Online Querying of Concept Hierarchies in P2P Systems

Katerina Doka, Athanasia Asiki, Dimitrios Tsoumakos and Nectarios Koziris

Computing Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens
{katerina, nasia, dtsouma, nkoziris}@cslab.ece.ntua.gr

Abstract. In this paper we describe *HIS*, a system that enables efficient storage and querying of data organized into concept hierarchies and dispersed over a network. Our scheme utilizes an adaptive algorithm that automatically adjusts the level of indexing according to the granularity of the incoming queries, without assuming any prior knowledge of the query workload. Efficient roll-up and drill-down operations increase the exact-match query ratio by shifting to the most favorable hierarchy level. Combined with soft-state indices created after query misses, our system achieves maximization of performance by minimizing query flooding. Extensive experimental evaluations show that, on top of the advantages that a distributed storage offers, our method answers the large majority of incoming queries without flooding the network and at the same time it manages to preserve the hierarchical nature of data. It shows remarkable performance especially for skewed workloads, which are frequently documented in the majority of Internet-scale applications. These characteristics are maintained even after sudden shifts in the workload.

1 Introduction

As the volume of produced data increases, so do the requirements for efficient processing in the various applications. Data warehouses, for example, host immense volumes of historical data, providing tools for their aggregation and management at different levels of granularity. Data are usually viewed in the form of multidimensional arrays (or *data cubes* [1]), which represent the basic abstraction in data-warehousing. Data cubes are characterized by their *dimensions*, which represent the notions that are important to an organization for managing its data (e.g., time, location, item, etc) and the *facts*, which are the numerical quantities to be analyzed (e.g., sales, profit, etc). Data cubes allow for efficient summarization of data by reducing the dimensions in the viewed data. However, data can be presented in an even more fine-grained manner through the use of *concept hierarchies*.

A concept hierarchy (or *taxonomy*) defines a sequence of mappings from more general to lower-level concepts. For example, Figure 1 shows a simple hierarchy for the `location` concept, where `Address` $<$ `ZipNo` $<$ `City` $<$ `Country` 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 a given cube at different levels of a dimension hierarchy: With the *roll-up* operation we climb up to a more summarized level of the hierarchy,

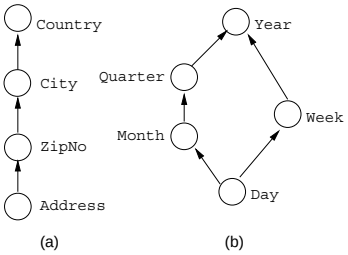

Fig. 1. A concept hierarchy for dimension (a) Location (b) Product.

Table 1. Sample data for our motivating scenario.

Location				*Measures*
Country	City	Zip	Address	Sales
Greece	Athens	16674	13 Promitheos st.	1,500
Greece	Athens	15341	40 Terpsitheas st.	9,900
Greece	Athens	16674	14 Ioanninon st.	2,450
Italy	Milan	20100	6 Modenna st.	12,100
Greece	Patras	26222	16 Tsamadou st.	1,990

while a *drill-down* defines the opposite operation (i.e., navigating to levels of the hierarchy with increased detail).

There has been significant effort in combining the advantages of a distributed and resilient solution such as P2P with the performance of storing large volumes of data in database systems. Peer-Database systems (e.g. [2–4]) represent a new trend in which peers maintain parts of a central or fully independent database and communicate with each other in a distributed, fault-tolerant manner. Nevertheless, there has been little effort in designing applications to store and query hierarchical data through a P2P system. Specifically, P2P databases that rely on DHT functionality are unable to directly support queries on dimension hierarchies.

In this paper we describe the *Hierarchical Indexing System (HIS)*, a DHT-based system that enables efficient storage and querying over dimensions characterized by specific hierarchies. *HIS* nodes actively monitor the granularity of posed queries in order to adjust the indexing level to the most beneficial one. Combined with soft-state indices which are dynamically created after query misses, our system manages to minimize the number of flooding operations necessary to provide exact answers. Furthermore, *HIS* does not invalidate the semantics of the stored hierarchies and allows for distributed knowledge mining. To our knowledge, this is the first attempt towards the support of concept hierarchies in DHTs.

The rest of the paper is organized as follows: Section 2 describes the problem and gives a preview of the proposed solution. In section 3 our method is thoroughly presented, while section 4 analyzes important aspects of our system, such as parameter selection and optimization. Our system is evaluated through experimental results in section 5. A brief overview of related literature follows in section 6 and finally the paper concludes in section 7.

2 Motivation and Problem Description

As a motivating scenario, let us consider the computational center of a supermarket chain that holds records of sales. Such historical data are aggregated (usually off-line) and queried for discovering interesting trends/associations. Instead of a centralized data warehouse, the management prefers a horizontal partitioning of the database (according

to some metric, e.g., geographically) so that they can perform on-line queries on the multiple dimensions. Moreover, it would be very important if simple mining operations (such as roll-up and drill-down) could be performed, as opposed to more complex queries that instead have to be processed off-line. Current work offers a variety of systems which can be used to distribute and query this information. DHTs offer a bandwidth-efficient, distributed substrate that robustly stores data and routes queries to a large number of nodes. Nevertheless, these schemes cannot be used to maintain the semantics of the hierarchy and efficiently retrieve views of the data at different granularities.

To see this, let us assume that the company's database contains a location dimension that relates to the suppliers' addresses. This dimension is organized along the hierarchy depicted in Figure 1(a). Table 1 shows some sample data relative to this dimension, with `Sales` being the fact of interest. In a plain DHT system, one would have to choose a level of the suggested hierarchy in order to hash all tuples to be inserted to the system. Assuming the tuples are hashed according to the `city` attribute, there will be a node responsible for tuples containing the value *Athens*, one for *Patras*, etc. This structure can be very effective when answering queries referring to the chosen attribute level, whereas queries concerning other levels of the hierarchy demand global processing.

The solution of multiple insertion of each tuple by hashing every hierarchy value is not efficient: As the number of levels increase, so does the redundancy of data and the storage sacrificed for this purpose. While exact-match queries would be answered without global processing, this scheme fails to encapsulate the hierarchy relationships: One cannot answer simple queries, such as "*Which country is Patras part of*" or "*Which zip-code does '15341' belong to*".

Our work intends to provide an indexing mechanism to facilitate storing and querying hierarchical data in DHT systems. *HIS* is an adaptive and bandwidth-efficient solution to this problem: Peers initially index at a default (*pivot*) level, but they internally store the inserted tuples in a hierarchy-preserving manner. Query misses are followed by soft-state pointer creations so that future queries can be served without re-flooding the network. Moreover, peers maintain local statistics which are checked in order to decide if a global re-indexing (to a lower or higher level of the hierarchy) is necessary. If the ratio of queries for `country` exceeds a threshold (assuming the pivot level is `city`), data would be re-indexed according to that level so that most requests would be directly answered.

It has been widely observed that most Internet-scale applications, including P2P ones, exhibit highly skewed workloads (e.g., [5–8], etc). *HIS* indexes popular levels for exact matches and uses indices to answer the less popular requests adapting to the incoming workload as a whole. Our extensive simulations show that our system effectively adapts the level of granularity of the indexing according to user requests, without prior knowledge of the query workloads. *HIS* achieves a high ratio of exact-match queries in a variety of workloads, even when these change dynamically with time. We show that our scheme is particularly efficient with highly skewed distributions (for both data and load), which are frequently documented in the majority of applications.

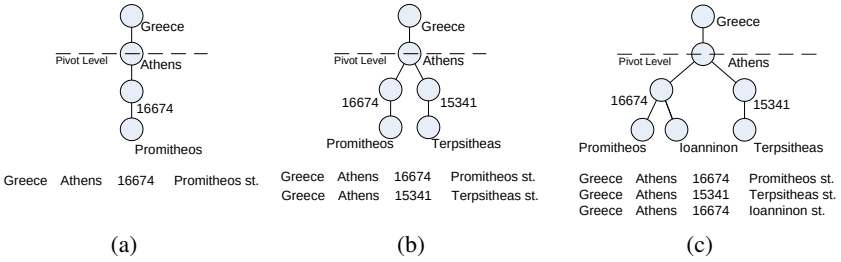

Fig. 2. Insertion of first (a), second (b) and third (c) tuple, hashed upon the `city` level, and creation of the tree structure in the node responsible for value *'Athens'*.

3 The Hierarchical Indexing System

HIS is a fully dynamic, self-adaptive protocol that can be applied over a P2P overlay in order to provide efficient mechanisms for storing, indexing and querying hierarchical data. Our goals are twofold: Efficient querying and preservation of the hierarchy semantics.

First, we address the efficiency of hierarchical data search. Our system should be able to provide with answers as efficiently as possible at various hierarchy levels of the data queried. Even though DHTs bind the number of query hops to the logarithm of the size of the overlay, they are unable to directly support queries on dimension hierarchies since they perform exact match lookups. Any other case would require message and time-consuming query flooding over the whole network.

Second, we intend to provide a hierarchy-aware system that will preserve useful hierarchy-specific information. Hash-based systems discard such information that exposes relationships between items. Hashing either on a single or on multiple levels of the hierarchy, a naive data insertion would fail to preserve the associations between the stored keys.

3.1 Data Insertion

The insertion of data items (by data item we refer to a tuple of the database) is performed as follows: Upon creation of the database, a level of the hierarchy, the *pivot* level, is globally selected. The ID of each tuple to be inserted is the hashed value of its pivot level. The DHT then assigns each data item to the node with ID numerically closest to that of the data item. For tuples inserted at a later stage, nodes can be informed of the global pivot level from one of their neighbors in the overlay.

Inserted data are stored in the form of trees that preserves their hierarchical nature. As a consequence, each distinct value of the pivot level corresponds to a tree that reveals part of the hierarchy. Let us assume the hierarchy described in Figure 1(a) with `city` as the globally defined pivot level. The first tuple to be inserted is assigned an ID that derives from applying our hash over the value *'Athens'* and forms a plain list (Figure 2(a)). As data items with the same ID keep arriving at this node, different values at levels lower in the hierarchy than the pivot level create branches, thus forming a tree structure. In Figure 2(b), a tuple with new `zipNo/address` values induces the creation of a new branch, while in Figure 2(c) the tree forks at the `address` level.

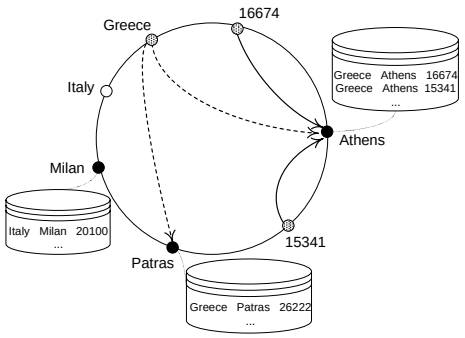

Fig. 3. Example of soft-state index creation.

3.2 Data Lookup and Soft-State Indexing

Queries concerning the pivot level are exact match queries and can be answered within O(log$N$) steps. Queries on any of the other levels cannot be answered unless flooded across the DHT. In order to facilitate this class of queries, we introduce *soft-state indices* to our proposed structure. These indices are created on demand, as soon as a query for non-pivot level data is answered. After the answers from the corresponding nodes are received through overlay flooding, the query initiator hashes the value of the requested key and sends the IDs of the nodes that answered the query to the node responsible for that key. In case of another query referring to the same value, the time and bandwidth consuming flooding is avoided and the response can be provided quickly and efficiently, within logN+C hops, C being the number of nodes the index points to.

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*), unless a new query for that specific value is initiated, in which case, the index is renewed. This mechanism ensures that changes in the system (e.g., data location, node unavailabilities, etc) will not result in stale indices, affecting its performance. 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. Calibrating I_{max} for performance without increasing it uncontrollably entails knowledge of our data (e.g., how skewed each hierarchy is, etc). The Lookup algorithm of *HIS* is presented in Algorithm 1.

Let us assume the same hierarchy as before, with `city` as the pivot level. When querying for the zip code with value *'16674'*, we discover that no such key exists in the DHT. Flooding is performed and the node *'Athens'* answers with the corresponding tuple. The initiator, which now knows the IDs of the nodes that answered the query, forwards them to the node responsible for the value *'16674'* which now has an index pointing to the node *'Athens'*.

The exact same procedure takes place when the query concerns a value that lies higher in the hierarchy than the pivot level. Both these cases are shown pictorially in Figure 3, where the black dots represent nodes that store the actual data, whereas nodes holding pointers are depicted in grey.

Algorithm 1 HIS Lookup Algorithm

Require: q: the query to be resolved
1: r: remote node
2: $K_{r,exact}$: set of keys stored by remote node, $K_{r,ind}$: set of keys indexed by remote node
3: $ID_q \leftarrow hash(q)$
4: DHT_route(LookupMessage(ID_q))
5: local processing by r
6: **if** $ID_q \notin K_{r,exact}$ **then**
7: check indices list $K_{r,ind}$
8: **if** $ID_q \notin K_{r,ind}$ **then**
9: no exact match found
10: flood(q)
11: local processing by r
12: answers returned by nodes having the corresponding tuples
13: DHT_route(IndexMessage($ID_q \rightarrow$ nodes that answered))
14: Receiver node builds index pointing to the nodes that answered and adds ID_q to $K_{r,ind}$
15: **else**
16: local processing, tuples returned
17: **end if**
18: **else**
19: tuples returned
20: **end if**

3.3 Reindexing operation

HIS is adaptive to the query distributions, supporting dynamic changes in the pivot level. By shifting to a higher or lower level (*rollup* and *drilldown* respectively), we aim to increase the ratio of exact-match queries, reduce floodings and boost performance.

If the number of queries initiated by a node regarding levels different than the pivot level exceeds the number of queries for the pivot level by some ratio, this node considers the possibility of a new partitioning. The mechanism functions as follows: Each node stores the number of queries (termed *popularity* and denoted p_{ℓ_i}) it initiates for each level of the L-level hierarchy ℓ_i, $0 \leq i \leq L-1$ within a restricted time-frame W. This time-frame should be properly selected to perceive variations of query distributions and, at the same time, stay immune to instant surges in load. By making use of these statistics, the most popular level ℓ_{max} is determined. If $p_{\ell_{max}}$ is more than *threshold%* of the total number of queries within this time frame, the node is positive to the potential of adopting another pivot level. This step is used as an indication of an imbalance that should be further investigated.

If this is the case, the reindexing enters its second phase, in which the local intuition must be confirmed (or not) using global statistics. The node whose local information indicates a possible shift of the pivot level sends a *SendStats* message to all system nodes. The initiator performs the same calculation using the global statistics this time, with the most popular level being ℓ_{max}. If ℓ_{max} is not *threshold%* or more of the total number of queries, no action is taken. In the opposite case, when

Algorithm 2 HIS Reindexing Algorithm

1: ℓ_p: current pivot level
2: p_{ℓ_i}: popularity of level ℓ_i
3: $L_{local} : \ell_0 < \ell_1 < \ldots < \ell_{max}$ levels ranked according to local popularity
4: **if** $p_{\ell_{max}} / \sum_{i=0}^{L-1} p_{\ell_i} \geq threshold$ **then**
5: flood(SendStatsMessage) and collect global statistics
6: $L_{global} : \ell_0 < \ell_1 < \ldots < \ell_{max}$ levels ranked according to global popularity
7: **if** $p_{\ell_{max}} / \sum_{i=0}^{L-1} p_{\ell_i} \geq threshold$ **then**
8: determine new pivot level ℓ_{p_new}
9: **if** $\ell_{p\_new} = \ell_p$ **then**
10: no action taken
11: **else if** ℓ_{p_new} higher in the hierarchy than ℓ_p **then**
12: flood(RollupMessage(ℓ_{p_new}))
13: $\ell_p \leftarrow \ell_{p\_new}$
14: deletion of all indices
15: rehashing of tuples
16: **else**
17: flood(DrilldownMessage(ℓ_{p_new}))
18: $\ell_p \leftarrow \ell_{p\_new}$
19: erasure of all indices
20: rehashing of tuples
21: **end if**
22: **else**
23: no action is taken
24: **end if**
25: **else**
26: no action is taken
27: **end if**

$$p_{\ell_{max}} / \sum_{i=0}^{L-1} p_{\ell_i} \geq threshold$$

then a rollup or drilldown is performed respectively by all nodes, depending on whether the new pivot level is higher or lower in the hierarchy than the current pivot level. Later on propose an enhanced mechanism for the new pivot level choice.

The node floods a *Rollup* or *Drilldown* message to all nodes of the system, to signal a shift to the new pivot level. Each node that receives this message traverses its tuples, finds all the values of the level that will constitute the new reference point and hashes them one by one, sending the tuples (or the indices to the real location of the data tuples) to the corresponding nodes. Assuming that the size of the dataset $|D| \gg N^2$, N being the size of the network, the preferred method to perform this is to send at most N messages per node, grouping the tuples by recipient. After the node completes the procedure, it erases all its data and indices. The algorithm is presented in Algorithm 2.

Back to our example, if the node *'Athens'* receives a *Drilldown* message for level `ZipNo`, it runs through its tuples and discovers that all values for that level are the ones

depicted in Figure 2(c), namely *'16674'* and *'15341'*. The values are hashed and the corresponding nodes are now responsible for the tuples containing these values.

3.4 Locking

In order to ensure the correctness of the answers during the rollup or drilldown procedure and to avoid simultaneous rollups or drilldowns by multiple nodes, we introduce a locking mechanism. After a node finally decides to perform reindexing according to the global statistics, it sends a *Lock* message to all nodes of the system and then proceeds to the rollup or drilldown. Once a node receives the *Lock* message, it changes its state to LOCKED and maintains it for a predefined period of time (related to the size of the system), which we assume adequate to cover the time needed for the whole system to finish the rollup or drilldown procedure and to reach a stable state. During this time, all queries are answered through flooding and no other operation is allowed.

4 Discussion

In this section we will briefly discuss some important aspects of *HIS* that relate to its parameters as well as optimization issues.

Memory requirements. Memory requirements for a *HIS* node include the space for the level statistics plus the storage required for the soft state indices: Specifically, each node requires $O(L) + O(I_{max}) = O(I_{max})$ memory, assuming that, in general, $L < 10$. As shown in the experimental section, the overhead may range from small to negligible, depending on the choice of I_{max}.

Parameter selection. A careful choice of the *TTL, W* and I_{max} parameters plays an important role in the performance of the system. A small *TTL* degrades the success ratio of the search mechanism, unnecessarily invalidating indices. Assuming the rate at which participating peers delete their data or disconnect is small (a reasonable assumption for our motivating application), a large value for *TTL* will not create a stale image that fails to reflect the infrequent changes.

The window parameter *W* represents the number of previous statistics that each node stores and uses in order to decide a pivot level change. A large value for *W* will fail to perceive load variations, whereas a very small value will possibly lead to frequent erroneous or conflicting reindexing decisions. In order to estimate its value, we set $W = O(1/\lambda)$, i.e., we connect the size of the window with the query interarrival time. The more frequent the requests, the smaller W can be and vice-versa. Finally, regarding the total amount of memory dedicated per node, this is dominated, as we mentioned before, by I_{max} and more specifically by the maximum number of non-pivot keys K_{max} that a node is responsible for (thus holds indices pointing to the relevant nodes). Assuming a (very optimistic) value of $N = 1K$ nodes for our application and that IDs and keys need 20 bytes (as outputs of SHA1 hash function), a node that will be responsible for 1K different keys will need at most 20KB of memory while for 10K keys a node will need at most 200KB of memory (certainly affordable by most modern desktop PCs).

Tuple insertion – Updates. Tuple updates are normally performed through an update of the tuple's measures at the corresponding node. One open issue relates to the

insertion of new tuples in the system. While hashing according to the current pivot level and storing the new item is trivial, there may exist indices that need to be updated since the new tuple must be included in the result set of various queries. As an example, consider an inserted tuple that documents sales in a new Greek zip code. An existing index for *'Greece'* should now include the ID of the node responsible for the new tuple. Since the creation of an index may be followed by one or more index deletions at the creating node (due to space constraints), the inserting node cannot know of the existence or not of an index relative to the new tuple a priori. This can be resolved in a variety of ways, according to the level of consistency that we require from our system. We distinguish the following two cases:

- *Weak consistency – allow for some incomplete answers:* Nodes periodically append the inserted tuples to a globally known location. Index-holding peers can then, asynchronously, retrieve this directory and update the required indices.
- *Strong consistency – require complete answers:* After each insertion, the node performs n lookups to identify the existence of all possible indices. Each node that holds a corresponding combination will update its value.

Enhanced pivot selection algorithm. Rollup or drilldown decisions have a large impact on our system in terms of bandwidth utilization and query performance: The reindexing mechanism not only requires global message exchange but also invalidates the so far created indices. Spikes in load can often produce such operations. Therefore, frequent index reorganizations should be avoided, yet beneficial reindexing should not be prevented. Besides the ranking of levels according to their popularity, the overall query distribution should be taken into account as well for the new pivot level choice, since it is possible that the system profits by choosing some less popular level than ℓ_{max}. Observing that the least costly reindexing process is the one that shifts the pivot level higher in the hierarchy, we propose an enhanced mechanism for choosing a new pivot level: After the collection of the global statistics, if the most popular level exceeds the *threshold* and the difference between that and some less popular levels is less than *diff*, all these levels are considered candidate levels. In that case, if one of the candidate levels is the pivot level, the system should make no change, otherwise, it should choose the level that is highest in the hierarchy from the candidate set.

Balancing Reindexing costs. Reindexing is a costly procedure, as it requires network flooding for the collection of statistics and the consecutive re-insertion of tuples. The latter dominates the complexity of the reindexing process which requires $\Omega(N^2)$ messages. Therefore, it is important to ensure that our gains from reducing query floodings outweigh this cost.

Minimizing Reindexing operations. In order to minimize the number of occasions where global statistics are collected due to nodes interested in suboptimal levels or malicious users, we introduce the $interval_n$ parameter for each node n, which defines the minimum interval between two consequent checks that can be initiated by n and coincides with the frequency of n checking its statistics. Its initial value T_s is the same for all nodes.

As mentioned earlier, it is possible that the globally optimal pivot level is contradictory to the interest of a single node. In such cases, local node statistics will be continuously dictating data repartitioning, refuted by the global statistics. In order to dis-

Table 2. Percentage of queries directed to the top 10% of the data values for various *levelDist*.

	Skewed towards	
levelDist/valueDist	$\rightarrow \ell_0$	$\rightarrow \ell_3$
0/0	52.3	52.3
0.5/0	63.3	40.6
1.0/0	73.4	31.1
1.5/0	82.5	24.5
2.0/0	88.8	22.6

Table 3. Percentage of queries directed to the top 10% of the data values for various *valueDist*.

	Skewed towards
levelDist/valueDist	$\rightarrow \ell_3$
1.0/0	31.1
1.0/0.5	40.5
1.0/1.0	78.3
1.0/1.5	98.9
1.0/2.0	100.0

courage consecutive reindexing attempts from the same node, we introduce a back-off mechanism, which multiplicatively increases the $interval_n$ when the processing of local and global statistics conclude in different results or in a no-change decision (i.e., $interval_n = 2T_s, 4T_s$, etc). Each time a *SendStats* message is flooded over the network, $interval_n$ is reset to the maximum between the current value and T_s, regardless of the outcome (whether a reindexing is decided or not).

5 Experimental Results

We now present a comprehensive simulation-based evaluation of *HIS*. Our performance results are based on a heavily modified version of the FreePastry simulator [9], although any DHT implementation could be used as a substrate. We assume a network size of 512 nodes, 100 of which are randomly chosen to initiate queries at any given time. Experiments conducted with up to 1K nodes showed little qualitative difference.

In our simulations, we use synthetically generated data. Our data is a tree with each value having at most one parent. We are using a uniform distribution per level, meaning that each distinct value of ℓ_i has a constant number of children in ℓ_{i+1}. By default, our data comprise of 10000 tuples, organized in a 4-level hierarchy (see Figure 1(a)) with one numerical fact (`sales`). The number of distinct values per level are $|\ell_0 = 500|, |\ell_1 = 1000|, |\ell_2 = 5000|$ and $|\ell_3 = 10000|$. The level of insertion is, by default, ℓ_1 (`city`), 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 both *levelDist* and *valueDist*. In the rest of this section, we use the notation a/b to identify workloads, where a and b correspond to the values of θ for *levelDist* and *valueDist* respectively. Tables 2 and 3 quantify the level of skew for commonly used workloads in our simulations, when the skew is directed towards the first (ℓ_0) or the last hierarchy level (ℓ_3).

Generated queries arrive at an average rate of $1\frac{query}{sec}$, in a 50000 sec total simulation time. We present results for queries on a single dimension with multiple levels of hierarchy. Our default *threshold* value is set to 30%, which is a large enough value to avoid very frequent reindexing attempts. Simulations with different values of threshold 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 500 seconds. Finally, for our experiments we assume a practically infinite value of *TTL* (indices never expire).

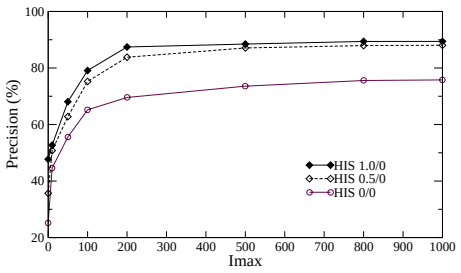

Fig. 4. Precision over variable values of the I_{max} parameter.

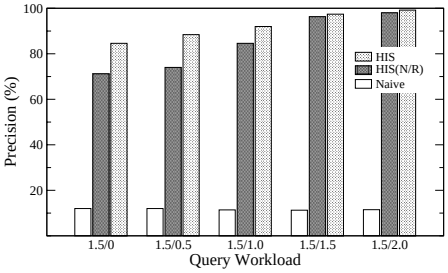

Fig. 5. Precision for varying *valueDist* (skew towards ℓ_3).

In this section, we intend to demonstrate the performance and adaptability of *HIS* under various conditions. Our goal is to show that *HIS* is 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*). We compare *HIS* with the naive protocol (referred to as *Naive*), where precision equals the ratio of queries concerning the insertion level, and a special case of *HIS*, where only the soft-state indices are utilized and no reindexing occurs (referred to as *HIS(N/R)*).

The effect of the I_{max} parameter

The value of the I_{max} parameter is very important as it specifies the maximum number of different non-pivot values that a node can index, and thus defines, as described earlier, the memory requirements of each node. The effect of the I_{max} parameter on the system's precision is examined in this set of experiments, with its value varying from 0 to 1000 for three workloads, 0/0, 0.5/0 and 1.0/0, directed towards ℓ_3. Results are depicted in Figure 4.

As expected, the system performance improves as I_{max} increases for all workload skews. As the number of indices increases, more queries can be answered using this mechanism. There exists a point I_{thres}, beyond which no significant improvement is observed. The I_{thres} value as well as the documented slope strongly depend on the data and query workloads. When no reindexing occurs (0/0 workload), the I_{thres} value is larger since *HIS* solely relies on indices to improve its performance, plus more distinct values are requested. In the two skewed workloads, *HIS* tracks the optimal pivot level and shifts to it, hence less space dedicated to indices is necessary to achieve high performance. Finally, it is worth noticing that the more biased the workload, the lower the performance gains. This is due to the fact that a greater θ value results in more duplicate queries, thus in fewer distinct keys that need to be indexed. The dominant performance mechanism in these cases is the indexing level.

For the rest of the experiments we assume that each node stores up to $I_{max} = 1$k indices, a large enough value which ensures that for the data and query workload of our experiments the majority of the created indices will remain in the system. This heavily favors the *N/R* method, since *HIS* discards all indices each time a reindexing occurs. With this value, each node needs to dedicate approximately 20KB of main memory for *HIS* (excluding the data of course, which may or may not be physically located at the DHT).

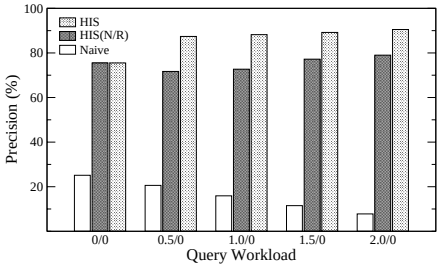

Fig. 6. Precision for varying *levelDist* (skew towards ℓ_3).

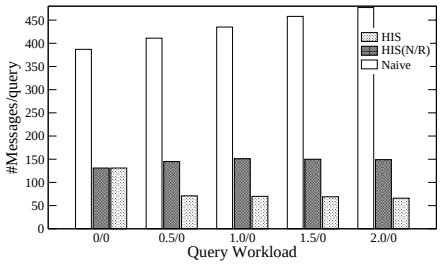

Fig. 7. Average number of messages required to answer a query (skew towards ℓ_3).

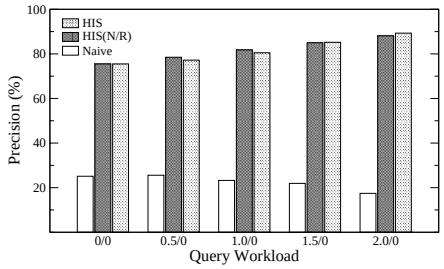

Fig. 8. Precision for varying *levelDist* (skew towards ℓ_0).

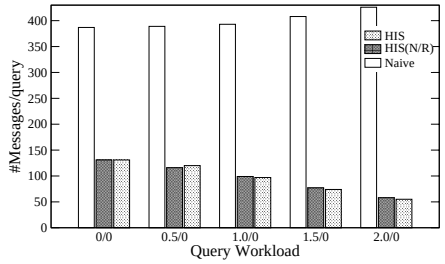

Fig. 9. Average number of messages required to answer a query (skew towards ℓ_0).

Performance with varying query distributions

In this set of simulations, we first vary the *levelDist* distribution, keeping $\theta = 0$ for *valueDist* (Figures 6-9). Afterwards, we vary the *valueDist* distribution, maintaining $\theta = 1.5$ for *levelDist* (Figure 5). In both cases we alter the direction of skew towards ℓ_0 or ℓ_3, using the default parameters otherwise.

In Figure 6, data are skewed towards the lowest level ℓ_3. As θ increases for *levelDist* , the performance of *HIS* improves: Reindexing is performed sooner and the exact matches due to the chosen pivot level increase. It reaches very high levels of precision (above 90%), while even with uniform requests (0/0) our method can directly answer close to 80% of all queries. *HIS* directly answers three to ten times more queries than the static method (i.e., *Naive*). In such cases, where *valueDist* is uniform and the most popular level consists of a large number of distinct values, indices alone do not suffice to serve queries, therefore *HIS(N/R)* demonstrates noticeably worse performance than *HIS*. However, as θ increases, *HIS(N/R)* starts to catch up with *HIS* due to query redundancy.

By increasing θ for *valueDist* (Figure 5), 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. The higher the θ, the more *HIS(N/R)* approaches *HIS*. This is due to the fact that indices can serve more queries since the number of distinct requests decreases.

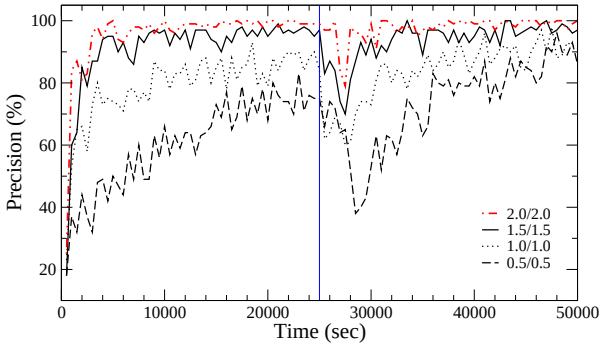

Fig. 10. Precision over time for various workloads when a sudden shift in skew occurs at $t_c = 25000sec$.

Figure 8 shows results where our workload favors ℓ_0. Again, we notice a similar trend in performance as the values for θ increase. Nevertheless, *HIS* is slightly less effective than *HIS(N/R)* (less than 5%), with this difference diminishing as θ increases. This is due to the limited number of distinct values of ℓ_0 which facilitates the maintenance of indices, while for *HIS* all created indices are erased during the rollup procedure. However, since we assume the same I_{max} parameter for both methods, *HIS* naturally outperforms its competition in the steady state, as it can increase its performance with time.

Figures 7 and 9 depict the number of messages that need to be exchanged in the system in order to answer a query, indicating a measure of bandwidth consumption. The numbers illustrated include the request as well as response messages for each query. While both methods are clearly more bandwidth efficient than the *Naive* method, *HIS* results in significantly less messages per query than *N/R*, as queries answered through indices, although preferable than floodings, require more messages than the ones answered directly (exact-matches). The difference is more noticeable in the workloads favoring ℓ_3 (Figure 7). The explanation is similar to that for the precision ratio: A skew towards the lowest level produces more distinct queries, thus more floodings, which strongly affect the communication overhead. *HIS* avoids that by shifting to the most popular level inducing some communication cost. Besides the creation and maintenance of indices, which is common in both *HIS* and *N/R*, *HIS* needs some extra messages in order to collect statistics and perform reindexing (referred to as *control* messages). However, in all experiments the documented ratio of control messages over the total number is less than 5%. Nevertheless, our gains in precision and reduction in floodings greatly outperform this small overhead.

Performance in dynamic environments

In the next experiment, we measure the performance and adaptivity of *HIS* in dynamic environments, namely sudden changes in the workload. We tailor our query distribution so that a sudden change occurs in the middle of the simulation ($t_c = 25000sec$): From a skewed workload towards ℓ_0 we shift to a skewed load towards ℓ_3. We show the results for various levels of skew in Figure 10.

Our results show that, in all cases, *HIS* quickly increases its precision due to the combination of automatic reindexing and soft-state indices. While floodings increase after t_c (hence the decline in precision), it quickly manages to recover and regain its perfor-

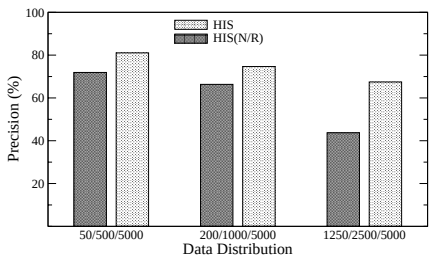

Fig. 11. Precision over various data distributions.

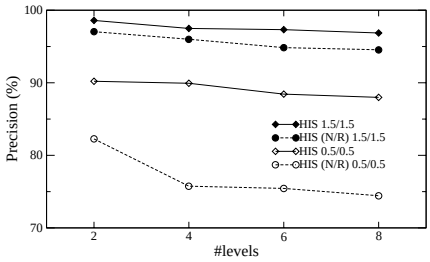

Fig. 12. Precision over variable number of hierarchy levels.

mance characteristics. The rate at which these events occur depends on the amount of skew: In the $2.0/2.0$ and $1.5/1.5$ cases, we show remarkable increase in precision (starting from the plain data insertion at $t = 0sec$), fast recovery after the change in skew and convergence to almost 100% precision. For less skewed distributions ($0.5/0.5, 1.0/1.0$), the results record a slight deterioration in the rate of convergence as well as a decline in precision from the change in skew: The decline ranges from less than 10% in the $2.0/2.0$ case to about 35% in the worst-case. Once again, we observe that *HIS* performs best in skewed workloads, but its performance in the steady state is invariably high, regardless of the workload.

Performance with varying data distributions

In this experiment, we show the impact of the distribution of data on the performance of our method. We vary the number of children a value can have in our data-trees: We shift from data with relatively balanced populations of the hierarchy levels to very skewed ones, e.g., $|\ell_n| \gg |\ell_i|, i < n$. We generate three datasets for a three-level hierarchy, with a different number of distinct values for each level except the bottom one (i.e., we maintain a constant number of tuples, $|\ell_2| = 5000$). We annotate each dataset using the notation $|\ell_0|/|\ell_1|/|\ell_2|$. The queries follow a zipfian $1.5/0$ distribution that changes midpoint, in a manner similar to the previous experiment: Half of the simulation time we favor ℓ_0, while the other half we favor ℓ_2. The results, for a simulation period of 10000 sec, are depicted in Figure 11.

Our method shows clear advantages over an indices-only scheme for all datasets. First, we note that *HIS* increases its performance as the dataset becomes more skewed towards the lower levels. This is due to the fact that, reindexing towards these levels enables *HIS* to answer almost all queries directly as the small number of distinct values for the remaining ones can be answered through indices. As the datasets become more "balanced", the creation and maintenance of indices fails to accommodate all queries, thus reducing the precision of both methods. Second, we note that *HIS(N/R)* is more heavily affected from the data distribution since its performance depends exclusively on the soft-state indexing, while our method shows a decline of less than 15%.

Testing *HIS* with different number of hierarchy levels

In this set of simulations we plan to identify the possible performance variations caused by datasets with different number of hierarchy levels. By altering the cardinality of each level, we create equal-size data and query-sets with the same *levelDist* and *valueDist*

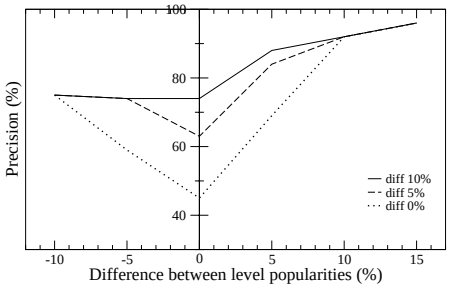

Fig. 13. Precision over various popularity differences for different values of *diff*.

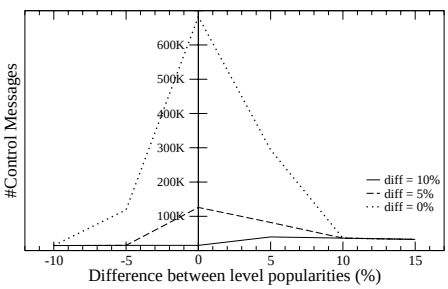

Fig. 14. Number of control messages exchanged for different values of *diff*.

values. Figure 12 depicts the results for 2, 4, 6 and 8 levels for two different workloads, skewed towards ℓ_3: 0.5/0.5 and 1.5/1.5.

As the number of levels increases linearly, the popular levels' request rates decrease, thus reducing the number of exact match queries for the level *HIS* chooses. Nevertheless, the performance degradation for *HIS* is negligible in both cases, because it relies on both the reindexing mechanism as well as soft-state indices. In the *N/R* version and for less skewed workloads, precision is more heavily affected by an increase in the number of levels, as the number of queries that cannot be accomodated by indices increases. For more skewed workloads, both schemes show very high precision. Nevertheless, variations in the value of θ have a much smaller effect on *HIS* compared to the *N/R* version.

Pivot level selection and the effect of the *diff* parameter

This experiment intends to prove the effectiveness of the enhanced pivot selection algorithm in terms of performance as well as bandwidth consumption. We produce workloads where the initial pivot level has a constant average popularity of 35%. Initially, the pivot level is the most popular one (ℓ_{max}=*pivot*), with the second most popular level (ℓ_3 in our case) falling short by 10%. The popularity of ℓ_3 gradually increases, reaching and finally exceeding that of the pivot by up to 15%. We measure the precision as well as the number of control messages, which include messages used for the creation and maintenance of indices as well as those exchanged to collect statistics and perform reindexing. The experiment is conducted with variable values of the *diff* parameter (0%, 5% and 10%).

As depicted in Figure 13, the performance noticeably degrades as the average popularity of ℓ_3 approaches that of the pivot levels for small values of *diff*. When *diff*=0, meaning that the system always adopts ℓ_{max} as the new pivot level, we observe a maximum degradation of 30% for the case where $p_{\ell_3} \simeq p_{\ell_{pivot}}$. This is due to the fact that minor temporal differences in popularities lead to frequent global reindexings (up to 50 such operations for the specific case). These consecutive reorganizations decrease the system's precision, also causing invalidation of the so far created indices and further decreasing the exact match rate. The communication overhead is heavily affected as well, as illustrated in Figure 14, which depicts the number of control messages exchanged throughout the simulation. When *diff* is small, besides an increase in messages enforcing rollups or drilldowns, the system suffers the increase in *SendStats* messages. When *diff*=0, equal level popularities result in the exchange of up to 60 times more control messages. The more concrete the difference between the two most popular levels, the

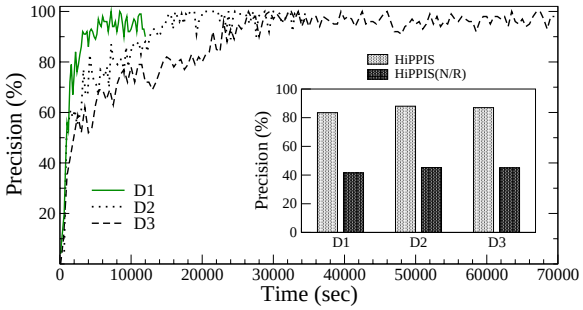

Fig. 15. Performance of HIS for the three APB datasets.

Table 4. Number of distinct values and number of tuples for the three APB-generated data and query sets.

	|Product|	# tuples	#queries
D1	9K	12M	12K
D2	45K	1500M	37K
D3	90K	12300M	75K

more confident the system is of the optimal pivot level and thereby the less unnecessary reindexings it performs.

As *diff* increases, the system's precision is less affected by similar average popularities of the two most popular levels. In the case of *diff*=10%, we notice that even with similar popularities, the system demonstrates no degradation. On the contrary, it maintains a steadily increasing precision, performing reindexing only when necessary. Furthermore, it demonstrates a significantly lower communication overhead (an order of magnitude fewer control messages).

Reindexing and flooding costs

As aforementioned, the cost of reindexing is non-negligible. Hence, it is very important that the system performs the minimum required reindexing rounds. *HIS* proves extremely efficient to that end, carrying out one reindexing process per direction of skew: This translates to a single reindexing for the workloads with no change and two for the workloads that change the direction of skew. Moreover, we calculated that one *HIS* reindexing equals 13 queries answered by flooding. Since we only perform the necessary reindexing(s) and less than 5 *SendStats* requests are produced per simulation, our method is substantially more bandwidth efficient.

APB Benchmark Datasets

We also test the performance of *HIS* using some realistic data and query sets generated by the APB-1 benchmark [10]. 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). We generate three datasets with increasing values for |`product`| and the produced data-cube (see Table 4). Results are depicted in Figure 15, with the enclosed graph showing the average precision.

In all cases, *HIS* achieves very high precisions (near 100%) with small ($< 10\%$) fluctuations. In all three simulations, the precision reaches 80% in about $1/3$ of the queries and reaches its peak after half the queries have taken place. It is noteworthy that this performance is registered for extremely voluminous datasets (in the order of billions of tuples), with a very small number of queries (compared to the size of the dataset) and using the same I_{max} parameter as with our default datasets (despite that the distinct values are 20 times as many). Another observation relates to the rate of convergence: The

smaller the dataset, the faster the convergence rate. This is again due to the fact that we use a limited amount of memory for the indices, therefore a big increase in the number of distinct values queried (this is natural as the dataset increases), cannot accommodate all requests.

6 Related Work

There has been significant work in the area of databases over P2P networks. PIER [3] 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 PIER platform is also used along with a Gnutella overlay in [11] for common file-sharing. The Chatty Web [12] 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. PeerDB [2] features relational data sharing without schema knowledge. Query matching and rewriting is based on keywords provided by the users. GridVine [13], and pSearch [14] 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 hierarchical data.

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.

In [16], hierarchies are exploited to enable faster computation of the possible views and a more compact representation of the data cube. Another approach is the DC-Tree [17], a fully dynamic index structure for data warehouses modeled as data cubes. In this approach, the attributes of a dimension are partially ordered with respect to the valid hierarchy schema for each dimension. 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.

7 Conclusions

In this paper we described *HIS*, an adaptive technique for storing, indexing and querying data organized in hierarchical dimensions for DHT overlays. *HIS* performs a combination of active reindexing according to the granularity of the incoming queries and soft-state indexing, achieving both high performance and adaptation to various shifts in workload, without assuming any prior knowledge of it and with minor communication and computation overhead. By preserving the nature of the stored tuples, it enables on-line querying on different granularities of voluminous data with 100% recall.

Our simulations, using a variety of workloads and data distributions, show remarkable performance in bandwidth efficiency (up to 10 times less query flooding and over 90% precision) compared to a naive DHT insertion scheme. Moreover, it shows very fast

adaptation to dynamic changes in load, especially for skewed ones, which are frequently documented in the majority of Internet-scale applications. Even with low amounts of skew, *HIS* manages to answer the majority of queries within $O(\log N)$ steps, by detecting the most popular level and shifting to it. We also demonstrated that our scheme achieves high performance for a number of different datasets, when both the number of levels or the distribution of values within each level vary.

Acknowledgments

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

References

1. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Min. Knowl. Discov. **1**(1) (1997) 29–53
2. Ooi, B., Shu, Y., Tan, K., Zhou, A.: PeerDB: A P2P-based System for Distributed Data Sharing. In: ICDE. (2003)
3. Huebsch, R., Hellerstein, J., Boon, N.L., Loo, T., Shenker, S., Stoica, I.: Querying the Internet with PIER. In: VLDB. (2003)
4. Halevy, A., Ives, Z., Madhavan, J., Mork, P., Suciu, D., Tatarinov, I.: The Piazza Peer Data Management System. In: IEEE Transactions on Knowledge and Data Engineering. (2003)
5. Cha, M., Kwak, H., Rodriguez, P., Ahn, Y., Moon, S.: I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In: IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement. (2007)
6. Ripeanu, M., Foster, I., Iamnitchi, A.: Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. IEEE Internet Computing Journal **6**(1) (2002)
7. Sen, S., Wong, J.: Analyzing peer-to-peer traffic across large networks. In: SIGCOMM Internet Measurments Workshop. (2002)
8. Chu, J., Labonte, K., Levine, B.: Availability and locality measurements of peer-to-peer file systems. In: SPIE. (2002)
9. FreePastry: http://freepastry.rice.edu/freepastry
10. APB-1: OLAP Council APB-1 Benchmark, http://www.olapcouncil.org/research/resrchly.htm
11. Loo, T., Hellerstein, J., Huebsch, R., Shenker, S., Stoica, I.: Enchancing p2p file-sharing with an internet-scale query processor. In: VLDB. (2004)
12. Aberer, K., Cudre-Mauroux, P., Hauswirth, M.: The Chatty Web: Emergent Semantics Through Gossiping. In: WWW Conference. (2003)
13. Aberer, K., Cudre-Mauroux, P., Hauswirth, M., Pelt, T.V.: Gridvine:Building internet-scale semantic overlay networks. In: International Semantic Web Conference. (2004)
14. Tang, C., Xu, Z., Dwarkadas, S.: Peer-to-peer information retrieval using self-organizing semantic overlay networks. In: SIGCOMM. (2003)
15. Koloniari, G., Pitoura, E.: Content-based routing of path quieries in peer-to-peer systems. In: EDBT. (2004)
16. Sismanis, Y., Deligiannakis, A., Kotidis, Y., Roussopoulos, N.: Hierarchical dwarfs for the rollup cube. In: DOLAP. (2003)
17. Ester, M., Kohlhammer, J., Kriegel, P.: The dc-tree: A fully dynamic index structure for data warehouses. In: ICDE. (2000)