Distributing the Power of OLAP*

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

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

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

ABSTRACT

In this paper we present the *Brown Dwarf*, a distributed system designed to efficiently store, query and update multidimensional data over an unstructured Peer-to-Peer overlay, without the use of any proprietary tool. *Brown Dwarf* manages to distribute a highly effective centralized structure among peers on-the-fly. Both point and aggregate queries are then naturally answered on-line through cooperating nodes that hold parts of a fully or partially materialized data cube. Updates are also performed on-line, eliminating the usually costly over-night process. Our initial evaluation on an actual testbed proves that *Brown Dwarf* manages to distribute the structure across the overlay nodes incurring only a small storage overhead compared to the centralized algorithm. Moreover, it accelerates cube creation up to 5 times and querying up to several tens of times by exploiting the capabilities of the available network nodes working in parallel.

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Systems and Software—*Distributed systems*; H.2.7 [**Information Systems**]: Database Administration—*Data warehouse and repository*

General Terms

Design, Management

Keywords

Data Warehousing, P2P, Data Cube

1. INTRODUCTION

Data warehousing has become a vital component of every organization, as it contributes to business-oriented decision-making. Large companies, scientific organizations (*NASA*, *WMO*, etc.) or even 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. Data warehouses store vast amounts of historical and operational data (in the form of multidimensional *cubes*), mainly due to the automation of business processes, the growing use of sensors and other data-producing devices along with the globalization of markets. Their candidate workloads usually consist of read-only queries interleaved with batch updates.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

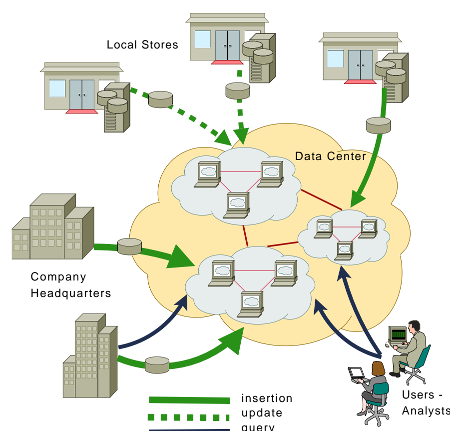

Figure 1: Motivating scenario of distributing a Data Warehouse

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. The challenge of scaling very large datasets along with the need for continuous data mining in order to detect real-time changes in trends, have given birth to the idea of creating distributed data-warehouse-like architectures.

As a motivating scenario, let us consider the computational center of a research or business establishment that maintains records over its operations. Instead of a centralized data warehouse, the management prefers a horizontal partitioning of the database (according to some metric, e.g., geographic), so that on-line queries on the multiple dimensions can be performed. Fig. 1 depicts a sample scenario where multiple establishments of a business insert, update and query such a distributed warehouse.

There are big challenges in proposing such a system. Centralized warehousing systems offer indexing schemes for storing and efficiently querying data cubes (e.g., [5, 6]), but only work in controlled environments, failing to scale. Some

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

DIM1	DIM2	DIM3	Measure
S1	C2	P2	$70
S1	C3	P1	$40
S2	C1	P1	$90
S2	C1	P2	$50

works in the field propose distributed warehousing systems (e.g., [1, 4]), 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 P2P computing [3], but with no a priori consideration for group-by queries.

Our goal is to create a distributed data warehousing system, where geographically spanned users, without the use of any proprietary tool, can share and query information. We intend to distribute a well known and highly efficient data structure, the *Dwarf* [6]. While the *Dwarf* 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, recent work [2] indicated that, depending on the cube's density, a *Dwarf* structure may take up orders of magnitude more space than the original tuples. To this end we propose the *Brown Dwarf*[1], a system that performs on-line distribution of *Dwarf* over network hosts in a way that all queries that were originally answered through the centralized structure are now distributed over the network. The distribution of the *Dwarf* structure relaxes its 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.

2. THE ORIGINAL DWARF

Dwarf [6] is a complete architecture for computing, storing, indexing, 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), while suffix redundancy occurs when some group-bys share a common suffix.

To better understand how *Dwarf* indexes the dataset and uses its properties to answer queries, we show in Fig. 2 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. Grey cells correspond to "*ALL*" values of that cell, used for aggregates on each dimension. Any group-by can be realized through traversing the structure and following the query attributes leading to a *leaf* node with the answer. For example, $\langle S1, C3, P1\rangle$ will return the $40 value while $\langle S2, ALL, ALL\rangle$ will return the aggregate value $140 following nodes (1)→(6)→(7).

[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.

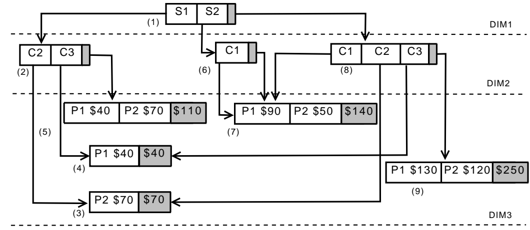

Figure 2: Centralized Dwarf for the fact table of Table 1, using the *sum* aggregation function

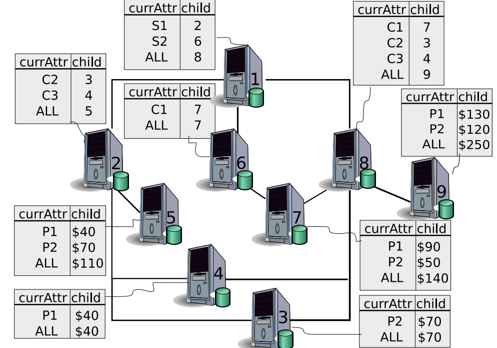

Figure 3: The distribution of the dwarf nodes in the Brown Dwarf of Table 1 and their *hint* tables

3. THE BROWN DWARF SYSTEM

The essence of *Brown Dwarf* is the distribution of the original, centralized dwarf structure over the nodes of an unstructured overlay in a way that guarantees equal storage and bandwidth consumption as well as query processing efficiency. The general approach 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. 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 the *currAttr* leads to. In case of a leaf node, *child* is the aggregate value. In order to route messages among network nodes, each of the peers maintains a routing table that maps UIDs to network IDs (e.g., IP address and port).

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.

Compared to the traditional *Dwarf*, *BD* offers significant advantages. The distribution of the *Dwarf* structure relaxes its overwhelming storage requirements and enables the computation of much larger cubes. Moreover, *BD* allows for on-line updates that can originate from any host participating in the dwarf structure.

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 algorithm of the original dwarf construction, distributing the dwarf nodes on-the-fly during the tuple-by-tuple processing, instead of keeping them in secondary storage. 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 $t_1 = \langle a_1, a_2 \dots a_d \rangle$ 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. 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 the newly allocated 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 the tuples are being processed one by one, new hint tables are created and existing ones are gradually modified.

Note that the proposed insertion mechanism does not entail an a priori 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, 4) must be registered in the node's hint table. Moreover, (3) is now closed, so *ALL* along with 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 hops. A DHT-like solution would require an average of $\log N$ steps for each dwarf node discovery (N being the size of the network), producing an average of $d \log N$ overlay hops for a query resolution.

Back to our example, let us consider the query $S_1 ALL P_2$. 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 attribute values and that new dwarf nodes are allocated when necessary. Moreover, *ALL* cells of dwarf nodes associated with the updated nodes are affected.

Assuming $u = \langle u_1, u_2 \dots u_d \rangle$ is the tuple to be added to an existing *BD*, the update procedure starts from N_{root} following the path designated by u_1, u_2 etc. Once the dwarf node containing the last common attribute u_i is discovered, a new entry for u_{i+1} must be registered to the node where 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.

4. EXPERIMENTAL RESULTS

We now present an evaluation of an initial implementation of *BD*, entirely written in Java, using an actual test-bed of $N = 16$ LAN commodity nodes (dual core, 2.0 GHz, 2GB of main memory). The centralized approach has also been implemented for direct comparison. In our experiments, using our own generator, we have created synthetic datasets consisting of a fact table representing multidimensional data with numerical facts. For the application workloads, we include both point and aggregate queries with varying proportions/distributions as well as batch updates.

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 construct *BD* and *Dwarf* cubes with variable number of dimensions d (5 up to 25), with cardinalities equal to 1K values. The datasets consist of 10K tuples, following uniform, self-similar (80-20) and Zipfian ($\theta = 0.95$) distributions. Storage consumption and insertion times are presented in Table 2.

First, we note that the total cube size is always bigger than the fact table by a factor that increases with dimensionality and skew. For $d = 25$, *Dwarf* takes up 152 times more storage than the fact table of the Zipfian distribution, while the *BD* expansion ratio reaches 195:1. This index growth, which constitutes an intrinsic characteristic of the method is an extra motivation for the distribution of the *Dwarf* cube.

The insertion time is in general proportional to the index sizes. High dimensional datasets take longer to insert and skew further slows down the process for both *Dwarf* and *BD*. However, 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

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

		Uniform				80-20				Zipf			
d	F. Tbl.	size(MB)		time(sec)		size(MB)		time(sec)		size(MB)		time(sec)	
	(MB)	*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	13	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	50	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: Effect of 1% increments over various dimensions

	Uniform			80-20		
d	time(sec)		msg/	time(sec)		msg/
	dwarf	*BD*	upd	*dwarf*	*BD*	upd
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 4: Query resolution times and communication cost over various 1K querysets.

	Uniform			Zipf		
d	time(sec)		msg/	time(sec)		msg/
	dwarf	*BD*	q	*dwarf*	*BD*	q
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

peer stores its part of the cube independently). The acceleration is more apparent as the number of dimensions and the skew grows: For instance, *BD* inserts the 25-d skewed cubes up to 3.5 times faster than *Dwarf*. The acceleration factor is not directly proportional to the number of participating nodes: The cube calculation remains serial and the network communication introduces latencies.

BD induces a storage overhead for all datasets. This overhead is mainly attributed to the mapping between the network IDs (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. This also explains why the overhead slightly increases with the number of dimensions. Nevertheless, this overhead is shared among the participating nodes: In the case of the 25-d Zipfian dataset, even though the overhead is 43MB, the burden of each of the 16 peers is only 2.7MB. Thus, the big advantage of *BD* is the fact that it can store almost N times as much data as *Dwarf*, using N computers similar to the central case.

Updates: In this section, we observe the behavior of *BD* when update batches are to be inserted to the distributed structure. Utilizing the same datasets, we apply 1% incremental updates which follow the uniform and the self-similar (80-20) distributions. Results are presented in Table 3.

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: The more the dimensions, the larger the *BD* created, thus the more dwarf nodes and cells are affected. As 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 affect more dwarf nodes and cells, thus slowing down the process and creating larger network traffic.

Query Processing: In this section we investigate the query performance of *BD* compared to that of *Dwarf*. Using the same datasets as in the above experiments, we pose two 1K 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 a dimension not participating in a query, is set to 0.3. Table 4 summarizes the results.

First, we 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 towards the answer and one to send the response back to the initiator.

5. 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 both point and group-by queries and update multidimensional bulk datasets on-line, without the use of any proprietary tool.

Our future work will focus on the improvement of the distributed characteristics of *Brown Dwarf*. Our goal is to address load balancing among nodes, content availability and resiliency under volatile environments. This dictates the design of a replication mechanism adaptive to both load skew and node failures. Furthermore, we intend to deploy *BD* to Cloud environments.

6. REFERENCES

[1] S. Abiteboul et al. WebContent: Efficient P2P Warehousing of Web Data. *VLDB'08*.

[2] J. Dittrich, L. Blunschi, and M. Salles. Dwarfs in the rearview mirror: how big are they really? *VLDB'08*.

[3] A. Doka, D. Tsoumakos, and N. Kozyris. HiPPIS: An Online P2P System for Efficient Lookups on d-Dimensional Hierarchies. In *WIDM'08*.

[4] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K. Tan. An Adaptive peer-to-peer Network for Distributed Caching of OLAP Results. In *SIGMOD'02*.

[5] L. Lakshmanan, J. Pei, and Y. Zhao. QC-trees: An Efficient Summary Structure for Semantic OLAP. In *SIGMOD'03*.

[6] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the PetaCube. In *SIGMOD'02*.