteName
```

The specific hierarchy has been used for all categories and projects, so as to maintain a common and general description. In order to assure that a value does not belong to more than one trees with different values in the root level, the data have been parsed and uniquely mapped to integer values taking into account the values between the VO level and each level until the examined level. This assumption can be justified, for example the Number of Jobs differs when the EGEE sites located in Greece are queried for two different VOs. Moreover, its a common practise in grid infrastructures to include the VO attribute in the queries for the Information Service. According to the information provided by this accounting portal, around 700 sites participate in various categories and projects of the infrastructure. Each site is considered as a different node in the simulation of our system. The size of the inserted dataset is 5789, and only the supported VOs by each site have been considered. For example, the possible values for a site belonging to the *EGEE* project may be:

[High-Energy Physics, atlas, EGEE, Production, South-Eastern, Greece, HG-01-GRNET].

Thus, the sites belonging to other categories may have not been registered with a value for each level of the hierarchy. In this case, the corresponding level has been filled with a value, which is never queried. For example, such a tuple for a site that belongs to *Tier2* has this form:

[Infrastructure, dteam, Tier2,Tier2Sub, UK, UK-London-Tier2, UKI-LT2-HEP].

The query workload has been created by real queries posed in the EGEE accounting portal during a two month period. Since this application further processes the collected data and aggregates them, it offers the potential that a query may refer to a group of VOs. In the presented results, each query is translated to the equivalent queries and a query for each VO of the group is generated. Each group of queries is submitted to the system at once and a standard rate for the execution of each group is assumed. The same convention has been followed in the simulation for the centralized model and *distributedCP* model. To the extend of our knowledge, it is a common practice to determine the VO when querying the various tools of the Information services. The performance of the proposed architecture is compared to the centralized model, where a central database gathers all the information and answers the queries and the *distributedCP* model, where a central Registry forwards the queries to all the nodes with relative data. The query workload comprises of around 20k group of queries, which have been translated into 250k of queries using the restriction for a unique VO value for each query. The majority of queries targets l_2, l_3 and l_4 around 85% of the time (19%,36% and 32% respectively).

Experiments have been carried out using all the possible levels as the default insertion level. The precision of our algorithms remains high, with only 3% flooded queries at most. The structure of the dataset and the biased query workload favor the resolution of the queries without flooding. The difference is in the percentage of the exact match queries and the indexed queries. The precision of exact match queries reaches around 65%, when the levels l_2, l_3 and l_4 are selected as the default *pivotlevel*. The selection of these levels as pivot levels appears as good solution for the distribution of the objects amongst the nodes as well.

Figure 18 depicts the variations of the average number of resolved queries per node during the simulation time. At the start of a time unit, a group of queries is posed to the system and thus the average load of the centralized solution is equal to the number of queries included in a group. The curve concerning the *distributedCP* model refers only to the

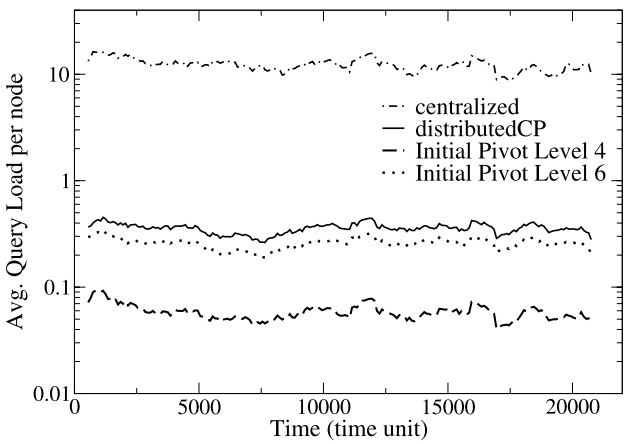

Fig. 18 Average load of queries per node

average query load of a producer. The load of the Registry node is equal to the node of the centralized solution or to a fraction of this load if a Registry node exists per VO. The indexing structure of our system results in a high precision, while each query is resolved by visiting less nodes and thus the query load per node is significantly smaller than in the *distributedCP* model. The average load per node of our system reaches the respective load of a node in *distributedCP* model, if l_6 is selected as the default *pivotlevel*. In this case, the indices forward the queries to an almost equal number of nodes as in the *distributedCP* model.

6.9 Performance for dataset of the APB benchmark

The adaptiveness of the system is also tested using another category of realistic data. For this reason, we generated query sets by the APB-1 benchmark [9]. APB-1 creates a database structure with multiple query sets by 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. For our experiments, we focus on the `product` dimension, a steep hierarchy of 6 levels (the bottom level contains 90% of the members). In more detail, the number of distinct values per level are $|\ell_0 = 50|$, $|\ell_1 = 150|$, $|\ell_2 = 800|$, $|\ell_3 = 3050|$, $|\ell_4 = 6950|$ and $|\ell_5 = 93050|$. Another characteristic of the specific dataset is that the number of children per node is not constant, as in the synthetically generated datasets of the previous experiments. The query load is skewed towards the lower levels of the hierarchy and 75% of queries refer to values of the ℓ_4 and ℓ_5.

The results are depicted in Fig. 19, where the precision over time for various initial levels of insertion is shown. It is remarkable that the system adapts to the query load and presents similar performance despite the selection of the level used as pivot level during initial insertions, thus the re-indexing operations -mainly drill-down operations towards

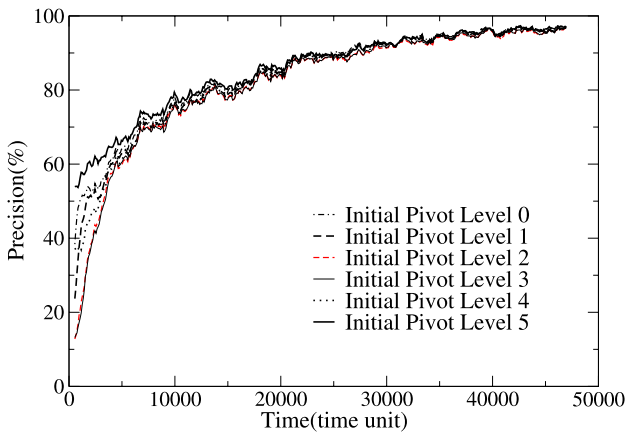

Fig. 19 Precision over time for the APB workload for different initial pivot levels

ℓ_4 and ℓ_5 and soft-state indices serve to the incremental precision, which reaches values near 100%.

6.10 Updates

In order to measure the cost of incrementally updating our dataset, we randomly select the 90% of the tuples, executed each of the described query workloads in 6.2 and finally update the dataset by inserting the remaining 10% of the tuples. We note here that the workload plays an important role in the update process as it affects the indexing levels of the stored tuples and, therefore, the update cost (as tuples may have common attributes with existing ones). The cost in messages for storing each of the initial tuples is one lookup message so as to locate the root key and one insertion message to store the tuple. Further lookup messages are not needed, since no other indices than the ones among the root keys and corresponding pivot levels have been created yet. The selected pivot level for the initial tuples is, by default, ℓ_1. The conducted experiments regard the update cost in terms of additional lookups operations to inform existing indices about the appearance of the new tuples. In these set of experiments, we modified the inserted dataset. Table 3 contains the average number of lookups per insertion for updating the soft-state indices. This cost can be considered as negligible when the skew is towards high levels of the hierarchy. The maximum documented cost for skewed workloads towards ℓ_3 and uniform valueDist is close to 2. The less skewed the distribution, the bigger the possibility of soft-index existence in levels other than the popular one is. As skew increases, this cost also diminishes.

6.11 Other experimental results

In this section, we briefly describe other conducted experiments. We experimented by varying the number of concurrent queries per time unit for the presented workloads. In this experiment, we divided the queries of the initial workloads of Sect. 6.2 into group of queries. Each group was posed to the system in the start of the time unit. The experimental results for workloads skewed towards l_0 and l_3 are shown in Fig. 20. The measured precision presents negligible variation, thus showing that the performance of the

Table 3 Number of average lookups for updating indices per insertion for different values of *valueDist*

valueDist	Skew towards ℓ_0		Skew towards ℓ_3	
	$\theta = 1.0$	$\theta = 2.0$	$\theta = 1.0$	$\theta = 2.0$
uniform	≈ 0	≈ 0	1.99	1.98
80/20	0.01	≈ 0	1.8	1.16
90/10	0.14	≈ 0	0.39	0.25
99/1	≈ 0	≈ 0	0.01	0.02

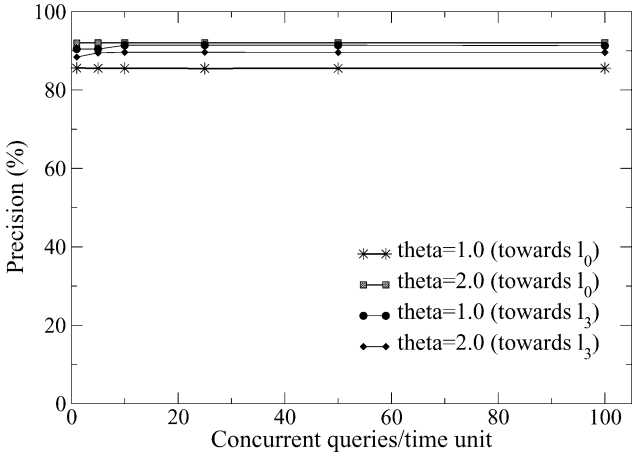

Fig. 20 Precision for concurrent queries for valueDist 90/10

system remains consistently high while the system scales to a considerable number of concurrent users.

Moreover, experiments conducted with up to 1k nodes showed little qualitative difference. Simulation results for different values of threshold showed that the fluctuation of the precision for $0{,}2 \leq thr \leq 0{,}4$ is at most 2%. For values $thr \geq 0{,}5$ and uniform workloads, the fluctuation reaches 10%. For $thr \leq 0{,}3$, the initial number of queries to allow re-indexing should increase in order to avoid redundant operations. Experiments for various data distributions with different number of distinct values per level showed no qualitative differences. Another important observation is that by varying the default pivot level the steady-state performance of our algorithm is not affected, since the re-indexing operations and soft-state indices adapt the pivot levels appropriately.

7 Conclusions

In this work we described a highly adaptive, scalable, online technique in order to store hierarchical data and do efficient query processing on them. Our scheme distributes large amounts of data over a DHT overlay in a way so that the hierarchy semantics are maintained while eliminating single points of failure and minimizing data unavailability. The distinctive characteristic of our method is the adaptive indexing over the data: The stored hierarchies are indexed over variable levels according to the granularity of the incoming queries. Using limited only knowledge, peers decide on the indexing level of their stored tuples so that flooding is minimized. Moreover, there is no need for off-line updates as our system consistently updates the dataset online at low cost.

We discussed one interesting application of our method over a Grid Information System: Distributing the sources of useful data over a grid system presents significant advantages over the existing approaches. Moreover, our unique re-indexing mechanism enables automatic aggregation of older data and more detailed views of recent ones.

Our experimental evaluation over multiple dynamic and challenging workloads confirmed our premise: Our system manages to efficiently answer the large majority of queries using very few messages. It is especially effective in skewed workloads, adapts to sudden shifts in skew and updates datasets in a fast, reliable and cost-efficient manner.

Acknowledgements We would like to thank the administration team of the CESGA EGEE Accounting Portal [1], who kindly provided us real data regarding the usage of their portal.

References

1. Egee accounting portal. http://www3.egee.cesga.es/gridsite/accounting/CESGA/
2. Ganglia Monitoring System. http://ganglia.info/
3. GT Information Services: Monitoring and Discovery System (MDS). http://www.globus.org/toolkit/mds/
4. Hawkeye: A Monitoring and Management Tool for Distributed Systems. http://www.cs.wisc.edu/condor/hawkeye/
5. R-GMA: Relational Grid Monitoring Architecture. http://www.r-gma.org/
6. The Globus Toolkit. http://www.globus.org/
7. Aberer, K., Cudre-Mauroux, P., Hauswirth, M.: The chatty web: emergent semantics through gossiping. In: WWW Conference (2003)
8. Aberer, K., Cudre-Mauroux, P., Hauswirth, M., Pelt, T.V.: Gridvine: building internet-scale semantic overlay networks. In: International Semantic Web Conference (2004)
9. OLAP Council, APB- 1 OLAP Benchmark. http://www.olapcouncil.org/research/resrchly.htm
10. Ester, M., Kohlhammer, J., Kriegel, P.: The dc-tree: a fully dynamic index structure for data warehouses. In: ICDE (2000)
11. Byrom, B. et al.: Apel: an implementation of grid accounting using r-gma. In: UK e-Science All Hands Conference (2005)
12. FreePastry. http://freepastry.rice.edu/FreePastry
13. Huebsch, R., Hellerstein, J.M., Lanham, N.L., Boon, T., Shenker, S., Stoica, I.: Querying the internet with PIER. In: VLDB (2003)
14. Kantere, V., Tsoumakos, D., Sellis, T., Roussopoulos, N.: GrouPeer: dynamic clustering of P2P databases. Inf. Syst. **34**(1), 62–86 (2009)
15. Koloniari, G., Pitoura, E.: Content-based routing of path queries in peer-to-peer systems. In: EDBT (2004)
16. Lakshmanan, L., Pei, J., Zhao, Y.: QC-trees: an efficient summary structure for semantic OLAP. In: SIGMOD (2003)
17. Ng, W.S., Ooi, B.C., Tan, K.L., Zhou, A.: PeerDB: a P2P-based system for distributed data sharing. In: ICDE (2003)
18. Sismanis, Y., Deligiannakis, A., Kotidis, Y., Roussopoulos, N.: Hierarchical dwarfs for the rollup cube. In: DOLAP (2003)
19. Tang, C., Xu, Z., Dwarkadas, S.: Peer-to-peer information retrieval using self-organizing semantic overlay networks. In: SIGCOMM (2003)
20. Tatarinov, I., Halevy, A.: Efficient query reformulation in peer-data management systems. In: SIGMOD (2004)
21. Wang, W., Lu, H., Feng, J., Yu, J.X.: Condensed cube: an effective approach to reducing data cube size. In: ICDE (2002)
22. Zhang, X., Freschl, J., Schopf, J.: Scalability analysis of three monitoring and information systems: MDS2, R-GMA, and Hawkeye. J. Parallel Distrib. Comput. **67**(8), 883–902 (2007)

Athanasia Asiki received her Diploma in Electrical and Computer Engineering (2005) from the National Technical University of Athens, Greece. She is currently a Ph.D. Student in the School of Electrical and Computer Engineering, National Technical University of Athens. Her research interests include large-scale distributed systems, grid middleware, development of grid applications, search techniques in Peer-to-Peer systems. She is a student member of the IEEE and member of the Technical Chamber of Greece.

Dimitrios Tsoumakos currently holds a visiting faculty position in the Computer Science department of the University of Cyprus. He is also a senior researcher in the Computing Systems Laboratory of the Department of Electrical and Computer Engineering of the National Technical University of Athens (NTUA). He received his Diploma in Electrical and Computer Engineering from NTUA in 1999, he joined the graduate program in Computer Sciences at the University of Maryland in 2000, where he received his M.Sc. (2002) and Ph.D.(2006).

Nectarios Koziris received his Diploma in Electrical Engineering from the National Technical University of Athens (NTUA) and his Ph.D. in Computer Engineering from NTUA (1997). He joined the Computer Science Department, School of Electrical and Computer Engineering at the National Technical University of Athens in 1998, where he currently serves as an Associate Professor. His research interests include parallel architectures, loop code optimizations, interaction between compilers, OS and architectures, communication architectures for clusters (OS and compiler support) and resource scheduling (cpu and storage) for large scale computer systems. He has published more than 100 research papers in international refereed journals and in the proceedings of international conferences and workshops. He has also published two Greek textbooks "Mapping Algorithms into Parallel Processing Architectures", and "Computer Architecture and Operating Systems". Nectarios Koziris is a recipient of the IEEE IPDPS 2001 best paper award for the paper "Minimising Completion Time for Loop Tiling with Computation and Communication Overlapping" (held at San Francisco, California). He serves as a reviewer in International Journals and various HPC Conferences (IPDPS, ICPP etc.). He served as a Chair and Program Committee member in various IEEE/ACM conferences and workshops. He is a project leader in several EU (FP5, FP6 and FP7) and national Research Programmes. He is a member of IEEE Computer Society, member of IEEE-CS TCPP and TCCA (Technical Committees on Parallel Processing and Computer Architecture), ACM and chairs the Greek IEEE Computer Society Chapter. He also serves as the Vice-Chairman for the Greek Research and Education Network (GRNET-Greek NREN, www.grnet.gr), Vice-Chairman for the Free/Libre/Open Source Software non-profit company (ELLAK www.ellak.gr), founded by the Greek Universities and Research Centers.