Leveraging 24/7 Availability and Performance for Distributed Real-Time Data Warehouses

Abstract—**Nowadays, most enterprises require near real-time Data Warehouses (DWs) that are able to deal with continuous updates while providing 24/7 availability. Distributed data using round-robin algorithms on clusters of shared-nothing machines is commonly used for improving performance. In this paper, we propose a solution for distributed DW databases that ensures its continuous availability and deals with frequent data loading requirements, introducing small performance overhead. We use a data striping and replication architecture to distribute portions of each fact table among pairs of slave nodes. Each slave node is an exact replica of its partner in the pair. This allows balancing query execution and replacing any defective node, ensuring the system's continuous availability. The size of each portion in a given node depends on its individual features, namely performance benchmark measures and dedicated database RAM. The estimated cost for executing each query workload in each slave node is also used for balancing and optimizing query performance. We include experiments using the TPC-H decision support benchmark to evaluate the scalability of our solution and show that it outperforms standard round-robin distributed DW setups.**

Keywords-Real-time data warehousing; availability; fault tolerance; data replication and redundancy; distributed and parallel databases; load balancing; performance optimization.

I. INTRODUCTION

Infrastructures such as the Internet have redefined most business models, pushing enterprises into a 24/7 business schedule, *i.e.*, while most business models functioned in an eight to twelve hours schedule in the past, business currently occurs in a non-stop fashion. This has shifted the decision support paradigm: business decisions are now made much more frequently and with the most recent business data. Consequently, today's Data Warehouse (DW) requires both ongoing availability (allowing to continuously provide information to decision makers) and highly frequent data loading (allowing the database to be able to include the most recent business data as quickly as possible), in order to fulfill its decision support purpose [21].

DWs typically have databases with millions of rows or more and take up many gigabytes or terabytes of storage space. Given these features, many decision support queries can run up to hours. Moreover, as the amount of stored data increases, database performance degrades. Distributed database architectures are used to improve performance in these environments, since they enable parallel processing and higher levels of scalability. The most common architecture is to distribute the database throughout shared-nothing clusters of inexpensive commodity PCs [6].

Typical distributed databases divide tables into separate chunks of data of approximately the same size using a round-robin algorithm, storing a chunk in each node [2, 3, 6, 14, 23]. To process a query, a central node receives it and splits it into partial queries to be processed by each node, sends each partial query to each node, collects all partial answers to build up the final answer and sends it back to the user that requested it [3]. We argue that while round-robin fixed-size data distribution seem good for homogeneous environments, many DW setups function in heterogeneous environments, in which nodes often have distinct individual performance. Thus, approaches that use this feature for load balancing in both data loading and querying may produce better results.

The critical issue in simultaneously executing loading and querying actions in DWs is that they introduce extremely large performance overheads, given the size and complexity of the databases [21]. This is the main reason why traditional DWs were updated only when offline to users. Recently, hardware speed and capacity have evolved up to a level where those overheads are now considered acceptable. Nowadays, most real-time DWs employ frequent micro-batch bulk loading of new data, while they are kept online for querying purposes [2, 20]. However, the frequency rate of these updates and how much data should be loaded in each load are very diverse [11]. Moreover, the overhead in loading and querying performance is also dependent from the storage size and hardware features of each clustered database. In fact, query performance in many distributed databases is poor mainly due to load balance problems [10].

In what concerns availability, there are several ways a DW can become fully or partially unavailable:

- Unexpected system failures (*e.g.* hardware failures, network problems, unexpected system shutdown, etc), resulting in *unplanned downtime*;
- Previously defined moments for executing tasks such as hardware, software and database maintenance (*e.g.* data loading, rebuilding indexes, backups, adding new storage hardware, etc), resulting in *planned downtime*.

To avoid both planned and unplanned downtime and ensure its 24/7 availability, a real-time DW must efficiently enable simultaneous data loading and querying, as well as hardware and database maintenance, including the execution of fault tolerant and self-healing actions, keeping the database online to its users in a non-stop fashion within a heterogeneous hardware and software environment. Pulling this together is not a trivial task. Our approach provides a feasible solution for improving the performance and availability of standard distributed real-time DWs and requires small implementation efforts.

A. Our proposal

In this paper, we engineer techniques such as data replicating and striping for enabling 24/7 availability (including fault tolerance and self-healing) in distributed real-time DWs. Our proposal focuses on ensuring the non-stop availability of the DW, as well as the improvement of both data loading and querying in the distributed database.

Our distributed database architecture is based on dividing data among pairs of nodes, called *slaves*. Each *slave* is an exact replica of its partner in the pair. This setup allows rebuilding a node from its partner, in case of integrity issues or hardware failures. Each *slave* can also alternatively process any query supposed to be processed by its respective pair, if this node is unavailable for any reason. Thus, each *slave* can act as a fault tolerance mechanism for its partner, ensuring the pair is functional unless both *slaves* are down. Moreover, using the DW-Striping (DW-S) technique for querying as shown in [3] and explained further in the paper, we may also produce approximate answers even if one or more pairs of *slaves* are down. Within each *slave*, there is also an exact duplicate of each database it holds. This enables executing maintenance tasks and reoptimizing each database (*e.g.*, rebuilding or updating materialized views and indexes) in a node without altering its availability, putting one database offline while its duplicate remains online.

Each cluster node has its own performance coefficient, in relation to the cluster's overall performance. We use this coefficient to balance data loading, by defining the amount of data each node stores in relation to the total size of the database. We also request faster nodes to process a larger number of queries than slower nodes. This leverages response time among the *slave nodes*, improving overall performance. Both data loading and query workload balancing are managed by a pair of nodes that coordinate the whole system, called *masters*. Each *master* is also a replica of its pair, ensuring the system is always available and works if one of *master* is down.

B. Main achievements and contributions

The main contributions of our proposal are as follows:

- We engineer classic techniques such as replication and striping, together with DW loading methods developed and published in former research, for building an efficient 24/7 available real-time distributed DW;
- Our solution optimizes system's overall performance, based on balancing data loading and query execution given each node's hardware and software features, as well as query workloads being executed;
- The DW is always online, for both planned and unplanned tasks such as adding, repairing or removing storage devices, as well as rebuilding or reoptimizing each node's database;
- The architecture is extremely flexible and can easily be applied in classical DWs with distributed databases for enabling 24/7 availability with fault tolerance, while updating the database in a (nearly) continuous fashion;
- Our experiments show our solution's feasibility and that it outperforms standard round-robin distributed database architecture such as DW-S.

C. Structure of the paper

The remainder of this paper is organized as follows. In Section II we summarize our previous work, which is the foundation for the proposed solution in this paper. In Section III we explain our proposal, pointing out the issues involved in its use. Section IV presents experimental evaluations of our solution using the TPC-H decision support benchmark and a real-world sales DW, comparing it to the round-robin DW-S. Section V describes related work on availability and real-time distributed DW solutions. Finally, in Section VI we present our conclusions and point out future work.

II. Building a 24/7 Real-Time Centralized DW

A. Changing a traditional centralized DW into a 24/7 real-time centralized DW

In this subsection we summarize previous work [17, 18], explaining how to change a traditional centralized DW with static data structures and offline updates into a centralized (near) real-time DW with a dynamic database, capable of dealing with the issues involving querying and frequent data loading at the same time. Our assumptions are very simple:

- Small tables (*i.e.*, with a small amount of rows) are able to load data faster than large tables;
- Tables that have no query performance optimization data structures such as primary keys and other indexes are capable of appending data much faster than tables that have those data structures;
- Using INSERT or appending data by bulk loading is much faster than UPDATE ELSE INSERT procedures (since UPDATE previously executes a table lookup), common to most commercial data loading tools.

Most DW schemas are star schemas [12], where business facts are stored in a central table called *Fact table* (*e.g.* Sales fact table) and the tables containing the business descriptors are called *Dimension tables* (*e.g.* Customer and Product tables). Dimension tables are linked to the fact table by their primary keys (*e.g.* CustomerID and ProductID). Since fact tables typically take up at least 90% of the total storage size [12], we focus on speeding up loading data into these tables. On the other hand, dimension tables are typically small sized and have a small amount of rows [12]. Thus, for updating dimension tables we use the standard UPDATE ELSE INSERT approach, since this will result in small delays that do not significantly affect the system's performance [13].

To store new fact table rows we use only INSERT statements or bulk loading into an extra temporary fact table, created empty of contents and without any constraint or optimization data structure (including primary keys, indexes, etc). The temporary fact table has the same data structure as the original fact table it concerns, plus an extra column that stores incremental identifiers for being able to identify the sequence of added rows.

Since our intention is to minimize the gap between what happens in the transactional systems and its propagation in the DW, a transaction can be changed at its origin after it has already been stored in the DW. Our solution deals with this by using only INSERT statements to update the DW. Given that business facts in fact tables are usually numerical values, to use only INSERT statements for updating fact tables we must ensure factual columns have additive properties. If this is assured, we can use SUM functions grouped by primary keys for obtaining the correct value of the transaction.

As an example, suppose a very simple sales DW with the schema shown in Figure 1. It has two dimensional tables

(Store and Customer) and one fact table (Sales). To simplify the figure, the Date dimension is not shown. This DW stores the sales value per store, per customer, per day. The primary keys are represented in bold, while referential integrity constraints with foreign keys are represented in italic. The factual attribute S_Value is additive.

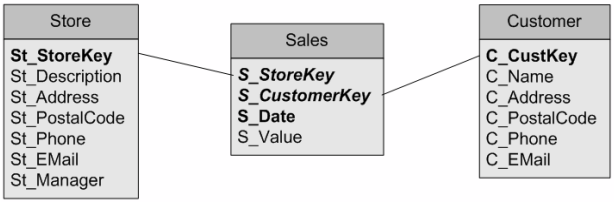

Figure 1. Sample sales data warehouse schema

A temporary fact table STmp_Sales is created for the original fact table Sales. A unique sequential identifier in STmp_Sales (STmp_Counter) records the sequence in which each row is appended. This identifies the sequence for each inserted row, useful for restoring prior data states in disaster recovery actions. With our method, any appending, updating or eliminating of business data on transactional systems only results in new row insertions in the DW, in order to minimize row, block, and table locks, as well as other concurrent data access problems. Physical database tablespace fragmentation is also avoided, since there is no deletion of data, just sequential increments. INSERT is much faster than UPDATE or DELETE actions, allowing us to state we use the fastest method to refresh the DW [13, 18].

Regarding Figure 1, we now describe an example for loading new data. Figure 2 presents the insertion of a row in the temporary fact table (STmp_Sales) for recording a sales transaction of value 100 occurred at 2008-05-02 in store St_StoreKey=1, related to customer with C_CustKey=10. This row is identified by STmp_Counter=1001. Meanwhile, other transactions occurred, and the transactional system recorded that instead of a value of 100 for the mentioned transaction, it should be 1000. The STmp_Sales rows with STmp_Counter=1011 and STmp_Counter=1012 reflect this modification. Summing STmp_Value grouped by primary key (STmp_StoreKey, STmpCustomerKey, STmpDate), the resulting value (100-100+1000 = 1000) is the new real value, due to the additivity of STmp_Value.

Row reflecting the insertion of a transactional record →

Rows reflecting the modification of the previous transac. record

STmp_StoreKey	STmp_CustomerKey	STmp_Date	STmp_Value	STmp_Counter
⋮	⋮	⋮	⋮	⋮
1	10	2008-05-02	100	1001
⋮	⋮	⋮	⋮	⋮
1	10	2008-05-02	-100	1011
1	10	2008-05-02	1000	1012
⋮	⋮	⋮	⋮	⋮

Figure 2. Partial contents of temporary fact table SalesTmp

Extraction-Transformation-Loading (ETL) tools are used for extracting the transactional business data, cleaning and transforming it into decision support format, and loading it into the DW. Many issues on the use of ETL tools do not focus only on performance problems (as would be expected), but also in aspects such as complexity, practicability and price [13]. By using only row insertion actions into empty or small sized tables without constraints or attached physical files related to it, we use the simplest and fastest logical and physical support for achieving our data loading goals [13].

To query the DW, users have three possibilities: 1) To query past data, queries remain unaltered, since the original schema is the same in the modified schema; 2) To query just the most recent data, queries just need to replace the original fact tables for the new temporary fact table replicas, since they are the ones that store that data; and 3) To query both past and most recent data, queries need to consider joining both the original and the new temporary fact tables.

Since data loading is done mainly into tables without performance optimization structures, as their size increases, query performance against them decreases. Then, the system is reoptimized, by transferring the data in the fact table replicas into the original ones, as explained in [17, 18], recreating the fact tables' replicas empty of contents once more and rebuild performance optimization structures in order to regain optimal performance. The issues of this method are thoroughly discussed in [17, 18].

B. Obtaining continuous availability in a centralized DW

The architecture of the latest version of our work, a 24/7 real-time DW (RTDW) system, published in [19], is shown in Figure 3. Using data replication and the techniques proposed in [17, 18], the 24/7 availability solution is based on the following:

- The server has a complete duplicate of each database;
- Both databases will be updated simultaneously by our 24/7 RTDW Tool, but only one at a time will be made available/online for users (although this is indifferent, since they query the middleware, which transparently redirects their queries to the appropriate DB and returns their responses to who requested them);
- Whenever the online DB needs to be reoptimized it is put offline, while its duplicate is put online, and vice-versa, in a continuous manner, assuring that there is always one of the DB which is available to the users;
- Since there is an exact replica of each database, they can act as fault tolerant solutions if one is damaged.

In [18], a middleware application transparently deals with both data loads and query processing. Each load is treated in a transaction-like fashion (instead of simple micro-batch bulk loading of individual rows) to solve referential integrity issues from our previous work [17]. A transaction is defined as a set of all dimension table rows needed for each fact table row. For instance, this means that before inserting a new as shown in Figure 2, the middleware would confirm that customer key 10 and store key 1 exist in Customer and Store, respectively. If they do not exist, they are created before updating the Sales fact table with the referred row.

The *24/7 RTDW Tool Manager* allows the DBA to manage the tool's database (*24/7 RTDW Tool Database*) and monitor data loading executions, as we previously described. It also allows the DBA to build the original DW schema duplicates and set up all the values when the tool is used for the first time, as well as to reoptimize any database at any time. The *24/7 RTDW Tool Database* stores the information

which is vital for managing the system, such as information on the DW tables and other data structures (indexes, materialized views, etc), definitions of transactions described in the former paragraph, how often is new data to be loaded, which is the current available database for users, status boolean flags indicating if any of the databases is being reoptimized, queried or updated, etc.

The *24/7 RTDW Tool Loader* component is responsible for executing the DW refreshment procedures, loading new data into its databases. The *24/7 RTDW Tool Query Executor* handles the user queries, selecting which replicated database to use. It simply redirects the requested queries, according to which is the available database for querying (defined in the tool's database), supplying the results to the query's request origin. All functional issues are thoroughly discussed in [19].

III. The 24/7 Real-Time Distributed DW

A. The 24/7 Real-Time Distributed DW Architecture

Figure 4 shows the conceptual hardware architecture for our solution. The system works by logically grouping nodes in pairs. Each node in a given pair is an exact replica of its partner. There are *slave pairs* and *master pairs* of nodes. The *master pair* contains *master nodes*, which are responsible for managing the system and interface the DW with the decision support users and ETL tools. The *slave nodes* are shared-nothing machines, each with its own DataBase Management System (DBMS) and independent database instances, storing part of the DW data. They work individually as explained in the previous section for processing data loading and querying, while maintaining their ongoing availability.

The *master nodes* maintain a list of all the *slave pairs* and respective *slave nodes*, and are responsible for defining the amount of data to load into each *slave node*'s database, sending them that data and verifying if each load process was successfully completed. They are also responsible for receiving user queries, splitting and sending them to be processed by each *slave node*, subsequently collecting those partial answers, building the complete final answers and returning those answers to the users which requested them.

Master nodes are also responsible for verifying if any *slave node* or *slave pair* is down, correcting defective *slave nodes* and alerting the DBA. Since each *slave node* in a given *slave pair* is an exact replica of its partner, this allows a *master node* to update, correct or restore any of its tables, or replace a jeopardized node that needs to be substituted by completely and correctly rebuilding a replacement node to take its place. A *master node* is also capable of rebuilding its partner in the respective *master pair*.

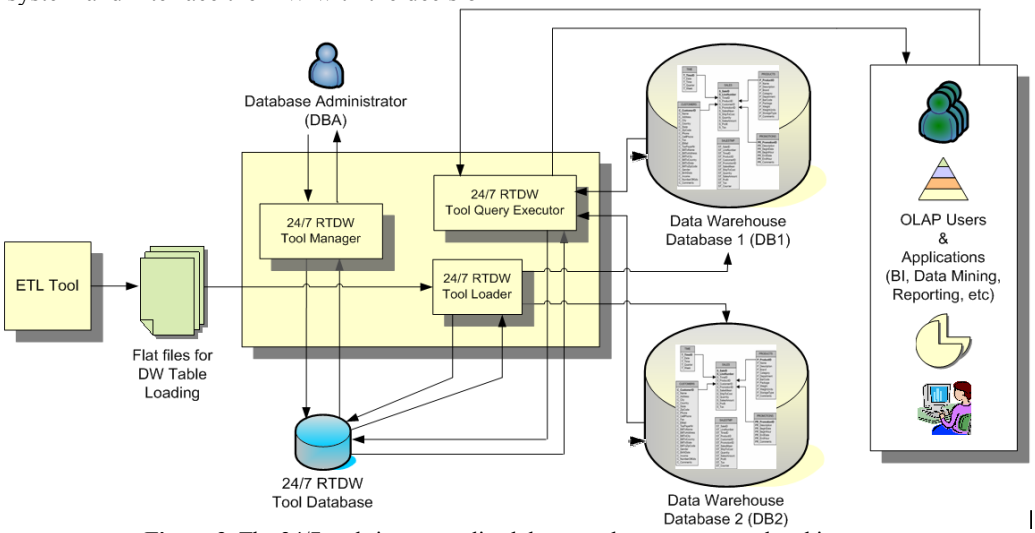

Figure 3. The 24/7 real-time centralized data warehouse conceptual architecture

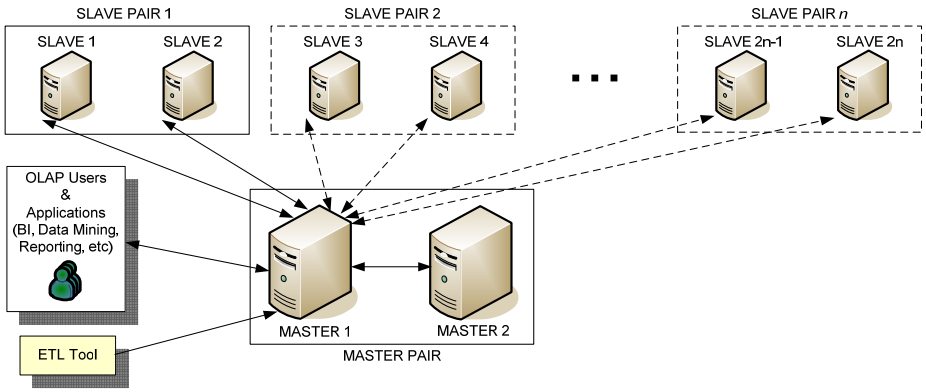

Figure 4. The 24/7 distributed real-time data warehouse hardware architecture

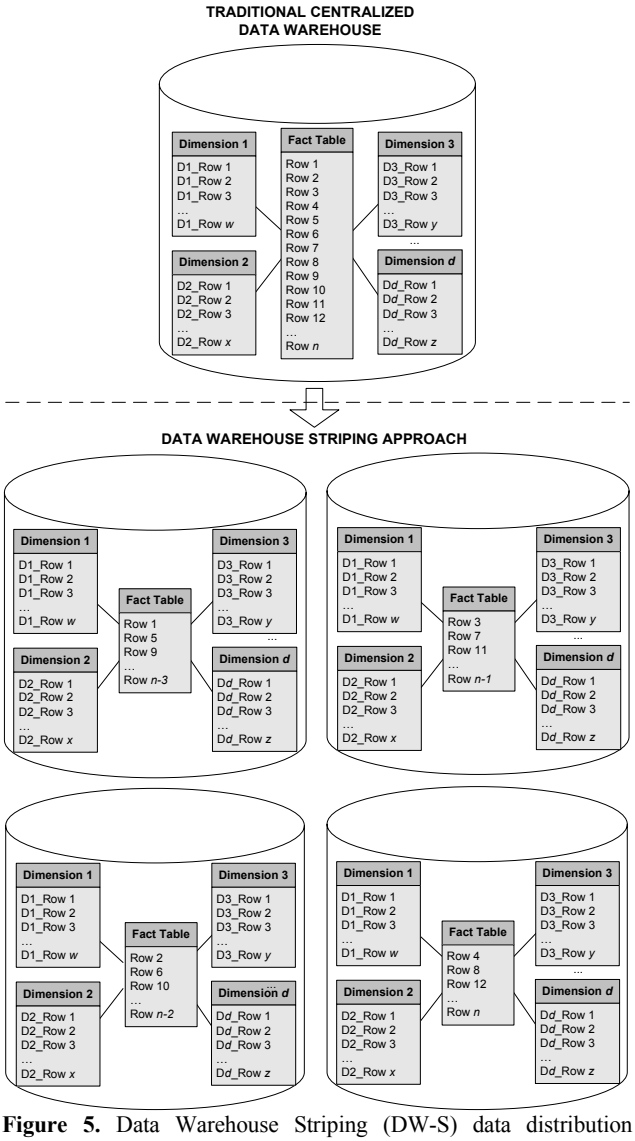

Figure 5. Data Warehouse Striping (DW-S) data distribution technique

B. Data distribution and load balancing

To distribute data among the slave nodes using the DW-Striping (DW-S) technique, explained in [3] and illustrated in Figure 5, a fact table is broken into equal-sized chunks of data using a round-robin algorithm. Each node stores one of those chunks, along with replicas of all dimension tables. The reason why the round-robin algorithm is used instead of dividing the data based on ranging values of a given attribute is to assure the data is randomly distributed independently from their values, so each node processes approximately the same amount of data as the remaining nodes and thus, avoids overloading specific nodes while others remain idle when processing range queries with that attribute.

However, we disagree with this form of distribution. We argue that faster nodes should process larger portions of data than slower nodes, in order to minimize waiting time until all partial query answers are returned to the *master*. For example, if there are two *slave* nodes, where one is two times faster than the other, then it should process the double of the amount of data of the other, to minimize the overall response time. Thus, to define the amount of data to load in each *slave node* we evaluate its performance capability (as shown further) and distribute data proportionally according to its relative individual performance coefficient within the cluster.

Proving that our data distribution proposal is more efficient than the standard round-robin algorithm is simple and straightforward. Consider the following scenario: in a cluster with two nodes, one machine is capable of processing data twice as fast as the other. Assuming the total size of the DW database is 1TB, the round-robin DW-S approach would store 500GB in each node. If the faster node takes 100 seconds to process a query, then the slower node (which works at half speed) takes 200 seconds. Since we have to wait for both nodes to finish processing the query, we can only build the final answer after 200 seconds (corresponding to the slowest node's processing time). In our approach, the slowest node would store only half the amount of the data of the fastest. Thus, the fastest node would store 667GB, while the slowest would store 333GB. Assuming that response time is approximately proportional to the amount of data that needs to be accessed by the node, then the fastest node would take 667/500*100 = 133 seconds, while the slowest node would take 333/500*200 = 133 seconds. Thus, both would be finished practically at the same time and none would have to wait for the other, completing both the tasks in 67 seconds earlier than the fixed-size round-robin approach.

An example of how our approach would divide a given fact table comparatively to the DW-S approach is shown in Figure 6, exemplifying a cluster of four *slave nodes*. While DW-S would generate four equal-sized chunks with $n/4$ rows each, our approach would generate four chunks with i, j, k, l rows for each *slave node*, dependent on their performance coefficient. In Figure 6, the fact table chunks in DW-S are named FTx, where x represents the *slave node* number, while in our approach each of those chunks is named FT$x.y$, where x also represents the *slave node* number, y=1 represents the original fact table and y=2 the temporary replica used for loading new fact table rows as explained in subsection II.A. As shown, each *slave node* stores a replica of its partner node from the *slave pair*, as explained previously. Since each *slave node* also has a replica of each of its database, Figure 7 illustrates an example of how data is stored in *slave pair* 1.

In real-world scenarios, clusters of shared-nothing nodes are heterogeneous environments. Many of the machines have distinct hardware (CPU, RAM, etc.) and software programs (operating system, resident applications, etc), all of which execute at different paces and with variable frequency. This implies that each machine/node has its own individual performance, which is mostly different from the rest and in contrast with the overall performance of the cluster. Besides these features, the amount of RAM reserved for the DBMS database cache and other critical operations in each node is also a very significant performance variable. Thus, each node can process queries faster or slower from the rest.

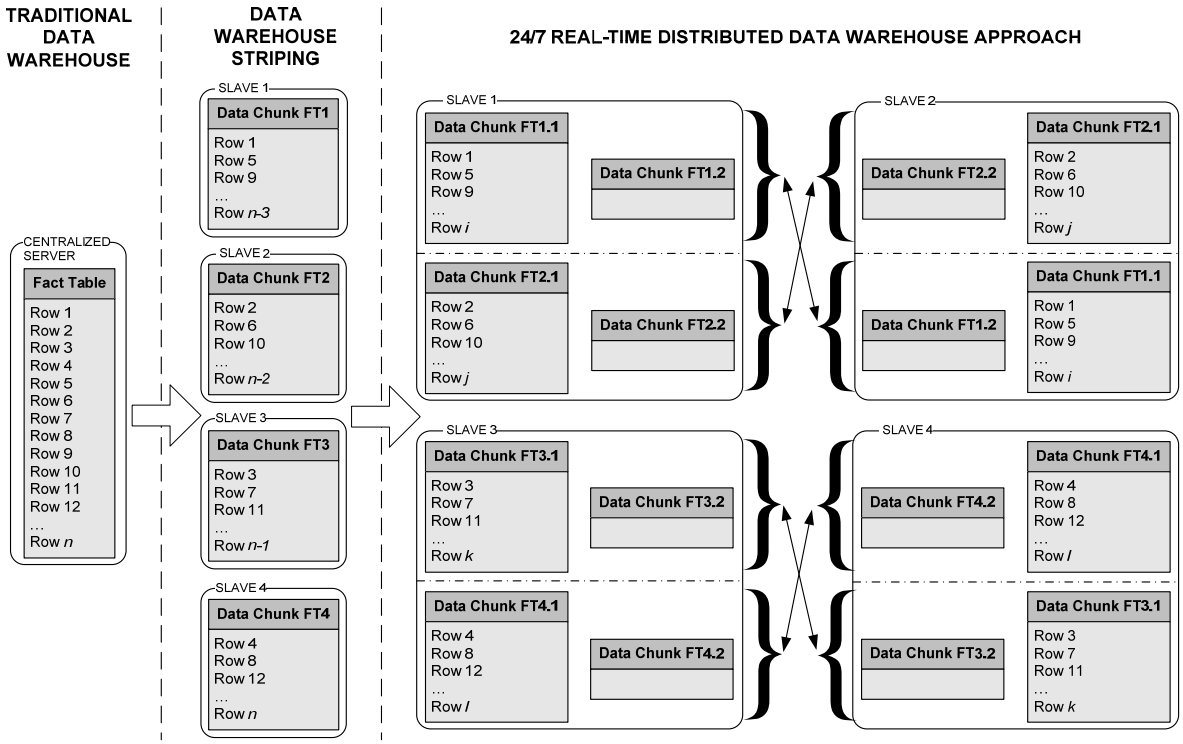

Figure 6. Comparing the Data Warehouse Striping data distribution method with the 24/7 Real-Time Distributed DW approach

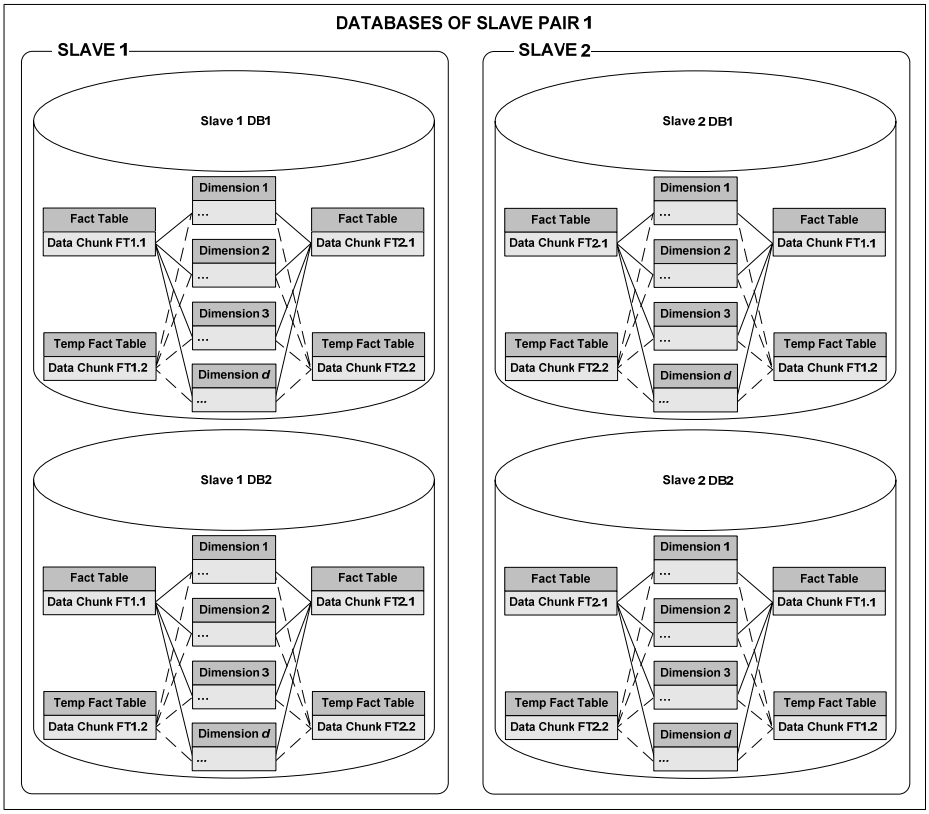

Figure 7. Example of the database setups in *slave pair* 1

The performance coefficient for each *slave node* results of a mix of equal weights from a measure given by a performance benchmark application such as [16] on the (Board, CPU, RAM, Hard Disk) components for each node and the amount of dedicated memory made available by the node for DBMS memory management such as database caching and user connection handling (*e.g.* the Oracle SGA and PGA). Suppose the performance benchmark measures and dedicated DBMS RAM for each *slave node* in a cluster of four *slave nodes* are as shown in Table I.

Given the resulting individual performance coefficients, for every set of 100 fact table rows, *slave node* 1 would store 29 rows, *slave node* 2 would store 22 rows, *slave node* 3 would store 17 rows, and *slave node* 4 would store 32 rows. Distributing data according to this will balance the amount of data to process by each *slave node* according to its individual performance features in relation to all remaining nodes and thus, balance response time amongst the nodes in a way that improves the cluster's overall response time.

Table I. Example of assessing individual performance coefficients

	Slave Node 1	**Slave Node 2**	**Slave Node 3**	**Slave Node 4**	**Σ**
Performance Benchmark	6.5	7.0	4.5	7.5	25.5
Perf. Bench. Coefficient	*6.5 / 25.5 = **25.5%***	*7.0 / 25.5 = **27.5%***	*4.5 / 25.5 = **17.6%***	*7.5 / 25.5 = **29.4%***	***100%***
DBMS Ded. Memory	512MB	256MB	256MB	512MB	1536MB
DBMS Mem. Coefficient	***33%***	***17%***	***17%***	***33%***	***100%***
Individual Coefficient	**58.5% / 2 = 29%**	**44.5% / 2 = 22%**	**34.6% / 2 = 17%**	**62.4% / 2 = 32%**	**100%**

C. Query workload execution and balancing

As we previously described, each *slave node* will be requested by a *master node* to execute each user query, producing a partial answer (if the query is to execute against a fact table) or a complete answer (if the query executes only against dimension tables). For the first case, all partial answers returned by the *slave nodes* will then be aggregated by a *master node* to produce the final answer, which will be returned to the user. For the second case, once an answer is received by the *master node*, all remaining queries are terminated at once, since all other responses will be exactly the same (given that all *slave nodes* have complete exact copies of all dimension tables).

If several nodes are occupied processing other queries when a new query is requested to execute, the *master node* will balance the workload by executing the partial queries against *slave nodes* that are idle or that are executing tasks with lower computational efforts, within each pair. To evaluate the computational efforts of a list of executing tasks in a *slave node*, we take the estimated query costs given by DBMS query cost estimator multiplied by the performance coefficient ratio between the nodes, at that node. As an example, consider the following scenario:

- We want to execute a new query Q3 against the nodes in *slave pair* 1, composed by *slave node* 1 and *slave node* 2;
- *Slave node* 1 is currently executing queries Q1 and Q2, which respectively have estimated costs of 2364.62 and 1222.1 (totalizing 3586.72), given by the DBMS query cost estimator;
- Query Q3 has an estimated cost of 345.2;
- *Slave node* 2 is currently executing queries Q1 and Q3, which respectively have costs of 2364.62 and 852.2 (totalizing 3216.82);
- If we refer to Table I shown in subsection III.B, *slave node* 1 and *slave node* 2 have performance coefficients of 29% and 22%, respectively.

The question for the *master node* is: should each partial query of Q3 be respectively processed by each *slave node* in *slave pair* 1, or should one of the *slave nodes* assume processing both partial queries? To answer this question, the *master node* needs to assess which hypothesis is potentially faster. With the mentioned measures from Table I, the effort ratio of *slave node* 1 relatively to *slave node* 2 is 22%/29% = 0.759, while the effort ratio of *slave node* 2 is 29%/22% = 1.318. Thus, the predictable computational effort measures of each node in the pair for processing their current tasks are:

Slave node 1: 3586.72*0.759 = 2722.32
Slave node 2: 3216.82*1.318 = 4239.77

Now, assuming that each *slave node* would subsequently process a partial Q3, we have an estimated cost of:

Slave node 1: 2722.32 + 345.2*0.759 = 2984.33
Slave node 2: 4239.77 + 345.2*1.318 = 4694.74

This means that we can assume Q3 to finish after a cost of 4694.74, since we can only build the final answer after both Q3 partial queries have been processed. Now, if both partial queries Q3 were processed only by *slave node* 1:

Slave node 1: 2722.32 + (345.2*0.759)*2 = 3246.33
Slave node 2 maintains its previous cost: 4239.77

In this hypothesis, we could build the final Q3 answer after a cost of 3246.33. Finally, if both partial queries Q3 were processed only by *slave node* 2:

Slave node 1 maintains its previous cost: 2722.32
Slave node 2: 4239.77 + (345.2*1.318)*2 = 5149.72

Regarding these calculus, the *master node* would choose only *slave node* 1 to process both Q3 partial queries, since this is the hypothesis in which both slave nodes would finish earlier because this hypothesis has smallest efforts to finish Q3 (3246.33<4694.74<5149.72). Besides presenting better response time, this query balancing approach would release *slave nodes* to an idle state earlier than the "one partial query – one slave node" DW-S approach, which would have an estimated cost of 4694.74.

D. Other functional issues

Every *slave node* in the system has a *24/7 RTDW Tool Database* such as it is described in [19] and in section II.B of this paper. Additionally to what is described in [19], it also stores communication and database access information on its partner within the *slave pair* to which it belongs.

In each *master node*, the *24/7 RTDW Tool Database* additionally stores all the data that exists in each of the *slave nodes 24/7 RTDW Tool Database*, plus: information on which is the performance coefficient of each *slave node*, as well as communication and database access information;

status on which user queries and what data loads are being executed, as well as a query history log and a data load log; a list of its *master node* pairs. When new *slave pairs* are added to the cluster, the DBA just needs to update the *master nodes' 24/7 RTDW Tool Database*.

In order to verify if any *slave node* is offline, a *master node* periodically tries to ping to each *slave node*. If a *slave node* does not respond, the *master node* stores that status in its *24/7 RTDW Tool Database* and the DBA is alerted. In what concerns data loading, as we mentioned previously, each *slave node* within a given *slave pair* is an exact replica of its partner. This means that both *slave nodes* in a pair receive all the new data loaded by its partner. When new data is received by a *master node* to update an offline *slave node*, the *master node* takes notice of the batch of rows referring to it and loads this data into the formerly unavailable node from its partner when it is restores its online status. The exception is if both *slave nodes* in the same *slave pair* are unavailable. In this case, the *master node* retains the data to load into that pair of *slave nodes* and subsequently executes the data load when they restore their online status.

In what concerns querying, an individual timeout is defined for each *slave node* to process each partial query, as well as a timeout for waiting for the remaining responses as the first nodes respond back to the *master node*, considering the estimated processing costs as shown in the previous subsection. In case any *slave node* exceeds those timeouts, the *master node* assumes the node has killed the partial query and builds an approximate answer from the partial results it has already received from the remaining available nodes, using statistical formulas according to the DW-S approach as explained in [3]. Thus, one or more nodes that are momentarily unavailable do not cause the system to stop.

For example, for a query to calculate a sum, if we have the partial answers $\{S_1, S_2, \ldots, S_i\}$ from i available nodes of a cluster with n nodes, the approximate answer is calculated by multiplying the average result from the available nodes by the total number of nodes: $((S_1+S_2+\ldots+S_i) / i) * n$. This approximate answer approach is valid for all mathematical calculus and is thoroughly discussed in [3], and due to lack of space will not be included here. In these cases, the users receive an estimated answer with a confidence interval for a given confidence probability. For example, SUM(estimated) = 233.45±12.5 with 95% confidence. The *master node* also alerts the DBA to check the *slave nodes* to which the timeouts refer. When ETL tools or users cannot access a *master node*, they can request its pair in the *master pair*.

IV. Experimental Evaluation

To evaluate our proposal, we implemented the 10GB size database of TPC-H decision support benchmark [22] using the Oracle 11g DBMS. All machines were identical Pentium Core2Duo 3GHz shared-nothing PCs, with 2GB of SDRAM and 160GB SATA hard disks. We evaluated the performance of the nodes with SiSoftware Sandra 2012 [16] and compare the results between our method and the standard DW-S round-robin technique. We shall not be concerned with the time spent in parsing and splitting the user query, communication costs in exchanging data between slaves and master node, and merging the partial answers. We focus on the response time of each node for each workload. The TPC-H query workload used in all tests was composed of TPC-H queries 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 19, 20 and 21, as a representative sample of both fact table and dimension table queries. We tested our approach and DW-S using clusters with 2, 4, 8 and 10 nodes, in scenarios with 1 to 10 users simultaneously querying the DW.

A. Query Execution without Data Loading

We have tested our proposal and the DW-S technique in setups that executed a predefined TPC-H query workload against static data (*i.e.*, processing queries without having to simultaneously handle data loading procedures), to compare them in what concerns purely query execution. The first tested setups were performed in a homogeneous cluster (all identical machines, with the same amount of predefined dedicated RAM for each node's database cache - 756MB). The results for these setups are shown in figure 8.

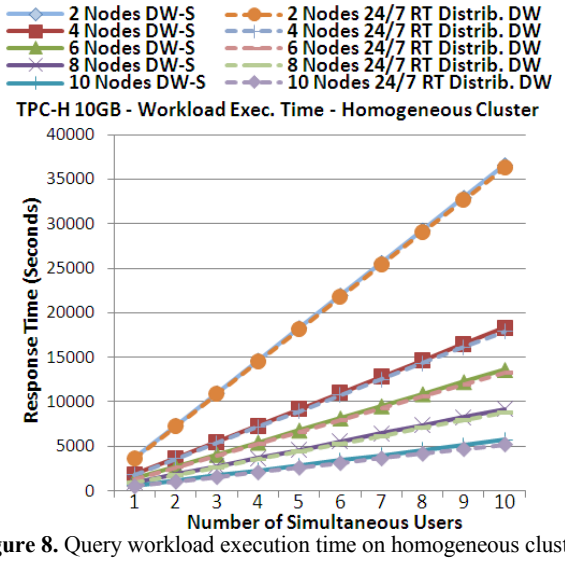

Figure 8. Query workload execution time on homogeneous cluster

Given that the cluster is composed of identical machines with identical software/hardware features, the results for DW-S and our approach are similar, as would be expected. However, it can be noticed that the results for our approach are slightly better. Although they are identical, the nodes have slightly different performance coefficients, meaning they store slightly different amounts of data in our approach, which leads to slightly better overall results, as we expected.

We repeated the same experiments using a cluster where half the nodes have 512MB dedicated database RAM and the other half 1024MB dedicated database RAM, composing a heterogeneous cluster. The results of these experiments are shown in figure 9. As can be seen, our approach presents much better results than DW-S. Since the master nodes need to wait for each node to finish processing each partial query and given that each node processes the same amount of data in the DW-S setups, the master needs to wait for the slower nodes, which represents a delay that does not occur in our approach, given that the faster nodes process more data than the slower nodes and therefore, minimizes waiting time.

This allows our approach to approximately maintain its overall performance, while DW-S introduces response time overheads from 20% (in the 10 node cluster) to 24% (in the 2 node cluster) for the tested scenarios. For example, in the 10 user 10 node cluster setup, the workload response time is approximately 5000 seconds using our approach, while the DW-S solution takes almost 7000 seconds. In the 10 user 2 node cluster setup, DW-S takes more than 45000 seconds, while our approach takes approximately 36000 seconds.

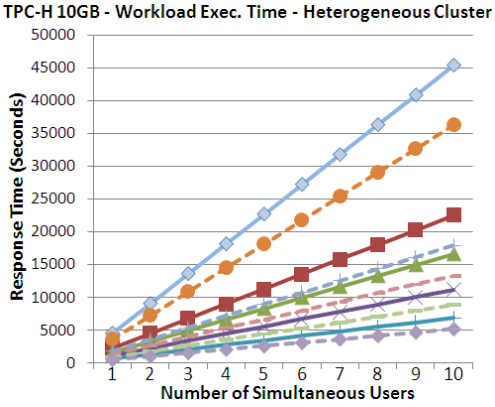

Figure 9. Query workload execution time on heterogeneous cluster

B. *Query Execution while executing Data Loading*

To measure the impact introduced in query response time by simultaneously loading data into the DW while the queries are processed, we also tested all the scenarios and setups mentioned in the previous subsection, while loading sets of 600 TPC-H transactions (approximately 65MB of data) every 30 seconds using Oracle's SQL*Loader. The results for the mentioned homogeneous and heterogeneous clusters are shown in Figures 10 and 11, respectively. The introduced overheads in the DW-S are even proportionally larger when compared to those introduced by our approach, as can be seen in the figures. While our approach manages to maintain response time overheads that range from 6% to 18% and are similar in both clusters, the DW-S solution will introduce from 6% to 31% in the homogeneous clusters and 9% to 39% in the heterogeneous clusters, for the tested setups and scenarios.

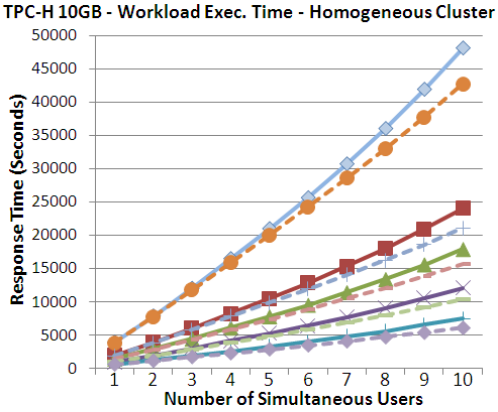

Figure 10. Query workload execution time on heterogeneous cluster while performing data loading

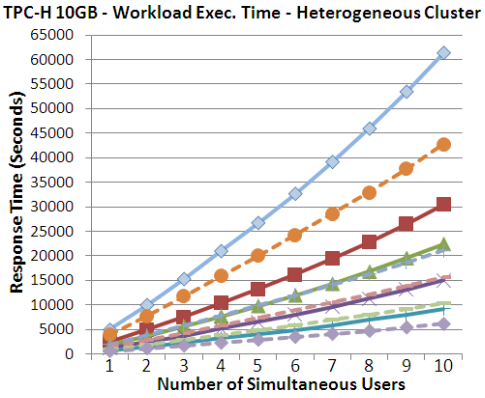

Figure 11. Query workload execution time on heterogeneous cluster while performing data loading

C. *Discussion and Remarks*

The DW-S proposal [3] means to achieve nearly optimal speed and scale up. This is due to the fact that typical queries lead to sub-queries executed in a completely independent way in each node and each query returns a partial result that is easy to merge to compute the final answer. That is, each node locally and independently processes the same amount of heterogeneous data as all the remaining. However, to obtain these results, the DW-S setup assumes that all nodes are similar machines processing approximately the same amount of data and thus, taking approximately the same time to process each partial query. As we previously mentioned, many real-world DWs use shared-nothing clusters composed by machines with distinct hardware and software features. As the results show, in these heterogeneous environments our proposal easily outperforms the standard fixed-size round-robin data distribution approach and presents better performance, taking advantage of each node's independent characteristics. If we consider that a change in the amount of dedicated database RAM in the nodes is effectively a slight difference between them, then we easily state that in a cluster composed of heterogeneous machines, each with distinct hardware components (CPU, RAM, etc) and executing different software applications at different rates, our approach can achieve much better results than standard round-robin fixed-size distributed databases.

V. Related Work

Our previous work [17, 18] is focused on optimizing data loading procedures for dealing with real-time data warehousing requirements, while [19] focuses on ensuring ongoing availability. In these papers, we thoroughly discuss the issues involved in those features and used in this paper. The concept of DW-S and its issues is described in [3].

Aster Data is a leading commercial DW with 24/7 availability [1]. This solution has built-in fault tolerance and self-healing capabilities to handle hardware and software failures. They use data replication with transparent fail-over and perform online restoration of backups whenever a fault is detected, executing online resynchronization. They also allow executing online data loads and exports, as well as adding new servers to the system without downtime.

Relating with Real-Time Data Warehousing, in [11] the authors present a architecture on how to define the types of update and time priorities (immediate, at specific time intervals or only on DW offline updates) and respective synchronization for each group of transactional data items. In [21] the authors propose using SQL INSERT-like loading instructions with bulk load speed, taking advantage of in-memory databases for data loading in the DW. An approach using SSD for caching updates in presented in [6], able to maintain query availability while adding fresh data to the DW. The work in [10] refers both query execution and data loads in analytical environments, discussing a comparison of techniques for loading and querying data simultaneously.

Solutions on data distribution and load balancing for data centers and transactional distributed databases are proposed in [4, 5, 7, 9]. These architectures typically use key-based hash or range methods to assign data to nodes in the cluster and work around consistency issues. The work in [8, 15] also focus on these aspects, but include replication strategies for ensuring high availability and fault tolerance. Using replication and virtualization for ensuring high availability and small performance overheads is proposed in [14].

MDHF [20] uses join-bitmap indexes with workload-based attribute-wise derived data distribution, hierarchy-aware processing and in-memory retention of dimensions for efficient processing of star schemas. A MapReduce-style fault tolerance strategy for improving query performance in case of node unavailability for long running queries is proposed in [23] for shared-nothing distributed databases, using data replication. A similar MapReduce-based solution is discussed in [6], including a set of related optimization techniques such as data storage and compression, as well as multi-query management, for a specific advertisement DW.

VI. Conclusions and Future Work

This paper proposes a solution for ensuring continuous availability while enabling simultaneously loading data and processing of decision support queries in distributed DWs. We achieve continuous availability by replicating databases within each node and within the assigned pairs of nodes, in which one node can respond on behalf of its partner for both data loading and querying, or can be used to rebuild its partner. This allows performing all kinds of planned and unplanned downtime tasks without taking the system offline. Approximate query answering is also used if any pair of nodes is down, ensuring the system will always produce a response to its users.

To improve the overall performance of a cluster, we distribute the amount of data in each node according to its individual performance coefficient relatively to all the nodes in the cluster, allowing faster nodes to process more data than slower nodes. We can also improve performance by processing both partial query responses of a pair in just one of the slave nodes, in case it is faster than the other or if the other is already occupied processing any other user query. This allows load balancing query workload jobs to improve the response time of the partial query responses in each pair. The experimental results show that our data distribution approach produces better performance results than the standard fixed-size round-robin distributed data approach used in most distributed DWs in the tested setups, for both homogeneous and heterogeneous clusters.

As future work, we intend to test and improve the data loading technique for using in-memory databases to speed up those procedures. We will also adapt our approach and evaluate it for virtualization and cloud DW setups.

References

[1] Aster Data, *Aster Data nCluster: "Always On" Availability*, Aster Data Systems, 2009.

[2] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica, *"MaSM: Efficient Online Updates in Data Warehouses"*, SIG Int. Conference on Management Of Data (SIGMOD), 2011.

[3] J. Bernardino, P. Furtado, and H. Madeira, *"Approximate Query Answering Using Data Warehouse Striping"*, Journal of Intelligent Information Systems (JIIS), Vol. 19, No. 2, pp. 145-167, 2002.

[4] D. G. Campbell, G. Kakivaya, and N. Ellis, *"Extreme Scale with full SQL Language Support in Microsoft SQL Azure"*, SIG Int. Conference on Management Of Data (SIGMOD), 2010.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *"Bigtable: A Distributed Storage System for Strcutured Data"*, Int. USENIX Symp. on Oper. Systems Design and Implementation (OSNI), 2006.

[6] S. Chen, *"Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce"*, Int. C. Very Large DataBases (VLDB),2010.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, *"PNUTS: Yahoo!'s Hosted Data Serving Platform"*, Proceedings of Very Large DataBases (PVLDB), 2008.

[8] C. Curino, E. Jones, Y. Zhang, and S. Madden, *"Schism: a Workload-Driven Approach to Database Replication and Partitioning"*, Int. Conference on Very Large DataBases (VLDB), 2010.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and M. Vogels, *"Dynamo: Amazon's Highly Available Key-Value Store"*, ACM Symposium on Operating Systems Principles (SOSP), 2007.

[10] G. Graefe, *"Fast Loads and Fast Queries"*, International Conf. on Data Warehousing and Knowledge Discovery (DAWAK), 2009.

[11] I. C. Italiano, and J. E. Ferreira, *"Synchronization Options for Data Warehouse Designs"*, IEEE Computer Magazine, 2006.

[12] R. Kimball, and M. Ross, The Data Warehouse Toolkit – The Complete Guide to Dimensional Modeling, John Wiley & Sons,2002.

[13] R. Kimball, and J. Caserta, The Data Warehouse ETL Toolkit, Wiley Computer Pub., 2004.

[14] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield, *"RemusDB: Transparent High Availability for Database Systems"*, Int. Conference on Very Large DataBases (VLDB), 2011.

[15] J. Rao, E. J. Shekita, and S. Tata, *"Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore"*, Int. Conference on Very Large DataBases (VLDB), 2011.

[16] SiSoftware, Sandra 2012 Computer Benchmark Application, http://www.sisoftware.net

[17] Our previous work. Not available due to double blind review.

[18] Our previous work. Not available due to double blind review.

[19] Our previous work. Not available due to double blind review.

[20] T. Stohr, H. Martens, and E. Rahm, *"Multi-Dimensional Database Allocation for Parallel Data Warehouses"*, Int. Conference on Very Large DataBases (VLDB), 2000.

[21] C. Thomsen, T. B. Pedersen, and W. Lehner, *"RiTE: Providing On-Demand Data for Right-Time Data Warehousing"*, Int. Conference on Data Engineering (ICDE), 2008.

[22] Transaction Processing Performance Council, TPC Benchmark H (TPC-H), http://www.tpc.org/tpch/.

[23] C. Yang, C. Yen, C. Tan, and S. Madden, *"Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database"*, Int. Conference on Data Engineering (ICDE), 2010.