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

Ricardo Jorge Santos  
CISUC – DEI – FCTUC  
University of Coimbra  
Coimbra, Portugal  
lionsoftware.ricardo@gmail.com

Jorge Bernardino  
CISUC – DEIS – ISEC  
Polytechnic Institute of Coimbra  
Coimbra, Portugal  
jorge@isec.pt

Marco Vieira  
CISUC – DEI – FCTUC  
University of Coimbra  
Coimbra, Portugal  
mvieira@isec.pt

Abstract—Real-time Data Warehouses (DWs) must be able to deal with continuous updates while ensuring 24/7 availability. To improve their performance, distributing data using round-robin algorithms on clusters of shared-nothing machines is normally used. This paper proposes a solution for distributed DW databases that ensures its continuous availability and deals with frequent data loading requirements, while adding small performance overhead. We use a data striping and replication architecture to distribute portions of each fact table among pairs of slave nodes, where each slave node is an exact replica of its partner. 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 query performance. We include experiments using the TPC-H decision support benchmark to evaluate the scalability of the proposed 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

Nowadays, many enterprises function on a 24/7 business schedule on a non-stop fashion. This has shifted the decision support paradigm: decisions need to be made much more frequently with the most recent business data. Thus, today’s Data Warehouse (DW) requires both ongoing availability (continuously providing information to decision makers) and highly frequent data loading (updating the database as quickly as possible to include the most recent business data), in order to fulfill its decision support purpose. They typically have databases with millions of rows or more and huge storage space, where many decision support queries can run up to hours. As stored data size increases, performance degrades. To improve performance in these environments, distributed database architectures are used, where the most common architecture is to distribute the database among shared-nothing clusters of cheap commodity PCs [6].

Typical distributed databases divide tables into separate chunks of data of roughly the same size using a round-robin algorithm, storing a chunk in each node [2, 3, 6, 12]. To process a query, a central node receives it, splits it into partial queries to process 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 equal fixed-size data distributions seem good for homogeneous environments, many DW setups function in heterogeneous environments, in which the nodes have distinct individual performance. Thus, approaches that use this feature for load balancing in both data loading and querying may produce better results.

Nowadays, most real-time DWs use frequent micro-batch bulk loading of new data, while they are kept online for querying purposes [2]. However, the update frequency rate and how much data should be loaded in each batch are extremely diverse [9]. Moreover, the overhead in loading and querying performance is also dependent on 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 [9].

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 downtime and ensure its 24/7 availability and data freshness, 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 in a non-stop fashion in heterogeneous hardware and software setups. Pulling this together is not a trivial task.

Our proposal

In this paper, we engineer data replication and striping techniques for enabling 24/7 availability (including fault tolerance and self-healing) in distributed real-time DWs. Our approach focuses on ensuring the non-stop availability and rapid updating of the DW, while improving performance 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.

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 by 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.

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. Section IV shows experimental evaluations of the proposed approach, comparing it with the round-robin DW-S. Section V describes related work on availability and real-time distributed DW solutions. Section VI presents our conclusions and points out future work.

II. BUILDING A 24/7 REAL-TIME CENTRALIZED DW
Changing a traditional centralized DW into a 24/7 real-time centralized DW
In this subsection we summarize previous work [15, 16], 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 with no performance optimization data structures such as primary keys and other indexes are capable of appending data much faster than tables with 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 [10], 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 [10], 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 [10]. 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 [11].

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, SUM functions grouped by primary keys can be used for producing the correct value of the transaction.

Our work in [15, 16] thoroughly explains the issues concerning this approach. With this method, any appending, updating or eliminating data on transactional systems results in new DW row insertions, minimizing 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, meaning that only the fastest methods to refresh the DW are used [9, 11, 16]. 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 [11].

Obtaining continuous availability in a centralized DW
The architecture of the latest version of our work, a 24/7 real-time centralized (single-server) DW (RTDW) system, published in [17], is shown in Figure 1. Using data replication and the techniques proposed in [15, 16], the 24/7 availability is based on the following:
- The server has a complete duplicate of each database;
- Both databases will be updated simultaneously, but only one at a time is made available/online for users (this is transparently handled by the middleware, which 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 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 [17], a set of middleware applications transparently deal with both data loads and query processing. Each load is treated in a transaction-like fashion (instead of individual row micro-batch bulk loading) to solve referential integrity issues from our previous work [15]. A transaction is defined as a set of all dimension table rows needed for each fact table row. For example, this means that before inserting a new Sales fact table row, the middleware would previously confirm that the rows for the CustomerID and ProductID key values, for instance, already exist in the respective Customer and Product dimension tables. If they do not exist, they are created before updating the Sales fact table.
The 24/7 RTDW Tool Manager allows the DBA to manage the 24/7 RTDW Tool Database and monitor data loading executions. It also allows building the original DW schema duplicates and setting up all values when the tool is used for the first time, as well as reoptimizing any database. The 24/7 RTDW Tool Database stores the information for system management, such as information on DW tables and other data structures (indexes, materialized views, etc.), definitions of transactions, how often new data is to be loaded, which is the current 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 [17].

### III. THE 24/7 REAL-TIME DISTRIBUTED DW

#### The 24/7 Real-Time Distributed DW Architecture

Figure 2 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 hold a list of all slave pairs and respective slave nodes and are responsible for defining the amount of data to load into each slave node’s database, sending the 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 the answers to the users which requested them. Master nodes are also responsible for verifying if a slave node or slave pair is down, correcting defective slave nodes and alerting 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.

#### Data Distribution and Load Balancing

To distribute data among the slave nodes using the DW-Striping (DW-S) technique, 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. 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.

An example of how our approach would divide a given fact table comparatively to the DW-S approach is shown in Figure 3, 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 3, the fact table chunks in DW-S are named FT\( x \), where \( x \) represents the slave.
node number, while in our approach each of those chunks is named \( F_{TXY} \), 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 4 illustrates an example of how data is stored in slave pair 1.

Figure 4. Example of the database setups 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. Besides these features, the amount of RAM reserved for the DBMS database cache and other 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 these 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 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 1. Example of assessing individual performance coefficients

<table>
<thead>
<tr>
<th>Performance Benchmark</th>
<th>Slave Node 1</th>
<th>Slave Node 2</th>
<th>Slave Node 3</th>
<th>Slave Node 4</th>
<th>( \Sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perf. Bench. Coefficient</td>
<td>6.5</td>
<td>7.0</td>
<td>4.5</td>
<td>7.5</td>
<td>25.5</td>
</tr>
<tr>
<td>DBMS Ded. Memory</td>
<td>512MB</td>
<td>256MB</td>
<td>256MB</td>
<td>512MB</td>
<td>1536MB</td>
</tr>
<tr>
<td>DBMS Mem. Coefficient</td>
<td>33%</td>
<td>17%</td>
<td>17%</td>
<td>33%</td>
<td>100%</td>
</tr>
<tr>
<td>Individual Coefficient</td>
<td>58.5% / 2</td>
<td>44.5% / 2</td>
<td>34.9% / 2</td>
<td>62.4% / 2</td>
<td>100%</td>
</tr>
</tbody>
</table>

A. 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).

Figure 3. Comparing the Data Warehouse Stripping data distribution method with the 24/7 Real-Time Distributed DW approach
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 former queries Q1 and Q2, which respectively have estimated costs of 2364.62 and 1222.1 (total 3586.72), given by 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 (total 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, the master node needs to assess which hypothesis is potentially faster. With the 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 to 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 and 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 and Slave node 2 = 4239.77 + 345.2*1.318 = 4694.74. This means Q3 would 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 and 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 2 = 4239.77 + (345.2*1.318)*2 = 5149.72.

Regarding these calculations, 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 (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.

IV. EXPERIMENTAL EVALUATION

To evaluate our proposal, we implemented the 10GB size database of TPC-H decision support benchmark [19] 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 [14] 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 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.

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. We configured the clusters with half the nodes having 512MB dedicated database RAM and the other half 1024MB dedicated database RAM, composing heterogeneous clusters. The results of these experiments are shown in Figure 5. 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.

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 are shown in Figure 6. 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.
node and within the assigned pairs of nodes. Each node can respond on behalf of its partner for both data loading and querying, or be used to rebuild its partner. This allows executing all sorts 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 always produces a response to its users. To improve cluster overall performance, the amount of data in each node is defined according to its individual performance coefficient relatively to the remaining nodes, allowing faster nodes to process more data than slower nodes. We also improve performance by load balancing workloads between nodes belonging to the same pair, to lower the response time of the partial query responses in each pair. The experiments show our data distribution approach produces better results than standard fixed-size round-robin distributed data approach used in most distributed DWs. As future work, we intend to improve data loading by using virtualization and in-memory databases to speed up those procedures. We will also adapt our approach and evaluate it for virtualization and cloud DW setups.

REFERENCES


Discussion and Remarks

The DW-S proposal [3] means to achieve nearly optimal speed and scale up, due that it assumes that nodes are similar machines and each node locally and independently processes the same amount of heterogeneous data and thus, takes nearly the same time to process each partial query. However, many real-world DWs use shared-nothing clusters composed by machines with distinct hardware and software features. As shown, in heterogeneous environments our proposal outperforms the standard fixed-size round-robin data distribution approach, taking advantage of each node’s distinct features. If we consider that a change in the amount of dedicated database RAM in the nodes is effectively a slight node's distinct features. 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 can 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 [15, 16] is focused on optimizing data loading procedures for dealing with real-time DW requirements, while [17] 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], using data replication with transparent fail-over and performing online restoration of backups when a fault is detected, executing online resynchronization. It also allows executing online data loads and exports, and adding new servers to the system without downtime. In [18] the authors propose SQL INSERT-like loading instructions using in-memory databases for DW loading. An approach using SSD for caching updates is proposed in [6], maintaining query availability while adding fresh data to the DW. The work in [9] discusses comparisons between techniques for loading and querying data simultaneously in analytical environments. Data distribution and load balancing solutions for data centers and transactional distributed databases are proposed in [4, 5, 7]. 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, 13] also focus on these aspects, but include replication strategies for ensuring high availability and fault tolerance. Using replication and virtualization for high availability and small performance overheads is proposed in [12].

VI. CONCLUSIONS AND FUTURE WORK

This paper proposes a solution for ensuring continuous availability while efficiently loading data in distributed DWs. We achieve continuous availability by replicating databases within each node and within the assigned pairs of nodes. Each node can respond on behalf of its partner for both data loading and querying, or be used to rebuild its partner. This allows executing all sorts 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 always produces a response to its users. To improve cluster overall performance, the amount of data in each node is defined according to its individual performance coefficient relatively to the remaining nodes, allowing faster nodes to process more data than slower nodes. We also improve performance by load balancing workloads between nodes belonging to the same pair, to lower the response time of the partial query responses in each pair. The experiments show our data distribution approach produces better results than standard fixed-size round-robin distributed data approach used in most distributed DWs. As future work, we intend to improve data loading by using virtualization and in-memory databases to speed up those procedures. We will also adapt our approach and evaluate it for virtualization and cloud DW setups.