J. Parallel Distrib. Comput. 71 (2011) 1434–1446

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

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

Brown Dwarf: A fully-distributed, fault-tolerant data warehousing system[☆]

Katerina Doka[*], Dimitrios Tsoumakos, Nectarios Koziris

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

A R T I C L E I N F O

Article history:
Received 2 July 2010
Received in revised form
11 May 2011
Accepted 10 July 2011
Available online 23 July 2011

Keywords:
Peer-to-Peer
Data warehousing
Data cube

A B S T R A C T

In this paper we present the *Brown Dwarf*, a distributed data analytics system designed to efficiently store, query and update multidimensional data over commodity network nodes, without the use of any proprietary tool. *Brown Dwarf* distributes a centralized indexing structure among peers on-the-fly, reducing cube creation and querying times by enforcing parallelization. Analytical queries are naturally performed on-line through cooperating nodes that form an unstructured Peer-to-Peer overlay. Updates are also performed on-line, eliminating the usually costly over-night process. Moreover, the system employs an adaptive replication scheme that adjusts to the workload skew as well as the network churn by expanding or shrinking the units of the distributed data structure. Our system has been thoroughly evaluated on an actual testbed: it manages to accelerate cube creation up and querying up to several tens of times compared to the centralized solution by exploiting the capabilities of the available network nodes working in parallel. It also manages to quickly adapt even after sudden bursts in load and remains unaffected with a considerable fraction of frequent node failures. These advantages are even more apparent for dense and skewed data cubes and workloads.

1. Introduction

The advent of the new century has been clearly marked by what IT people refer to as "data explosion" [20]. Market globalization, business process automation, the growing use of sensors and other data-producing devices [29], along with the increasing affordability of hardware have contributed to this continuous trend. Large companies, scientific organizations (e.g., *NASA* or *WMO*) as well as more specialized enterprises (such as government, Internet-related, etc.) heavily rely on data analysis in order to identify behavioral patterns and discover interesting trends/associations. New compliance regulations mandate even more data retention as a brute force method to reduce risk, while an increasing number of organizations begin to digitize their records. It is not surprising that enterprises are adding capacity at an astounding rate.

Data warehousing has thus become a vital component of every organization, as it contributes to business-oriented decision-making. Data warehouses store vast amounts [24] of historical and operational data in the form of multidimensional *cubes*. Their workloads (often referred to as *analytical* queries) usually consist of read-only queries interleaved with batch updates. Besides the well-documented need for off-line analytics, the requirement for constant data analysis in order to immediately detect real-time changes in trends is ever-growing [6,20]. The satisfaction of both modes implies the need for an always-on, real-time data access and support system for concurrent processing of large query rates without significant deterioration in response times.

Traditional warehouses present a strictly centralized and off-line approach in terms of data location and processing [21,35,30]: views are usually calculated on a daily or weekly basis after the operational data have been transferred from various locations. Furthermore, the need to keep large volumes of historical data online and ensure their availability and fast access even under heavy workload dictates a continuous investment in hardware, electrical power and software maintenance. Some works in the field propose distributed warehousing systems [18,4,2], but the warehouse and its aggregation, update and querying functionality remain centralized. Recently, effort has been made to distribute the data warehouse itself by applying techniques from the field of Peer-to-Peer (P2P) computing [12], but with no a priori consideration for group-by queries. Moreover, none of these approaches deals with the query performance versus variable data availability or load skew.

Recently, a new class of large scale analytics engines [3,33] has emerged to leverage the recent innovation in the industry around large-scale data management. Deployed on shared-nothing, commodity hardware architectures, they cover the newly added requirement for scalability, robustness and availability at low cost. Yet, as they are based on the MapReduce programming model, they target mostly batch-mode analytics jobs rather than real-time, "per-tuple" processing [23]. Lastly, parallel databases [27,32] offer great efficiency at the cost of elasticity and robustness in failures [28].

[☆] This is an extended version of the short paper presented in HPDC'10.

[*] 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 © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.07.008

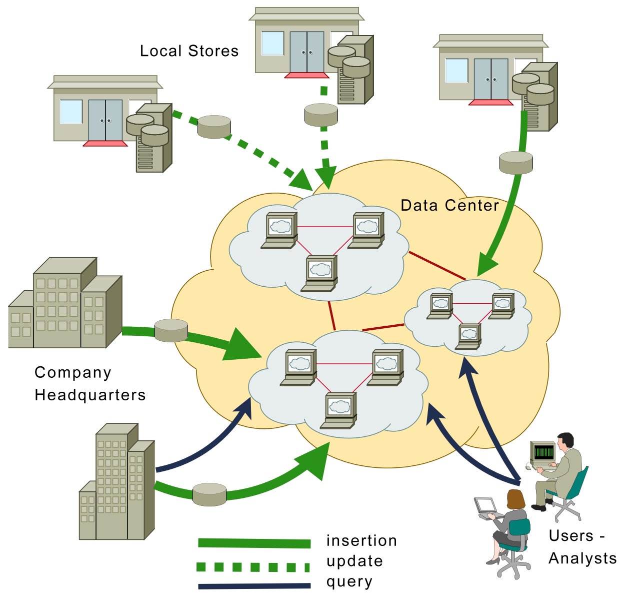

Fig. 1. Motivating scenario of distributing a data warehouse.

Table 1
A sample fact table with three dimensions and one measure.

DIM1	DIM2	DIM3	Measure
S_1	C_2	P_2	\$70
S_1	C_3	P_1	\$40
S_2	C_1	P_1	\$90
S_2	C_1	P_2	\$50

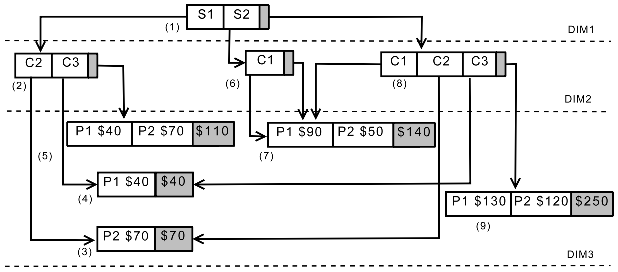

Fig. 2. Centralized Dwarf for the fact table of Table 1, using the *sum* aggregation function.

Our goal is to create an on-demand version of a highly efficient data warehousing system, where geographically spanned users, without the use of any proprietary tool, can share and query information. As a motivating scenario, let us consider a business establishment that maintains records of its operations. These records could well be security, network or system event logs making the search and analysis of that data an essential part of managing, securing and auditing how this company's technology infrastructure is used. Instead of creating a centralized data warehouse on site with a large upfront (and maintenance) cost, the management chooses to distribute data and computation to possibly multiple location-transparent facilities of commodity nodes and access it more easily and ubiquitously. Fig. 1 depicts this scenario where multiple establishments of a business insert, update and query such a distributed warehouse.

To this end, we propose the *Brown Dwarf*,[1] a system that performs on-line distribution of a centralized warehousing structure, the *Dwarf* [30], over network hosts in a way that all queries that were originally answered through the centralized structure are now distributed over an unstructured P2P network of commodity nodes. *Dwarf* is an approach to compute, index and query large volumes of multidimensional data. While it offers many advantages, like data compression and efficiency in answering aggregate queries, it exhibits certain limitations that prohibit its use as a solution for our motivating problem. Besides the lack of fault-tolerance and decentralization, a *Dwarf* structure may take up orders of magnitude more space than the original tuples [11]. Our *Brown Dwarf* system relaxes these storage requirements and enables the computation of much larger cubes. Moreover, it allows for on-line updates that can originate from any host that accesses the particular service. Finally, the proposed system can handle significantly larger query rates and actively protect against failing or uncooperative peers, as it offers multiple entry points and adaptive replication of the most loaded parts of the cube. In summary, the contributions of this paper are the following:

- A complete indexing, query processing and update system for data cubes over a distributed environment. The cube is created with just one pass over the data, while updates are processed on-line. Commodity PCs can participate in this distributed data store, while users need no proprietary tool to access it.
- A robust and efficient adaptive replication scheme, perceptive to workload skew as well as node churn. *Brown Dwarf* nodes achieve this using only local load measurements and overlay knowledge.
- A thorough validation of the proposed system using an actual deployment. Our findings show that *Brown Dwarf* can be as much as 36 times faster in creating the cube and 60 times faster in querying it compared to the centralized version. Moreover, it offers a fair distribution of the original dwarf and the induced query load at low cost (shared among many nodes), it shows impressively prompt adaptation to query skew and proves resilient to a considerable fraction of node failures even with low replication ratios.

[1] A *Brown Dwarf* is an object which has a size between that of a giant planet and that of a small star. It is possible that a non-negligible portion of the mass in the Universe is in the form of Brown Dwarfs.

2. The centralized Dwarf and overview of the Brown Dwarf

Dwarf [30] is a centralized structure for indexing, storing, querying and updating both fully and partially materialized data cubes. *Dwarf*'s main advantage is the fact that it eliminates *both* prefix and suffix redundancies among the dimension values of multiple views. Prefix redundancy happens when a value of a single or multiple dimensions occurs in multiple group-bys (and possibly many times in each group-by). For example, for the raw data in Table 1 (referred to as the *fact table*), the value S_2 appears in $\langle S_2, C_1, P_1\rangle$, $\langle S_2, C_1, P_2\rangle$ but also in many group-bys (e.g., $\langle S_2, C_1\rangle$, $\langle S_2\rangle$, etc.). On the other hand, suffix redundancy occurs when some group-bys share a common suffix. For example, we can see in Table 1 that the $\langle C_1, x\rangle$ group-by has the same value as the $\langle S_2, C_1, x\rangle$ one, with x being any value in the third dimension.

Fig. 2 shows the cube created by this algorithm for the fact table of Table 1. The structure is divided in as many levels as the number of dimensions. The root node contains all distinct values of the first dimension. Each cell value points to a node in the next level that contains all the distinct values that are associated with its value. Gray cells correspond to *ALL* values of that cell, used for aggregates on each dimension. Any group-by query can be answered through traversing the structure and following the query attributes, leading to a *leaf* node with the answer. For example, query $\langle S_1, C_3, P_1\rangle$ will return the \$40 value while $\langle S_2, ALL, ALL\rangle$ will return the aggregate value \$140 following the path (1) $\rightarrow$ (6) $\rightarrow$ (7).

Brown Dwarf (or *BD* for short) is a system that distributes *Dwarf* over a network of interconnected nodes. The goal is to have the ease of constructing, querying and updating this structure in an on-line fashion over a network overlay instead of a central location.

The *BD* construction algorithm distributes dwarf nodes to network hosts on-the-fly as tuples are parsed in a single pass. Pictorially, Fig. 3 shows that nodes (1) through (9) are selected in this order to store the corresponding dwarf nodes of Fig. 2. These nodes form an unstructured P2P overlay, using the indexing induced by the centralized creation algorithm. Queries and updates are then naturally handled using the same path that would be utilized in *Dwarf*, with overlay links now being followed: if the incoming query asks about S_1 it will be forwarded to node (2). From there, depending on the requested group-by (ALL, C_2 or C_3), terminal nodes (3), (4) or (5) can be visited.

Compared to the traditional *Dwarf* (or other centralized indexing methods used in data-warehousing), *BD* offers the following advantages:

- The existence of more than one hosting nodes offers the ability to parallelize the cube construction, querying and update process.
- The distribution of the structure enables the computation and storage of much larger cubes.
- *BD* allows for on-line updates that can originate from any host accessing the update service of the system.
- The proposed system can handle significantly larger rate of requests without having to replicate the whole structure, as it offers multiple entry points and adaptive replication of the most loaded parts of the cube.

3. The Brown Dwarf system

The essence of *Brown Dwarf* (or *BD*) is the distribution of the original, centralized structure over the nodes of an unstructured overlay in a way that guarantees query processing efficiency even under node churn.

While structured P2P overlays provide efficient lookup operations for (*key, value*) pairs, they do not directly support the storage of more complex data structures. Moreover, they require full control over the induced topology as well as the storage distribution, thus proving unfitting for many realistic scenarios. Contrarily, unstructured overlays [14] offer more loose constraints on topology and data management (nodes are responsible for their own repositories), making them particularly appealing for our application.

The general approach of *BD* is the following: each vertex of the dwarf graph (henceforth termed as *dwarf node*) is designated with a unique ID (*UID*) and assigned to an overlay (or network) node. We assume that each network node n is aware of the existence of a number of other network nodes, which form its *Neighbor Set*, NS_n. Adjacent dwarf nodes are stored in adjacent network nodes in the P2P layer by adding overlay links. Thus, each edge of the centralized structure represents a network link between n and a node in NS_n. Each peer maintains a *hint* table, necessary to guide a query from one network node to another until the answer is reached and a *parent* list, required by our replication process to avoid inconsistencies.

The *hint* table is of the form (*currAttr, child*), where *currAttr* is the current attribute of the query to be resolved and *child* is the UID of the dwarf node which the *currAttr* leads to. If the dwarf node containing *currAttr* constitutes a leaf node, *child* is the aggregate value. The parent list contains the UID(s) of the dwarf node's parent node(s) along with the *currAttr*, whose child led to the specific node. In order to route messages among network nodes, each of the peers maintains a routing table that maps UIDs to *NIDs* (i.e., network IDs, e.g., IP address and port).

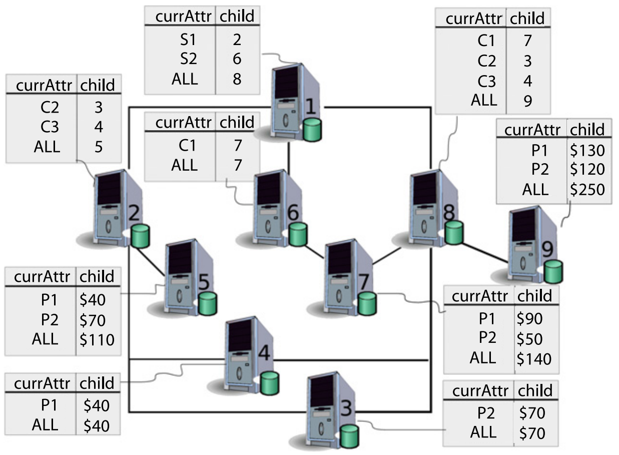

Fig. 3. The distribution of the dwarf nodes in the Brown Dwarf of Table 1 and their *hint* tables.

3.1. Insertion

The creation of the data cube is undertaken by a specific node (*creator*), that has access to the fact table. The creator follows the dwarf insertion algorithm, distributing the dwarf nodes on-the-fly during the tuple-by-tuple processing, instead of keeping them in secondary storage. In general, the creation of a cell in the original dwarf corresponds to the insertion of a value under *currAttr* in the hint table. The creation of a dwarf node corresponds to the registration of a value under *child*. Thus, all distinct values of the cells belonging to a dwarf node are eventually registered under *currAttr*. Moreover, the node each *currAttr* points to is kept under the *child* attribute. In the case of a dwarf leaf node, *child* corresponds to the measure or the aggregate value.

Let d be the number of dimensions and $t_1 = \langle a_1, a_2, \ldots, a_d \rangle$ be the first tuple of the fact table. Upon processing of t_1, a_1 triggers the creation of the root node, meaning that a network node from the creator's NS is allocated (let it be node N_{root}). A new hint table is created and stored in N_{root} under a randomly chosen UID. At this point, only the *currAttr* can be filled in with a_1. Moreover, the parent list of N_{root} is set to null (as the root node has no parents). Moving to a_2, a new node is allocated from the neighborhood of N_{root} and a new hint table is created following the previous procedure. The UID of N_{root} is appended to the list of parents of the newly allocated node and the UID of the node is added to N_{root}'s hint table under a_1. The same procedure is followed by all dimension attributes of t_1 (plus the special *ALL* attribute wherever needed). As tuples of the fact table are being processed one by one, new hint tables are created and existing ones are gradually modified (see Algorithm 1).

Note that our insertion mechanism does not demand the creation of the centralized dwarf. Nodes are created and hint tables are filled in gradually as tuples are processed. The only information the creator needs to hold at each moment is that of d dwarf nodes (the nodes of the path that t_i traverses).

For the first tuple of Table 1, the corresponding nodes and cells are created on all levels of the dwarf structure (Fig. 2). Each of the created nodes (1), (2), (3) are assigned to respective overlay nodes. In the hint table of (1), S_1 is placed under *currAttr* and (2) under *child*. Following the same procedure, the routing table for (2) is filled in with C_2 and (3) and that of (3) with P_2 and \$70 (the measure attribute, since it is a leaf node). Insertion moves on to the next tuple, which shares only prefix S_1 with the previous one. This means that the C_3 needs to be inserted to the same node as C_2, namely (2), and (4) needs to be allocated. Thus, C_3 must be registered in the node's hint table as a new *currAttr* and (4) as a new child value. Moreover, (3) is now closed, so *ALL* along with

Algorithm 1 BD Insertion

```
$t_{\text{curr}}$: the tuple to be inserted, $t_{\text{pre}}$: the tuple previously inserted

$N_{\text{creator}}$: the node that initiates the insertion
while unprocessed tuples exist do
  $t_{\text{cur}} \leftarrow$ the next tuple of the fact table
  while next attribute of $t_{\text{cur}}$ exists in $t_{\text{pre}}$ do
    find the network node $N_{\text{last}}$ that holds the corresponding dwarf node
  end while
  add the first uncommon attribute to the hint table of $N_{\text{last}}$ under currAttr
  for all remaining attributes of $t_{\text{cur}}$ do
    create dwarf node and assign it a UID
    add UID to the hint table of $N_{\text{last}}$
    randomly pick a node $N_{\text{next}}$ from $NS_{N_{\text{last}}}$
    send dwarf node to $N_{\text{next}}$
    $N_{\text{last}} \leftarrow N_{\text{next}}$
  end for
  beginning bottom up add the ALL cells and create new dwarf nodes according to original SuffixCoalesce
end while
```

the aggregate value \$70 are registered in its hint table. Gradually, all necessary nodes are allocated and their hint tables are filled in with the appropriate routing information (see Fig. 3).

3.2. Query resolution

Queries are resolved by following their path along the *BD* system attribute by attribute. Each attribute value of the query belongs to a dwarf node which, through its hint table, leads to the network node responsible for the next one.

A node initiating a query $q = \langle q_1, q_2 \dots q_d \rangle$, with q_i being either a value of dimension i or *ALL*, forwards it to N_{root}. There, the hint table is looked up for q_1 under *currAttr*. If it exists, *child* will be the next node the query visits. The above procedure is followed until a measure is reached. Note here that, since adjacent dwarf nodes belong to overlay neighbors, the answer to any point or group-by query is discovered within at most d overlay hops. This happens because adjacent dwarf nodes may actually reside on the same peer.

From the above description, it is clear that the system requires an entry point, meaning that query initiators should be aware of N_{root}, where the resolution of any query starts from. This can be achieved through an advertising mechanism, invoked by N_{root} upon allocation. The existence of a single entry-point for *BD*, which constitutes a single point of failure, is tackled by our replication strategy, thoroughly described in the following sections.

Back to our example, let us consider the query S_1ALLP_2. Beginning the search from (1), and consulting the child value corresponding to S_1, we end up at (2). There, since the second dimension value is *ALL*, the query follows the path indicated by the third entry of the hint table, thus visiting (5). P_2 narrows the possible options down to the second entry of the hint table, namely \$70.

3.3. Incremental updates

The procedure of incremental updates is similar to the insertion process, only now the longest common prefix between the new tuple and existing ones must be discovered following overlay links. Once the network node that stores the last common attribute is discovered, underlying nodes are recursively updated. This means that nodes are expanded to accommodate new cells for new attribute values and that new dwarf nodes are allocated when

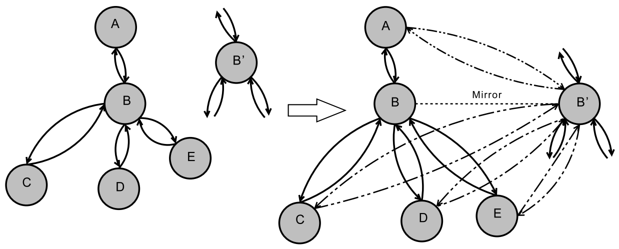

Fig. 4. Example of mirroring.

Algorithm 2 mirror

```
Require: $dn_s$: dwarf node to be replicated
  $N$: network node that hosts $dn_s$
  $N$ sends hint table of $dn_s$ to $N' \in NS_N$
  $N'$ informs parents and mirrors of $dn_s$
```

necessary. Moreover, the insertion of new tuple to an existing *BD* affects the *ALL* cells of dwarf nodes associated with the updated nodes.

Assuming $u = \langle u_1, u_2 \dots u_d \rangle$ is the tuple to be added to an existing *BD*, the incremental update procedure starts from the root of the structure following the path designated by u_1, u_2 etc. Once the dwarf node containing the last attribute that is already present u_i is discovered, a new entry for u_{i+1} must be registered to the node where the child of u_i points to. The following attributes $(u_{i+2} \dots u_d)$ will trigger the creation of new dwarf nodes. The special *ALL* cells are recursively updated for all nodes affected by the change.

It is obvious that the update procedure is bandwidth-costly, since the insertion of new tuples trigger the updating of multiple dwarf nodes and cells, therefore we assume it is invoked when several batches of updates are collected.

3.4. Mirroring

In order to ensure availability and eliminate single points of failure, especially in the case of N_{root} that represents the single entry point for queries in our system, we assume a global replication parameter k. This parameter defines the degree of data redundancy in the system: During the insertion phase, each dwarf node is stored in $k + 1$ network nodes instead of just one. Thus, in its initial state, the system hosts $k + 1$ instances (*mirrors*) of each dwarf node. The query forwarding algorithm is now amended: a node that receives a query randomly chooses from a list of mirrors the one to forward the query to.

To achieve correct behavior after a mirroring operation, the parent, children and mirror nodes of the original node(s) must be informed of this creation: parents must know of the new node in order to include it in the forwarding process. The mirror node must be informed of the peers that it precedes according to *BD* as well as its parent(s). Finally, the children must be informed of this new parent. Fig. 4 describes this process pictorially: node B chooses to replicate one of its dwarf nodes to B', invoking mirror (Algorithm 2). B' receives the hint table of B regarding the specific dwarf node and is thus informed of A as well as C, D and E, creating overlay links to them accordingly (parent and children). Node A is informed of the new child, adding B' to its list of children for this dwarf node. Moreover, parent links are also updated (shown in dotted lines). With the exchange of hint tables between B and B', the latter is able to discover all other mirrors and notify them of its existence.

From that point on, the system is responsible for preserving the number of each dwarf nodes' *mirrors* above k. To validate their (un)availability, *mirrors* periodically ping each other. This is also achieved through normal queries, when forwarding to a node that fails to reply. If a network node perceives that the *mirror* of a dwarf node it hosts is unavailable (because of a network failure for example), it initiates the mirroring operation for that dwarf node through invocation of the `mirror` function: the node chooses another peer in order to replicate the dwarf node to it. The node to receive the replica can be chosen either from the pool of locally known peers or using another independent service. It is important to note here that the whole process does not affect the behavior of the system, as all queries continue to be normally resolved.

In the event of updates over replicated nodes, the following strategy is adopted: during the insertion of a new tuple, only one mirror of each affected node is updated. The rest are updated asynchronously through the pinging process, where updated nodes propagate their changes to their mirrors. Since data are read-only, updates generally translate to the addition of new (*currAttr, child*) pairs, therefore the merging of updates between two mirrors of the same dwarf node is trivial. The only case where the *child* of an existing *currAttr* changes is when a suffix redundancy is destroyed. In this case, the mirrors resolve any conflict by choosing the change with the latest timestamp. This scheme offers eventual consistency, which we prefer for our target applications over a more strict and costly scheme that synchronously updates all mirrors.

3.5. Handling node failures and query skew

A basic requirement for every distributed application is fault-tolerance. Node clusters typically consist of commodity, failure-prone hardware. Especially in the case of data analysis, where complex workloads can take hours to complete, the probability of a failure occurring becomes higher. Apart from node churn, data skew is another factor that stresses the system's ability to operate smoothly, as it can degrade the performance of overloaded nodes, having a disproportionate effect on total query latency. In *BD*, we utilize a replication scheme adaptive to both skew and node churn to address both issues in a unified way, *expanding* popular or unavailable elements of the structure and *shrinking* others that receive few requests.

3.5.1. Node churn

When a node wishes to leave the system, it initiates the `graceful-depart` function, given in Algorithm 3: for each of the dwarf nodes it stores, the respective parents and children are notified to revise their links. The same is true for the list of mirror nodes. Messages can be grouped per recipient since, in the majority of cases, we anticipate that some nodes will be parents for multiple dwarf nodes or both parents and children.

Given this process, we may now describe how random departures or node failures are handled. A departed node is discovered either through the periodic ping procedure, or through query routing. In the first case, the mirror node receives no acknowledgment from the departed node within a period of time, considering it unavailable. It then initiates `graceful-depart` with the failed node's NID as a parameter. In the second case, the departed node is, at some point, selected during the routing process. The parent node will be informed of this failure after a timeout occurs. The parent then forwards the current query along with a `graceful-depart` request to another child, mirror of the departing node (relative to the queried dwarf node). Besides processing the query, this mirror initiates `graceful-depart` with the failed node's NID as a parameter. In the end, all of the failed node's parents, children and mirrors will be notified of this event and update their links.

3.5.2. Load-driven mirroring

In *BD*, network nodes utilize adaptive mirroring according to the load received on a per-dwarf node basis. A network node hosting an overloaded dwarf node can create additional mirrors through the *expansion* process. The node to receive the new mirror can be

Algorithm 3 `graceful-depart`

```
Require: $dn_s$: dwarf node to be deleted
  $N$: network node that initiates deletion
  $N$ informs parents, children and mirrors of $dn_s$
  $N$ deletes hint table of $dn_s$
```

Algorithm 4 BD Adaptive Mirroring

```
  $DN_N$: the set of dwarf nodes that network node $N$ hosts
  $k$: the replication degree, $r_s$: number of mirrors for $dn_s$
  for all $dn_s \in DN_N$ do
    $l_s(t)$ the current load for $dn_s$
    if $l_s(t) > \text{Limit}^s_{\text{exp}} \vee r_s < k$ then
      for $i = 1$ to $\max(\lceil l_s(t)/\text{Limit}^s_{\text{exp}} \rceil, k - r_s)$ do
        mirror($dn_s$)
      end for
    else if $l_s(t) < \text{Limit}^s_{\text{shr}} \wedge r_s > k + 1$ then
      graceful-depart($dn_s$)
    end if
  end for
```

chosen from the node's *NS* either randomly, or following some more advanced policy, which takes into account parameters like storage, utilization and load. Such a policy could, for instance, dictate the selection of the most underloaded peer, or the peer with the largest amount of free disk space. The newly created mirror will be used by the parent node(s) in order to receive some of the requests. In the opposite case, an underloaded dwarf node can be deleted from the system through the *shrink* process, as long as the total number of its mirrors remains over *k*.

These procedures require that each peer participating in *BD* monitors the incoming load for each of the dwarf nodes dn_s, $s \in (i, i+1, \ldots, j)$, it hosts. Let $l_s(t)$ be the current load for dn_s. The two procedures can be described as follows:

`Expansion`: As load increases due to incoming requests, some dwarf nodes reach their self-imposed limits, which we assume are expressed by parameter $\text{Limit}^s_{\text{exp}}$: it represents the maximum number of requests that dwarf node dn_s can accept per time unit. When this limit is exceeded, the hosting node invokes `mirror` in order to replicate it according to its demand and relative to $\text{Limit}^s_{\text{exp}}$. Specifically, each node dn_s, with $l_s(t) > \text{Limit}^s_{\text{exp}}$ will be replicated $\lceil l_s(t)/\text{Limit}^s_{\text{exp}} \rceil$ times. This mechanism allows for adaptive expansion of the network-wide storage according to demand and helps overloaded nodes to offload part of their workload to other server instances.

`Shrink`: Temporal changes in workload may result in the creation of mirror nodes which eventually become underutilized. The system should be able to delete such nodes, provided that their deletion will not result in less than $k+1$ mirrors. Assuming $\text{Limit}^s_{\text{shr}}$ is the limit, under which dn_s is considered underloaded, and r_s is the number of the mirrors of dn_s that the storing node is aware of, then each dn_s with $l_s(t) < \text{Limit}^s_{\text{shr}}$ and $r_s > k+1$ will be deleted, through `graceful-depart`. To ensure that the deletion of dn_s will not cause the overloading of its mirrors, we estimate $\text{Limit}^s_{\text{shr}}$ using the following rationale: when deleting a replica, we get from r_s to $r_s - 1$ mirrors. Estimating the total load for dn_s to be $r_s l_s(t)$, we require that $l_s(t') \simeq r_s l_s(t)/(r_s - 1) < \text{Limit}^s_{\text{exp}}$. Thus we choose $\text{Limit}^s_{\text{shr}} = \text{Limit}^s_{\text{exp}} \cdot r_s/(r_s + c)$, where c is a positive constant.

In essence, $\text{Limit}_{\text{exp}}$ and $\text{Limit}_{\text{shr}}$, relative to the context and values that are assigned to them, implement the application's policy with respect to the quality of service. Indeed, they regulate how reactive (and thus query-efficient) we want the application to be at the cost of more or less storage and data transfers. A more formal description of the adaptive mirroring algorithm can be seen in Algorithm 4.

4. Optimizations—discussion

BD utilizes a series of optimizations to enhance its performance by reducing communication costs.

4.1. Query performance optimization

In general, we expect the number of dwarf nodes created to be much larger than the number of participating hosts, with each peer hosting multiple dwarf nodes. This fact, along with the load-driven expansion and shrink of the system may result in individual peers hosting dwarf nodes that are connected in the logical level (a dwarf node and some of its children's mirrors). In this case, a query resolution that follows a random path from a dwarf node to one of its children may result in unnecessary message exchange between peers. Alternatively, upon reception of a query, the peer can choose from the list of children the one that resides in the same node, thus reducing the network messages needed.

4.2. Dimension grouping optimization

This optimization intends to reduce the communication cost between network nodes during all the operations of *BD*. Instead of storing the data structure at the dwarf node granularity, the system chooses to group related dwarf nodes and store them together as an entity. Due to the lack of an a priori knowledge of the graph (since it is constructed and distributed on the fly) and the distributed nature of our system, we employ a simple heuristic: starting from the leaf nodes of the structure, we create groups that contain sub-graphs of height h, $h < d$. This practically means that a dwarf node *dn* and all its descendants of depth h reside at the same physical node. Nevertheless, this might not be the case for dwarf nodes along an aggregate path, since an *ALL* cell might point to an existing dwarf node that belongs to a different group.

Both data load and update procedures are now affected, since a group, rather than an individual dwarf node, is assigned to a network node. During querying, the number of network hops until the answer is reached is significantly reduced. For point queries, the number of hops is $\lceil \frac{d}{h} \rceil$, while for aggregate queries it may range from $\lceil \frac{d}{h} \rceil$ to d at most. Replication takes great benefit from this enhancement as well, since the memory needed for statistics as well as the overall replication cost is reduced. However, it must be noted that such grouping may also reduce the effect of decentralization and thus the potential of the system to exploit parallelization (relative to the choice of h and the structure of the dwarf graph). This approach is deemed most beneficial for high-dimensional datasets with the value of h set significantly smaller than that of d.

4.3. Consistency issues

The expand–shrink scheme raises some consistency issues relating to the precision of the information that each node has of the available mirrors. Obviously, an almost concurrent creation of two or more mirror nodes or a series of expansions and shrinks at different parts of the overlay may result in nodes with different and incomplete knowledge of both the number and the identities of available mirrors. Yet, these inconsistencies are temporary. The higher the query rate that triggers such occurrences, the larger the probability that newly created or unavailable replicas are discovered through the query forwarding process. Moreover, it is reasonable to assume that, for our target applications, we do not expect a high churn rate from participating peers and that some nodes (possibly the initial ones) will be more stable, server-like nodes that rarely disconnect. The granularity of the replication scheme as well as its consistency guarantees under various conditions are also issues that concern *BD*'s performance and are subjects of future work.

5. Experimental results

We now present a comprehensive evaluation of *BD*. The system is developed entirely in Java, using the socket API for inter-node communication and deployed in an actual test-bed of $N = 16$ commodity nodes (dual core, 2.0 GHz, 2 GB main memory), which act as the storage/computation infrastructure. We assume that the *NS* of each node is set to 15, which practically means that all network nodes are aware of each other. The centralized approach has also been implemented for direct comparison.

In our experiments, we use both synthetic and real datasets consisting of a fact table representing multidimensional data with numerical facts. The synthetic datasets have been generated with our own and the APB-1 benchmark generator [5]. Our generator creates the tuples from combinations of the different dimension values (*cardinality*), plus a random numerical fact. Furthermore, we may choose to create tuples that combine dimension values uniformly or with bias (creating 80/20, 90/10 and 99/1 distributions and zipfian distributions with $\theta = 0.95$). The aggregate function used in the results is `sum`. The real and the APB datasets are described in the corresponding sections.

For the application workloads, we include both point and aggregate queries with varying proportions/distributions as well as batch updates. We either query the available dimension values uniformly or with skew, following the zipfian distribution with various θ values.

5.1. Cube creation

In the first set of experiments, we evaluate the creation of the distributed *BD* structure in terms of construction time, storage and communication. We also prove the fairness of the data distribution over various datasets and local neighborhood information.

5.1.1. Varying the number of dimensions

Assuming no replication (i.e., $k = 0$), we construct *BD* and *Dwarf* cubes with variable number of dimensions d (5 up to 25), with cardinalities equal to 1 k values. The datasets consist of 10 k tuples, following uniform, self-similar (80–20) and zipfian ($\theta = 0.95$) distributions. For the *BD* evaluation, the experiments are conducted in our distributed testbed, consisting of 16 LAN nodes. Storage consumption and insertion times are presented in Table 2.

Our system exhibits impressively faster creation compared to the centralized method, due to the fact that *BD* allows for overlapping of the store process (each peer stores its part of the cube independently). The acceleration is more apparent as the number of dimensions and the skew grows, since such datasets result in larger cubes. For instance, *BD* inserts the 25-*d* skewed cubes up to 3.5 times faster than *Dwarf*. The acceleration factor of course is not directly proportional to the number of participating nodes. The cube calculation remains serial and network communication introduces latencies. Experimenting in WAN environments with various latencies as well as parallelizing the cube calculation process is a subject of future work.

Note that the total cube size is always bigger than the fact table by a factor that increases with dimensionality and skew (152 times for the central and 195 times for *BD* for the worst case). This observation confirms previous findings documenting that *Dwarf* blows up the size of some datasets. In addition to that, *BD* induces a small storage overhead. This overhead is mainly attributed to the mapping between the UIDs (set to 4 bytes each in our implementation) that every Dwarf node needs to keep in order to be accessible by network peers and Dwarf node IDs, as well as the parent list for the `mirror` process. This also explains why the overhead slightly increases with the number of dimensions. Nevertheless, this overhead is shared among the participating

Table 2
Storage requirements and creation time for *Dwarf* and *Brown Dwarf* data cubes of various dimensionalities.

d	Fact Tbl. (MB)	Uniform				80–20				Zipf			
		Size (MB)		Time (s)		Size (MB)		Time (s)		Size (MB)		Time (s)	
		Dwarf	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
5	0.2	1	1	4	4	1	1	8	7	1	1	3	4
10	0.4	4	5	31	10	4	5	28	14	6	7	54	21
15	0.6	7	9	63	29	10	13	96	43	22	27	226	74
20	0.8	13	17	122	55	18	23	352	82	54	69	543	204
25	1.0	18	23	198	88	29	37	729	196	152	195	1206	535

Table 3
Creation time (in s) for *Brown Dwarf* data cubes of various dimensionalities varying the *k* parameter.

d	*k*			
	0	1	2	4
5	4	5	6	8
10	10	13	19	23
15	29	36	49	76
20	55	75	104	152
25	88	110	150	220

Table 4
Effect of various dataset sizes on 5-*d* cubes.

# Tuples	Size (MB)		Time (s)	
	Dwarf	*BD*	*Dwarf*	*BD*
10 k	1.2	1.5	4	4
100 k	4.8	5.9	71	20
1 M	78.2	96.4	559	79
10 M	402.9	482.1	18 188	498

Table 5
Effect of various densities on 5-*d* cubes.

Dens (%)	Size (MB)		Time (s)	
	Dwarf	*BD*	*Dwarf*	*BD*
0.01	5.4	6.6	54	13
0.1	4.6	5.5	39	9
1.0	3.0	3.5	19	5
10.0	1.4	1.6	5	3

nodes. Thus, the big advantage of *BD* is the fact that it can store almost *N* times as much data as *Dwarf* (for $k = 0$), using *N* computers similar to the central case.

Table 3 presents the cube insertion times for the uniform datasets when the replication parameter *k* ranges from 0 to 4. Although the cube is inserted $k + 1$ times and the storage consumption as well as the communication cost sustain a *k*-fold increase, the increase in the total insertion time is not proportional to *k*. Once again, parallel disk I/O operations alleviate the impact of the linear increase in the size of the data to be stored, resulting in an average 2–2.5 factor increase in insertion times in the worst case (comparing $k = 0$ and $k = 4$).

5.1.2. *Varying the number of participating nodes*

To examine the gains in insertion time caused by the parallelization of the storage process and to analyze the evolution of the communication cost, we vary the number of participating commodity nodes. As input data, we use two 5-*d*, uniformly distributed cubes, consisting of 100 k and 500 k tuples respectively. Cardinalities for all dimensions are set to 100.

The graphs in Fig. 5 reveal on one hand that the increase in the number of participating nodes enhances the system performance (over 5 times faster insertion times). However, the speedup is not linear and there is a point beyond which no dramatic improvement is demonstrated. This is due to the fact that the serial nature of the cube creation algorithm poses a limit in the parallelization of the storage process itself, since Dwarf nodes cannot be stored faster than their calculation. On the other hand, larger number of nodes induce bigger total communication costs, as shown in the included graph. Still, the total number of messages per insertion shows similar behavior to that of the insertion time. Even though the communication cost increases, the overhead per network node decreases as messages are scattered among more peers.

5.1.3. *Varying the number of tuples*

We now examine how the *BD* cube behaves when scaling the number of tuples in the fact table. Keeping a constant $d = 5$ dimensions and all cardinalities equal to 100, we create datasets of 10 k, 100 k, 1 M and 10 M tuples. Table 4 summarizes the results. As the number of tuples increases, this leads to higher acceleration factors for our insertion method. While the insertion of the 10 k dataset lasts the same in both systems, the 10 M dataset is inserted almost 36 times faster. We observe that our system maintains roughly the same storage overhead for all dataset sizes, since the increase in the number of tuples results in an increase in storage for both *Dwarf* and *BD* cubes.

5.1.4. *Varying the density of the cube*

We now examine how the density of the data cube affects its insertion in the *BD* system. We create 5-*d* datasets of 100 k tuples following a uniform distribution, with variable densities ranging from 0.01% to 10%. We document the amount of storage allotted by each network node for *Dwarf* and *BD* and present it in Table 5. Our system remains faster in creating and distributing the structure by a constant factor of about four. The next observation is that the lower the density, the larger the total amount of storage for the *BD*, since sparse cubes leave little room for redundancies, thus resulting in larger Dwarf structures. Therefore the denser the cubes, the less noticeable the difference in storage between *Dwarf* and *BD*.

5.2. *Updates*

In this section, we observe the behavior of *BD* when update batches are to be inserted to the distributed structure using our real testbed of 16 nodes. Utilizing the same 10 k-tuple datasets of varying dimensions described before, we present measurements for two different settings. In the first setting, we apply 1% incremental updates which follow the uniform and the self-similar (80–20) distribution. In the second setting, we apply increments of sizes from 0.1% up to 10% to the 10-*d* cube. These increments are of two types, consisting of tuples generated either by combining existing dimension values (*old*), or by adding new values to the dimensions' domains (*fresh*). For both types, we record the total update time as well as the number of messages required per individual update. We present the results for these two settings in Tables 6 and 7 respectively.

Taking advantage of the inherent parallelization that updates (similar to insertions) exhibit, *BD* is up to 2.3 times faster in the high-dimensional sets. Dimensionality plays a big role in both the time and the cost of updates. This observation is clearly documented: the more the dimensions, the larger the *BD* created, thus the more Dwarf nodes and cells are affected (see Table 6). As

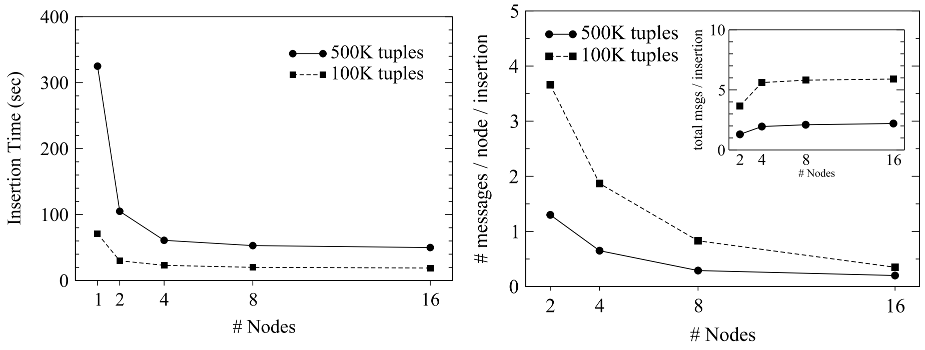

Fig. 5. Time and incoming messages per network node for various network sizes and data cubes.

Table 6
Effect of 1% increments over various dimensions.

d	Uniform			80–20		
	Time (s)		msg/upd	Time (s)		msg/upd
	Dwarf	*BD*	*BD*	*Dwarf*	*BD*	*BD*
5	7.1	7.2	14.6	7.5	6.4	13.7
10	17.7	14.3	50.8	21.3	14.4	49.8
15	30.8	21.8	111.0	43.4	31.2	120.4
20	48.6	27.9	193.3	104.1	65.8	200.2
25	89.1	39.1	300.7	172.1	103.6	305.7

Table 7
Effect of various update types and sizes on the 10-*d* dataset.

Size (%)	Time (s)		msg/upd	
	Old	*Fresh*	*Old*	*Fresh*
0.1	1.9	1.8	49.5	40.2
1.0	14.3	11.6	50.8	41.5
10.0	127.2	78.0	61.2	56.6

Table 8
Query resolution times and communication cost over various 1 k querysets.

d	Uniform			Zipf		
	Time (s)		msg/query	Time (s)		msg/query
	Dwarf	*BD*	*BD*	*Dwarf*	*BD*	*BD*
5	5.2	4.0	5.8	1.9	1.7	5.5
10	30.1	2.6	10.9	29	1.2	10.6
15	65.2	2.9	15.6	55.4	1.2	15.5
20	102.1	3.0	20.8	88.3	1.5	20.3
25	182.5	13.2	25.9	172.1	9.2	25.6

observed in the case of cube creation, skewed datasets take longer to update, due to the fact that updates in a dense part of the cube affects more Dwarf nodes and cells, thus slowing down the process and creating larger network traffic.

Table 7 shows that the size of the update over the original cube has negligible effect on the communication cost per update. However, it is interesting to note that while *fresh* updates over the original cube create more nodes and cells in the structure, yet the cost is inversely proportional. The number of update messages per update are almost 20% less compared to *old* updates. This is due to the fact that the *fresh* batch contains new attribute values and fails to find redundancies with the originally inserted *BD* structure. This results in more new nodes, yet less recursive updates of the affected *ALL* values, hence less messages.

5.3. Query processing

In this section we investigate the query performance of *BD* compared to that of *Dwarf* and examine the effect of adaptive mirroring on the load distribution among the system nodes.

5.3.1. Varying the number of dimensions

Using the same datasets as in the insertion and update experiments, we pose two 1 k querysets that follow the uniform and zipfian ($\theta = 0.95$) distributions respectively, with the ratio of point queries set to 0.5. Moreover, P_d, which we define as the probability of dimension i not participating in a query (i.e., $q_i =$ *ALL*), is set to 0.3. Table 8 summarizes the results. It should be noted that the Dwarf index does not remain in memory for either method, thus I/O is performed for every query.

We first notice that, in all cases, *BD* resolves the workload noticeably faster than the centralized version. While the query response times rise with the dimensionality for *Dwarf*, *BD* times remain almost constant and only the 25-*d* workloads cause a slight slowdown. The resolution of each dimension of the query is an atomic operation that may be performed by separate peers. Thus, having 16 nodes perform I/O operations in parallel instead of just one significantly boosts performance. Especially in the case of biased and high dimensional workloads, where there is more room for parallelization, *BD* exhibits impressive acceleration factors, performing up to 60 times faster than the original *Dwarf*. It is thus apparent, that *BD* is able to handle a significantly (by orders of magnitude) larger request rate than its centralized version.

Moreover, the number of messages per query is in all cases bound by $d+1$: d messages to forward the query to the Dwarf nodes along the path toward the answer and one to send the response back to the initiator.

5.3.2. Varying the number of nodes

For the 20-*d* dataset of the previous experiment, we plot the response times and the per query communication when scaling the number of network nodes from 1 (centralized) to 16. Apart from the uniform workload of 1 k queries used before, we also pose a workload of 10 k queries with the same characteristics, to further stress the system. Fig. 6 pictorially presents the results.

The first graph plots total query response times. As observed, an initial increase in nodes dramatically accelerates responses. The performance gains become smaller as the size grows. This is due to the fact that, depending on the specific dataset and its dimensions, the parallelization ability of *BD* is saturated after a certain number of nodes. Yet, larger overlays with more than a single entry point (i.e., using mirroring for N_{root}) will scale better in the case of high-rate workloads. The second graph presents the average number of messages needed to answer a query for both workloads. We notice that as the number of peers increases, the communication cost increases too, but not uncontrollably, since it converges to $d+1$ messages per query. Furthermore, the cost is scattered among peers, resulting in less load per network node.

5.4. Benchmarks and real datasets

Next, we examine the behavior of *BD* with more realistic input sets. We utilize 6 different datasets. Using the APB-1 data generator

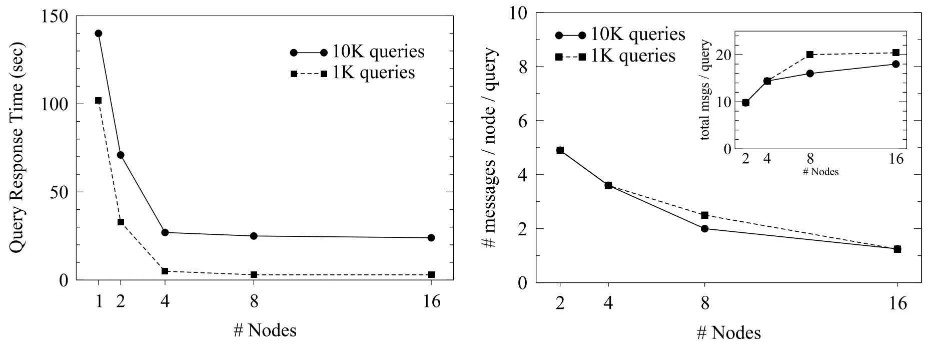

Fig. 6. Resolution time and messages per network node for various network sizes and query workloads.

Table 9
Measurements for various APB datasets.

Density	#Tuples (M)	F.Tbl size	Size (MB)		Time (s)		Qu.time (s)	
			Dwarf	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
0.1	1.2	24 MB	17	20	42	16	40	12
0.5	6.2	131 MB	84	98	314	48	93	13
1	12	240 MB	167	195	612	89	107	14
5	62	1.3 GB	821	965	3204	247	135	14

Table 10
Measurements for the real datasets.

Dataset	Size (MB)		Time (s)		*A* (s)		*B* (s)		*C* (s)	
	Dwarf	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
Weather	9.3	11.4	120	23	234	11	165	12	114	12
Forest	8.0	9.8	66	20	144	11	111	11	70	12

and different densities, we produce 4 datasets ($d = 4$) with dimension cardinalities 9000, 900, 9 and 24 and one measure attribute. The other 2 datasets are subsets of the *Weather* [36] and *Forest* [7] datasets (10 k tuples each). The forest dataset has 10 dimensions with cardinalities as reported in [11], while the weather dataset has 9 dimensions, corresponding to truncated ocean weather measurements for Sept. 1984.

We use the APB query generator to produce 1 k query workloads, both point and aggregate ones. For the real datasets, in order to produce our workloads, we first order the tuples and then use the zipfian distribution to select those that will form 10 k querysets. We vary θ from 0 (uniform distribution) to 2, producing three workloads (denoted as *A*, *B* and *C* respectively). The ratio of point queries is set to 0.5, while for aggregate ones $P_d = 0.3$.

The results, presented in Tables 9 and 10, are in line with the findings of the previous experiments. First, we notice that the Dwarf algorithm can, depending on the input data, perform efficient compression of the cube. The storage overhead is at most 144 MB (for the APB dataset of 1.3 GB), a steady 17% increase compared to the centralized case. Nevertheless, the cube is now shared among each of the 16 peers participating in the system. For the construction times, *BD* is obviously faster than *Dwarf*. Our results show that the distributed version is 13 times faster compared to the centralized run, giving impressive cube creation times (about 4 min for the APB dataset of 62 million tuples). Query response times are up to 20 times faster for *BD*, which is able to handle almost 1 k queries per second.

5.5. Adaptive mirroring

Using a 10-*d* cube with 10 k tuples (uniformly distributed) and pose 5 k query-sets, following the uniform and the zipfian (with various θ values) distributions respectively, with the ratio of point queries set to 0.5 and $P_d = 0.3$. Queries arrive at an average rate of $\lambda = 100$ queries/s. Fig. 7 displays the number of the created replicas as well as the induced system-wide communication overhead over time for different query-set distributions and for $\text{Limit}^s_{\text{exp}} = 10$.

In all cases, our scheme increases the number of replicas at overloaded parts of the structure through its *expansion* mechanism in order to bring the system to a balance and eliminate the instances of overloaded Dwarf nodes. The more skewed the query-set, the more replicas *BD* produces. This is natural since, as we showed before, overloaded Dwarf nodes for skewed distributions have substantially higher load. The rate at which replicas are created decreases with time and reaches a steady state where the number of mirrors remains almost constant. It is worth noticing that *BD* reaches the steady state fairly quickly (within a few seconds—less than 10 in our experiments), due to the ability of the *expansion* mechanism to create multiple mirrors according to the amount of overload. However, until steady state is reached, we observe a short period of fluctuation in the number of replicas, which is more apparent for workloads of high skew. The simultaneous initiation of the mirror process leads to temporary inconsistencies with regard to which mirror each node knows of. In this case, it takes some time until all mirrors discover each other, creating an uneven load distribution among them. It is also very important to stress that the mirroring process diminishes substantial load inequalities with minimal storage overhead. About 100 replicas are created at most in an initial *BD* structure of 130 k Dwarf nodes, which translates to less than 0.1% of extra storage consumption.

The second graph of Fig. 7 depicts the total number of control messages in the course of time. These are the messages required to inform parent, children and mirror nodes of the creation of a new replica or the deletion of an existing one, as well as the insertion message itself in case an *expansion* occurs. As expected, *expansion* and *shrink* comes with a certain communication cost and the more the replicas created or deleted, the more the control messages required. However, this message burst lasts only for a short period of time, until the system reaches a steady state. Moreover, the communication cost is shared among the resources.

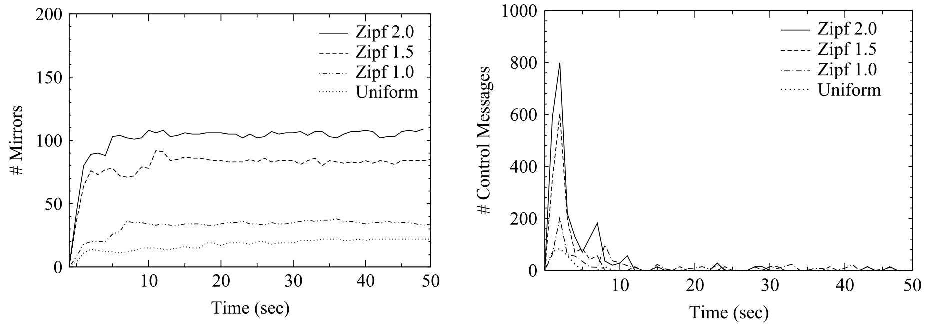

Fig. 7. Number of replicas over time and control message overhead for different query distributions (adaptive mirroring, with $\text{Limit}^s_{\text{exp}} = 10$).

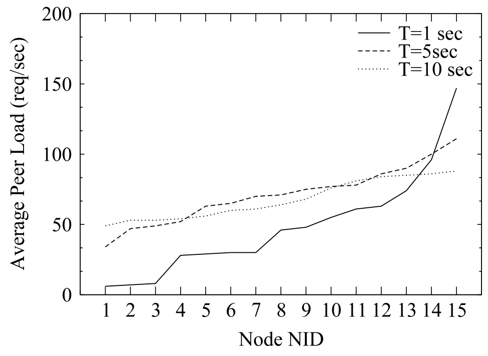

Fig. 8. Load distribution before, during and after mirroring (Zipf $\theta = 1.5$, $\text{Limit}^s_{\text{exp}} = 10$).

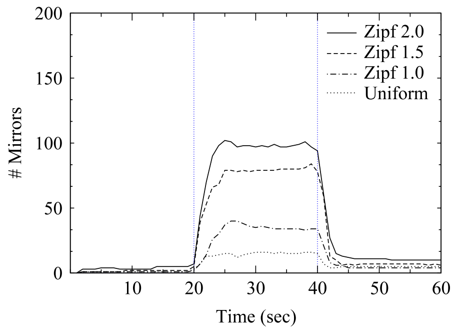

Fig. 9. Number of replicas over time for a pulse-like query rate with $\text{Limit}^s_{\text{exp}} = 10$.

To further examine the behavior of our system under stress conditions and sudden changes in load, we conduct another series of experiments. Using the same workloads and an initial query rate of 10 queries/s, we suddenly increase the query rate by a factor of ten (λ reaches 100 queries/s) after 20 s of querying time. After another 20 secs, the rate decreases again to its initial value. We evaluate the efficiency of the *expansion* as well as the *shrink* mechanisms to perceive the change and adapt the number of replicas accordingly, in order to perform as required with minimum storage consumption.

Fig. 9 presents the number of existing replicas over time throughout the simulation for $\text{Limit}^s_{\text{exp}} = 10$. Almost immediately after the increase in λ, the number of replicas increase rapidly, almost 10 times as much. After the end of the pulse, the *shrink* mechanism erases underloaded Dwarf nodes, freeing up disk space. Again, we observe how quickly *BD* manages to detect the change in load. Within a few seconds, the mirrors decrease dramatically and keep decreasing gradually, tending to reach the state that existed before the pulse was applied. However, the more biased the workload, the more the steady state before and after the pulse differs.

Fig. 8 also shows that *BD* moves toward a more balanced load distribution with each step. A load snapshot of a skewed workload ($\theta = 1.5$) at the beginning, the middle and a random point in the steady state show that our method manages to decrease disparity between node loads. That is its main advantage compared to static mirroring. The system gradually moves toward states where more server instances will be involved in query processing and less overloaded Dwarf nodes exist.

5.6. Node failures

Our system relies on the cooperation of commodity nodes forming an unstructured P2P overlay. It is likely that failures will occur throughout the execution of a workload, making it important to examine the impacts of such occurrences on our system. Using the 10-*d* dataset (with $k = 3$) and a uniformly distributed queryset of 5 k queries that arrive at a rate of 10 queries/s, we enforce node failures as follows: every T_{fail} sec, a subset N_{fail} of the online peers fails in a circular way, while previously offline nodes are reinserted to the network. Note that, by failing we mean that nodes depart ungracefully, without informing any other peer. Starting from $|N_{\text{fail}}| = 1$ we gradually increase it up to the value of 4 (since each Dwarf node exists in $k + 1 = 4$ different network nodes), aiming to test *BD*'s fault tolerance and examine its performance under volatile conditions. The adaptive mirroring mode is turned off, in order to better interpret the results. It is worth noticing that the parameters used in these experiments are far more pessimistic than the reported ones. Google, for instance, experiences failures in 0.76% of the average number of machines allocated for an analysis job, with T_{fail} being almost 500 s on average [10]. Table 11 summarizes our findings. Note that the query time column represents the absolute time for a single query to be resolved (not the average completion time of many queries run in batch mode).

We observe that the system maintains the theoretical guarantee that for any failure level below the replication no data/query loss will occur (see lines with $|N_{\text{fail}}| < 4$). Even when 25% of the nodes fail a very small portion of the queries has to be restarted (less than 5% in the worst case). This happens because, for a query *all* the replicas of at least one the respective Dwarf nodes that reside on the answering path must be offline in order for the query to fail. Because of the automatic replenishment of the replica-set whenever a Dwarf node falls below $k + 1$ copies, query loss is very small. The number of messages needed per query now tops *d* due to the redirections needed for some requests. Redirections and mostly the induced timeouts increase the average response time by a factor of roughly 13 compared to the no-failures run. Still, this number does not incorporate the query resolution speed-up *BD* exhibits when many queries are sent in batches.

In Fig. 10, we plot the total number of replicas in the system over time, for the various $|N_{\text{fail}}|$. We observe that the number of mirrors remains stable, despite of the random node departures and very close to the initial value for $k = 3$ (represented by the horizontal red line). In fact, the number of replicas is somewhat larger than the theoretical value. This happens because more than one peers might initiate a mirroring process at the same time, thus producing more replicas. This becomes more obvious as the ratio of failing

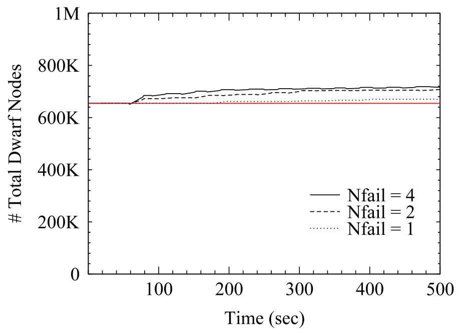

Fig. 10. Number of replicas over time with various numbers of failing nodes.

Table 11
Implications on data and query processing for increasing number of failures and different T_{fail} values.

$\lvert N_{\mathrm{fail}}\rvert$	T_{fail} (s)	Query loss(%)	Total redir	msg/q	q.time (ms)
0	–	0	0	9.8	57
1	90	0	11	9.8	79
2	90	0	204	10.4	257
4	90	2.9	841	11.1	734
1	60	0	21	9.9	107
2	60	0	258	10.5	304
4	60	4.3	894	11.2	812

nodes increases. The small fluctuations are due to the deletion of mirrors as nodes fail.

5.7. *Effect of dimension grouping*

Here we evaluate the impact of dimension grouping on performance as well as fairness in both storage and load. Using the 20-d dataset and varying the h parameter from 1 to 10, we apply a batch of 20 k updates in an existing *BD* structure of 100 k tuples. Afterward, we pose 1 k uniformly distributed queries. We measure the time and communication cost of the operations, as well as the value of the Gini coefficient G for both the storage consumption and the produced load (Table 12). G is a summary statistic that serves as a measure of inequality in a population. It is calculated as the sum of the differences between every possible pair of individuals, divided by the mean size. Its value ranges between 0 and 1, where 0 corresponds to perfect equality. Assuming our population comprises of the size of the stored data and the number of received requests by each node, we calculate the value of G as an index of storage and load distribution among servers respectively.

As h increases the communication cost of all operations noticeably decreases, since paths of h nodes reside in the same host, consequently reducing the operation times. Naturally, as the grouping becomes more coarse grained it causes imbalance in both storage and load. Although the effect on storage is much more severe, the load remains more balanced among servers, even in the exaggerated case of $h = 10$. This experiments proves the trade-off between performance gain and balance, suggesting a choice of h much smaller than d.

6. Related work

Brown Dwarf is a system for addressing the analytics of modern, network-centric enterprises. It extends traditional, well-established approaches to analytics and combines them with P2P techniques, being therefore relevant to several diverse fields of related works.

The sharing of relational data using both structured and unstructured P2P overlays is addressed in a number of papers. PIER [17] 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. On top of the DHT overlay, a Prefix Hash Tree (PHT) is built for secondary indexing. The PIER platform is also used along with a Gnutella overlay in [22] 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. In GroupPeer [19], 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 [25] also features relational data sharing without schema knowledge. Query matching and rewriting is based on keywords provided by the users. GridVine [1] hashes and indexes RDF data and schemas, and pSearch [31] represents documents as well as queries as semantic vectors. A work by Vaisman et al. [34] stressing the need for P2P OLAP 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 multidimensional data and aggregate queries over voluminous datasets.

Gray et al. introduced the data cube operator in 1997 [15]. 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. A basic problem related to the data cube is the complexity of its computation and the corresponding storage requirements, since the number of possible views increases exponentially to 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 [21,35,30]. Another approach is the DC-Tree [13], a fully dynamic index structure for data warehouses modeled as data cubes. In this work, the attributes of a dimension are partially ordered with respect to the valid hierarchy schema for each dimension. The DC-tree stores one concept hierarchy per dimension and assigns an ID to every attribute value of a data record that is inserted. These approaches are efficient in answering both point and aggregate queries over various data granularities but do so in a strictly centralized and controlled environment.

Closest to our distributed system are works that have proposed multiple cooperating data-warehouses. In [18], 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 the work of [4], the authors define the distributed data warehouse as a structure that consists of multiple local data warehouses adjacent to data collection points, and of 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-layered architecture consisting of multiple local data warehouses and a global one is proposed. WebContent [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. All these approaches perform some hybrid query processing model by allowing requests to route to different cites. Yet, unlike *BD*, they do not distribute the warehouse structure itself, with the processing cites remaining centralized.

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. This new class of analytics engines leverages the recent innovation in the industry around large-scale data management. Deployed on shared-nothing, commodity hardware architectures, they cover the newly added requirement for scalability, robustness and availability at low cost. The Pig project at Yahoo [26], the SCOPE project at Microsoft [8] and the open-source Hive project [33] mainly focus on

Table 12
Measurements for various values of *h* using the 20-*d* dataset.

h	Storage *G*	Update			Querying		
		Time/update (ms)	msg/update	*G*	Time/query (ms)	msg/query	*G*
1	0.01	153	35.6	0.05	109	17.2	0.05
2	0.20	148	30.3	0.13	38	8.6	0.08
5	0.39	102	23.4	0.29	15	3.9	0.18
10	0.61	91	20.8	0.30	13	2.3	0.32

language issues, addressing the creation of SQL interfaces on top of Hadoop [16]. 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. However, such technologies are inherently batch-oriented, as they can provide large amount of processing power, but do not guarantee per tuple processing nor real-time responses like *BD* does.

To conclude with relative solutions, parallel database solutions [27,32] offer great efficiency at the cost of elasticity and robustness in failures [28]. Indeed, resources cannot be automatically allocated (nor released) according to demand and the addition of new machines to the system requires significant effort as well as downtime. Lastly, parallel databases do not operate on heterogeneous environments. Contrarily, based on a shared nothing architecture, *BD* guarantees resilience as well as scalability on top of the advantages that a distributed storage offers, without compromising query and update efficiency. Its load-driven replication mechanism ensures data availability despite node failures.

7. Conclusions

In this paper we presented *Brown Dwarf*, a system that distributes a data cube across peers in an unstructured P2P overlay. To our knowledge, this is a unique approach that enables users to pose group-by queries and update multidimensional bulk datasets on-line, without the use of any proprietary tool.

Our system employs many plausible features required by an application and its respective hardware: it is scalable, as it can use an unbounded number of cooperating nodes, distributing computation and storage; it provides data availability through the adaptive replication scheme according to both workload and node failures; it efficiently answers all point and aggregate queries in a bounded number of steps; finally it is cost-effective, as it uses only commodity hardware, while with its expand–shrink scheme each dataset takes up only the necessary amount of storage.

Our evaluation shows that the data cube is evenly distributed across a number of cooperating peers. Both creation and querying times are significantly reduced (often by an order of magnitude) due to the parallel paths taken in the overlay. Moreover, it expands popular parts of the structure using local only load measurements while constantly monitoring the whole set to remain above an acceptable replication threshold. Finally, it minimizes node overloads and query processing times even in very demanding and dynamic workload/churn conditions.

References

[1] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, T. VanPelt, Gridvine: building internet-scale semantic overlay networks, Lecture Notes in Computer Science (2004) 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, VLDB'08, in: Proceedings of the 34th International Conference on Very Large Data Bases, vol. 1, VLDB Endowment, 2008, pp. 1428–1431.
[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin, HadoopDB: an architectural hybrid of Map-Reduce and DBMS technologies for analytical workloads, in: Proceedings of the 35th International Conference on Very Large Data Bases, VLDB'09, VLDB Endowment, 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.
[5] OLAP Council APB-1 OLAP Benchmark.
[6] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the clouds: a Berkeley view of cloud computing, Tech. Rep. UCB/EECS-2009–28, EECS Department, University of California, Berkeley, February 2009.
[7] J.A. Blackard, The Forest CoverType dataset. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype.
[8] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, SCOPE: easy and efficient parallel processing of massive data sets, VLDB'08, in: Proceedings of the 34th International Conference on Very Large Data Bases, vol. 1, VLDB Endowment, 2008, pp. 1265–1276.
[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] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107.
[11] J. Dittrich, L. Blunschi, M. Salles, Dwarfs in the rearview mirror: how big are they really?, VLDB'08, in: Proceedings of the 34th International Conference on Very Large Data Bases, vol. 1, VLDB Endowment, 2008, pp. 1586–1597.
[12] K. Doka, D. Tsoumakos, N. Koziris, Online querying of *d*-dimensional hierarchies, Journal of Parallel and Distributed Computing 71 (3) (2011) 424–437. http://dx.doi.org/DOI:10.1016/j.jpdc.2010.10.005.
[13] M. Ester, J. Kohlhammer, H. Kriegel, The DC-tree: a fully dynamic index structure for data warehouses, in: Proceedings of the 16th International Conference on Data Engineering, ICDE'00, IEEE Computer Society Press, 2000, pp. 379–388.
[14] Gnutella File-sharing and Distribution Network, 2003. http://rfc-gnutella.sourceforge.net/.
[15] 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.
[16] Hadoop Web Page. http://hadoop.apache.org/core/.
[17] 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, VLDB'03, VLDB Endowment, 2003, p. 332.
[18] 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, SIGMOD'02, ACM, New York, NY, USA, 2002, pp. 25–36.
[19] V. Kantere, D. Tsoumakos, T. Sellis, N. Roussopoulos, GrouPeer: dynamic clustering of P2P databases, Information Systems 34 (1) (2009) 62–86.
[20] E. Knorr, Dealing with the Data Explosion, Infoworld, 2009. http://www.infoworld.com/d/storage/dealing-data-explosion-690.
[21] 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, SIGMOD'03, ACM, 2003, p. 75.
[22] 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, VLDB'04, VLDB Endowment, 2004, p. 443.
[23] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: interactive analysis of web-scale datasets, VLDB'10, in: Proceedings of the 36th International Conference on Very Large Data Bases, vol. 3, VLDB Endowment, 2010.
[24] C. Monash, The 1-petabyte Barrier is Crumbling. http://www.networkworld.com/community/node/31439.
[25] W.S. Ng, B.C. Ooi, K.L. Tan, A. Zhou, Peerdb: a P2P-based system for distributed data sharing, ICDE'03, in: Proceedings of the 19th International Conference on Data Engineering, vol. 1063, IEEE Computer Society Press, 2003, pp. 633–644.
[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, ACM, 2008, pp. 1099–1110.
[27] Oracle Exadata. http://www.oracle.com.
[28] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, M. Stonebraker, A comparison of approaches to large-scale data analysis, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD'09, ACM, 2009.
[29] L. Siegele, Let it Rise, 2008. http://www.economist.com/node/12411882.
[30] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, Y. Kotidis, Dwarf: shrinking the petacube, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD'02, ACM, 2002, pp. 464–475.
[31] C. Tang, Z. Xu, S. Dwarkadas, Peer-to-peer information retrieval using self-organizing semantic overlay NSetworks, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, 2003, pp. 175–186.

[32] Teradata. http://www.teradata.com/.

[33] 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 Map-Reduce framework, VLDB'09, in: Proceedings of the 35th International Conference on Very Large Data Bases, vol. 2, VLDB Endowment, 2009, pp. 1626–1629.

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

[35] W. Wang, H. Lu, J. Feng, J. Yu, Condensed cube: an effective approach to reducing data cube size, in: Proceedings of the 18th International Conference on Data Engineering, ICDE'02, IEEE Computer Society Press, 2002.

[36] Kx Systems. http://www.kx.com/.

Katerina Doka is a post-doctoral researcher at the Computing Systems Laboratory of the Department of Electrical and Computer Engineering of the National Technical University of Athens (NTUA), involved in several European and National R&D projects. She received her Diploma in Electrical and Computer Engineering in 2005 and her Ph.D. in 2011 from NTUA. Her research interests lie in the fields of Data Management in Large Scale Distributed Systems, Peer-to-Peer Technologies and Cloud Computing and she has contributed to 13 academic publications in international conferences and journals. She is a member of the IEEE Computer Society, the ACM SIGMOD and the Technical Chamber of Greece. More information can be found in http://www.cslab.ece.ntua.gr/doka/.

Dimitrios Tsoumakos holds an academic position in the Department of Informatics of the Ionian University. He is also a senior researcher at 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). His research interests lie in the area of distributed systems/algorithms, particularly in designing and implementing adaptive, scalable and bandwidth-efficient schemes for data storage, retrieval and dissemination. Applications over Peer-to-Peer systems, Grid and Cloud Computing are considered. He is also actively involved in Database research, especially in designing distributed indexing schemes for shared databases, distributed RDF and datacube stores and open-linked data.

Nectarios Koziris, Associate Professor, received his Diploma in Electrical Engineering from the National Technical University of Athens (NTUA) and his Ph.D. in Computer Engineering from NTUA (1997). He joined the Computer Science Department, School of Electrical and Computer Engineering at the National Technical University of Athens in 1998. His research interests include parallel architectures, loop code optimizations, interaction between compilers, OS and architectures, communication architectures for clusters (OS and compiler support), OS virtualization, large scale computer and storage systems, cloud infrastructures, distributed systems and algorithms, distributed data management. Nectarios Koziris has co-authored more than 100 research papers in international refereed journals (including TPDS, TACO, JPDC, ParCO, JSC etc.) and in the proceedings of international conferences and workshops (including IPDPS, SC, PPoPP, ICPP, etc.). From 2002 he has been involved in the organization of more than 50 international conferences as Chair/co-Chair, Program Chair, Program Committee Member, Publicity Chair, including SPAA, IPDPS, ACM Supercomputing, ICPP, PDSEC & CAC workshops etc. He has also published two Greek textbooks "Mapping Algorithms into Parallel Processing Architectures", and "Computer Architecture and Operating Systems". Nectarios Koziris is the recipient of the IEEE IPDPS 2001 best paper award for the paper "Minimising Completion Time for Loop Tiling with Computation and Communication Overlapping" (held at San Francisco, California). He has been a project coordinator/key researcher in numerous EU (FP5, FP6 and FP7) and national Research Programmes, in the area of high performance computing and large scale computer systems. He is a member of the IEEE Computer Society, member of IEEE-CS TCPP and TCCA (Technical Committees on Parallel Processing and Computer Architecture), Senior Member of the ACM and chairs the Greek IEEE Chapter Computer Society. He also serves as the Vice-Chairman of the Board of Directors for the Greek Research and Education Network (GRNET-www.grnet.gr) and Vice-Chairman of the Board of Directors in EEL/LAK (Free/Libre/Open Source Software non-profit organization-www.ellak.gr), founded by the Greek Universities and Research Centers.