HiPPIS: An Online P2P System for Efficient Lookups on d-Dimensional Hierarchies

Katerina Doka
School of Electrical and Computer Engineering
National Technical University of Athens, Greece
katerina@cslab.ntua.gr

Dimitrios Tsoumakos
School of Electrical and Computer Engineering
National Technical University of Athens, Greece
dtsouma@cslab.ntua.gr

Nectarios Koziris
School of Electrical and Computer Engineering
National Technical University of Athens, Greece
nkoziris@cslab.ntua.gr

ABSTRACT

In this paper we describe *HiPPIS*, a system that enables efficient storage and on-line querying of multidimensional 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 workload. Efficient roll-up and drill-down operations take place in order to maximize the 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, both point and aggregate ones, without flooding the network. At the same time, it manages to preserve the hierarchical nature of data. These characteristics are maintained even after sudden shifts in the workload.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Design, Performance

Keywords

Distributed Hash Table, Data Warehousing, Concept Hierarchies

1. INTRODUCTION

As the volume of produced data increases, so do the requirements for efficient data processing by the various applications. In both the business and the research domain, data are usually viewed in the form of multidimensional arrays (or *data cubes* [8]). 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, product, customer, 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* defines a sequence of mappings from more general to lower-level concepts. For example, a simple hierarchy for the `location` dimension could be: `Address` < `ZipNo` < `City` < `Country`; and one for `product`: `Product` < `Brand` < `Category`. Hierarchies allow users to view a given cube at different levels of granularity: With the *roll-up* operation we climb up to a more summarized level of the hierarchy, while a *drill-down* navigates to lower levels of 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.

Data warehouses usually host immense volumes of historical data and provide tools for their aggregation and management at different levels of granularity. Yet, this is a strictly centralized (in terms of data location and processing) and off-line approach (views are usually calculated on a daily or weekly basis after the operational data have been transferred from various locations). On the other hand, there has been considerable work in sharing relational data using both structured (i.e. *DHTs*) and unstructured (i.e., *Gnutella*-style) Peer-to-Peer overlays (e.g., [9, 10, 13]). Nevertheless, no special consideration has been given to multidimensional data supporting hierarchies. Specifically, 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, for *both* point and aggregate queries.

As a motivating scenario, let us consider a geographically dispersed business or application that produces immense amounts of data such as a multinational sales corporation or a data-collection facility that processes data from Internet routers. We argue for a completely decentralized approach, where users can perform *on-line* queries on the multiple dimensions, simple yet important mining operations (such as roll-up and drill-down on the defined hierarchies) and calculate aggregate views that return important data summaries. Such an application, besides eliminating the central storage and processing bottleneck and minimizing human coordination efforts, enables querying the data in real time, even if some of the resources are unavailable.

Let us assume that the company's database contains data organized along the location and product dimensions. 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 and repeat this for each dimension. Assuming the tuples are hashed according to the `city` and `category` attributes, there will be a node responsible for tuples containing the value *Athens*, one for *Patras*, etc, as well as nodes responsible for *Electronics*, *Household*, etc. This structure can be very effective when answering queries referring to the chosen levels of insertion (and even so, intersection of tuples will be necessary), whereas queries concerning other levels of the hierarchy demand global processing.

The solution of multiple insertion of each tuple by hashing every hierarchy value of each dimension is not viable: As the number of dimensions and levels increase, so does the redundancy of data and the storage sacrificed for this purpose. Furthermore, while point 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 "*What is the total revenue for 'Electronics' products anywhere sold*".

Contribution Summary Our work intends to describe a complete system that enables storing and querying hierarchical data in DHTs. The *Hierarchical Peer-to-Peer Indexing System (HiPPIS)* undertakes the task of storing and indexing bulk data in the form of a fact table (see Table 1) to multiple sites over the network. Peers initially index at a default (*pivot*) level combination. Inserted tuples are internally stored 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. Peers maintain local statistics which are used in order to decide if a re-indexing (to a different combination of hierarchy levels) is necessary, according to the current query trend. If the ratio of queries for `country, brand` exceeds a threshold (assuming the pivot level is `city, category`), data would be re-indexed according to that level combination so that most requests would be directly answered. Besides answering point queries at different levels of granularity, *HiPPIS* can answer group-by queries, such as "*Give me the sales registered for 'Greece' for ALL products*".

It has been widely observed that most Internet-scale applications, including P2P ones, exhibit highly skewed workloads (e.g., [4, 15], etc). *HiPPIS* indexes popular levels and uses indices to answer the less popular requests. It adapts to the incoming workload as a whole, without assuming any prior knowledge of the data or workload distributions and without any precomputations on the data. Our extensive simulations show that our system effectively adapts the level of granularity of the indexing according to user requests. *HiPPIS* 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 data distributions which are frequently documented in the majority of applications.

2. THE HIERARCHICAL PEER-TO-PEER INDEXING SYSTEM

HiPPIS 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. In this section, we first give the necessary notation to the problem and then describe in detail the *HiPPIS* protocol: Data indexing during insertion, query lookups and system adaptation to the incoming queries.

2.1 Necessary Notation

Our data spawn the d-dimensional space. Each dimension i is organized along $L_i + 1$ hierarchy levels: $H_{i0}, H_{i1}, \ldots, H_{iL_i}$, with H_{i0} being the special *ALL* $(*)$ value. We assume that our database comprises of fact table tuples of the form:
$\langle tupleID, D_{11} \ldots D_{1L_1}, \ldots, D_{d1} \ldots D_{dL_d}, fact_1, \ldots, fact_k \rangle$, where $D_{ij}, 1 \leq i \leq d$ and $1 \leq j \leq L_i$ is the value of the j^{th} level of the i^{th} dimension of this tuple and $fact_i, 0 \leq i \leq k$ are the numerical facts that correspond to it (we assume that the numeric values correspond to the more detailed level of the cube). Our goal is to efficiently insert and index these tuples so that we can answer queries of the form: $q = \langle q_1, q_2, \ldots, q_d \rangle$, where each query element q_i can be a value from a valid hierarchy level of the i^{th} dimension, including the $*$ value (dimensionality reduction): $q_i = D_{ix}, 0 \leq x \leq L_i$.

Table 1: Sample fact table

	Location			*Product*		Fact
TupleID	`Country`	`City`	`Zip`	`Category`	`Brand`	`Sales`
ID2	Greece	Athens	16674	Electronics	Apple	11,500
ID5	Greece	Athens	15341	Electronics	Sony	1,900
ID51	Greece	Athens	15341	Electronics	Philips	22,900
ID31	Greece	Athens	16732	Household	AEG	2,450
ID190	Greece	Patras	19712	Household	Unilever	1,990
ID324	Greece	Athens	17732	Electronics	Philips	2,450
ID501	Greece	Athens	17843	Electronics	Sony	12,000
ID712	Greece	Athens	17843	Electronics	Apple	32,000

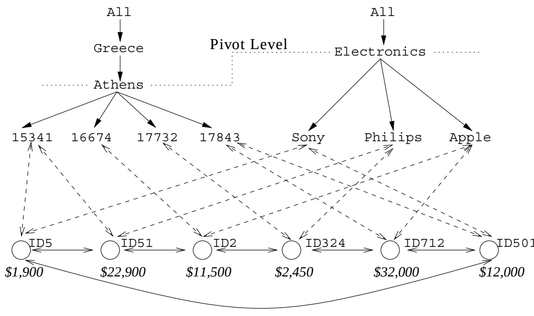

Figure 1: Sample data stored at node responsible for `Athens,Electronics` using the data of Table 1

2.2 Data Insertion

The insertion of a data tuple is performed as follows: Upon creation of the database, a combination of levels is globally selected. This is called *pivot level* $P = \langle p_1, p_2, \ldots, p_d \rangle$ where each pivot element p_i can be a valid hierarchy level of the i^{th} dimension (including the special $*$ value): $p_i = H_{iy}, 0 \leq y \leq L_i$. The ID of each tuple to be inserted is the hashed value of the tuple values corresponding to the pivot level. The DHT then assigns each tuple to the node with ID numerically closest to this value. 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. Nodes store multiple forests, one for each d-valued combination it is responsible for. As a consequence, each distinct value of the pivot level combination corresponds to a forest that reveals part of the hierarchy. Each forest consists of d rooted trees, one for each dimension. To see this pictorially, let us refer to the example depicted in Figure 1. Let us assume the data contained in Table 1 and the hierarchy described in Section 1 (without the last level of each dimension) with $\langle city, category \rangle$ 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'*||*'Electronics'* and forms a forest with two plain lists. 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. The trees of a forest are connected (in order to retrieve the corresponding facts) through the tuple IDs, depicted as a linked list in Figure 1.

2.3 Data Lookup and Indexing Mechanism

Queries concerning the pivot level are defined as *exact match* queries and can be answered within O(log$N$) forwarding steps.

Since we have included the $*$ as the top level of the hierarchy of each dimension, the pivot level combination may include $*$ in any of its d possible values. Therefore, assuming the query elements $q_i = D_{ix}$ and the respective pivot level elements $p_i = H_{iy}$, the query is an exact-match one if $x = y$ in the case it comprises of exact values or if $p_i = *$. Queries on any of the other level combinations cannot be answered unless flooded across the DHT. In order to amortize the cost of this operation and facilitate such requests, 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.

Soft-state indices give users the illusion that the queried values are actually hashed and retrieved in a fast manner. In reality, $O(\log N)$ steps are required to locate the indices which are then used to retrieve the multiple tuples required to compute the correct result set. The number of indices followed depends on the query and the pivot level: If the query attributes are of equal/smaller level than the respective pivot level elements, only a single pointer will exist. Otherwise multiple (the exact number depends on the data) pointers need be followed.

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. After the number of created indices per node has reached the 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).

Summarizing, we can categorize the queries that can be posed to the system into *exact match* ones and those that require forwarding to more than a single node. The latter can be answered either through flooding or using the soft-state indices. In any case, we assume that the computation of the correct fact values is done locally through addition of the returned numeric values.

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

The same procedure takes place when the query concerns a value that lies higher in the hierarchy than the pivot level. The query for ⟨*'Greece'*, $*$⟩ is routed to the node responsible, where no answer is available. Flooding is performed and the nodes that contain relevant tuples are discovered. Finally, the data satisfying the query are returned to the initiator and multiple indices are built. Both these cases are shown pictorially in Figure 2, where the black nodes are the ones that store the actual data, whereas the nodes holding pointers are depicted in gray.

2.4 Reindexing operation

HiPPIS is adaptive to the query distribution, supporting dynamic changes in the pivot level, without assuming any prior knowledge

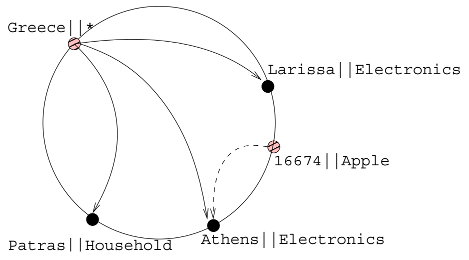

Figure 2: Example of soft-state index creation using our running example

Algorithm 1 HiPPIS Reindexing Algorithm

```
P: current pivot level combination
$popularity_{c_i}$: popularity of level combination $c_i$
$C_{local} : c_0 < c_1 < \ldots < c_{max}$ ranked level combinations according
to local popularity
**if** $popularity_{c_{max}}$ - $popularity_P$ > $threshold$ **then**
  flood(SendStatsMessage) and collect global statistics
  $C_{global} : c_0 < c_1 < \ldots < c_{max}$ ranked level combinations according to global popularity
  calculate $\Delta$
  $threshold \leftarrow k \cdot \Delta$
  **if** $popularity_{c_{max}}$-$popularity_P$ > $threshold$ **then**
    determine new pivot level $P_{new}$
    **if** $P_{new} = P$ **then**
      no action taken
    **else**
      flood(ReindexingMessage($P_{new}$))
      $P \leftarrow P_{new}$
      rehashing of tuples
    **end if**
  **end if**
**else**
  no action is taken
**end if**
```

and solely based on locally maintained statistics. By shifting to a different level combination we aim at increasing the ratio of exact-match queries, reducing floodings and boosting performance. The exact procedure is presented in Algorithm 1.

If the number of queries initiated by a node regarding level combinations different than the pivot level exceeds the number of queries for the pivot level by some *threshold*, this node considers the possibility of a new partitioning. Each node determines the *popularity* of each level combination ($\prod_{i=0}^{d} L_i$ exist) by measuring the number of queries it has locally initiated within the most recent 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.

If the percentage of the queries on the most popular level combination c_{max} is more than *threshold%* of the respective pivot level popularity, 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, after collecting the statistics from all nodes, redefines c_{max} and repeats the aforementioned procedure, enhanced with a strategy for the optimal pivot level selection, thoroughly described in the next section. In case of a new pivot level selection, reindexing is performed respectively by all nodes.

The initiating node floods a *Reindex* message, to force all nodes to change their pivot level. Each node that receives this message traverses its tuples, finds all the values of the level combination that will constitute the new reference point and rehashes them one by one, sending the 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 $N-1$ messages per node, grouping the tuples by recipient. After the node completes the procedure, it erases all its data and indices.

2.5 Locking

In order to ensure the correctness of the answers during the Reindexing process and to avoid simultaneous Reindexings 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 Reindexing. 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 is adequate to cover the time needed for the whole system to finish the Reindexing process and to reach a stable state. During this time, all queries are answered through flooding and no other operation is allowed.

2.6 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 of electronics in a new Greek zip code. An existing index for $q = \langle$*'Greece', 'Electronics'*$\rangle$ 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 $\prod_{i=0}^{d} L_i - 1$ lookups to identify the existence of all possible index combinations. Each node that holds a corresponding combination will update its value.

3. DISCUSSION – ENHANCEMENTS

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

Memory requirements A node running *HiPPIS* requires space for the combination statistics ($O(\prod_{i=0}^{d} L_i)$ modulo the window W) plus the storage required for the soft state indices. Each created index for a specific key holds, besides the key itself and its time of creation, the IDs of the nodes that hold the relative tuples. The number of different IDs is bound by the size of the network N. Hence, if K_{max} the maximum number of non-pivot keys held by a node, each node requires $O(NK_{max})$ bytes. Note here that in this calculation we have not included the amount of space reserved for the data at each node (which are usually not stored in main memory). Nodes can either physically store the data or pointers to their original locations. Whichever the case, the amount of space per forest depends on the pivot level (besides the data distribution of course): The higher the hierarchy levels in P, the larger the number of tuples that correspond to each tree.

Parameter Selection A careful choice of the *TTL, W,* K_{max} parameters plays an important role in the performance of the system. A small *TTL* degrades the success ratio of the search mechanism, invalidating indices unnecessarily. 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 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 20MB of memory while for 10K keys a node will need at most 200MB of memory (certainly affordable by most modern desktop PCs).

Reindexing cost and Load Balancing 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.

Furthermore, following our previous discussion, there is a clear trade-off between the amount of space per forest (via the choice of the pivot level) and the amount of processing corresponding to each node: The higher the pivot level, the more requests are handled through a single node. In this work, we do not deal with the load-balancing problem (caused either by uneven load or data distribution), as this is orthogonal and can be handled in a variety of well-documented ways in a DHT (e.g., [7, 14], etc). Nevertheless and for our target applications, we believe that an uneven data distribution is unlikely: The number of participating peers is not expected to be very high so that a uniform hashing of the existing combinations even at the highest levels will result in a uniform data distribution.

Minimize 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 define the $interval_n$ parameter for each node n. This parameter defines the minimum time-stretch 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: $interval_n \geq T_s$. In order to discourage consecutive reindexing attempts from the same node, this parameter is multiplicatively increased when the processing of global statistics (per p's indication) 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* is reset to the maximum between the current value and T_s, regardless of the outcome (whether *Reindexing* is decided or not).

Threshold Selection The *threshold* is of vital importance for the efficiency of the system, and should therefore be carefully determined in order to avoid unnecessary reindexing decisions. Frequent index reorganizations should be avoided, yet beneficial reindexing should not be prevented. The node having initiated the collection of global statistics calculates the *popularity* of each level combination, that is, the percentage of queries concerning that specific level combination, and ranks them according to this metric ($C : c_0 < c_1 < \ldots < c_{max}$). The overall query distribution should be taken into account as well, since it is possible that the system profits by choosing some less popular combination than c_{max}. This conclusion derives from the following observations:

- Remaining at the current pivot level spares the reindexing process as well as the invalidation of the so far created indices.
- A $*$ subsumes all levels of a dimension's hierarchy, since queries for other levels can be answered from the ALL data stored: For example for a pivot level $P = \langle H_{11}, * \rangle$ all queries $q = \langle D_{11}, q_2 \rangle$ can be answered (with q_2 being any possible value from any level of dimension 2).

The pivot choice is shaped as follows: The level combinations that lie within *threshold* from c_{max} are considered as pivot candidates. More formally, $\{\forall c_i \in C, 0 \leq i \leq max \mid popularity_{c_{max}}\text{-}popularity_{c_i} < threshold \implies c_i \in C_{cand}\}$, where C_{cand} is the set of candidate level combinations.

The threshold value is proportional to the Mean Difference (Δ) of the popularity values, in particular *threshold*$=k\Delta$, $k \leq 1$. The parameter Δ, which equals the average absolute difference of two independent values, is chosen as a measure of statistical dispersion: $\Delta = \frac{1}{max \cdot (max+1)} \sum_{i=0}^{max} \sum_{j=0}^{max} |c_i - c_j|$. Among all $c_i \in C_{cand}$, the new pivot level is chosen through the following strategy:

1. If the current level $P \in C_{cand}$, the system takes no action.
2. Otherwise, from all $c \in C_{cand}$ containing $*$ in one or more dimensions, we consider only combinations that include up to $\lceil \frac{D}{2} \rceil$ ones and exclude the rest. This is in order to ensure that no excessive local processing will be needed for incoming queries. For each of the remaining combinations containing $*$, we recalculate their *popularity* adding the *popularity* of other candidate combinations that are subsumed by it. For instance, let us assume $\langle \texttt{Country}, \texttt{Brand} \rangle$, $\langle \texttt{City}, * \rangle$ and $\langle *, \texttt{Brand} \rangle$ are the candidate pivot levels, with *popularities* of 10%, 20% and 15% respectively. Comparing the two levels with $*$, $\langle *, \texttt{Brand} \rangle$ can answer $\langle \texttt{Country}, \texttt{Brand} \rangle$ queries, thus its *popularity* rises to 25%, and is therefore chosen over $\langle \texttt{City}, * \rangle$ as the new pivot combination.
3. If none of the above holds, the system shifts to the level combination with the highest popularity.

4. EXPERIMENTAL RESULTS

We now present a comprehensive simulation-based evaluation of *HiPPIS*. Our performance results are based on a heavily modified version of the FreePastry simulator [6], although any DHT implementation could be used as a substrate. By default, we assume a network size of 256 nodes. Results were collected with up to 1K nodes with little qualitative difference.

In our simulations, we use synthetically generated data. Each dimension is represented as a tree with each value having a single parent and at most *mul* children in the next level. The tuples of the fact table to be stored are created from combinations of the leaf values of each dimension tree plus a randomly generated numerical fact (`sales`). By default, our data comprise of 22k tuples, organized in a 3-dimensional, 3-level hierarchy. The number of distinct values of the top level is $|H_1| = 20$ and *mul*=2. The initial pivot level is, by default, $P = \langle H_{12}, H_{22}, H_{32} \rangle$.

Table 2: Percentage of queries directed towards the 27 level combinations of our initial simulation

θ	% most popular	% least popular	#combs
0	3.7	3.7	27
0.5	11.1	2.1	27
1.5	44.8	0.3	27
2.5	74.9	0.01	27
3.5	88.8	0.01	12

For our query workloads, we consider a two-stage approach: we first identify the probability of querying each level combination according to the *levelDist* distribution; a query is then chosen from that combination following the *valueDist* distribution. In our experiments, we order the different combinations lexicographically, i.e., combination $\langle H_{13}, H_{21}, H_{31} \rangle > \langle H_{11}, H_{23}, H_{33} \rangle$ and we use the Zipfian distribution for *levelDist* where #queries for combination $i \sim 1/i^{\theta}$. We vary the value of θ as well as the direction of the ordering to control the amount and target of skew of our workloads. For *valueDist* we use the 80/20 rule by default, unless stated otherwise. Table 2 gives an overview of the workloads we frequently use in this section. We document the percentage of queries directed towards the most and least popular combination, as well as the number of combinations that receive at least one query (out of the total 27 existing).

Our default workload comprises of 35k queries which arrive at an average rate of 1 $\frac{query}{sec}$. For our experiments, W is set to 500sec and *TTL* is given a practically infinite value (indices never expire).

In this section, we intend to demonstrate the performance and adaptability of *HiPPIS* under various conditions. To that direction, we measure the percentage of queries which are answered directly, i.e., without flooding (*precision*) and we trace the average number of exchanged messages per query, as well as the overhead of control messages needed by our protocol. We compare *HiPPIS* with the naive protocol (referred to as *Naive*), where precision equals the ratio of queries on the initial pivot level, and a special case of *HiPPIS*, where only the indices are utilized and no reindexing occurs (referred to as *HiPPIS(N/R)* or plain *N/R*).

Performance with varying query distributions
In this initial set of simulations, we vary the θ parameter for levelDist as well as the direction of skew, using the default parameters otherwise.

In Figure 3, data are skewed towards the "lowest" level ($\langle H_{13}, H_{23}, H_{33} \rangle$). As θ increases, the workload becomes more skewed and the performance of *HiPPIS* improves: Reindexing is performed sooner, as the ratio of popular queries increases, resulting in a rise of the exact matches due to the chosen combination. Moreover indices contribute more to system's precision, since the number of distinct queries for non pivot level tuples decreases. For uniform distributions, the number of distinct queries does not allow our method to capitalize on the indexing scheme.

Figure 4 shows results where our workload favors $\langle H_{11}, H_{21}, H_{31} \rangle$. Again, we notice a similar trend in performance as the values for θ increase. Nevertheless, *HiPPIS* is slightly more effective than before, with its difference from *N/R* increasing as θ increases. This is due to the limited number of distinct values of $\langle H_{11}, H_{21}, H_{31} \rangle$, which facilitates the maintenance of indices, favoring *N/R* against *HiPPIS*. The latter erases all created indices during the reindexing process. However, *HiPPIS* naturally outperforms its competition in the steady state, as it can increase its performance with time.

Figures 5 and 6 depict the number of messages exchanged per query in the system, indicating a measure of bandwidth consumption. Messages regarding query resolution (including requests as well as responses) and control messages, which include those needed

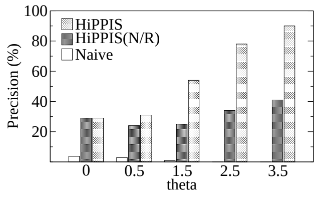

Figure 3: Precision for varying levels of skew (most popular combination is $\langle H_{13}, H_{23}, H_{33} \rangle$)

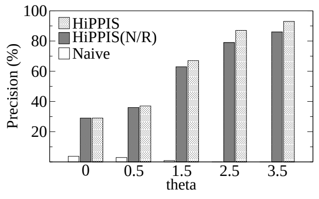

Figure 4: Precision for varying levels of skew (most popular combination is $\langle H_{11}, H_{21}, H_{31} \rangle$)

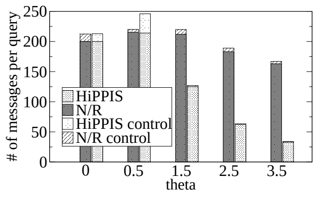

Figure 5: Average number of messages required to answer a query (skew towards $\langle H_{13}, H_{23}, H_{33} \rangle$)

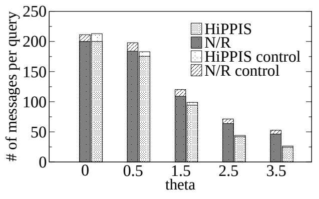

Figure 6: Average number of messages required to answer a query (skew towards $\langle H_{11}, H_{21}, H_{31} \rangle$)

to build indices, collect statistics, notify of a Reindexing and reinsert tuples, are presented separately. Qualitatively, the total number of messages per query is inversely proportional to the system's precision. As observed in all experiments, the overhead of control messages is small and outweighed by the gains in precision (less than 8% over the total number of messages). This is due to the fact that *HiPPIS* carries out the minimum required reindexing rounds, which translates to one reindexing process per direction of skew. We also notice that the overhead of the control messages decreases as the workload becomes more skewed (almost negligible for $\theta > 1.5$). This can be explained by the fact that *HiPPIS* becomes more confident in the level of reindexing it chooses as θ increases. Finally, we calculated that one *HiPPIS* reindexing equals less than 100 queries answered by flooding (in the number of messages needed for these two operations). Given that we only perform the necessary reindexing and less than 5 *SendStats* requests are produced per simulation, our method makes near-optimal use of its bandwidth-intensive operations.

Effect of recurring queries

We plan to identify the effectiveness of our system's indexing mechanism under workloads with varying ratio of recurring queries. We believe that this will be the case for the majority of workloads for our target applications, with users temporarily interested in a small number (or set) of (aggregate) data. We consider two different scenarios for the distribution of the duplicate queries. In the first case, for two levels of skew ($\theta = \{1.0, 3.0\}$), we vary the percentage of unique queries by increasing duplicate ones, following the same distribution. In the second case, for three different values of θ for *levelDist*, namely 0.0, 1.0 and 3.0, the *valueDist* distribution varies from uniform to 99/1, creating within each level combination the same amount of skew. The documented precision for both cases is depicted in Figures 7 and 8 respectively.

In both cases we notice that, as more queries recur in the workload, the performance increases. In the first case, recurring queries follow the *levelDist* distribution, meaning that duplicate queries primarily concern the most popular level combinations. Since *HiPPIS* reindexes to the most beneficial level, it naturally increases its exact answers compared to *N/R*. Nevertheless, the gains decrease as replication increases, unlike *N/R*, which shows almost linear improvement. This is due to the fact that there exists less "room" for *HiPPIS* to take advantage of the indexed queries since it has already moved to the best combination which takes up significantly more requests. As θ increases, we normally expect an increase in performance.

In the second case (Figure 8), as the bias of queried values within a level combination increases, we observe that our system benefits more and more from the soft-state indices, exploited by duplicate queries. Small replication in queries results in significant differences in precision for the various θ values, as for these kinds of workloads precision is dominated by exact matches. Nevertheless, all three distributions seem to converge to very high precision levels as the ratio of duplicate queries augments.

The effect of aggregate queries

In this experiment we intend to examine how our system behaves when we inject an increasing number of aggregate queries (from zero up to 50% of the total number of queries). We assume two different distributions as to how $*$ are distributed in those queries: In scenario 1 (S1), a $*$ appears in the three dimensions with probabilities (0.73, 0.18, 0.09) respectively (i.e., we heavily favor an aggregate view on the first dimension). In the second one (S2), each dimension is given an equal probability. The workload skew is set to $\theta = 2.0$. Results are presented in Figure 9.

We notice that both methods increase in performance as the percentage of aggregate queries increases in both distributions. This is due to the fact that the different combinations that these queries can produce are less than those of point queries. Therefore, increasing their ratio enables the indexing mechanism to store and answer a larger amount or requests without flooding. This is evident from *N/R*'s precision increase. In the latter case, the reindexing process invalidates all created indices, thus mitigating the beneficial effect we described before. Furthermore, the skew in the star distribution affects, although slightly, the system's precision: greater skew leads to greater probability of duplicate queries, favoring the indexing mechanism. Since *HiPPIS* is less dependent on this mechanism, the increase in precision is less noticeable than in the case of *HiPPIS(N/R)*.

Performance in dynamic environments

In the next experiment, we measure the performance and adaptivity of *HiPPIS* 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 ($t_c = 31000sec$): From a skewed workload towards $\langle H_{13}, H_{23}, H_{33} \rangle$ we shift to a skewed load towards $\langle H_{11}, H_{21}, H_{31} \rangle$. We show the results for two levels of skew in Figure 10.

Our results show that, in all cases, *HiPPIS* quickly increases its precision due to the combination of automatic reindexing and soft-state indices. Floodings increase after t_c, since neither the pivot combination nor the so far created indices can efficiently serve queries with different direction of skew (hence the decline in precision). However, it quickly manages to recover and regain its performance characteristics, as a Reindexing is performed and new indices are built. The rate at which these events occur depends on the amount of skew: In the $\theta = 3.0$ case, 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 the less skewed distribution ($\theta = 1.0$), 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 30% in the $\theta = 3.0$ case to about 40% in the worst-case. Once again, we observe that *HiPPIS* performs best in skewed

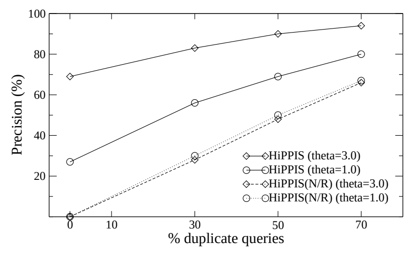

Figure 7: Precision over variable percentage of duplicate queries

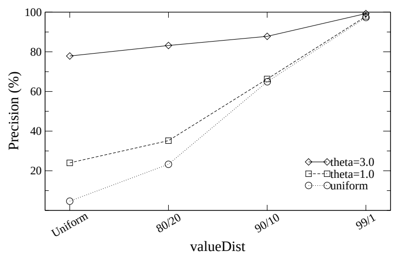

Figure 8: Precision for various distributions of *valueDist*

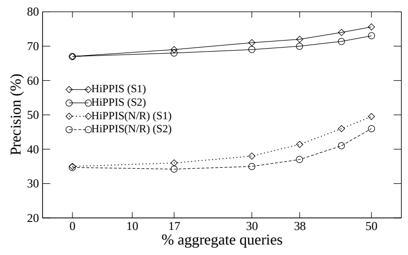

Figure 9: Precision over variable number and skew of aggregate queries

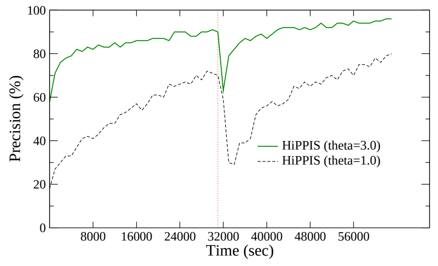

Figure 10: Precision over time for various workloads when a sudden shift in skew occurs in $t_c = 31000$sec

workloads, but its performance in the steady state is invariably high, regardless the workload.

Testing *HiPPIS* with different number of dimensions
In this set of simulations we plan to investigate the possible performance variations caused by datasets with variable dimensionality. We assume that each dimension is described by a 3-level hierarchy. By varying the *mul* parameter we try to create equal-size data and query-sets with the same θ value. Figure 11 depicts the results for 2 to 8 dimensions for two different values of θ: 1.0 and 3.0.

As the number of dimensions increases linearly, the number of combinations increases exponentially. This radically the popular levels' request rates, especially for less skewed workloads, thus reducing the number of exact match queries for the level combination *HiPPIS* chooses. This becomes obvious when θ increases: the slope becomes more parallel to the dimension axis. *HiPPIS* ranges between 40% and 70% in the low skew case while for bigger skew this becomes 80% to 93%. A pure indexing scheme solely relies on the duplicate queries and (to a lesser extent) to the exact match queries of the random pivot level, thus producing poor results.

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, where its value varies from 0 to 4000 for the standard workload, for two levels of skew, $\theta = \{1.0, 3.0\}$, directed towards $\langle H_{13}, H_{23}, H_{33}\rangle$. Results are depicted in Figure 12.

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. For the less skewed workload, the I_{thres} value is larger since more distinct values are requested, thus *HiPPIS* relies more on indices to improve its performance. In the more skewed workloads *HiPPIS* tracks the optimal pivot level sooner 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.

Given this analysis, a value of $I_{max} = 2$k indices is deemed adequate, ensuring that the majority of the created indices will remain in the system. This heavily favors the *N/R* method, since *HiPPIS* discards all indices each time a reindexing occurs. With this value, used in all this experimental section, each node needs to dedicate at most 100KB of main memory for the soft-state indices.

APB Benchmark Datasets
Finally, we test the performance of *HiPPIS* using some more realistic data and query sets generated by the APB-1 benchmark [3]. Running the APB-1 data generator with the density parameter set to 0.1, we produced a 4-dimensional dataset with cardinalities 9000, 900, 9 and 24 and two measure attributes. Each dimension comprises of a hierarchy of 7, 4, 2 and 3 levels respectively. The produced data-cube contains 1,239,300 tuples, while the produced workload comprises of 25k queries (queries with $*$ were filtered out from the original query workload) with 1% replication ratio. Results are depicted in Figure 13.

We clearly notice that *HiPPIS* exhibits very high performance, reaching over 90% of precision in its steady state after about 4000 queries. This experiment shows that for more realistic scenarios, even with more dimensions *HiPPIS* quickly adapts and serves the vast majority of user requests without flooding. Using plain indices reduces precision by over 20%, while there is a substantial delay in reaching the steady state (twice as many queries needed).

5. RELATED WORK

There has been significant work in the area of databases over P2P networks. PIER [9] proposes a distributed architecture for relational databases supporting operators such as join and aggregation of stored tuples. A DHT-based overlay is used for query routing. The Chatty Web [1] 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 GroupPeer [10], 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 [13] also features relational data sharing without schema knowledge. Query matching and rewriting is based on keywords provided by the users. GridVine [2], and pSearch [17] are based on structured P2P overlays. GridVine hashes and indexes RDF data and schemas, and pSearch represents documents as well as queries as semantic

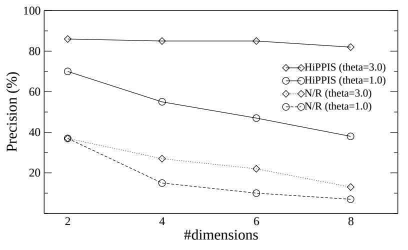

Figure 11: Precision over variable dimensionality datasets

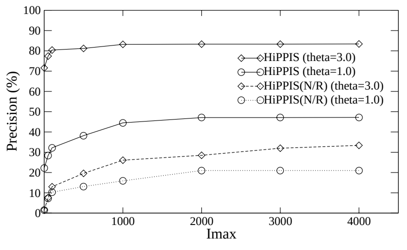

Figure 12: Precision over variable *Imax*

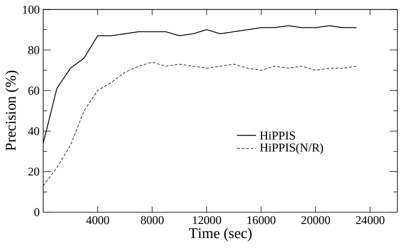

Figure 13: Precision of *HiPPIS* for the one APB query workload

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 multidimensional datasets.

An interesting method for representing hierarchical data is presented in [11]. 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., [12, 18]). However, only few support both aggregate queries and hierarchies. In [16], 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 [5]. In this work, one concept hierarchy is stored per dimension and an ID is assigned 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.

6. CONCLUSIONS

In this paper we described *HiPPIS*, a distributed system that stores and indexes data organized in hierarchical dimensions for DHT overlays. *HiPPIS*, assuming no prior knowledge of the workload nor any precomputations, enables on-line queries on the different dimensions and granularities of the data. Our system dynamically adjusts to the workload by reindexing the stored data according to the incoming queries. With the combination of adaptive indexing and soft-state pointers, *HiPPIS* manages to avoid the network-disastrous flooding in most cases, while enabling both real-time querying and update capabilities on voluminous data.

Our simulations, using a variety of workloads and data distributions, show good performance and bandwidth efficiency. *HiPPIS* is especially effective with skewed workloads, achieving very high precision and fast adaptation to dynamic changes in the direction of skew. Even with low query replication ratios, *HiPPIS* manages to answer the majority of queries within $O(\log N)$ steps, by detecting the most popular level combination and shifting to it. Moreover, a significant increase in the number of aggregate queries does not degrade system performance, but on the contrary, leads to higher precision.

Acknowledgments

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

7. REFERENCES

[1] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth. The Chatty Web: Emergent Semantics Through Gossiping. In *WWW Conference*, 2003.

[2] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. V. Pelt. Gridvine:Building internet-scale semantic overlay networks. In *International Semantic Web Conference*, 2004.

[3] *OLAP Council APB-1 OLAP Benchmark.* http://www.olapcouncil.org/research/resrchly.htm.

[4] M. Cha, H. Kwak, P. Rodriguez, Y. Ahn, and S. Moon. 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.

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

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

[7] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *ICDCS*, 2004.

[8] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

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

[10] V. Kantere, D. Tsoumakos, and T. Sellis. GrouPeer: Dynamic Clustering of P2P Databases. *Information Systems*, 2008.

[11] G. Koloniari and E. Pitoura. Content-based routing of path quieries in peer-to-peer systems. In *EDBT*, 2004.

[12] L. Lakshmanan, J. Pei, and Y. Zhao. QC-trees: An Efficient Summary Structure for Semantic OLAP. In *SIGMOD*, 2003.

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

[14] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. In *EDBT*, 2006.

[15] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

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

[17] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM*, 2003.

[18] W. Wang, H. Lu, J. Feng, and J. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *ICDE*, 2002.