Relaxing Freshness to Improve Load Balancing in a Cluster of Autonomous Replicated Databases

Stéphane Gançarsi† - Cécile Le Pape† - Patrick Valduriez‡

† Firstname.Lastname@lip6.fr, LIP6, Paris, France
‡ Patrick.Valduriez@inria.fr, INRIA and IRIN, Nantes, France

1 Introduction

In [GNV02, GNPV02], we proposed a solution for hosting autonomous applications and databases in a cluster system. This solution exploits database replication to provide both high-availability and high-performance. This work is done in the context of the Leg@net project † whose objective is to demonstrate the viability of the Application Service Provider model for pharmacy applications. In particular, we must efficiently support mixed workloads composed of front-office update-intensive transactions (e.g., drug sales) and back-office read-intensive queries (e.g., statistics on drugs sold).

With optimistic replication [OV99], different transactions (and queries) can be performed in parallel at different nodes without any waiting. However, conflicting updates can create copy consistency problems, i.e. replicas diverge, which need be solved by reconciliation. Replica divergence must be controlled for two reasons. First, reconciliation gets more difficult and time-consuming as divergence increases. Application semantics may help in this case. For instance, if transactions \( t_1 \) and \( t_2 \), executed at nodes \( n_1 \) and \( n_2 \) respectively, are commutative, reconciliation can simply perform \( t_1 \) at \( n_2 \) and \( t_2 \) at \( n_1 \). Second, read-only queries do not always require perfectly consistent data and may tolerate some inconsistency, for instance “computing the daily turn-over of Tylenol with a tolerated imprecision of 5%”. In this case, application semantics is modelled by consistency requirements which express how much the result of a given query may differ from the expected results if the query was sent to a consistent node.

In this paper, we address the problem of expressing and exploiting consistency requirements in order to optimize the execution of read-only queries. It can be stated as follows: given an autonomous database replicated in optimistic mode, evaluate the level of copy consistency to route a query to a node such that (1) the copy consistency level guarantees that the query result will meet the query consistency requirements and (2) the choice of the node optimizes load balancing. We focus on an important type of consistency, called view consistency, which states that transactions read the appropriate value with respect to their serialization order. It may be violated if a transaction reads data at a node where reconciliation is needed. For simplicity, we use a mono-master replication scheme: update transactions are all sent to a single master node while read-only queries may be sent to any node. Slave nodes are updated asynchronously through refresh transactions and may contain stale data until the refresh process is completed. In mono-master replication, consistency reduces to freshness. Data at a slave node is totally fresh if it has the same value as the same data at the master node, i.e. all the refresh transactions for those data have been propagated to the slave node. The freshness level reflects the distance between the data value at the slave node and that at the master node. For instance, assume a transaction \( T(id\_product, q) \) to sale quantity \( q \) of product \( id\_product \) which updates table \( STOCK(id, quantity, threshold, date\_last\_update) \). Assume now that query \( Q \) asks for the products to be re-ordered, i.e. the tuples in table \( STOCK \) with \( quantity < threshold \), and that \( Q \) tolerates a divergence of 5 units with respect to the actual stock. Thus, \( Q \) can be sent to a slave node \( N \) if \( T \) does not update any item in table \( STOCK \) by more than 5 units, i.e. \( N \) is fresh enough with respect to \( Q \).

This paper makes two main contributions. First, it proposes a freshness model for users to specify freshness requirements for queries. It also allows modelling the changes made by update transactions, in order to evaluate the freshness of slave nodes. Second, it proposes algorithms to evaluate data freshness and compute the minimum set of refresh transactions needed to guarantee that a node is fresh enough with

† Sponsored by the RNTL between LIP6, Prologue Software and ASPLIne
respect to a given query. The paper is organized as follows. Section 2 gives an overview of our system architecture. Section 3 presents the freshness model while section 4 gives the algorithms based on that model to optimize load balancing. Section 5 concludes.

2 Replicated database architecture

Figure 1 gives an overview of our cluster system Architecture which is derived from [GNPV02] . Metadata useful for load balancing is provided and managed by the DB administrator. It includes for instance the default level of freshness required by a read-only query. It also includes information on update transactions, so that the system can compute for each transaction, the maximum of change it will produce on the master copy. Clients send either transactions (which perform updates) or read-only queries to the respective manager. They may define several tries for the same query: each try may have a different level of required freshness. Tries are organized as a graph (DAG), so that the system can perform tries according to its partial order, until satisfying the corresponding level of freshness. The default freshness level may be overwritten by the client when defining the query try graph. When the load balancer receives a transaction, it sends it to the master node. It also sends the maximum of change that the transaction will produce to the freshness evaluation module which maintains, for each replica at a slave node, an upper bound of its divergence with respect to the master node. This information is used, together with usual information (CPU, disk) on nodes’ load, to send query tries to the least loaded node. Refresh transactions are sent to the slave node periodically or may be forced by the load balancer in order to make a slave node fresh enough for a given query try.

![Diagram of mono-master replicated database architecture](image)

**Figure 1: Mono-master replicated database architecture**

3 Freshness Model

In this section, we present a freshness model for queries and transactions. First, we describe how freshness requirements are specified for a query with a flexible freshness policy. We detail a set of freshness measures which allow users to specify the freshness of data that matches the semantics of the application. Second, we propose a simple model of transactions as sets of updates. We use this model to evaluate data freshness on slave nodes through confidence levels.
3.1 Specification of freshness requirements

Freshness requirements of queries are specified through a flexible model. Users specify execution units for queries called tries. Each try contains both a minimal freshness level for the results of the query and a maximal execution delay. Tries are organized as a directed acyclic graph called freshness policy: when a try fails, i.e. if the results cannot be retrieved at the required freshness level within the specified delay, a next try is issued according to the partial order of the freshness policy.

The freshness level of a query is defined as a logical formula: the results of the query at a slave node are fresh enough if the formula is satisfied at the node. A freshness level is composed of logical nodes, called freshness atoms. A freshness atom \( \mu(a) < t \) is a condition upon an access atom (relation, tuple, column or tuple attribute -also called element-) which bounds the staleness of the access atom \( a \) under a certain threshold \( t \) for a given freshness measure \( \mu \).

Classifications of freshness measures have already been proposed in the literature [ABGM90, BGM94, GN95, WYP92, YV02]. We propose an adaptation of these measures to our context.

Let \( a \) be a logical access atom ; \( a_i \) is a stale secondary copy of \( a \) at a slave node \( S_i \); \( \hat{a} \) is the fresh state of \( a \); Refresh\( i(a) \) is the sequence of refresh transactions updating \( a \) which have committed at the master node but have not yet been propagated to \( S_i \).

We consider the following measures:

1. \textbf{Numerical measure Num:} If \( a \) is a numerical tuple attribute, the freshness of \( a_i \) is the difference between \( a_i \) and \( \hat{a} \) : \( \text{Num}_{\text{elt}}(a_i) = \hat{a} - a_i \)

   If \( a \) is a column associated to a numerical attribute, the freshness of \( a_i \) is an aggregation of the numerical freshness of its elements \( e_k \) : \( \text{Num}_{\text{col}}(a_i) = \sum_{e_k \in a_i} \text{Num}_{\text{elt}}(e_k) \).

   If \( a \) is a tuple, the numerical freshness of \( a_i \) is the number of stale elements in \( a_i \).

   If \( a \) is a relation, the numerical freshness of \( a_i \) is the number of stale tuples in \( a_i \).

2. \textbf{Time measure Age:} If \( a \) is a numerical tuple attribute, the age of \( a_i \) is the time since \( a_i \) has been waiting for the first refresh transaction, \( t_k \). If \( a \) is a column, a tuple or a relation, the age of \( a_i \) is the age of the oldest element of \( a_i \).

3. \textbf{Ordering measure Order:} If \( a \) is a tuple attribute, the ordering measure of \( a_i \) is the cardinal of Refresh\( i(a) \). If \( a \) is a column, a tuple or a relation, the ordering measure of \( a \) is the maximum of the ordering measures of its elements.

4. \textbf{Complex measures} are linear combinations of \textit{Num}, \textit{Age} and \textit{Order}.

3.2 Update model

An update transaction \( t_j \) is associated to a set \( U_j \) of update atoms. An update atom \( (\mu(a) = \text{val}) \) is an association between an access atom \( a \), a measure \( \mu \) and a value \textit{val}. For example, \( (\text{Num} (\text{STOCK}) = +2) \) is an update atom of \( U_j \) which means that two tuples of the relation \textit{STOCK} are intended to be updated by transaction \( t_j \).

Freshness atoms cannot be evaluated exactly \textit{a priori}, but we can define an upper bound for them which we call confidence level. The confidence level of an access atom \( a \) for a freshness measure \( \mu \), denoted by \( \text{conf}(a, \mu) \), is a value which guarantees that: \( \mu(a) \leq \text{conf}(a, \mu) \). Therefore the following holds: \( \text{conf}(a, \mu) \leq \text{threshold} \Rightarrow (\mu(a) \leq \text{threshold}) \). Confidence levels are computed by aggregating all the update atoms from the refresh transactions in Refresh\( i(a) \).

4 Trading freshness for load balancing

In this section, we give algorithms that use the freshness model to optimize load balancing. A query is sent to a slave node only if the node satisfies the freshness level of the current try. Therefore, when choosing an execution node, the load balancer needs to know for every slave node (1) if the freshness level is already
satisfied by the node and (2) which refresh transactions must be sent to the node if it is not fresh enough.

evaluate_refresh(try, i) {
    transaction tk = ptr_i.trans;
    transaction tn = ptr_0.trans; //master node
    while (tk != null) do
        if fresh_enough(get_formula(try), tk, tn) then return (tk, ..., t);
        tk := tk.next;
    done
}

Figure 2: Evaluating refresh requirements of a slave node for a try

Figure 2 presents our algorithm to achieve this goal. Function evaluate_refresh(try, i) computes a sequence of transactions to be sent to node Si in order to guarantee the freshness level of a try try. This algorithm is based on a queue where incoming transactions are placed in the order of processing by the master node (which is not necessary the same as the incoming order). The refresh level of a slave node Si is represented by a “stack pointer” ptr_j on the queue: all the transactions preceding the transaction pointed to by ptr_j in the queue have already been executed at Si. Node Si is perfectly fresh when ptr_i meets ptr_0, the master node pointer. For example, Figure 3 shows a set of transactions T1, T2, ..., T6. The global execution order is (T2, T1, T3, T6, T4, T5). There are four slave nodes: S1 and S2 have processed transactions (T2, T1, T3); S3 has not been updated since the beginning may be due to a network failure and S1 is the only slave node perfectly fresh.

Figure 3: Transactions global ordering.

Function evaluate_refresh(try, i) simulates partial refresh of Si until it is fresh enough to satisfy the freshness level associated to try. It uses function fresh_enough(fresh_jv, tk, tn) which tests if the freshness level fresh_jv of a try is satisfied at a node where the sequence of transactions (tk, ..., tn) has not been propagated yet. The confidence level is computed as specified in section 3.2 and compared to the given threshold. If the value returned by the function is true, the system guarantees that the freshness requirement of the try is satisfied. Due to space limitations, function fresh_enough is not detailed here.

Using the preceding two functions, the load balancer knows, for each node, how many refresh transactions must be sent to reach the required freshness of a given query try. This information is used together with usual load information (node CPU load, network link load, ...) to determine the best node to perform a query try, with forced refresh if the node is not fresh enough. Such node selection requires additional computation such as cost function evaluation and is beyond the goal of this paper. If the result of the query

\footnote{The master node execution order is the global order of the transactions in a mono-master configuration.}
try is not given by the local DBMS within the required delay, a next try is then considered, at a possibly
different slave node.

5 Conclusion

In this paper, we addressed the problem of load balancing of read-only queries in a cluster of autonomous
databases with optimistic replication. We observed that many queries do not need to access perfectly fresh
data. This is the case in our ASP context with pharmacy applications. And accessing perfectly fresh data
may severely restrict load balancing because only a few nodes store those data. Thus, we strive to exploit
user requirements on copy consistency and thus relax the stronger constraint of data freshness.

Assuming mono-master replication, we propose a freshness model for users to specify the required fresh-
ness level for queries. The freshness model also allows modelling the changes made by update transactions,
in order to evaluate the freshness of the slave nodes. It is a flexible model since it allows users defining
different tries for the same query, each try with a different level of required freshness. We propose algorithms
to evaluate data freshness and compute the minimum set of refresh transactions needed to guarantee that
a node is fresh enough with respect to a given query.

We have started to implement the proposed solution on LIP6’s cluster architecture running Linux and
Oracle 8i. To validate our solution, we plan to run a number of queries on our pharmacy database with and
without replication, and study the effect of relaxing freshness on load balancing and performance. We will
also compare our solution to Oracle built-in replication service. We expect to have more gain on two typical
scenarios: speed up OLTP transaction by sending OLAP queries on slave nodes, speed up concurrent OLAP
queries by load balancing on different, fresh enough, nodes.

References

[ABGM90] Rafael Alonso, Daniel Barbará, and Hector Garcia-Molina. Data caching issues in an information

[BGM94] Daniel Barbará and Hector Garcia-Molina. The demarcation protocol: A technique for main-

[GN95] Rainer Gallensdörfer and Matthias Nicola. Improving performance in replicated databases
through relaxed coherency. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, edi-

[GNPV02] Stéphane Gançarski, Hubert Naacke, Esther Pacitti, and Patrick Valduriez. Parallel process-
ing with autonomous databases in a cluster system. In Proc. CoopIS (LNCS 2519), Irvine

[GNV02] Stéphane Gançarski, Hubert Naacke, and Patrick Valduriez. Load balancing of autonomous
applications and databases in a cluster system. In Proc. Distributed Data and Structure (WDAS),

1999.


[YV02] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency
model for replicated services. ACM Transactions on Computer Systems (TOCS), 20(3):239–282,
2002.