J. Parallel Distrib. Comput. 71 (2011) 424–437

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Online querying of *d*-dimensional hierarchies☆

Katerina Doka*, Dimitrios Tsoumakos, Nectarios Koziris

Computing Systems Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens, Greece

ARTICLE INFO

Article history:
Received 4 February 2010
Received in revised form
3 October 2010
Accepted 12 October 2010
Available online 16 October 2010

Keywords:
Peer-to-Peer
Distributed hash tables
Concept hierarchies
Data warehousing

ABSTRACT

In this paper we describe a distributed system designed to efficiently store, query and update multidimensional data organized into concept hierarchies and dispersed over a network. Our system employs an adaptive scheme 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. Updates are performed on-line, with minimal communication overhead, depending on the level of consistency needed. Extensive experimental evaluation shows that, on top of the advantages that a distributed storage offers, our method answers the vast majority of incoming queries, both point and aggregate ones, without flooding the network and without causing significant storage or load imbalance. Our scheme proves to be especially efficient in cases of skewed workloads, even when these change dynamically with time. At the same time, it manages to preserve the hierarchical nature of data. To the best of our knowledge, this is the first attempt towards the support of concept hierarchies in DHTs.

1. Introduction

Nowadays data are produced at an astounding rate [17]: Market globalization, business process automation, the growing use of sensors and other data-producing devices, along with the increasing affordability of hardware have contributed to this continuous trend. On the other hand, information environments themselves are distributed. Business groups consist of multiple companies around the world, which, although operating autonomously, still need to provide the headquarters with summarized information for decision making. Indeed, large companies as well as scientific organizations heavily rely on data analysis in order to identify behavioral patterns and discover interesting trends/associations.

Data warehousing has become a vital component of every organization, as it contributes to business-oriented decision-making. A data warehouse is a central repository that hosts immense volumes of historical data from multiple sources and provides tools for their aggregation and management at different levels of granularity. The basic abstraction in data-warehousing is *data cubes*, multidimensional arrays in the form of which data is usually viewed. 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.). They allow for efficient summarization of data by reducing the dimensions and producing aggregate views of the 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. Fig. 1 shows a simple hierarchy for the `location` dimension, where `Address` < `ZipNo` < `City` < `Country` and one for `product`, where `Product` < `Brand` < `Category`. Concept hierarchies are important because they allow the structuring of information into categories, thus enabling its search and reuse. They 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.

Yet, data warehouses present a strictly centralized and off-line approach in terms of data location and processing: Views are usually calculated on a daily or weekly basis after the operational data have been transferred from various locations and, surprisingly, this practice is still considered to be state-of-the-art. The ever-growing volumes of data along with the requirement for constant data analysis in order to immediately detect real-time changes in trends imply the need for an always-on, real-time data access and support system for concurrent processing of queries. These challenges have given birth to the idea of creating distributed data-warehouse-like systems deployed on a shared-nothing, commodity hardware architecture, giving the advantage of scalability and robustness at low cost.

☆ This is an extended version of the work presented in WIDM'08.

* Corresponding author.
E-mail addresses: katerina@cslab.ece.ntua.gr (K. Doka), dtsouma@cslab.ece.ntua.gr (D. Tsoumakos), nkoziris@cslab.ece.ntua.gr (N. Koziris).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.10.005

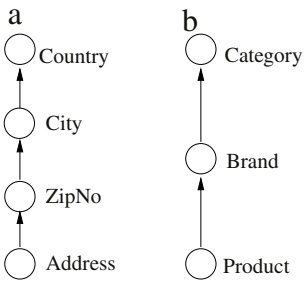

Fig. 1. A concept hierarchy for dimension (a) `Location` (b) `Product`.

Some works in the field propose distributed warehousing systems (e.g., [15,4,2]), but the warehouse and its aggregation, update and querying functionality remain centralized. 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, combining the advantages of a distributed and resilient solution with the performance of storing large volumes of data in database systems. Peer Database systems (e.g., [16,14,24]) represent a new trend in which peers maintain parts of a central database and communicate with each other in a distributed, fault-tolerant manner. Nevertheless, no special consideration has been given to multidimensional data supporting hierarchies and, until now, Peer Databases that rely on DHT functionality are unable to directly support queries on multiple dimension hierarchies.

In this paper 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. To that end, we propose the *Hierarchical Peer-to-Peer Indexing System (HiPPIS)*, a DHT-based system that enables efficient storage and querying over multiple dimensions characterized by specific hierarchies. Thus the system benefits from the inherent characteristics of the Peer-to-Peer architecture, such as scalability, fault tolerance and availability relying solely on commodity nodes. *HiPPIS* 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, *HiPPIS* 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.

1.1. Motivation and problem description

As a motivating scenario, let us consider a geographically dispersed business or application that produces immense amounts of data, e.g., 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, 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 (see Fig. 1). 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 *Milan*, 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 hierarchy levels demand global processing.

The solution of multiple insertions 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 sold anywhere*".

1.2. Sketch of HiPPIS and contribution summary

Our work intends to describe a complete system that enables storing and querying hierarchical data in DHTs. *HiPPIS* undertakes the task of storing and indexing bulk data in the form of a fact table (e.g., 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. For instance, 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 granularities, *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., [8,26], etc.). *HiPPIS* indexes popular levels and uses soft-state 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. The contribution of our work can be summarized in the following:

- It addresses the problem of hierarchical data search in DHT systems. 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. Our technique, taking into account user preferences and sensing potential overall tendencies, allows reorganization of the indexing structure in favor of resolving queries for the most popular data. It also manages to preserve the useful hierarchy-specific information that hashing destroys. Either 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. By using a tree-like data structure to store data and maintain indices to related keys, our system is able to respond to more complex, hierarchy-based queries.
- It allows for on-line updates, unlike the conventional update technique in data warehousing, which dictates an offline update application on a daily or weekly basis. The communication overhead depends on the level of consistency needed by the application.
- It presents a thorough experimental section where we clearly identify the advantages of our proposed system in a variety

of workloads (variable levels of skew, dynamic changes, etc.), datasets and update setups. We also register the induced data and load distributions across the nodes of the overlay. *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, without inducing significant load or storage imbalance among the network nodes. Moreover, even under high update rates, the freshness of the query responses remains acceptable.

The rest of the paper is organized as follows: The following section goes through the related literature. Section 3 describes our approach in a more detailed manner. In Section 4 we discuss its requirements together with protocol enhancements. Section 5 presents our evaluation. Finally, Section 6 concludes the paper.

2. Related work

In [12], the data cube operator is introduced. The data cube generalizes many useful operators, namely aggregation, group by, roll-ups, drill-downs, histograms and cross-tabs. It consists of several independent attributes grouped into dimensions and some dependent attributes which are called measurements. Since the number of the possible views increases exponentially with the number of dimensions, materialization is commonly used in order to speed-up query processing. This approach fails in a fully dynamic environment where the queries are not known in advance or when the number of possible queries becomes very large.

Several indexing schemes have been presented for storing data cubes (e.g., [18,33]). However, only few support both aggregate queries and hierarchies. In [27], 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. Another approach is the DC-Tree [10], a fully dynamic index structure for data warehouses modeled as data cubes. It exploits concept hierarchies across the dimensions of a data cube. In this work, the attributes of a dimension are partially ordered with respect to the valid hierarchy schema for each dimension. The DC-tree stores one concept hierarchy per dimension and assigns an ID to every attribute value of a data record that is inserted. In [20,21], the authors present a novel lattice traversal scheme, in order to construct complete data cubes with arbitrary hierarchies. 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.

Recently, effort has been made to exploit parallel processing techniques for data analysis by integrating query constructs from the database community into MapReduce-like software. The Pig project at Yahoo [23], the SCOPE project at Microsoft [7] and the open-source Hive project [31] mainly focus on language issues, addressing the creation of SQL interfaces on top of Hadoop [13]. HadoopDB [3] proposes a system-level hybrid approach, where MapReduce and parallel DBMSs are combined. SQL queries are translated with the use of Hive into MapReduce jobs, which are eventually executed on a single node, running a DBMS. Some vendors (e.g., Vertica, AsterData, etc.) have already presented shared-nothing parallel databases with regards to Cloud computing. However, such technologies are inherently batch-oriented, as they can provide large amount of processing power, but do not guarantee real-time responses. Parallel database solutions, on the other hand, exhibit reduced robustness in failures and do not operate in heterogeneous environments.

The notion of a distributed Data Warehouse has been used in the past, although a more accurate characterization for the proposed systems would be 'cooperative', rather than 'distributed'. In [15], the authors consider a number of DWs and peers, forming an unstructured P2P overlay for caching OLAP views. Views are divided in chunks and peers retrieve cached chunks from the network and the DW if needed. In [4], the authors define the distributed data warehouse as a structure that consists of multiple local data warehouses adjacent to data collection points and a coordinator site, responsible for interacting with each of the local sites and for correlating and aggregating query results. A similar approach is described in [9], where a two-layer architecture consisting of multiple local data warehouses and a global one is proposed. All these approaches perform some hybrid query processing model by allowing requests to route to different cites. [2] describes a P2P platform for managing distributed repositories of XML and semantic Web data, where various data processing building blocks are integrated as Web services. Yet, none of the above works distributes the warehouse structure itself, keeping the processing cites centralized.

The sharing of relational data using both structured and unstructured P2P overlays is addressed in a number of papers. PIER [14] 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. Combined with a Gnutella-like overlay in [19], PIER has also been used for common file-sharing. The unstructured overlay is used for locating popular items while the PIER search engine favors the publishing and discovery of rare items. PeerDB [24] and GrouPeer [16] feature relational data sharing without schema knowledge, utilizing query rewriting. GridVine [1] hashes and indexes RDF data and schemas, and pSearch [29] represents documents as well as queries as semantic vectors. A recent work stressing the need for P2P OLAP is [32], which mainly focuses on answering OLAP queries over a network of data warehouses that do not share the same schema. 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.

3. The hierarchical peer-to-peer indexing system

3.1. 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 jth level of the ith 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 ith dimension, including the $*$ value (dimensionality reduction): $q_i = D_{ix}$, $0 \leq x \leq L_i$.

3.2. Data insertion

The insertion of a data tuple (or a pointer to the real location of it) is performed as follows: Upon creation of the database, a combination of levels is globally selected. This is called *pivot* $P = \langle p_1, p_2, \ldots, p_d \rangle$ where each pivot element p_i can be a valid hierarchy level of the ith 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

Table 1
Sample fact table.

TupleID	Location			Product		Fact
	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
ID55	Greece	Larissa	20100	Electronics	Sony	12,100
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

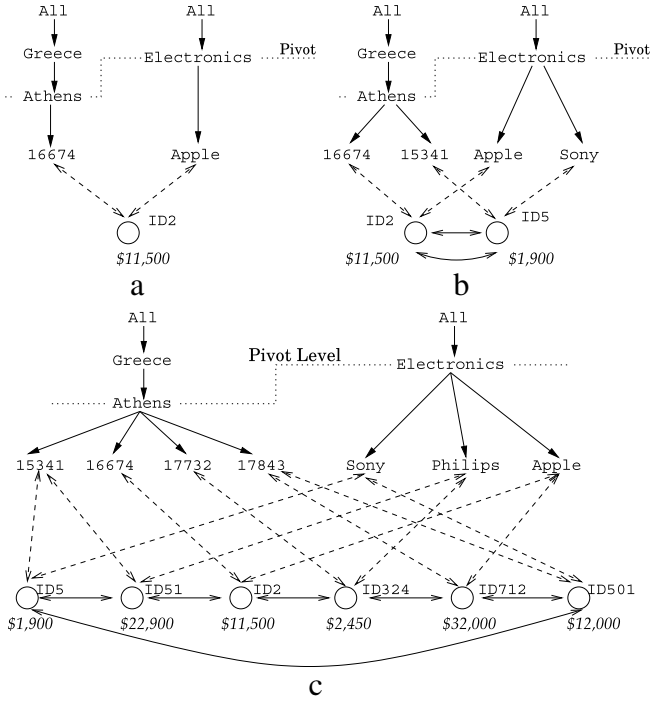

Fig. 2. The forest structure at node responsible for `Athens, Electronics` after the insertion of (a) the first tuple, (b) the second tuple and (c) all tuples of Table 1.

closest to this value. For tuples inserted at a later stage, nodes can be informed of P 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 Fig. 2. Let us assume the data contained in Table 1 and the hierarchy described in Section 1 (without the last level of each dimension) with ⟨*city*, *category*⟩ to be 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 (Fig. 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 (Fig. 2(b) and (c)). The trees of a forest are connected (in order to retrieve the corresponding facts) through the tuple IDs, depicted as a linked list in Fig. 2.

3.3. Data lookup and indexing mechanism

Queries concerning P 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, P 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. So, essentially, we term indices the pointers from a node that should hold the answer to a query, had the pivot been the queried level combination, to the node or nodes that actually store the answer.

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 P: 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. Apparently, in cases of very large datasets and uniform query distributions the index size can grow large. 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). Thus, the system tends to preserve the most "useful" indices, namely the ones that refer to the most frequently used data items. The HiPPIS lookup and indexing algorithm is presented in Algorithm 1.

As an example, let us assume the same hierarchy as before, with ⟨`city, category`⟩ as P. 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*'. Thus, in case of another query referring to the same value, the time and bandwidth consuming

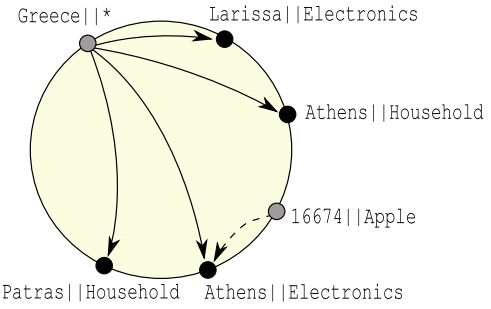

Fig. 3. Example of soft-state index creation using our example.

flooding is avoided and the response can be provided quickly and efficiently, within logN+C hops.

Algorithm 1 HiPPIS Lookup and Indexing Algorithm

$q = \langle q_1, q_2, \ldots, q_d \rangle$: the query to be resolved
$P = \langle p_1, p_2, \ldots, p_d \rangle$: the pivot level combination
r: remote node
$K_{r,exact}$, $K_{r,ind}$: set of keys held and indexed by remote node respectively
if $\exists P_s \subseteq P : \forall p_i \in P_s, p_i = * \wedge q_i \neq *$ and the rest of the attributes in $(P - P_s)$ are of the same level **then**
  $ID_q \leftarrow$ hash(q) where q_i is replaced by $*$
  DHT_route(LookupMessage) to r responsible for ID_q
  local processing by r and possible answers returned
else
  $ID_q \leftarrow hash(q)$,
  DHT_route(LookupMessage(ID_q))
  local processing by r
  if $ID_q \notin K_{r,exact}$ **then**
    if $ID_q \notin K_{r,ind}$ **then**
      flood(q), local processing by each r
      answers returned by set of nodes R
      DHT_route(IndexMessage($ID_q \rightarrow R$))
      Receiver nodes add ID_q to $K_{r,ind}$
    else
      local processing, tuples returned
    end if
  else
    tuples returned
  end if
end if

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 Fig. 3, where the black nodes are the ones that store the actual data, whereas the nodes holding pointers are depicted in gray.

3.4. Reindexing operation

In a data warehouse, the distribution of data or queries may vary over time. Thus, it is possible that the choice of P, which is done once at the beginning, does not favor performance. *HiPPIS* is adaptive to the query distribution, supporting dynamic changes in the pivot level, without assuming any prior knowledge, being solely based on locally maintained statistics. By shifting to a different level combination we aim at increasing the ratio of exact-match queries, reducing flooding and boosting performance. The exact procedure is presented in Algorithm 2.

Algorithm 2 HiPPIS Reindexing Algorithm

P: current pivot level combination
$popularity_{c_i}$: popularity of level combination c_i
$C_{local} : c_0 < c_1 < \cdots < c_{\max}$ ranked level combinations according to local popularity
if $popularity_{c_{\max}}$ - $popularity_P > threshold$ **then**
  flood(SendStatsMessage) and collect global statistics
  $C_{global} : c_0 < c_1 < \cdots < c_{\max}$ ranked level combinations according to global popularity
  calculate *threshold*
  if $popularity_{c_{\max}}$-$popularity_P > threshold$ **then**
    determine new pivot level P_{new}
    if $P_{new} \neq P$ **then**
      flood(ReindexingMessage(P_{new}))
      $P \leftarrow P_{new}$, rehashing of tuples
    end if
  end if
end if

If the number of queries initiated by a node regarding level combinations different than P exceeds the number of queries for P 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 popularity, the node is positive to the potential of adopting another pivot. If this is the case, 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 P 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 selection, thoroughly described in the next section. In the case that a new P is selected, reindexing is performed respectively by all nodes.

It should be noted here that the first phase of the reindexing process is not decisive for the selection of the new potential pivot level; it is rather used as an indication of an imbalance that should be further investigated. Thus, we assume that nodes act altruistically, not only by reporting their true statistics, but also in the sense that they may trigger a change of pivot that may not reflect their personal preferences.

The initiating node floods a *Reindex* message to force all nodes to change their pivot. 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.

Back to our example, if the node '*Athens*' ‖ '*Electronics*' receives a *Reindex* message for ⟨`city`, `brand`⟩, it runs through its tuples and discovers that the values corresponding to that level combination are '*Athens*' ‖ '*Sony*', '*Athens*' ‖ '*Philips*' and '*Athens*' ‖ '*Apple*'. The values are hashed and the corresponding nodes are now responsible for the tuples containing these values (Fig. 4).

3.5. Locking

In order to ensure the correctness of the answers during the reindexing process and to avoid simultaneous reindexings by

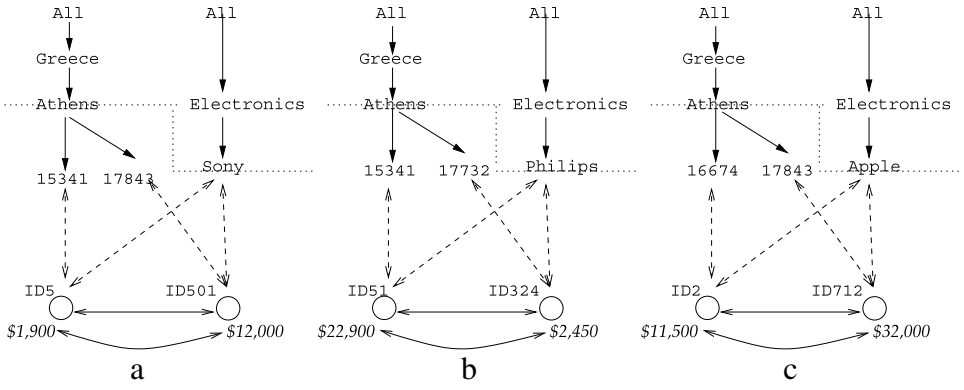

Fig. 4. The produced forest structures after reindexing.

multiple nodes, we introduce a locking mechanism. After a node decides to perform reindexing according to the global statistics, it first sends a *Lock* message to all system nodes and then proceeds to it. 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 network size), which we assume is adequate to cover the time needed for the whole system to finish reindexing and to reach a stable state. During this time, locked nodes continue answering queries through flooding. Therefore the system constantly remains online.

To cope with the issue of possible concurrent locks, we adopt a simple resolution mechanism: Since each *Lock* message, upon creation by the initiator, is identified by a (local) timestamp and the initiator's ID, nodes receiving more than one *Lock* messages within a small time frame may assume as valid the one with the earliest timestamp (or the one coming from the lowest ID) and accept *Reindex* messages only by its initiator. If the newly received *Lock* is not valid, the node stops forwarding it. An already LOCKED node is not allowed to initiate another locking.

We should note here that since each node collects global statistics before a new pivot decision, it is impossible that two non-malicious nodes come to a different decision. This would only be possible (with low probability) if sampling were used for global statistics collection. Even so, the locking mechanism makes sure that only one node at a time can instruct reindexing.

3.6. 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 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 city. An existing index for $q = \langle$'Greece', 'Electronics'$\rangle$ should now include the ID of the node responsible for the new tuple. It should be noted that inconsistencies may arise only by tuples that contain new pivot level combinations and thus create new forests. 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 identify the following two cases:

- *Strong consistency*: For applications that rely on constant data analysis and immediate detection of changes in trends, it is crucial that, at any time, any query to the data warehouse returns the complete and most up-to-date answers. For instance, in case of an intrusion detection application which analyzes data created by geographically disperse routers, denial-of-service (DoS) attacks must be tracked immediately in order to protect the routers from collapsing. To achieve strong consistency, after each tuple 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. Thus, consistency is guaranteed in exchange of a higher communication cost, which depends on the rate λ_{upd} at which updates are being performed.
- *Weak consistency*: When the application can afford some "staleness" in the data, a weak consistency scheme can be applied. Nodes append the inserted tuples to a globally known location. Index-holding peers can then, asynchronously, retrieve this directory and update the required indices. During the time period between the new tuple insertion and the asynchronous index update, it is possible that some answers are not 100% up-to-date. The freshness of the responses depends on λ_{upd}, as well as on the rate λ_{index} at which each node contacts the central directory and updates its indices. The communication cost is smaller than that of the strong consistency scheme, since $\lambda_{index} < \lambda_{upd}$. Therefore, this approach is recommended in cases where bandwidth resources are limited and 100% accuracy is not required.

4. Discussion—enhancements

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

4.1. 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 $K_{\max}$ is 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 (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 P (besides the data distribution of course): The higher the hierarchy levels in P, the larger the number of tuples that correspond to each tree.

4.2. 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 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 inter-arrival time. The more frequent the requests, the smaller W can be and vice versa.

In order to estimate λ, we need the zeroth and first frequency moment (F_0 and F_1 respectively) of the request sequence arriving at a node. F_0 is the number of distinct IDs that appear in the sequence, while F_1 is the length of the sequence (number of requests). Nodes can easily monitor the number of incoming requests inside a time interval. Many efficient schemes to estimate F_0 within a factor of $1 \pm \epsilon$ have been proposed (e.g., [6,5]). We use one of the schemes in [6], which requires only $O(1/\epsilon^2 + \log(m))$ memory bits, where m is the number of distinct node IDs. In reality, m is in the order of the network's size, since all nodes may possibly reach it in the DHT.

Finally, regarding the total amount of memory dedicated per node, this is dominated by the maximum number of non-pivot keys $K_{\max}$ that a node is responsible for. Assuming a value of $N = 1$K nodes for our application and that IDs and keys need 20 bytes (as outputs of SHA1 hash function), a node that is responsible for 1K different keys will need at most 20 MB of memory while for 10K keys a node will need at most 200 MB of memory (certainly affordable by most modern desktop PCs).

4.3. 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. Instead of crawling the entire network, the global statistics collection could be based on uniform sampling, thus decreasing the number of required messages. Random sampling in DHTs can be achieved simply by generating identifiers at random and finding the peers closest to them. Because peer identifiers are generated uniformly, we know they are uncorrelated with any other property. This technique is simple and effective, as long as there is little variation in the amount of the identifier space that each peer is responsible for. Such a sampling technique was used in various studies of widely deployed DHTs (e.g., [28]). However, the re-insertion of tuples is the part that dominates the complexity of the reindexing process, requiring $\Omega(N^2)$ messages. Therefore, it is important to ensure that our gains from reducing query flooding outweigh this cost.

More formally, let us assume that x and y are the pivot level combinations before and after reindexing respectively. As $Gain_{x\to y}(t)$ we denote the gain in messages after reindexing as a function of time and as $Cost_{x\to y}$ the cost of reindexing in messages. To conclude that a reindexing was indeed beneficial for the system, the following statement should be true:

$$
\begin{aligned}
&Cost_{x\to y} < Gain_{x\to y}(t) \Rightarrow \\
&Cost_{x\to y} < \mathrm{EM}_x \cdot \log(N) + \mathrm{Fl}_x \cdot N - \mathrm{EM}_y \cdot \log(N) - \mathrm{Fl}_y \cdot N \Rightarrow \\
&Cost_{x\to y} < \lambda_x \cdot t \cdot \log(N) + (\lambda - \lambda_x) \cdot t \cdot N \\
&\quad -\lambda_y \cdot t \cdot \log(N) - (\lambda - \lambda_y) \cdot t \cdot N \Rightarrow \\
&Cost_{x\to y} < (N - \log(N))(\lambda_y - \lambda_x) \cdot t
\end{aligned}
$$

where EM_i and Fl_i represent the exact match and flooded queries respectively for level combination i. Moreover, we assume no soft-state indices, a steady query arrival rate λ and steady query rates λ_x and λ_y targeted towards x and y respectively. $Cost_{x\to y}$ is bound by N^2, since the size of the dataset $|D| \gg N^2$ and thus messages are grouped by recipient. So, in the worst case:

$$
\begin{aligned}
&N^2 \cdot \log(N) < (N - \log(N))(\lambda_y - \lambda_x) \cdot t \Rightarrow \\
&(\lambda_y - \lambda_x) \cdot t > \frac{N^2 \cdot \log(N)}{N - \log(N)}
\end{aligned}
$$

and for large N values, $N - \log(N) \cong N$

$$
(\lambda_y - \lambda_x) \cdot t > N \cdot \log(N). \tag{1}
$$

From (1) we derive that a reindexing is beneficial in terms of messages when the difference in the number of exact matches before and after reindexing is greater than a number depending on the network size. This can be achieved either when there is a reasonable difference between the query arrival rates of the two level combinations or when adequate time has elapsed before a new reindexing takes place. We must stress that this formula represents the worst case scenario for *HiPPIS*, since we have assumed that soft-state indices do not contribute to the system's gain. However we expect that, depending on the posed workload, soft-state indices can significantly decrease the number of messages exchanged and thus lead to a balance between reindexing *Cost* and *Gain* more quickly.

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 explicitly 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., [25,11], etc.). Nevertheless and for our target applications, we believe and prove in our experimental section 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.

4.4. Minimize global statistics collection

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 $interval_{n,t}$ parameter for each node n at the tth time it checks its statistics. 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: $interval_{n,0} = T_s$. In order to discourage consecutive reindexing attempts from the same node, this parameter is multiplicatively increased when the processing of global statistics concludes in different results or in a no-change decision and reset to T_s otherwise. Specifically:

$$
interval_{n,t} = \begin{cases} 2 \times interval_{n,t-1}, & \text{if conflict between} \\ & \text{local and global stats} \\ T_s, & \text{otherwise} \end{cases}
$$

4.5. 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 discouraged, 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 ($C : c_0 < c_1 < \cdots < c_{\max}$). 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 P 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,

$$\{\forall c_i \in C, 0 \leq i \leq \max \mid popularity_{c_{\max}} - popularity_{c_i} < threshold \Rightarrow c_i \in C_{cand}\}$$

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 $threshold = k \cdot \Delta, k \geq 1$. 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 $\langle$`Country`, `Brand`$\rangle$, $\langle$`City`, $*\rangle$ and $\langle *$, `Brand`$\rangle$ are the candidate pivot levels, with *popularities* of 10%, 20% and 15% respectively. Comparing the two levels with $*$, $\langle *$, `Brand`$\rangle$ can answer $\langle$`Country`, `Brand`$\rangle$ queries, thus its *popularity* rises to 25%, and is therefore chosen over $\langle$`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.

5. 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 [30], although any DHT implementation could be used as a substrate. By default, we assume a network size of 256 nodes, but results were collected with up to 8K nodes. In our simulations, we use synthetically generated data, produced by our own as well as the APB-1 benchmark generator [22]. In the former case, each dimension is represented as a tree with each value having a single parent and *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 is, by default, $\langle H_{12}, H_{22}, H_{32} \rangle$. The APB-1 generated datasets are described in the corresponding subsection.

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).

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

Our default workload comprises of 35k queries which arrive at an average rate λ_{query} of 10 queries per simulated time unit. For simplicity reasons we have set the time unit equal to 1 s, therefore $\lambda_{\text{query}} = 10 \frac{\text{queries}}{\text{s}}$. For our experiments, W is set to 50 s and *TTL* is given a practically infinite value (indices never expire). Finally, the value of $I_{\max}$, which is heavily data and query-dependent, has been experimented on and set to 2k (each node dedicates at most 100 KB of memory on soft-state indices).

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, 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).

5.1. 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 the first graph of Fig. 5, 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 exact matches due to the chosen combination. Moreover indices contribute more to the system's precision, since the number of distinct queries for non pivot tuples decreases. For uniform distributions, the number of distinct queries does not allow our method to capitalize on the indexing scheme.

The next graph 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.

Fig. 6 depicts the number of messages exchanged per query in the system, indicating a measure of bandwidth consumption.

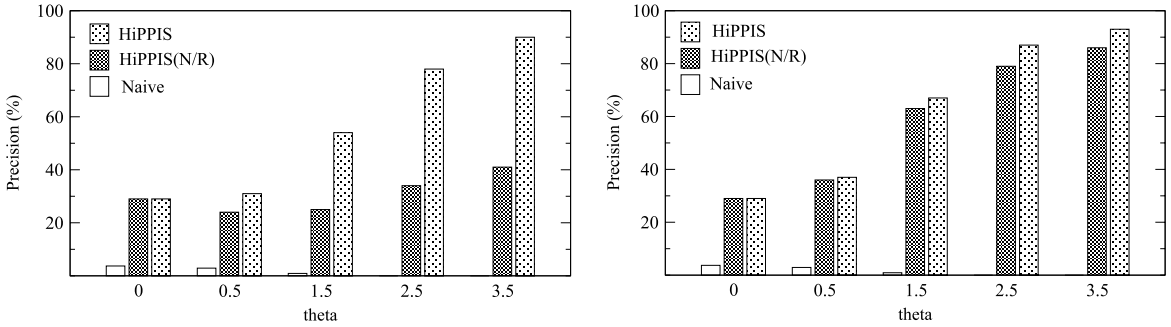

Fig. 5. Precision for varying levels of skew (most popular combination is $\langle H_{13}, H_{23}, H_{33}\rangle$ and $\langle H_{11}, H_{21}, H_{31}\rangle$ respectively).

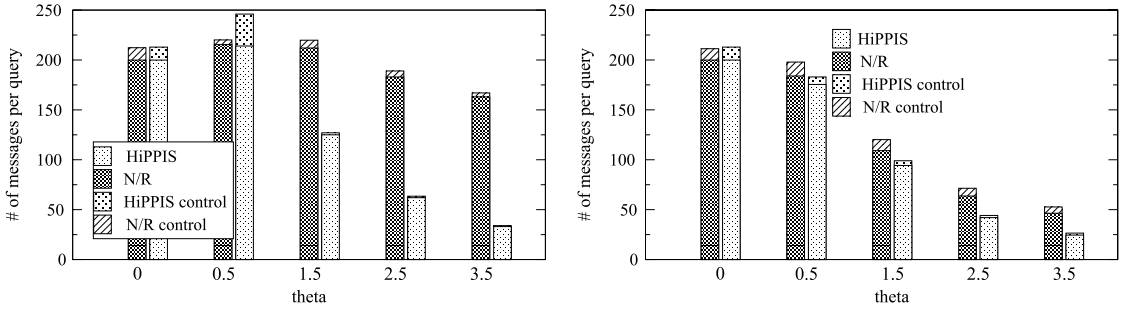

Fig. 6. Average number of messages required to answer a query for varying levels of skew (most popular combination is $\langle H_{13}, H_{23}, H_{33}\rangle$ and $\langle H_{11}, H_{21}, H_{31}\rangle$ respectively).

Table 3
Statistics for various datasets.

Direction of skew	θ	#Global stats	#Reindexings	#Reinsertions	Simulation time (s)	BW (KB)
Up	1.5	5	1	11 746	5.1	755
Up	2.5	4	1	11 678	5.5	746
Up	3.5	2	1	11 521	5.5	730
Down	1.5	5	1	16 824	4.2	743
Down	2.5	4	1	16 933	4.4	735
Down	3.5	3	1	16 701	4.3	740

Messages regarding query resolution (including requests as well as responses) and control messages, which include those needed 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.

5.2. Reindexing cost

Table 3 presents statistics concerning the reindexing process during the workloads of the previous experiment. The workloads directed towards $\langle H_{13}, H_{23}, H_{33}\rangle$ are denoted as *down* and the ones towards $\langle H_{11}, H_{21}, H_{31}\rangle$ as *up*. As aforementioned, the cost of reindexing is non-negligible. Hence, it is very important that the system performs the minimum required reindexing rounds. *HiPPIS* proves extremely efficient to that end: Only one reindexing process is carried out per direction of skew and less than 5 *SendStats* requests are produced per simulation, thanks to the *interval* selection strategy presented in Section 4.4. Thus, our method makes near-optimal use of its bandwidth-intensive operations. It is also worth noting that reindexings towards the lowest hierarchy levels cause more reinsertions than those directed towards the upper ones. This is due to the fact that the dataset used has a limited number of tuples. For a large number of tuples, reinsertions for all directions converge to N^2. However, the total consumed bandwidth (denoted as *BW*) remains the same regardless the skew and its direction, since, in all cases, the initial dataset is reinserted. The time measurements may not adequately reflect reality due to the fact that the experimental evaluation is based on a simulation rather than a real, deployed system. In a real system of N nodes, we would expect a significant acceleration in computation (almost N-fold) and a communication cost depending on the topology of the underlying network.

Trying to identify the circumstances under which our system benefits from the reindexing process, we plot the *Cumulative Gain* and the *Cumulative Cost* of reindexing in messages for various datasets and workloads (Fig. 7). By *Cumulative Gain* we signify the total number of messages spared when using *HiPPIS* instead of N/R and in *Cumulative Cost* we include messages for global statistics collection, locking and reinsertion of tuples. Our first observation is that the *Cumulative Gain* increases more rapidly with the increase in skew. This is natural, since highly skewed workloads translate to bigger differences between the most popular and the rest level combinations. Moreover, the workloads directed towards $\langle H_{13}, H_{23}, H_{33}\rangle$ exhibit a higher increase rate in *Cumulative Gain* compared to the ones towards $\langle H_{11}, H_{21}, H_{31}\rangle$. This is due to the fact that the soft-state indexing mechanism of N/R is more effective in the latter case (less distinct values for the specific level combination). However, for highly biased workloads, regardless

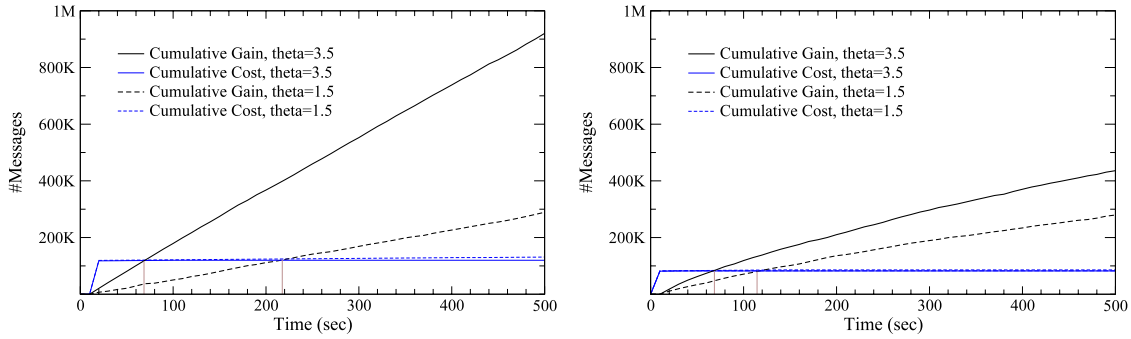

Fig. 7. Balance between reindexing cost and gain in messages over time (skew towards $\langle H_{13}, H_{23}, H_{33}\rangle$ and $\langle H_{11}, H_{21}, H_{31}\rangle$ respectively).

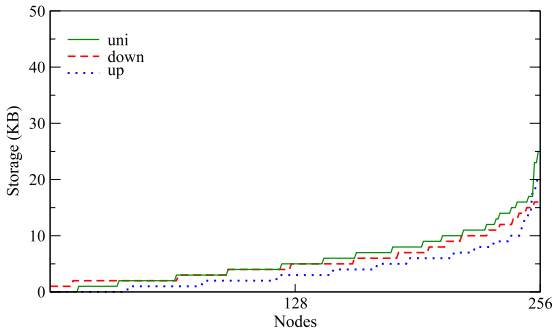

Fig. 8. Storage distribution over the network nodes.

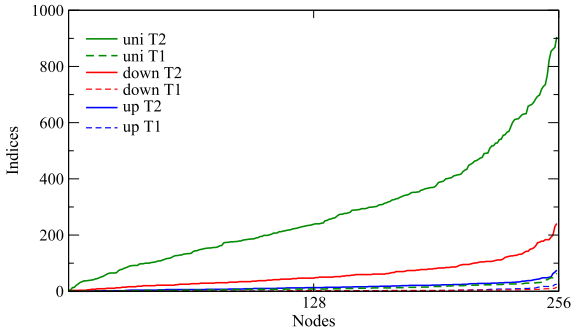

Fig. 9. Index distribution over the network nodes.

the direction of skew, our system manages to outweigh the reindexing cost in less than 100 s.

The results of this experiment conform to the conclusions derived by our cost benefit analysis of Section 4.3: The higher the workload skew, the more quickly *HiPPIS* starts gaining benefit from a reindexing.

5.3. Storage and load distribution

This set of experiments aims to evaluate *HiPPIS* in terms of storage and load distribution among the participating network nodes. Using the default dataset, we utilize three of the workloads generated for the previous experiments: the one with levelDist of $\theta = 0$ (denoted as *uni*), and the ones with $\theta = 3.5$, directed towards $\langle H_{11}, H_{21}, H_{31}\rangle$ and $\langle H_{13}, H_{23}, H_{33}\rangle$ (denoted as *up* and *down* respectively).

Fig. 8 depicts the space dedicated by each node for storing the actual data (in the form of the forest-like structures presented in Section 3.2) after the end of the simulation. The measured quantities for each of the 256 nodes are sorted in ascending order. After the necessary reindexings have occurred, the final pivot level combinations are $\langle H_{12}, H_{22}, H_{32}\rangle$, $\langle H_{11}, H_{21}, H_{31}\rangle$ and $\langle H_{13}, H_{23}, H_{33}\rangle$ for *uni*, *up* and *down* respectively. The more numerous the different values of P, the more balanced is the storage distribution among the nodes. In the case of the *down* workload, the majority of nodes host similar quantities of storage space. However, even for the *up* workload, no major differences are documented, since the number of different value combinations is still much larger than the size of the network. This leaves the fairness of the distribution mainly on the hash function.

P affects the total disk space needed to store the distributed data structure: The same dataset requires more space when stored under $\langle H_{13}, H_{23}, H_{33}\rangle$ than under $\langle H_{11}, H_{21}, H_{31}\rangle$. This is due to the fact that a P close to the root of the forest eliminates redundancies in all levels lying below it in the hierarchy. This can be clarified by observing Figs. 2(c) and 4. While the value 'Athens' is stored just once for $\langle$`city`, `category`$\rangle$, it needs to be stored 3 times for $\langle$`city`, `brand`$\rangle$.

Fig. 9 shows the amount of indices stored by each network node in ascending order at two distinct points in time, at $T_1 = 100$ s and $T_2 = 3000$ s. T_1 corresponds to an initial point before any reindexing has occurred, whereas T_2 to a moment close to the end of the simulation. In all cases (except *uni*), indices are distributed pretty evenly among the nodes, with more skewed loads registering the most balanced results. *Uni* exhibits a remarkable increase in indices over time (almost 18 times as many indices in T_2 than in T_1). Since the queries can contain any value with equal probability and no reindexing is performed, very few queries are being repeated and thus indices are constantly being built. The smallest increase is documented for the *up* workload, since it is the workload with the least possible value combinations that can be queried. Fig. 11 depicts the average number of messages per second handled by each network node over time, including control messages. For a uniform workload, the simulation starts with an average load almost as high as the query arrival rate, since the majority of the queries are answered though flooding. As time progresses and indices are being created, this measure decreases almost linearly. For the skewed workloads, we observe a spike in load shortly after the simulation starts (see embedded graph). This is due to the reindexing process and mainly to the reinsertion of the dataset according to the new P. However, the average load per node remains within acceptable limits (less than 30 msg/s) and can easily be handled by the network nodes. Moreover, the decrease is more abrupt during the first 100 s, while after that point, no significant improvement is documented. This can be explained by the fact that reindexing occurs very quickly, thus leaving little room for refinement through index creation. Finally, as seen in Fig. 10, the individual load (sorted in ascending order) is very evenly distributed among the network nodes at all times.

5.4. Scaling the network and dataset size

In this set of experiments we aim to examine how well our system scales with regard to the number of participating nodes and the number of tuples in the dataset. First, having inserted the default dataset, we vary the network size from 128 to 8192 nodes.

Table 4
Statistics for various network sizes.

#Nodes	#Global stats	#Reindexings	Avg. load/node (msg/s)	ReIndLoad/node (msg/s)	#Msg/query	Precision (%)
128	5	1	2.2	95	28	84.7
256	4	1	2.0	54	51	84.4
1024	2	1	1.8	19	182	83.5
2048	2	1	1.7	12	348	83.3
4096	2	1	1.6	9	655	83.3
8192	2	1	1.6	6	1202	83.3

Table 5
Statistics for various dataset sizes.

#Tuples	#Global stats	#Reindexings	Avg. load/node (msg/s)	ReIndLoad/node (msg/s)	BW (MB)	#Msg/query	Precision (%)
100K	3	1	2	204	4	52	83
1M	4	1	2	235	40	49	83
10M	3	1	2	254	400	50	83
100M	3	1	2	255	4000	49	83

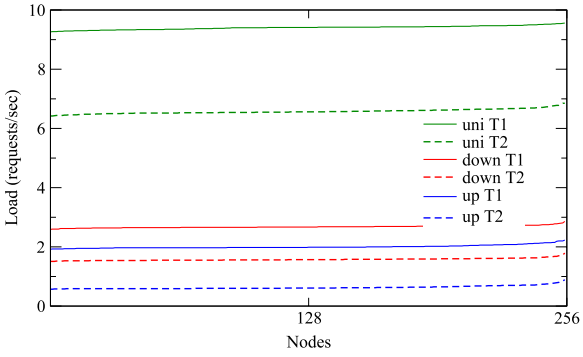

Fig. 10. Load distribution over the network nodes.

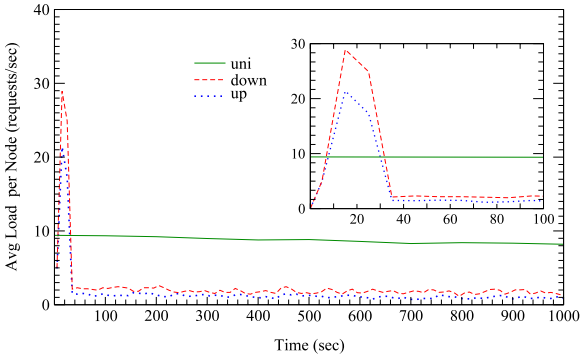

Fig. 11. Average load per network node over time.

We believe that for a data warehousing application, a system consisting of 8K nodes is an already exaggerated scenario. Also, we vary the dataset size from 100K to 100M tuples and insert it in our default system of 256 nodes. In all cases we pose workloads with levelDist of $\theta = 3$, directed towards $\langle H_{13}, H_{23}, H_{33}\rangle$.

As Table 4 proves, *HiPPIS* manages to maintain a steadily high precision, performing only one reindexing and collecting global statistics less than 5 times throughout the simulation, regardless of the network size. Of course, the average number of messages required to resolve a query increases with the increase in network nodes, as flooding becomes more costly. However, this number is scattered over the network nodes, resulting in a decreasing average load per node. The same is true for the load caused by the reindexing process, since the number of reinsertions remains the same in all cases due to the dataset size ($|D| < N^2$).

When the number of tuples increases by orders of magnitude, only the bandwidth consumed during the reindexing process

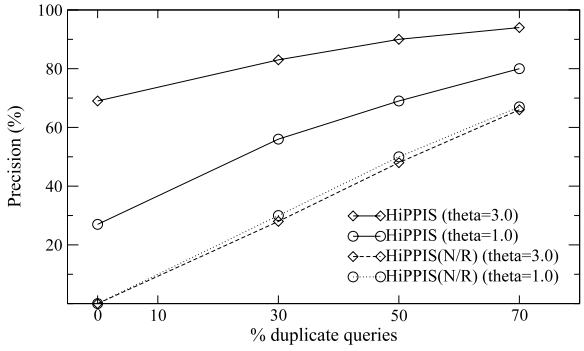

Fig. 12. Precision over variable percentage of duplicate queries.

shows a proportional increase, due to the reinsertion of the dataset. The rest of the statistics presented in Table 5 remain stable: Invariably high precision, steady average load per node and number of messages per query, and reindexing load per node converging to N.

5.5. The 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 Figs. 12 and 13 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 P which takes up significantly more requests. As θ increases, we normally expect an increase in performance.

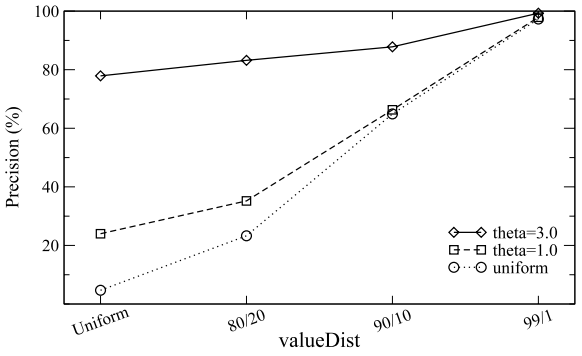

Fig. 13. Precision for varying distributions of *valueDist*.

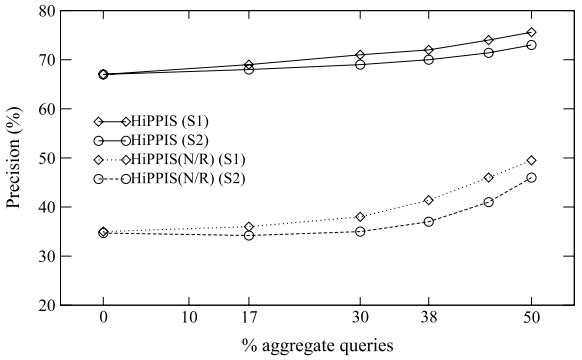

Fig. 14. Precision over variable number and skew of aggregate queries.

In the second case (Fig. 13), 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 repetition 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.

5.6. 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 Fig. 14.

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).

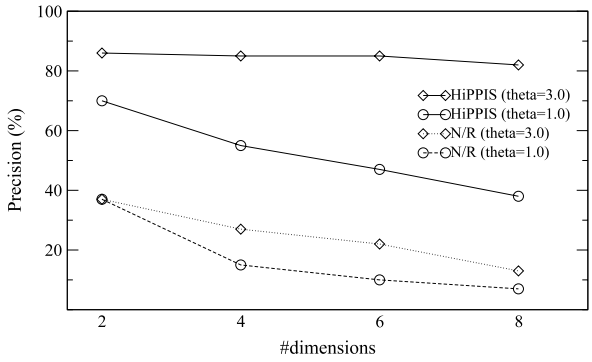

Fig. 15. Precision over variable dimensionality datasets.

5.7. 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 ($t_c = 3100$ s): 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 Fig. 16.

Our results show that, in all cases, *HiPPIS* quickly increases its precision due to the combination of automatic reindexing and soft-state indices. Flooding increases 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 a remarkable increase in precision (starting from the plain data-insertion at $t = 0$ s), 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 workloads, but its performance in the steady state is invariably high, regardless the workload.

5.8. Varying the 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. Fig. 15 depicts the results for 2–8 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 reduces 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 with the dimension's axis. *HiPPIS* ranges between 40% and 70% in the low skew case while for bigger skew this becomes 80%–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.

5.9. Updates

In this subsection we focus on the evaluation of the weak consistency update mechanism of *HiPPIS*. Specifically, we run the

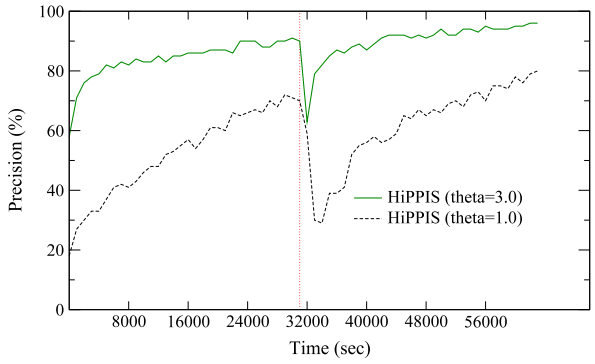

Fig. 16. Precision over time for various workloads when a sudden shift in skew occurs in $t_c = 31\,000$ s.

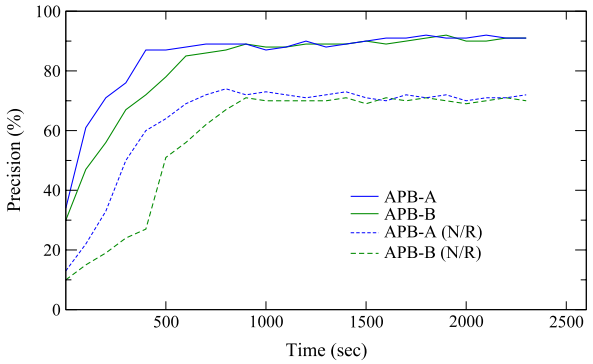

Fig. 17. Precision of *HiPPIS* for the APB query workload.

Table 6
Percentage of inconsistent answers for various λ_u.

λ_{upd} (updates/s)	Inconsistency (%)	
	U_1	U_2
0.01	0.10	1.18
0.1	1.26	5.02
1.0	8.15	18.23
10.0	19.21	20.01

simulator using the default dataset and two workloads, U_1 and U_2 with skew set to $\theta = 2.0$. U_1 contains exclusively point queries, whereas in U_2, 30% of the workload's queries are aggregate ones. During he simulation, we apply incremental updates at a rate λ_{upd} that varies from 0.01 to 10 $\frac{\text{updates}}{\text{s}}$. The period of the index update procedure is set to 100 s for all network nodes in all cases. Bearing in mind that the incoming query rate $\lambda_{\text{query}} = 10\ \frac{\text{queries}}{\text{s}}$, this translates to queries being posed 1000 times more frequently than index updates are checked. Incremental updates also occur up to 1000 times more often than index checking. We measure the percentage of queries whose answers are incomplete, henceforth termed as *inconsistency* and present the results in Table 6. Note that a response is considered inconsistent, when at least one record is missing, regardless the total number of missing records.

Naturally, the faster the update rate, the higher the number of inconsistent answers. However, inconsistency tends to converge as λ_{upd} increases. Even when λ_{upd} is equal to the incoming query rate, meaning that every update is followed by a query, the inconsistency remains in tolerable levels, due to the fact that after the necessary reindexings have occurred, most of the queries are answered directly, without the use of indices. The impact of reindexing is evidently heavier for U_1. Since it does not contain any aggregate queries, the workload is more targeted to the new pivot level values, thus requiring less use of indices. As a result, the inconsistency ratio is noticeably lower than that of U_2.

Finally, it is worth noticing that when following the weak consistency scheme, the cost of updates is independent of λ_{upd} and equals N messages every period of the index update procedure (2.56 $\frac{\text{msg}}{\text{s}}$ in this case). On the contrary, a strong consistency scheme would provide 100% accuracy, but require $(\prod_{i=0}^{d} L_i - 1) \cdot \log(N)$ messages per update, resulting in an average rate ranging from 2.16 to 2160 $\frac{\text{msg}}{\text{s}}$ for our simulation settings, depending on λ_{upd}. Therefore, for high λ_{upd} the strong consistency scheme should be avoided due to the considerable communication cost it produces.

5.10. APB benchmark datasets

Finally, we test the performance of *HiPPIS* using some more realistic data and query sets generated by the APB-1 benchmark [22]. APB-1 creates a database structure with multiple dimensions and generates a set of business operations reflecting the basic functionality of OLAP applications. Running the APB-1 data generator with the density parameter set to 0.1 and 1, we produced two 4-dimensional datasets (APB-A and APB-B) with cardinalities 9000, 900, 9 and 24 and two measure attributes. Each dimension is comprised of a hierarchy of 7, 4, 2 and 3 levels respectively. APB-A contains 1.2M and APB-B 12M tuples respectively, 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 Fig. 17.

We clearly notice that *HiPPIS* exhibits very high performance, reaching over 90% of precision in its steady state after about 400 s for APB-A. 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).

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. Depending on the needs of the application, *HiPPIS* can also deploy variable consistency update schemes to achieve the desired accuracy in replies without excessive communication overhead.

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 few recurring queries, *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. At the same time, the system manages to avoid a substantial load imbalance or uneven storage distribution. Finally, adopting a weak consistency update scheme does not significantly degrade the freshness of the responses (less than 20% of the answers are incomplete), even when updates occur as often as queries are posed.

Business analytics are believed to represent particularly good opportunities for Cloud Computing applications with many companies moving towards this direction. We believe that our system is a particularly good candidate for deployment in the Cloud and this is what we currently pursue. Indeed, HiPPIS provides several architectural characteristics required for the Cloud, such as cost-efficiency, meaning high-performance query processing and updates, elasticity, meaning seemingly infinite resources through the use of a shared-nothing architecture and content availability, meaning that the application should be able to continue running in the event of multiple node failures.

References

[1] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, T. Van Pelt, Gridvine: Building Internet-Scale Semantic Overlay Networks, in: Lecture Notes in Computer Science, 2004, pp. 107–121.

[2] S. Abiteboul, T. Allard, P. Chatalic, G. Gardarin, A. Ghitescu, F. Goasdoué, I. Manolescu, B. Nguyen, M. Ouazara, A. Somani, N. Travers, G. Vasile, S. Zoupanos, WebContent: efficient P2P warehousing of web data, Proceedings of the VLDB Endowment Archive 1 (2) (2008) 1428–1431.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin, HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads, in: Proceedings of the 35th International Conference on Very Large Data Bases–Volume 35, 2009, pp. 1084–1095.

[4] M. Akinde, M. Böhlen, T. Johnson, L. Lakshmanan, D. Srivastava, Efficient OLAP query processing in distributed data warehouses, Information Systems 28 (1–2) (2003) 111–135. doi:http://dx.doi.org/10.1016/S0306–4379(02)00051–0.

[5] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, Journal of Computer and System Sciences 58 (1) (1999) 137–147.

[6] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, L. Trevisan, Counting Distinct Elements in a Data Stream, in: Lecture Notes in Computer Science, vol. 2483, 2002, pp. 1–10.

[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, SCOPE: easy and efficient parallel processing of massive data sets, Proceedings of the VLDB Endowment 1 (2) (2008) 1265–1276. doi:http://doi.acm.org/10.1145/1454159.1454166.

[8] M. Cha, H. Kwak, P. Rodriguez, Y. Ahn, S. Moon, I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system, in: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, ACM, 2007, p. 14.

[9] Q. Chen, U. Dayal, M. Hsu, A distributed OLAP infrastructure for e-commerce, in: Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems, IEEE Computer Society, 1999, pp. 209–220.

[10] M. Ester, J. Kohlhammer, H. Kriegel, The DC-tree: a fully dynamic index structure for data warehouses, in: Proceedings of the International Conference on Data Engineering, IEEE Computer Society Press, 2000, pp. 379–388.

[11] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, P. Keleher, Adaptive replication in peer-to-peer systems, in: International Conference on Distributed Computing Systems, vol. 24, 2004, pp. 360–371.

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

[13] Hadoop Web page. http://hadoop.apache.org/core/.

[14] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, I. Stoica, Querying the Internet with PIER, in: Proceedings of the 29th International Conference on Very Large Data Bases–Volume 29, VLDB Endowment, 2003, p. 332.

[15] P. Kalnis, W. Ng, B. Ooi, D. Papadias, K. Tan, An adaptive peer-to-peer network for distributed caching of OLAP results, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2002, pp. 25–36.

[16] V. Kantere, D. Tsoumakos, T. Sellis, N. Roussopoulos, GrouPeer: dynamic clustering of P2P databases, Information Systems 34 (1) (2009) 62–86.

[17] E. Knorr, Dealing with the data explosion. URL: http://www.infoworld.com/d/storage/dealing-data-explosion-690?page=0,0.

[18] L. Lakshmanan, J. Pei, Y. Zhao, QC-trees: an efficient summary structure for semantic OLAP, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, ACM, 2003, p. 75.

[19] B. Loo, J. Hellerstein, R. Huebsch, S. Shenker, I. Stoica, Enhancing P2P file-sharing with an Internet-scale query processor, in: Proceedings of the 30th International Conference on Very Large Data Bases–Volume 30, VLDB Endowment, 2004, p. 443.

[20] K. Morfonios, Y. Ioannidis, CURE for cubes: cubing using a ROLAP engine, in: Proceedings of the 32nd International Conference on Very Large Data Bases–Volume 32, VLDB Endowment, 2006, p. 390.

[21] K. Morfonios, Y. Ioannidis, Revisiting the cube lifecycle in the presence of hierarchies, The VLDB Journal The International Journal on Very Large Data Bases 19 (2) (2010) 257–282.

[22] OLAP council APB-1 OLAP benchmark. http://www.olapcouncil.org/research/resrchly.htm.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: SIGMOD'08, ACM, New York, NY, USA, 2008, pp. 1099–1110. doi:http://doi.acm.org/10.1145/1376616.1376726.

[24] W. Ooi, K. Zhou, PeerDB: a P2P-based system for distributed data sharing, in: Proceedings of the 19th International Conference on Data Engineering, ICDE'03, vol. 1063, 2003, pp. 633–644.

[25] T. Pitoura, N. Ntarmos, P. Triantafillou, Replication, Load Balancing and Efficient Range Query Processing in DHTs, in: Lecture Notes in Computer Science, vol. 3896, 2006, p. 131.

[26] M. Ripeanu, I. Foster, 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) 50–57.

[27] Y. Sismanis, A. Deligiannakis, Y. Kotidis, N. Roussopoulos, Hierarchical dwarfs for the rollup cube, in: Proceedings of the 6th ACM International Workshop on Data Warehousing and OLAP, ACM, New York, NY, USA, 2003, pp. 17–24.

[28] D. Stutzbach, R. Rejaie, Improving lookup performance over a widely-deployed DHT, in: Proceedings of Infocom, vol. 6, 2006.

[29] C. Tang, Z. Xu, S. Dwarkadas, Peer-to-peer information retrieval using self-organizing semantic overlay networks, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, New York, NY, USA, 2003, pp. 175–186.

[30] The FreePastry project. http://freepastry.rice.edu/FreePastry.

[31] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive—a warehousing solution over a MapReduce framework, Proceedings of the VLDB Endowment 2 (2) (2009) 1626–1629.

[32] A.A. Vaisman, M.M. Espil, M. Paradela, P2P OLAP: data model, implementation and case study, Information Systems 34 (2) (2009) 231–257.

[33] W. Wang, H. Lu, J. Feng, J. Yu, Condensed cube: an effective approach to reducing data cube size, in: Proceedings of the International Conference on Data Engineering, 2002.

Katerina Doka received her Diploma in Electrical and Computer Engineering from the National Technical University of Athens in July 2005. She is currently a Ph.D. candidate at the Computing Systems Laboratory, Department of Electrical and Computer Engineering of the National Technical University of Athens, conducting research in the field of Large Scale Distributed Systems, Peer-to-Peer Technologies and Grid/Cloud Computing.

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

Nectarios Koziris Associate Professor, NTUA. His research interests include parallel architectures, scalable distributed systems and data & resource management for large scale Internet systems. He has published more than 90 papers in international journals and in the proceedings of international conferences. Nectarios Koziris is a recipient of the IEEE IPDPS 2001 best paper award. He served as a Chair and Program Committee member in various IEEE/ACM conferences. He is a member of IEEE, senior member of ACM and chairs the Greek IEEE CS Chapter. He also serves as the Vice-Chairman for the Greek Research and Education Network.