

Efficient Updates for a Shared Nothing Analytics Platform*

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

Dimitrios Tsoumakos
Department of Computer
Science
University of Cyprus
dtsouma@cs.ucy.ac.cy

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

ABSTRACT

In this paper we describe a cloud-based data-warehouse-like system especially targeted to time series data. Apart from the benefits that a distributed storage built on top of a shared-nothing architecture offers, our system is designed to efficiently cope with continuous, on-line updates of temporally ordered data without compromising the query throughput. Through a totally customizable process performing asynchronous aggregation of past records, we achieve significant gains in storage and update times compared to traditional methods, maintaining a high accuracy in query responses for our target application. Experiments using our prototype implementation over an actual testbed prove that our scheme considerably accelerates (by a factor above 3) the update procedure and reduces required storage by at least 30%. We also show how these gains are related to the level and rate of aggregation performed.

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

Distributed Systems, Data Warehousing

Keywords

Data Cube, Time Series, Updates

1. INTRODUCTION

Data warehousing has become a vital component of every organization in the scientific as well as the business domain, as it provides tools for data analysis, summarization and prediction of future trends in areas such as retail, finance, network/Web services, etc. The basic abstraction is the *data cube* that allows for fast analysis from multiple perspectives.

*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.

MDAC '10, April 26, 2010 Raleigh, NC, USA
Copyright 2010 ACM 978-1-60558-991-6/10/04 ...\$10.00.

Typical workloads usually consist of read-only queries interleaved with batch updates.

An important class of data is the *time series* data that typically contain a time attribute, such as the date in a stream of sales data or the time of a credit card purchase. Such data can be presented in an even more fine-grained manner through the use of *concept hierarchies*, which define a sequence of mappings from more general to lower-level concepts (e.g., *Day<Month<Quarter<Year*) and allow users to view a given cube at different levels of granularity.

Besides the well-documented need for off-line analytics, the requirement for data analysis in order to immediately detect interesting trends/associations is ever-growing [4, 9]. Market globalization, business process automation, new compliance regulations, etc., mandate even more data retention from companies and organizations alike as a brute force method to reduce risk and render real-time analytics a necessity [8]. Thus, to satisfy both modes would imply 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.

Yet, traditional data warehouses present a strictly centralized and off-line approach in terms of data location and processing [12, 11]: Views are usually calculated on a daily or weekly basis after the operational data have been transferred from various locations. Even if some works in the field propose distributed warehousing systems [2, 1], the warehouse and its main functionality remain centralized. Moreover, the interpretation of time series data is crucial for monitoring a process of interest and detecting the occurrence of certain temporal patterns, e.g., for denial-of-service (DoS) attack or intrusion detection [10]. To that end, constant data analysis is required in order to detect real-time changes in trends. However, updating existing warehouse structures is a rigorous task, fact that discourages their use by applications that require frequent on-line updates.

These challenges have given birth to the idea of creating distributed data-warehouse-like systems deployed on a shared-nothing, commodity hardware architecture, giving the advantage of scalability, robustness and availability at low cost. The *Cloud Computing* paradigm seems an ideal candidate platform: Scalable and often virtualized resources are provided as a service over the Internet by large infrastructure companies. The particularly appealing pay-as-you-go pricing model based on direct storage and CPU consumption largely alleviates customers from the cumbersome and expensive maintenance costs. Resource availability is typically elastic, with a seemingly infinite amount of compute

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

power and storage available on demand.

Analytical applications are particularly well-suited for the Cloud [4]. In our efforts to provide an always-on, real-time data access system for concurrent update and query processing with fast response times we have developed the *Brown Dwarf* platform, where we applied techniques from the field of P2P computing to distribute and dynamically replicate a data cube. However, updating such a structure is still costly, discouraging its use under frequent updates, as in the case of time series data.

In this work, we describe a prototype implementation of *Time Series Dwarf*, a system that maintains all the characteristics of applications to be deployed in the Cloud: Geographically spanned users, without the use of any proprietary tool, can share information that arrives from distributed locations at a high rate in the form of time-series and query it in different levels of granularity. Our scheme provides two important mechanisms: First, it organizes the cube indexing so that update costs and times are minimized. Second, it provides an adaptive materialization scheme that summarizes older data according to the level of detail they are requested, in order to minimize storage consumption and maximize query throughput. In summary, the contributions of this paper are the following:

- A complete indexing, query processing and update system for multidimensional time-series data over a distributed environment where frequent updates are performed. The cube is created with just one pass over the fact table; aggregate queries can target multiple hierarchy levels, while updates are processed on-line with a small cost in terms of bandwidth consumption as well as time. Commodity PCs can participate in this distributed data store, while users need no proprietary tool to access it.
- An adaptive scheme, that adjusts the granularity of materialization utilizing the common observation that queries over time tend to be less detailed as time progresses. Thus, query resolution is accelerated and storage consumption is minimized according to demand.
- An initial validation of the proposed system using an actual deployment on synthetic and real datasets. Our findings show that our scheme considerably accelerates the update procedure and reduces required storage according to the level and rate of aggregation performed.

2. DWARF EVOLUTION

Dwarf [11] is an architecture for storing, querying and updating 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, thus reducing the size of the cube. Fig. 1 shows the *Dwarf* created 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 points to a node in the next level that contains all the distinct values that are associated with its value. Grey cells cor-

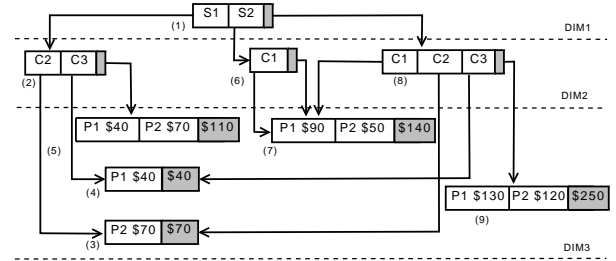


Figure 1: Centralized Dwarf for the fact table of Table 1, using the *sum* aggregation function

respond to “*ALL*” values, used for aggregates on each dimension. Any point or 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).

While *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 decentralization, recent work [6] indicated that, depending on the cube’s density, a *Dwarf* structure may take up orders of magnitude more space than the original tuples. Moreover, updating such a structure is very costly and inefficient, due to the a-priori materialization. Therefore updates are usually performed in batches. Even so, sometimes it is more efficient to reconstruct the whole dwarf structure from scratch.

Brown Dwarf (*BD* for short) [7] is a system that distributes *Dwarf* over a network of interconnected nodes on-the-fly, in a way that all queries that were originally answered through the centralized structure are now distributed over a P2P overlay. The general approach is that each vertex of the dwarf graph (henceforth termed as dwarf node) is designated with a unique ID (UID) and assigned to a network node. Adjacent dwarf nodes are stored in adjacent network nodes in the P2P layer by adding overlay links, which represent the edges of the centralized structure. 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. If the dwarf node containing *currAttr* constitutes a leaf node, *child* is the aggregate value.

Pictorially, Fig. 2 shows that nodes (1)–(9) are selected in this order to store the corresponding dwarf nodes of Fig. 1, forming an unstructured P2P overlay, using the indexing induced by the centralized creation algorithm. Queries and updates are then handled using the same path that would be utilized in *Dwarf*, with overlay links now being followed: An incoming query about S1 will be forwarded to node (2). From there, depending on the requested group-by (*ALL*, C2 or C3), terminal nodes (3), (4) or (5) can be visited.

The distribution of the *Dwarf* structure relaxes its storage requirements and enables the computation of much larger cubes. However, the update procedure remains costly in terms of time as well as bandwidth.

3. TIME SERIES DWARF DESIGN

Time Series Dwarf (*TSD*), especially targeted to time series data, modifies the *BD* structure in a way that favors frequent updates over temporally ordered information, pro-

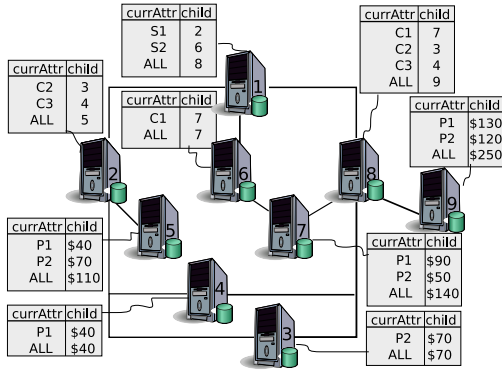


Figure 2: The distribution of the dwarf nodes in the Brown Dwarf of Table 1 and their hint tables

duced in a distributed manner at a high rate and adjusts the granularity of the materialization according to the data’s recency. Apart from the benefits that the distribution of the Dwarf offers, such as the acceleration of the cube construction, the ability to store much larger cubes and the dramatic reduction in query response times, our system allows for online, cost-efficient updates that can originate from any host accessing the update service of the system even at very high rates. Moreover, it makes better use of the available storage and bandwidth, as the granularity of aggregation can be adjusted, according to the application needs.

Some necessary assumptions that we believe to be reasonable for our motivating scenario and justify some of our design and implementation choices are the following:

- In a data cube, any dimension can be characterized by concept hierarchies. For simplicity reasons we only consider hierarchies for the Time dimension in our use cases. However, our system can be generalized to support hierarchies in any dimension.
- No updates for past events are posed to the system. Since updates can be issued in a distributed manner by any host accessing the service, it is possible that some of them arrive out-of-order, without significant delay though, as it is expected in time series data. To ensure temporal order, the system could introduce a window of time, within which all arriving updates will be sorted before their integration to the existing materialized structure.
- As time progresses, the temporal granularity of the posed queries is relative to the time the query is being posed. This practically translates to more detailed queries for recent events, and more coarse grained queries for records concerning past events.

Table 2 contains a sample fact table of 3 dimensions (Time, Store, Product) and a measure of interest (Sales). It also includes the declaration of the hierarchy imposed on the Time dimension of the sample dataset, as well as the metadata that map the values of each hierarchy level to values of the level above. As far as the notation is concerned, if N is the number of levels of the Time hierarchy, each hierarchy level is denoted as ℓ_i , $0 \leq i \leq N - 1$ with ℓ_0 corresponding to the most detailed level.

3.1 Insertion

The insertion operation refers to the initial cube creation using historical data of the past. Time is chosen first in the dimension ordering, while the rest of the dimensions are placed in a descending cardinality order. As proven in the original paper, dimensions with higher cardinalities result in

Table 2: Data and Metadata sample for our use case

Time Hierarchy	Fact Table				Metadata
	DIM ₁	DIM ₂	DIM ₃	Measure	
Year	M_1	S_1	P_1	\$10	$M_1 \rightarrow Q_1$
↑	M_1	S_1	P_2	\$20	$M_2 \rightarrow Q_1$
Quarter	M_2	S_2	P_2	\$30	$M_3 \rightarrow Q_2$
↑	M_3	S_2	P_2	\$40	$Q_1 \rightarrow Y_1$
Month	M_4	S_1	P_1	\$50	$Q_2 \rightarrow Y_1$
					$Q_3 \rightarrow Y_2$

a smaller Dwarf cube when placed on the higher levels of the Dwarf cube. The construction is preceded by a sort on the fact table, as in the original Dwarf. Figure 3 depicts the Dwarf structure according to the original algorithms for the data of Table 2, which we present for comparison reasons.

The main difference of our system’s cube construction is the lack of the ALL cell in the Time dimension. Tuples are being processed one by one, first according to ℓ_0 (Month in our example). As soon as all the values of ℓ_0 that are mapped to the same value of ℓ_1 have been processed, our algorithm creates an aggregate cell for that specific value of ℓ_1 . This procedure is followed for all N levels of the hierarchy. In general, the aggregate of a value of ℓ_i is created by calling the SuffixCoalesce routine of the original Dwarf (see [11]), providing as input the set of Dwarfs of the ℓ_{i-1} that correspond to the specific value of ℓ_i . For the sample data of Table 2, after having created the nodes and cells for the first two tuples according to the original cube construction algorithm, the system proceeds to the third. It then realizes that all tuples belonging to Q_1 have been processed, thus it creates the aggregate cell and the corresponding subdwarf for Q_1 . Similarly, upon reception of the last tuple, the system constructs the subdwarf for Y_1 . The final graph created can be seen in Figure 4. Note that the highest level of aggregation is defined by the highest level of the hierarchy.

3.2 Querying

Queries are resolved by following their path along the system attribute by attribute. Resolution starts from a specific entry point, namely the network node that hosts the root dwarf node of the distributed structure (N_{root}). A node initiating a query $q = \langle q_1, q_2 \dots q_d \rangle$, with q_1 being of any ℓ_i and q_j , $j \neq 1$ being either a value of dimension j or ALL, forwards it to N_{root} . There, the hint table is looked up for q_1 under currAttr. If q_1 is of ℓ_0 and exists, child is the next node the query visits. The above procedure is followed until a measure is reached. Since adjacent dwarf nodes belong to overlay neighbors, the answer to any point or group-by query is discovered within at most d hops. If q_1 is of a level ℓ_i , $i \neq 0$, the same procedure is followed if this aggregate exists, with the answer being reached within d hops. If the query concerns a very recent event and thus the aggregate cell has not been created yet, the initial roll-up query must be substituted by multiple queries of ℓ_0 . Let $V_{\ell_0 \rightarrow q_1}$ be the set of values of ℓ_0 that are descendants of the queried value of ℓ_i , $i \neq 0$. Then the requester must issue $|V_{\ell_0 \rightarrow q_1}|$ queries with $q_1 \in V_{\ell_0 \rightarrow q_1}$ and q_j , $j \neq 1$ being the same as in the original query, gather the results and spend some post-processing to calculate their aggregate. In that case, the answer is at most $d \cdot |V_{\ell_0 \rightarrow q_1}|$ hops and an aggregation away.

In our sample case, a query for $\langle Q_1, ALL, P_1 \rangle$ follows the path (1)→(6)→(7) and returns \$10, while $\langle Q_3, S_1, P_1 \rangle$ is translated to $\langle M_4, S_1, P_1 \rangle$ through the metadata informa-

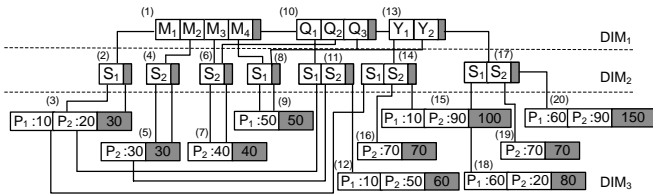


Figure 3: Dwarf cube for the fact table of Table 2

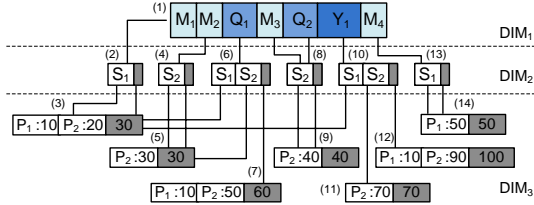


Figure 4: TSD Cube for the fact table of Table 2

tion and returns \$50 after visiting (1), (13) and (14).

The existence of a single entry point creates load imbalance among the network nodes and constitutes a single point of failure. In this work, we do not explicitly deal with load-balancing nor fault tolerance, as we believe it is orthogonal. An initial approach to tackle such issues is discussed in [7].

3.3 Updating

This is an important operation, as, by nature, time series data undergo very frequent updates. Assuming that already inserted tuples are read-only and can neither be changed nor deleted, as the common practice in data warehouses dictates, the update procedure translates to the insertion of new tuples in the existent cube. The difference is that 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 necessary. As in the insertion case, tuples are initially inserted in ℓ_0 and as soon as a value of ℓ_i , $i \neq 0$ is complete, the specific aggregate is constructed.

The important benefit of the proposed system, compared to the original *Dwarf* is that it significantly reduces the update cost, due to the arrival of tuples in temporal order, combined with the lack of an *ALL* cell in the first dimension. The temporal order guarantees that the first attribute of the new tuple will either create a new cell in the first dimension, or it will coincide with the last cell of N_{root} . Therefore, no aggregate cell of the *Time* dimension will be affected. This is not the case for the rest of the dimensions though, where all the affected aggregates have to be recalculated.

In our example, if $(M_5, S_2, P_1, \$60) \rightarrow Q_3$, the system will create a new cell in (1) for M_5 and two new dwarf nodes for the rest of the attribute values. Besides these three dwarf nodes accessed in total, no other node of the system is affected. Contrarily, for the original *Dwarf* of Figure 3, apart from the above mentioned changes, new sub-dwarfs for Q_3 and Y_2 would be created. Moreover, following the *ALL* path of the *Time* dimension, nodes (19) and (20) would be updated with the newly inserted values. In total, 12 dwarf nodes would be accessed. It is thus apparent that updates in our system are less costly, as they affect fewer dwarf nodes and cells compared to the original *Dwarf*.

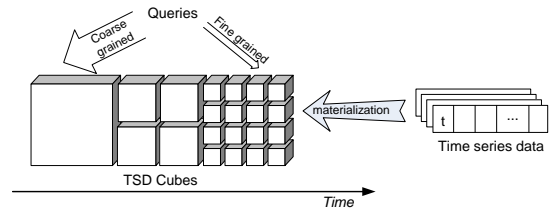


Figure 5: Materialization over time

3.4 Adaptive Materialization

The proposed system, as described above, follows a static strategy for materialization: A roll-up view in the *Time* hierarchy is created as soon as all required data are available, without destroying the drill-down views, which remain in the system. Instead, a more dynamic approach can be adopted: A daemon process periodically and asynchronously creates the roll-up views and erases the corresponding drill-down ones. The period of this process is chosen taking into account the characteristics of the application. Depending on the application, not all data are queried upon in the same level of granularity. Indeed it is often the case that recent data are demanded in a fine-grained manner, whereas queries for historical data usually concern aggregated periods of time. Therefore, the materialization process gradually follows the roll-up path and eliminates the more detailed views as time passes, as depicted in Figure 5.

The time thresholds beyond which a roll-up view is created and the corresponding finer grained ones are erased are denoted as $(T_{\ell_1}, T_{\ell_2} \dots T_{\ell_{N-1}})$. This practically means that for records stored more than T_{ℓ_i} the system constructs and maintains only the aggregate views that belong to ℓ_i , erasing all views from ℓ_0 to ℓ_{i-1} . The values of the thresholds can either be set by the administrator a priori, taking into account the needs of the specific application or dynamically adjusted according to the monitored query trend.

In the case of adaptive materialization, there exists a tradeoff between the size and the complexity of the cube, which consequently affects update and query response times, and the accuracy of the responses. Creating and erasing the aggregate views periodically spares significant amount of storage space. However, the main advantage of the method is the acceleration of updates and the increase in query throughput, due to the smaller overall size of the distributed cube: Keeping dwarf nodes small helps the system navigate more quickly through them. On the other hand, queries that follow an opposite trend than the one expected either are more costly, or cannot be answered accurately. More precisely, coarse grained queries concerning very recent events that only exist in the finest granularity translate to multiple fine grained ones. The system, apart from the bandwidth cost, has to pay a post-processing cost as well, calculating the aggregate of the gathered responses. On the contrary, data that are stored according to a granularity level higher than ℓ_0 suffer an irreversible information loss. Therefore fine grained queries concerning them are only answered in approximation, for instance with the ratio of the aggregate fact to the number of distinct values of the queried level that belong to the data aggregation level.

4. EXPERIMENTAL EVALUATION

In this section we provide an initial evaluation of our prototype, entirely written in Java and deployed on 25 commodity nodes of our lab infrastructure (dual core, 2.0 GHz, 2GB of main memory).

Table 3: Measurements for various dataset insertions

Dataset	#Tup.	size(MB)			time(sec)				msg/ins			
		<i>Dwarf</i>	<i>BD</i>	<i>TSD</i>	<i>TSD_{ad}</i>	<i>Dwarf</i>	<i>BD</i>	<i>TSD</i>	<i>TSD_{ad}</i>	<i>BD</i>	<i>TSD</i>	<i>TSD_{ad}</i>
APB-A	1.2M	56	59	53	9	485	101	100	57	2.3	1.5	0.3
APB-B	2.5M	102	115	93	24	957	220	198	123	2.4	1.7	0.4
APB-C	3.7M	163	182	146	32	1530	321	289	167	2.4	1.6	0.4
DARPA	1.1M	178	191	156	127	614	222	208	189	5.9	5.2	1.2

In our experiments, we use both synthetic and real datasets. Using the APB-1 benchmark generator [3] we produce three 4-d datasets (A, B and C with densities 0.1, 0.2 and 0.3 respectively) with dimension cardinalities 24, 9000, 900 and 9 and one measure attribute. Time dimension consists of 3 hierarchy levels, *Month*<*Quarter*<*Year* and covers the period from January 1995 to June 1996. The real dataset contains publicly available network audit data for the 1998 DARPA Intrusion Detection Evaluation Program [5]. It includes 1 million records, collected over a period of 6 weeks and organized in 7 dimensions. The *Time* hierarchy is organized along *Day*<*Hour*<*Minute*. The aggregate function used in the results is *sum*. For the application workloads, we include both point and aggregate queries with varying granularities and distributions, as well as continuous updates.

Experiments are conducted using both the static and the adaptive mode of *TSD*: In the former case, materialization is performed for all hierarchy levels in a synchronous way, whereas in the latter case, denoted as *TSD_{ad}*, materialization is performed through a daemon process, with thresholds statically set according to the dataset: For the APB datasets, (T_{ℓ_1}, T_{ℓ_2}) is set to (1 quarter, 1 year) and for the DARPA dataset (2 hours, 2 days). For direct comparison we have also implemented the centralized *Dwarf*, as well as its distributed approach, *BD*.

Cube construction In the first set of experiments we construct the *Dwarf*, *BD* and *TSD* cubes to evaluate their creation in terms of time, storage and communication overhead. For *TSD_{ad}* we assume that only the aggregates that conform to the T_{ℓ_i} thresholds are being created in the first place. For instance, for the DARPA dataset this means that records of the last 2 hours are stored in the granularity of minutes, of the last 2 days in the granularity of hours and the rest as days. Results are presented in Table 3.

We observe that *TSD* noticeably reduces the size of the created cube compared to *BD* (providing smaller cubes even from the centralized *Dwarf*), due to the lack of the *ALL* cell in the first dimension. The reduction is even more impressive with *TSD_{ad}* (reaching almost 85% for the APB and 34% for the DARPA datasets). In this case, the choice of the T_{ℓ_i} thresholds plays a decisive role in the cube size reduction: The smaller the T_{ℓ_i} , the more coarse grained the views of the stored data, thus the smaller the cube size. In the worst case, the size of a *TSD_{ad}* cube is equal to that of *TSD*.

Analogous to the cube size reduction is the acceleration of creation time. First, we confirm that the distribution of the cube leads to a better exploitation of existing resources, enforces parallelization and thus reduces cube construction times. *TSD* and *TSD_{ad}* exhibit a further reduction (reaching 10% and 45% respectively) compared to *BD*. *TSD_{ad}* is 89% and 70% faster in storing the cube compared to the centralized system using the APB and DARPA datasets respectively. The absence of the *ALL* cell in *Time* in both cases and the selective materialization of *TSD_{ad}* result in smaller cubes with less dwarf nodes, thus requiring less communica-

Table 4: Measurements for 10k updates over various datasets

Dataset	time(sec)				msg/update		
	<i>Dwarf</i>	<i>BD</i>	<i>TSD</i>	<i>TSD_{ad}</i>	<i>BD</i>	<i>TSD</i>	<i>TSD_{ad}</i>
APB-A	1123	603	404	315	21.5	9.1	8.2
APB-B	1158	611	418	323	23.1	10.3	8.8
APB-C	1203	624	424	328	25.2	10.9	9.1
DARPA	1535	649	458	380	28.6	13.6	9.3

tion and I/O cost. This is in line with the results concerning the messages per tuple insertion.

Updates To test the behavior of *TSD* under continuous updates, we apply 10k tuple insertions over the APB and DARPA datasets and measure the total time needed to process the update batch as well as the communication cost of the procedure. Table 4 summarizes the results. It is worth noting that, in the case of the static *TSD*, the update process includes the creation of roll-up aggregate views, as it happens synchronously, as soon as a hierarchy level value is complete. On the contrary, the creation of roll-up views and the deletion of drill-down ones are not considered part of the update procedure of *TSD_{ad}*, since these are undertaken by the daemon process and take place periodically and independently of the updates.

Both modes of *TSD* drastically improve the update performance, accelerating the process compared to the centralized *Dwarf* and *BD*. *TSD* is about 3 times faster than the central algorithm and over 30% than the simple distributed dwarf. Apart from the parallelization of the process, which is enabled through the distribution of the cube structure, the updates come in order, guaranteeing that no update will affect an already created roll-up view. Furthermore, recursive updates of affected *ALL* cells take place only in the dimensions other than the first, reducing the communication cost. Indeed, the required messages per update drop almost 3 times compared to *BD*. *TSD_{ad}* proves even faster (about 20%) and more cost-efficient than *TSD*, due to the fact that materialization in the various hierarchy levels is performed asynchronously. Lastly, from the APB datasets we conclude that the larger the cube, the more the nodes and cells affected by updates, thus the more costly the process in terms of time and messages.

Figure 6 plots the communication overhead of the *TSD_{ad}* daemon process. The period of its invocation is naturally set to the smallest of the T_{ℓ_i} values, in our case 2 hours. Upon its call, the daemon creates a burst in messages. These messages are needed to create the new aggregates and erase the finer-grained ones. However, this burst lasts only a few minutes (less than 5) and messages are scattered over the network, leading to tolerable per node communication cost ($\sim 100 \frac{\text{queries}}{\text{sec}}$ for our setting).

Querying We now investigate the query performance of *TSD* in both the static and the adaptive mode compared to that of *BD*. Furthermore, we examine the overhead introduced by the lack of the *ALL* cell in the *Time* dimension and the information loss caused by erasing fine-grained views in the case of *TSD_{ad}*.

First, over the DARPA dataset we pose 3 different query-sets (Q_1, Q_2 and Q_3). Q_1 is a workload that ideally conforms to the set of T_{ℓ_i} thresholds of *TSD_{ad}*. Q_2 is created by following a two-step approach. First, a tuple of the dataset is selected using a Zipfian distribution of $\theta = 1.0$, favoring the most recent records. The probability that corresponds to the selected tuple according to our zipfian distribution is used

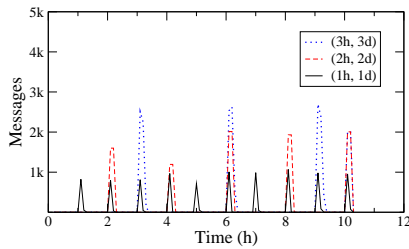


Figure 6: Messages over time for the TSD_{ad} daemon

Table 5: Measurements for various workloads over the DARPA and APB datasets

Queryset	time(sec)			msg/query			%Inaccur. Queries	%Resp. Deviation
	BD	TSD	TSD_{ad}	BD	TSD	TSD_{ad}		
Q_1	5	6	6	6.9	6.9	6.9	0%	0%
Q_2	5	9	8	6.9	9.2	9.1	15%	19%
Q_3	5	24	21	6.9	32.3	32.3	33%	32%
A	13	21	19	3.8	9.3	9.3	19%	23%
B	15	23	19	3.9	9.5	9.5	21%	25%
C	16	24	20	3.9	9.6	9.5	20%	22%

to choose the level of **Time** that will be used in the query: Recent records are queried upon in more detail than older ones. Q_3 follows the Uniform distribution. For all querysets, we set $P_d = 0.1$, which we define as the probability of a dimension not participating in a query.

As seen in Table 5, workload Q_1 does not affect the query throughput nor the per query communication cost, since all queried levels exist. However, as the queryset approximates the uniform distribution (Q_2 and Q_3), the messages needed to answer a query increase, naturally affecting the response times. This is attributed to the fact that queries concerning aggregates of recent data cannot be answered directly, since these aggregates have not been constructed yet, but need to be translated to multiple queries of a lower hierarchy level, thus issuing more messages. TSD_{ad} proves slightly more efficient, mainly due to the more limited size of the cube, which allows queries to navigate faster through it.

In the case of TSD_{ad} and for the querysets Q_2 and Q_3 , it is possible that a query concerns a more fine-grained level of **Time** that is no longer available. In such an event, only an approximation of the answer will be returned. As a convention, TSD_{ad} replies with the corresponding portion of the aggregate value, e.g., returns 1/12 of the value if queried for a **month** and only **year** granularity exists. The percentage of the queries that are answered in approximation reaches 32% in the worse case (uniform distribution). The last column of Table 5 contains the percentile deviation of the returned responses compared to the correct ones. As observed, this measure follows the percentage of inaccurate queries.

The workload posed over the APB dataset is the one generated by the APB benchmark generator. It consists of point as well as aggregate queries in every possible dimension. The ratio of aggregate queries on the **Time** dimension is 50%, which does not favor TSD . However, the results gathered in the lower part of Table 5 show that the slowdown compared to BD is not significant and the percentage of inaccurate queries lies within acceptable limits for applications tolerant of some degree of inconsistency.

There clearly exists a trade-off between the size of the created cube and the accuracy of the query responses. Figure 7 depicts the effect of different T_{ℓ_i} values on these measures for the DARPA dataset under the workloads Q_2 and Q_3 . The smaller the T_{ℓ_i} , the faster the coarse grained aggregates re-

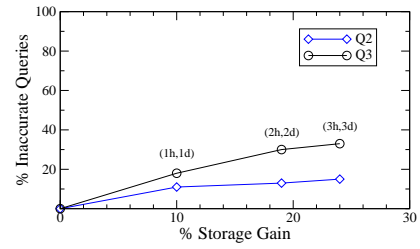


Figure 7: Storage gain vs. % of inaccurate queries for various T_{ℓ_i} in TSD_{ad} (DARPA)

place the finer grained ones. This leads to a gain in storage, yet a loss of detailed information, thus an increase in inaccurate answers. It becomes apparent that in order to find the golden mean, the T_{ℓ_i} thresholds need to fit the expected workloads. A dynamic threshold selection according to the monitored query trend is a subject of future work.

5. CONCLUSIONS

In this paper we described a data-warehouse-like system deployed on a shared-nothing architecture especially designed to handle time series data produced at a high rate in a distributed manner. The asynchronous construction of aggregates in various granularities according to the recency of the temporally ordered data spares storage, while preserving a high accuracy in query responses. Results from our prototype implementation show that our system processes updates 3 to 4 times faster than traditional, centralized solutions, while considerably reducing storage consumption. Tuning the materialization parameters achieves the desired balance between storage consumption and response accuracy. An adaptive, real-time adjustment of these parameters relative to the posed workloads is what we currently pursue.

6. REFERENCES

- [1] S. Abiteboul et al. WebContent: Efficient P2P Warehousing of Web Data. *VLDB'08*.
- [2] M. Akinde et al. Efficient OLAP Query Processing in Distributed Data Warehouses. *Information Systems*, 28(1-2):111–135, 2003.
- [3] *OLAP Council APB-1 OLAP Benchmark*. <http://www.olapcouncil.org/research/resrchly.htm>.
- [4] M. Armbrust et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, Berkeley, 2009.
- [5] DARPA Intrusion Detection Evaluation. <http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/index.html>.
- [6] J. Dittrich, L. Blunski, and M. Salles. Dwarfs in the rearview mirror: how big are they really? *VLDB'08*.
- [7] K. Doka, D. Tsoumakos, and N. Koziris. Brown Dwarf: Distributing the Power of OLAP. Technical report, NTUA, 2009.
- [8] T. Economist. Technology: The data deluge.
- [9] E. Knorr. Dealing with the data explosion, Infoworld.
- [10] A. Singhal and S. Jajodia. Data warehousing and data mining techniques for intrusion detection systems. *Distributed and Parallel Databases*, 20:149–166, 2006.
- [11] Y. Sismanis et al. Dwarf: Shrinking the PetaCube. In *SIGMOD'02*.
- [12] W. Wang, H. Lu, J. Feng, and J. X. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *ICDE'02*.