Support for Concept Hierarchies in DHTs

Abstract

Concept hierarchies greatly help in the organization and reuse of information and are widely used in data warehouses. 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 incoming queries. Roll-up and drill-down operations are performed on a per-node basis in order to minimize the number of floods for answering queries on varying levels of granularity. Initial experimental results support this argument on a variety of workloads.

1 Introduction

A concept hierarchy (or *taxonomy*) defines a sequence of mappings from more general to lower-level concepts. For example, Figure 1(a) 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 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 hierarchy with increased detail).

While there has been considerable work in sharing simple relational data using both structured and unstructured overlays (e.g., [3–5]), no special consideration has been given to data supporting hierarchies. In [6], 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 [1], a fully dynamic index structure for data warehouses modeled as data cubes. Nevertheless, these are strictly centralized solutions. 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.

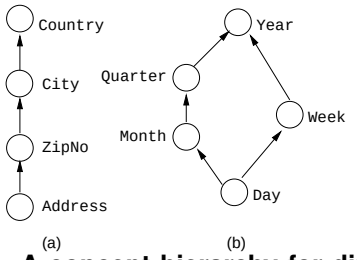

Figure 1. A concept hierarchy for dimension (a) Location (b) Time (lattice).

Consider, as an example, 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.

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) and we also assume that `Sales` is 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 *Milan*, 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. If the majority of queries concerns a level other than city, it is obvious that the system does not perform well.

The solution of multiple insertion of each tuple by hashing every hierarchy value is not viable: 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 Athens part of*" or "*Which addresses correspond to zip-code '15341'*".

In this paper, we present a first attempt to produce a system that efficiently stores and queries multi-dimensional data organized in hierarchies. First, we address the efficiency of lookups: Our system should be able to provide with answers on different levels of the data hierarchies. As we mentioned before, queries over a hierarchy level other than the default one will result in multiple floods. 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. Hash-based systems naturally discard such information: Through hashing on a single or multiple levels of the hierarchy, a naive data insertion would fail to preserve the associations between the stored keys. 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.

2 An Adaptive Indexing Scheme to Support Concept Hierarchies

In this paper, we propose a method for storing, indexing and querying hierarchical data in a DHT-based network. Our goal is to enable efficient querying while preserving the hierarchy semantics. The data items are structured according to a concept hierarchy with *L levels*, which defines the mapping from general to lower-level concepts. We consider that the root's hierarchy level (hence *root level*) is l_0. We define that $l_a < l_b$, if and only if l_a is closer to l_0 than l_b, namely l_a is higher in the concept hierarchy than l_b. Each tuple to be stored in the DHT is actually a record in the fact table of our data warehouse. It contains values for each level l_i of the hierarchy and numerical values corresponding to the facts of interest (e.g., sales). Our data is a tree with each value having at most one parent (e.g., see Figure 2(b)). In this section, we describe in detail how the data insertion, indexing and lookup is performed.

2.1 Data Insertion

The insertion of tuples is performed as follows: Initially, a level of the concept hierarchy is selected. The values of this level are hashed to be used as keys. In the rest of the paper, we refer to this level as the *pivot level*. Each group of tuples with same value in the pivot level is assigned to the node with ID numerically closest to its key.

Each peer organizes the tuples with same value in the *pivot level* in tree structures 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.

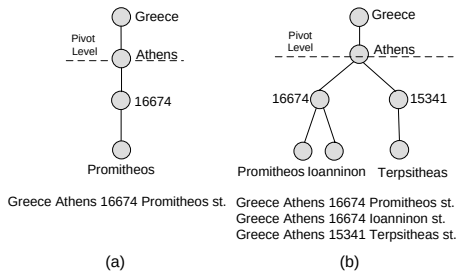

Figure 2. Evolution of the tree structure in the node responsible for 'Athens' during the insertion of tuples hashed upon the `city` level

An example of insertion is shown in Figure 2. Let us assume that tuples follow the concept hierarchy for location depicted 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 function over the value *'Athens'* and forms a plain list (see 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 (see Figure 2(b)).

2.2 Data Lookup and Soft-state Indices

Queries concerning the pivot level are exact match queries and can be answered within O($\log N$) steps, N being the number of participating peers. Queries concerning any of the other levels cannot be answered unless flooded across the DHT. In order to exploit the knowledge acquired from flooded queries, we introduce *soft-state bidirectional indices* to our proposed structure. These indices are created on demand according to the following procedure, when a flooded query is answered. When a node answers a query received through overlay flooding, 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 a soft-state index, as soon as it receives the complete answer. It hashes the value of the requested key found through flooding and inserts it along with IDs of nodes having the actual tuples in the DHT. The node that receives the index informs the nodes having the tuples for the successful index creation. 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 unavailabilities, etc) will not result in stale indices, affecting the validity of the lookup mechanism.

The nodes holding the tuples of the indexed value need to

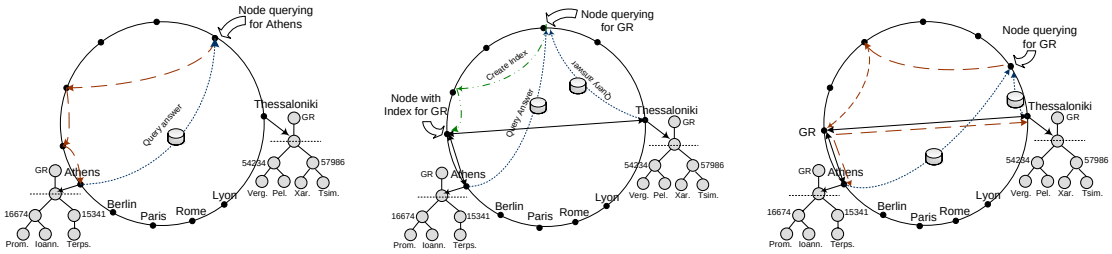

Figure 3. (a) Lookup in the DHT overlay for *'Athens'* when the `city` is selected as pivot level. (b)Lookup for value country *'GR'*, which ends up to flooding and creation of index when the query is answered. (c) Lookup for the indexed value *'GR'*.

know the existence of an index, in order to erase it if a roll-up or drill-down takes place. The bidirectional indices are introduced only in order to ensure data consistency, even though the indices are soft-state. We want to prevent the existence of stale indices after a roll-up or drill-down and at the same time to avoid increasing the complexity of the system by updating them.

In the example of Figure 3, the lookup for the hashed value of *'GR'* ends up with no results. The next step is the flooding of the query and the nodes with the keys *'Athens'* and *'Thessaloniki'* answer with the corresponding tuples. The query initiator, which now knows the IDs of the nodes that answered the query, inserts these IDs to the node responsible for the value *'GR'*. This node now has an index pointing to the node *'Athens'* and another to node *'Thessaloniki'*. In the future, queries for *'GR'* will be answered without flooding. The node responsible for the *'GR'* key will forward the query to all relative nodes directly, when it is reached by a subsequent lookup operation for *'GR'* (see Figure 3(c)).

2.3 Reindexing Operation

The proposed indexing method is adaptive to the query distribution and supports partial changes of pivot levels. Our aim is to adapt the pivot level of the inserted tuples dynamically and independently for each tree, in order to increase the ratio of the exact-match queries.

The re-indexing mechanism works as follows: Each node is responsible for a number of data trees. It maintains a record per tree containing the number of queries per level 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. When a node answers a flooded query, it checks if the number of queries for a level exceeds the number of queries for the pivot level. In order to decide if a roll-up or drill-down is needed, the level l_a of the queried value plays an important role.

If the queried level l_a is lower than the *pivot level* of the tree, then there exists only one tree consisting of tuples that answer the query. A drill-down is performed, if the total number of queries on the candidate level is more than a *threshold%* of the total number of queries for that tree within the time frame W. If this is the case, 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 in 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.

In the trees of Figure 4(a), we assume that queries for `zip code` of *'Paris'* exceed the threshold, while this is not the case for the other trees. A flooding query either for the `zip code` level or the `address` level will trigger the re-indexing mechanism. Since the threshold condition is satisfied for the `zip code`, a drill-down is performed resulting in a new state of the existing trees depicted in Figure 4(b). The performed drill-down does not affect the lookup method, which will continue to return the correct results for all levels.

If the queried level l_a is higher that the *pivot level*, then there are more than one trees with this value which have to participate in the roll-up. Otherwise, lookups for this value will not return complete results. A node checks for roll-up when it answers a flooded query for a level l_a higher than the *pivot level*. If the percentage of the queries for this level is more than *threshold%* of the total queries, then the node is positive to the potential of adopting another *pivot level* for this tree. This step is indicative of an imbalance and the query initiator is informed about this. If the query initiator is aware of at least one node willing to roll-up to this level, it starts a procedure to confirm the local intuition by using statistics from all the nodes having answered the query. If the candidate *pivot level* is *threshold%* or more of the total number of queries, then the query initiator messages the in-

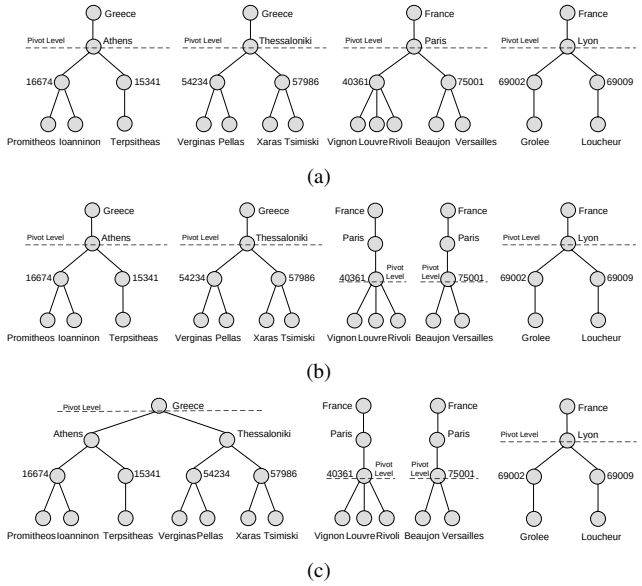

Figure 4. Sample data and roll-up / drill-down operations

volved nodes to roll-up the corresponding trees to this level by re-inserting these tuples. In the opposite case, no other action is taken than the creation of a soft-state index for the value of this level. The partial roll-up to the country level for trees containing the value *'Greece'* is shown in Figure 4(c).

3 Experimental Results

We now present an early simulation-based evaluation of the proposed method. We utilize a modified version of FreePastry [2], although any DHT implementation can be used. We assume a network of 512 nodes, which are randomly chosen to initiate queries. We use synthetically generated data on a single dimension, consisting of 5000 tuples organized in a 4-level hierarchy with one numerical fact. The number of distinct values per level are $|\ell_0 = 100|, |\ell_1 = 500|, |\ell_2 = 1000|$ and $|\ell_3 = 5000|$. The level of insertion is, by default, l_1 (`city`). The values per level are uniformly distributed and each distinct value of level l_i has a constant number of children in l_{i+1}. The *threshold%* is set to 20%, which is considered an acceptable value in order to avoid frequent drill-down and roll-up operations.

Queries follow the Zipfian distribution, i.e., #queries for item $i \sim 1/i^{\theta}$. We vary the value of θ in order to control the amount of skew of the queried levels and the queried values from each level. We measure the ratio of queries answered without flooding (*precision*) and the results are shown in Figure 5. The top graph represents the situation when the workload is skewed towards ℓ_0 and the bottom one when it is skewed towards ℓ_3. As θ increases, the workload becomes more skewed and the performance of our method

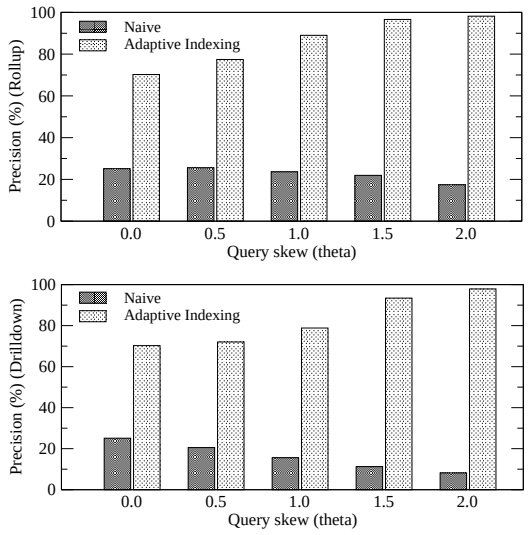

Figure 5. Precision for variably skewed workload

improves: Reindexing is performed sooner and indices are more frequently refreshed, thus serving more queries. We achieve very high levels of precision (close to 100% for high skew and over 70% when all levels and their values are uniformly requested). The results clearly show that the proposed method outperforms the "naive" method, where no soft-state indices are used and drill-down and roll-up operations are not performed. The performance is slightly better when l_0 is the most popular level, since there are less distinct values for lower levels, making the re-indexing decision easier to be made. Even if a roll-up is not performed, the soft-state indices can serve more queries for these levels.

We are currently implementing a locking mechanism to assure full data consistency during the reindexing operations. We also work on algorithmic improvements to reduce communication costs in combining node statistics as well as specific mechanisms for new tuple insertion and updates. Our goal is to evaluate the performance of the proposed method for more dynamic workloads and multiple dimensions.

References

[1] M. Ester, J. Kohlhammer, and P. Kriegel. The dc-tree: A fully dynamic index structure for data warehouses. In *ICDE*, 2000.

[2] FreePastry. http://freepastry.rice.edu/freepastry.

[3] R. Huebsch, J. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.

[4] V. Kantere, D. Tsoumakos, and T. Sellis. Semantic grouping of social networks in p2p database settings. In *DEXA*, 2007.

[5] B. Ooi, Y. Shu, K. Tan, and A. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *ICDE*, 2003.

[6] Y. Sismanis, A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical dwarfs for the rollup cube. In *DOLAP*, 2003.