Refresco: Improving Query Performance Through Freshness Control in a Database Cluster

Cécile Le Pape[1], Stéphane Gançarski[1], and Patrick Valduriez[2]

[1] Laboratoire d'Informatique de Paris 6, Paris, France
`Firstname.Lastname@lip6.fr`
[2] INRIA and LINA, Nantes, France
`Patrick.Valduriez@inria.fr`

Abstract. We consider the use of a cluster system for managing autonomous databases. In order to improve the performance of read-only queries, we strive to exploit user requirements on replica freshness. Assuming mono-master lazy replication, we propose a freshness model to help specifying the required freshness level for queries. We propose an algorithm to optimize the routing of queries on slave nodes based on the freshness requirements. Our approach uses non intrusive techniques that preserve application and database autonomy. We provide an experimental validation based on our prototype *Refresco*. The results show that freshness control can help increase query throughput significantly. They also show significant improvement when freshness requirements are specified at the relation level rather than at the database level.

1 Introduction

Recently, the database cluster approach [4,8,9], *i.e.* cluster systems with off-the-shelf (black-box) DBMS nodes, has gained much interest for various applications such as Application Service Provider (ASP). In the ASP model, applications and databases are hosted at the provider site and accessed by customers, typically through the Internet, who are no longer concerned with data and application maintenance tasks. Through replication of customers' databases at several nodes, a database cluster can yield high-availability and high-performance at a much lower cost than with a DBMS on a tightly-coupled multiprocessor. In the Leg@net project[1], the objective is to demonstrate the viability of the ASP model using a database cluster for pharmacy applications in France. In particular, we must 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). In practice, front-office processing has priority over back-office processing which usually has to be performed during closing hours. Preserving

[1] Project sponsored by the RNTL between LIP6, Prologue Software and ASPLine.

autonomy is often of major importance in a database cluster. In the ASP context, autonomy means that applications and databases must remain unchanged when hosted at the provider site, in order to avoid high costs in migration and maintenance. Thus, our challenge is to exploit the cluster's parallelism to allow both front-office and back-office to be performed on-line, as efficiently as if they were local to the pharmacy site. Our approach is to capture application semantics for optimizing load balancing within the cluster system.

In [4], we discussed the architectural issues underlying our approach. We showed that using a transaction processing monitor or a parallel DBMS does not address the autonomy requirements. We also showed that synchronous (eager) replication is not appropriate for the ASP model, and we proposed an asynchronous (lazy) replication scheme. In order to avoid consistency problems, we use a mono-master (primary copy) replication scheme: *update transactions* (or *transactions* for short) are all sent to a single master node while *read-only queries* (or *queries* for short) 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. However, as the serialization order of refresh transactions on any slave node is the same as the serialization order of the corresponding transactions on the master node, we guarantee that queries always read a consistent state, though maybe stale, on slave nodes. This is obtained by sending refresh transactions sequentially to each slave node, according to the serialization (commit) order obtained on the master node. Mono-master replication has the advantage of simplicity and is sufficient in many cases where most of the conflicts occur between OLTP transactions and OLAP queries, as in our pharmacy application and most of ASP potential applications. In mono-master replication, one main dimension for data quality is *freshness* which is defined through *freshness level*. The data at a slave node is totally fresh if it has the same value as that at the master node, *i.e.* all the corresponding refresh transactions have been propagated to the slave nodes. Otherwise, the freshness level reflects the distance between the data value at the slave node and that at the master node.

In this paper, we address the problem of expressing and exploiting freshness requirements in order to optimize the execution of queries. An obvious observation is that queries do not always require perfectly fresh data and may tolerate to read some stale data. For instance, assume a query Q computing the average quantity sold per product and per day over the last six months, on a table *SALE* containing the sales history. As it covers a large time interval, computing Q may be acceptable even if it misses the last tuples inserted in table *SALE*. In this case, application semantics is modelled by freshness requirements which express how many missing tuples in *SALE* are tolerated in order to compute Q. Another observation is that a slave node does not always need to be refreshed in order to comply with the freshness requirements of a query, even if the query requires perfect freshness. For instance, if all the transactions executed on the master node and waiting for refresh on a slave node S_i do not access table *SALE*, *e.g.* they access table *PRODUCT*, S_i is still perfectly fresh for Q, but may be not fresh enough for a query accessing table *PRODUCT*. In this case, detecting potential conflicts between transactions and queries helps reducing the refresh sequence to apply to a node to get it fresh enough for a query. Hence, application semantics is also modelled by the potential conflicts between queries and transactions. In this context, we want to increase efficiency by allowing queries to be sent to slave nodes even if they are not up-to-date, according to application requirements on data freshness. The problem can be stated as follows: given an autonomous database replicated in mono-master mode, evaluate the level of copy freshness of slave nodes to route a query to and select a node such that (1) the copy freshness level guarantees that the query result will satisfy the query freshness requirements and (2) the choice of the node optimizes query response time.

There are several projects close to our approach [1,2,7,12,5,8,9,6]. However, they all have one or more of the following limitations: are specific to some kind of data (*e.g.* XML documents), do not allow to model several kinds of freshness level, do not take updates into account, require substantial modification of the DBMS transaction manager, or do not model conflicts between OLAP and OLTP loads at a granularity finer than the entire database.

In this paper, we make three main contributions. First, we define a freshness model for users to specify freshness requirements for queries. This model allows capturing conflicts between queries and transactions. Second, we propose an algorithm to optimize the routing of queries on slave nodes based on the freshness requirements and the conflicts. Third, we provide an experimental validation using *Refresco* (*Routing Enhancer through FREShness COntrol*), a middleware prototype which implements our approach.

The paper is organized as follows. Section 2 gives an overview of our database cluster architecture. Section 3 defines the freshness model. Section 4 gives the algorithms to optimize query routing. Section 5 presents our experimental validation. Section 6 compares our approach with related work. Section 7 concludes.

2 Database Cluster Architecture

Figure 1 gives an overview of our database cluster architecture, derived from [4]. As shown, our middleware preserves the autonomy constraint because it interfers neither with client's applications nor with existing databases and DBMS: it receives requests from the application and sends them to nodes. Results are returned from nodes to the load balancer which forwards them to clients. The database is fully replicated on nodes $S_1, S_2, \ldots, S_N$. S_0 is the *master node* which is used to perform transactions and queries. The other nodes are *slave nodes* used for queries. They are updated only through *refresh transactions*. Refresh transactions are sent sequentially, according to the serialization (commit) order obtained on the master node, in order to guarantee the same serialization order on slave nodes. Metadata useful for the load balancer is provided and managed by the DBA using the metadata repository. It includes for instance the default level of freshness required by a query. It also includes information about which

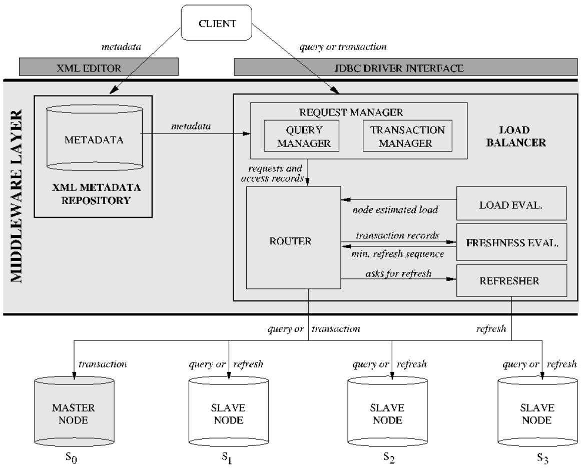


Fig. 1. Mono-master replicated database architecture

part of the database is updated by the transactions and read by the queries, enabling the detection of potential conflicts between updates and queries.

The load balancer which receives clients' requests performs two main functions: request management and routing. The request manager prepares specific *access records* for transactions and queries: the transaction manager and the query manager prepare respectively *transaction records* and *query records*. Access records are built using metadata and dynamic information provided by the clients (*e.g.* parameters for SQL programs) or resulting from the execution of transactions on the master node or obtained by parsing application code when available.

The router uses access records to send requests to nodes. Whenever a request is sent to a slave node, its estimated duration is maintained by the load evaluation module. Transactions are sent to the master node. Transaction records are enriched with dynamic information about the transaction execution on the master node (commit time of the transaction, number of tuples changed, ...). They are stored by the freshness evaluation module until every node has executed the corresponding refresh transaction and then removed.

When the router receives a query Q, it asks the freshness evaluation module to compute the corresponding minimum refresh sequence for every node. It also asks the load evaluation module to compute the current node's load. Then, it computes a cost function for Q for every slave node, including the cost of the possible execution of refresh transactions in order to make the slave node fresh enough for Q. Then the router sends the query and possible refresh transactions to the slave node which minimizes the cost function, thus minimizing the query response time. Since queries are only sent to slave nodes, they do not interfer with the transaction stream on the master node. In our application example, this yields an important advantage since transactions represent front-office applications with high priority.

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. Based on a definition of transaction precedence, we define refresh sequences. Then we give three freshness measures which allow users to specify the freshness of data that matches the semantics of the application. Second, we define conflict classes to model potential conflicts between transactions and queries. Most of the concepts used in this section are shown below.

$$
\begin{aligned}
\text{Freshness Level} &::= \text{Freshness Atom} \\
&\quad \mid \text{Freshness Level} \lor \text{Freshness Level} \\
&\quad \mid \text{Freshness Level} \land \text{Freshness Level} \\
\text{Freshness Atom} &::= (\text{Access Atom, Freshness Measure, Threshold}) \\
\text{Freshness Measure} &::= \text{Age} \mid \text{Order} \mid \text{Card} \\
\text{Access Atom} &::= \text{Database} \mid \text{Relation} \mid \text{Attribute} \\
\text{Conflict Class} &::= \{\text{Access Atom}\}
\end{aligned}
$$

3.1 Freshness Requirements

Freshness requirements of queries are specified through a flexible model which allows the user (database programmer or DBA) to define the staleness allowed for each part of the database read by the query depending on the desired granularity and freshness measure. First, the user determines the *access atoms* of the query, *i.e.* the parts of the database accessed by the query. Depending on the granularity desired, an access atom can be the entire database, a relation or even a relation attribute. For each access atom a, the user gives a condition on a which bounds the staleness of a under a certain threshold t for a given *freshness measure* μ, *i.e.* such as $\mu(a) < t$. The *freshness level* of a query Q is then defined as a logical formula composed of every freshness atom and denoted by $\mathit{Fresh}(Q, S_i)$: the results of Q at a slave node S_i are fresh enough if $\mathit{Fresh}(Q, S_i)$ is satisfied, *i.e.* if it returns *true.*

3.2 Precedence Order and Refresh Sequences

We now define a precedence order among requests, in order to define the freshness on slave nodes. This order reflects the global serialization order among transactions over the cluster, *i.e.* the serialization order obtained on the master node and reproduced on the slave nodes. It is used to define refresh sequences for a node, which contain the refresh transactions necessary to make the copy of the node fresh enough. First, we define state sequences for requests (transactions or queries) : *accepted, running, done* and *notified.* A request is *accepted* when the connection between the client and the system is successful. The request is given a global identifier i and is denoted by r_i. Request r_i is *running* if its beginning is recorded in the DBMS log, at the master node if r_i is a transaction, at a slave node if it is a query. If r_i is a transaction, it is *done* when its commit or abort is recorded in the DBMS log. If r_i is a query, it is done when it has committed at a slave node and returned a result with a satisfying level of freshness. Finally, a request is *notified* when its results are returned to the client.

As said in Section 1, we must ensure that queries always read a consistent (possibly stale) state of the database. In a mono-master configuration, the local concurrency control at the master node always produces consistent states. Thus, ensuring global consistency is equivalent to ensuring that refresh transactions are executed on a slave node in the serialization order of the master node, which we obtain by sending refresh transactions sequentially, according to this order. In practice, retrieving the serialization order on the master node depends on the isolation protocol used by the local DBMS. If it provides commit-order serializability, this is straightforward by reading commit log records[2]. We base our precedence order and thus our freshness computation on the commit log record of a transaction for two main reasons. First, since this information is available in most existing DBMS, this makes our approach generic. Second, reading a log is a non-intrusive method, which is important to preserve autonomy.

The *precedence order among transactions* is defined as follows: let T and T' be two transactions, we say that T *precedes* T', denoted by $T \prec T'$, if T and T' have committed on the master node, and T is done before T' is done. Note that, as it is based on commit time, $\prec$ is a *total order* among transactions.

The *precedence order between transactions and queries* is defined as follows: let T be a transaction and Q be a query, we say that T *precedes* Q, denoted by $T \prec Q$, if T is done before Q starts running. Note that there is no need of an order among queries.

Let *seq* be a transaction sequence. *Head(seq)* denotes *seq* without its last element, *Apply(seq,*S_i*)* denotes the state of node S_i after applying the transactions of *seq* on S_i. We define $\mathit{MinRefresh}(S_i, Q)$ the minimal refresh sequence to apply on S_i according to the $\prec$ order defined above, in order to make it fresh enough for query Q as the sequence of transactions t such as:

$$\forall t \in \mathit{MinRefresh}(S_i, Q),\ t \text{ has committed on } S_0, \text{ and}$$
$$\mathit{Fresh}(Q,\ \mathit{Apply}(\mathit{MinRefresh}(S_i, Q), S_i)), \text{ and}$$
$$\neg\mathit{Fresh}(Q,\ \mathit{Apply}(\mathit{Head}(\mathit{MinRefresh}(S_i, Q)), S_i))$$

We also define $\mathit{PerfectRefresh}(S_i)$ the refresh sequence to apply on S_i according to the $\prec$ order defined above, in order to make it perfectly fresh for any query. It is the sequence of transactions t such as:
$\forall t \in \mathit{PerfectRefresh}(S_i), t$ has committed on $S_0 \wedge \neg(t$ is done on $S_i)$

[2] We use the Oracle's SERIALIZABLE ISOLATION_LEVEL.

3.3 Freshness Measures

Classifications of freshness measures can be found in [1,2,3,11,12]. We adapt these measures to our context because we cannot use internal information of the DBMS transaction manager to evaluate them.

Let a be an access atom. We consider different measures μ and define them, *at a given instant* t, for a being either an attribute, a relation or the entire database. First, we define $U(a_i)$ as the set of transactions updating an access atom a_i, $U(a_i) = \{T \in \mathit{PerfectRefresh}(S_i) \wedge T\ \mathit{updates}\ a_i\}$, where $T\ \mathit{updates}\ a_i$ is defined as follows:

- if a_i is an attribute $R.att$, $T\ \mathit{updates}\ a_i$ if it inserts or deletes at least one tuple in R_i, or modifies *att* in at least one tuple of R_i,
- if a_i is a relation R_i, $T\ \mathit{updates}\ a_i$ if it inserts, deletes or modifies at least one tuple in R_i,
- if a_i is the database, $T\ \mathit{updates}\ a_i$ if it inserts, deletes or modifies at least one tuple.

We define three freshness measures *Order*, *Age* and *Card* as follows:

***Order*(a_i)**: the ordering measure of a_i is the number of transactions updating a which have committed on the master node and have not yet been propagated on slave node S_i at instant t, *i.e.*

$$\mathit{Order}(a_i) = |U| \text{ (the cardinal of } U\text{)}$$

***Age*(a_i)**: the age of a_i is the maximum time since at least one transaction updating a has committed on the master node and has not yet been propagated on slave node S_i at instant t, *i.e.*

$$\mathit{Age}(a_i) = \mathit{Max}(t - T.\mathit{commit\_time}), T \in U$$

***Card*(a_i)**: this measure reflects the number of stale elements in a_i. If a_i is an attribute $R.att$, *Card*(a_i) is the number of tuples in R_i inserted, deleted or having *att* being modified by all the transactions in U. If a_i is a relation R, *Card*(a_i) is the number of tuples in R_i inserted, deleted or updated by all the transactions in U. If a_i is the database, *Card*(a_i) is the number of tuples inserted, deleted or updated by all the transactions in U.

These different measures correspond to different user requirements. Measure *Order* is useful, for instance, for queries involving history relations, since it can estimate the number of missing inserted tuples. Measure *Age* allows modelling queries such as "Give the value of X as it was no later than Y minutes ago". It is also useful for queries accessing history relations. For instance, if a query wants data as of last week, the results will be correct if computed on a node stale since one hour. Measure *Card* is more relevant for estimating the accuracy of a query result, since it is able to count the number of individual updates missing to get a copy perfectly fresh. These measures can also be combined to define complex measures. Note that, by definition, freshness is computed just before a query is sent to the best node: transactions sent to the master node after this moment are not taken into account.

3.4 Conflict Classes

Conflict classes are used to detect potential conflicts between transactions and queries, before query execution. They are stored in the metadata repository. They may be given by the user or inferred by parsing the transactions' source code (when available). They can also be deduced from the access atoms used to model transactions and queries.

Let r be a request. The *conflict class* of r, denoted by $CC(r)$, is defined as a set of access atoms potentially accessed by r. The conflict class of a request r is a superset of the data set which the request will actually access. As transactions are serialized at the master node, we are not interested in the data read transactions. Thus, $CC(r)$ is the data which r will potentially write (resp. read) if r is a transaction (resp. a query). Conflict classes may be defined in different ways, depending on the granularity needed by applications. Consider transaction T_1, and queries Q_1 and Q_2 defined as follow:

T_1*: update PRODUCT set price=price*1.1 where id=1234;*
Q_1*: select id, avg(quantity) from SALE where date between to_date('07/01/2003')*
and to_date('12/31/2003') group by id;
Q_2*: select id from PRODUCT where type='Lotion';*

The table below shows the conflict classes for T_1, Q_1 and Q_2 according to the selected granularity level. When specified at the database level, T_1 and Q_1 potentially conflict. But they do not potentially conflict when specified at the relation level because Q_1 reads data from table *SALE* when T_1 updates data in table *PRODUCT*. Q_2 and T_1 potentially conflict at the relation level when they do not conflit at the attribute level. This example shows that the choice of the granularity level impacts potential conflicts: the finer the granularity, the less potential conflicts exist.

granularity	CC(Q_1)	CC(T_1)	CC(Q_2)
database	{database}	{database}	{database}
relation	{SALE}	{PRODUCT}	{PRODUCT}
attribute	{SALE.id, SALE.quantity, SALE.date}	{PRODUCT.price}	{PRODUCT.id, PRODUCT.type}

Conflict classes allow defining *potential conflicts* between requests. Since transactions are serialized on the master node, there is no need for our middleware to handle write/write conflicts. Thus, since a query cannot conflict with another query, we only need to define potential conflicts between a transaction and a query. A query Q and a transaction T potentially conflict if a least one access atom of $CC(Q)$ conflicts with one of $CC(T)$ conflicts, according to the following rules:

◇ the database potentially conflicts with any other access atom
◇ a relation R_i potentially conflicts with a relation R_j iff $R_i = R_j$

- ◇ a relation R_i potentially conflicts with an attribute $R_j.col_k$ iff $R_i = R_j$
- ◇ an attribute $R_i.att_k$ potentially conflicts with an attribute $R_j.att_l$ iff $R_i = R_j \wedge att_k = att_l$

In other words, an access atom potentially conflicts with another one if they are the same or if one is included in the other. Potential conflicts include real conflicts, *i.e.* conflicts at execution time. This means that whenever a transaction and a query actually conflict, a potential conflict has been detected *a priori*. The reverse is not true. Consider query Q_3: *select * from PRODUCT where id=4567;*. Even at the finest granularity of our model (attribute), a potential conflict is detected on *PRODUCT.id* between Q_3 and transaction T_1 defined above. However, at execution time, T_1 and Q_3 do not access the same tuple and will not actually conflict. This problem could be solved in some cases by defining conflict classes at finer granularity levels (*e.g.* tuple), but this would make freshness evaluation much more complex and costly in terms of metadata management.

4 Trading Freshness for Load Balancing

In this section, we show how freshness can be evaluated and give an algorithm that use the freshness model to optimize query load balancing.

4.1 Evaluating Freshness

Computing the measures defined above is relatively straightforward. *Order* is evaluated by counting the number of transactions necessary to get an access atom copy perfectly fresh. *Age* is evaluated using the commit time of transactions. *Card* evaluation uses the number of tuples modified by a transaction returned by the database driver after the transaction commit on the master node. However, freshness atoms can not be evaluated with a perfect precision. The main reason is that we must evaluate them before the query is sent to a given slave node. At that time, we do not know which tuples will be accessed by the query. As discussed in Section 3.4, it is thus impossible to determine which transactions not already propagated on the node will really conflict with the query. Our solution to this problem is to compute an upper bound for freshness atoms, called *confidence level*. The confidence level of a freshness atom (a, μ, t), denoted by $conf(a, \mu)$, is a value which guarantees that $\mu(a) \leq conf(a, \mu)$. Therefore the following holds: $(conf(a, \mu) \leq t) \Rightarrow (\mu(a) \leq t)$.

Confidence levels are computed using potential conflicts between queries and transactions, as defined in Section 3.4, based on the conflict classes stored in the metadata repository. As potential conflicts include real conflicts, this guarantees that freshness atom evaluation is over-estimated. Note however that transactions which do not potentially conflict with the access atoms included in the freshness atom are not considered in the computation.

4.2 Computing the Minimum Refresh Sequence for a Query *MinRefresh*(S_i, Q)

A query is sent to a slave node only if the node satisfies the freshness level of the query. Therefore, when choosing an execution node, the router needs to know for every slave node which refresh transactions must be sent to the node if it is not fresh enough. To this end, it asks the freshness evaluation module to compute the corresponding minimum refresh sequence for every node. Figure 2 shows the queue managed by the freshness evaluation module where incoming transactions are stored until every slave node has executed it. They are placed in the queue in the global serialization order, i.e. the serialization order on the master node. The refresh level of a slave node S_i is represented by a "stack pointer" *level*$_i$: all transactions preceding the transaction pointed by *level*$_i$ have already been executed at S_i. Node S_i is perfectly fresh when *level*$_i$ meets the master node pointer, *master_level*.

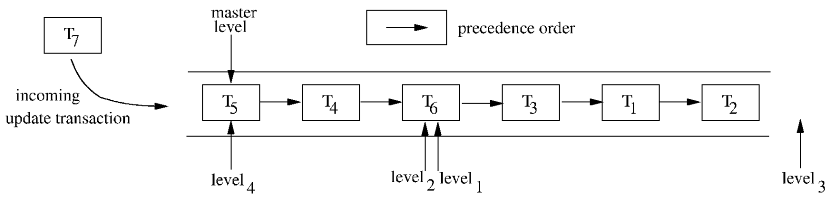


Fig. 2. Transactions global ordering.

In this example, the set of running transactions is $T_1, T_2, ..., T_6$ while an incoming transaction T_7 is about to be inserted. The global execution order is $(T_2, T_1, T_3, T_6, T_4, T_5)$. There are four slave nodes: S_1 and S_2 have processed transactions (T_2, T_1, T_3, T_6), S_3 has not been updated since the beginning may be due to a network failure and S_4 is the only slave node perfectly fresh.

This data structure minimizes memory utilization. First, since there is only one queue for all the slave nodes, adding a node to the cluster only implies adding a new pointer. Second, as soon as an transaction has been propagated to all slave nodes, it is deleted from the queue. Based on this queue, function *getMinRefresh* (see Figure 3.a) computes the minimum refresh sequence of a slave node S_i for the freshness level f of a given query, which is available in the query record. It returns a pointer to a level between the node current level and the master node level. This means that the sequence of transactions between the node current level and the level computed must be applied to the slave node in order to make it fresh enough for the query. If the freshness level is a freshness atom, the algorithm tries to decrease the refresh level needed, from the master level to the lowest possible level. The best case is to reach the current node level: no refresh is necessary for this query on this node. For each level reached, the confidence level of this freshness atom is updated when some potential conflict is detected with the corresponding transaction. The process ends when the threshold for the freshness measure is exceeded.

```
getMinRefresh(f, i){
  // f : freshness level of a query
  // i : slave node identifier
  if (f.type == ''AND'')
    return max(getMinRefresh(f.left_op, i),
           getMinRefresh(f.right_op, i));
  if (f.type == ''OR'')
    return min(getMinRefresh(f.left_op,i),
           getMinRefresh(f.right_op, i));
  // the freshness level is a freshness atom
  m = f.freshness_measure;
  t = f.threshold;
  a = f.access_atom;
  node_level = getLevel(i);
  master_level = getLevel(master_node);
  tmp_level = master_level;
  current_change=0;
  // find the first transaction over the threshold
  while (tmp_level != node_level) {
    if (conflicts(f,tmp_level)) {
    switch (m) {
      ''AGE'' : change = getCurrentTime()−level.commit_time;
      ''ORDER'' : change = change + 1;
       ''CARD'': change += level.nb_tuples_modified;  }
    if (current_change > a.threshold)  return tmp_level;     }
    tmp_level = tmp_level.next;
  }
  return node_level;
}
           Freshness Evaluation Module
```

```
route (query){
  // compute the best node for this query
  min_cost=+infinity;
  for (node_id in slave nodes) {
    cost = getNodeLoad(node_id);
    refresh_level = minRefresh(query.fresh_level, node_id);
    refresh_cost = refreshCost(refresh_level, node_id);
    cost += refresh_cost;
    if (cost<min_cost) {
       min_cost = cost;
       chosen_node = node_id;
    }
  }
  // refresh the choosen node until the
  // minimum refresh level computed
  asksRefresh(getLevel(chosen_node), refresh_level, query);
  return chosen_node;
}
                    Router
```

Fig. 3. Computing the minimum refresh sequence and routing algorithm for a query

4.3 Routing Algorithm

Figure 3.b shows the routing algorithm which evaluates query refreshment and execution cost on every slave node in order to choose the best node. First, based on previous executions of the query, function *getAvgTime(query)* evaluates the query execution time on the node. Then the current load of the node is added. It is estimated by the load evaluation module which sums the remaining execution time of all the running transactions on the node. Finally, the total cost is increased with the refresh cost, given by expression *refreshLoad(getMinRefresh(Q.freshness_level,node),node)* which evaluates the execution time of the minimum refresh sequence for the query Q on this node. The best node is the one minimizing the total cost. If more than one node have the same cost, we make a random choice. Before the function *route()* returns, it calls function *asksRefresh()* which asynchronously sends a refresh demand to the refresher. The query execution starts on the node when the refreshment is done.

In our approach, routing is a multi-criteria decision. It takes into account simultaneously the query freshness criterion, but also the current nodes load criterion and the refreshment cost criterion. Hence, the router may decide that refreshing a stale node is better than choosing a node fresh enough, for example when the refresh sequence is small and the node sligthly loaded. All these criteria are considered at the same time, which is more efficient than optimizing one criterion after each other. Our refresh strategy is embedded in the routing process. It is different from [9], where routing is independent from the refresh strategy. The strategy in [9] proceeds as follows. It first selects the nodes which are fresh enough and then elects the least loaded node. If there is no node fresh enough for a query, the query waits until refresh is activated upon time-out. Thus, it does not take into account, as we do, cases when the refreshment cost is lower than sending the query to a node fresh enough but very loaded.

5 Experimental Validation

In order to validate our approach, we developped a prototype, called *Refresco*, which implements our architecture and routing algorithm. We evaluate the influence of freshness on global performance, with different freshness measures. Then we focus on the impact of freshness threshold. Finally, we study the impact of different cluster sizes to show significant benefits even with small numbers of nodes.

5.1 Prototype Environment

The prototype is implemented in Java (jdk 1.4). The database is fully replicated on four nodes, each running the Oracle 8i server under Linux. The middleware layer runs on a fifth node. All nodes (Pentium IV 2Ghz, 512 Mb RAM) are interconnected by a switched 1 GBit/s Fast-Ethernet local area network. We generated the database according to the TPC-R benchmark [10] with a scaling factor of 1. The workload contains OLTP transactions and OLAP queries sending one SQL request (transaction or query) every 5 ms. The transactions correspond to the TPC-R refresh function RF1 while the queries are randomized TPC-R queries. The workload is composed of six transaction streams and six query streams. The average response time, obtained by executing transactions on a load-free single Oracle Server node is about 4ms for OLTP transactions while it is more than two minutes for OLAP queries. Each experiment has a duration of 20 minutes.

5.2 Experiment Parameters and Performance Measures

Experiment parameters are described in the table below. Within the same experiment, every query has the same freshness policy: the freshness level is a logical AND formula and access atoms are defined by the same freshness measure, the same freshness threshold and the same granularity.

Threshold	0, ..., $+\infty$
Granularity	*database / relation*
Freshness measure	*age / order / card*
Number of nodes	*1, 2, 3, 4*

For every experiment, we made the following measurements:

– *Query throughput*: the number of queries executed per hour.
– *QMRT*: the mean response time per query in seconds.
– *Transaction throughput*: the number of transactions executed on the master node per minute.
– *TMRT*: the mean response time per transaction on the master node in milliseconds.

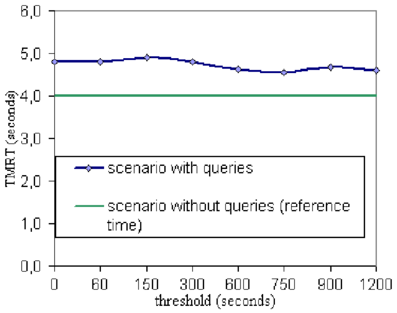
The total response time of a query is detailed as the time to choose the best node (routing time), the time to refresh the node (refresh time) and the time to execute the query on the selected node (DB time).

5.3 Impact of Freshness Threshold

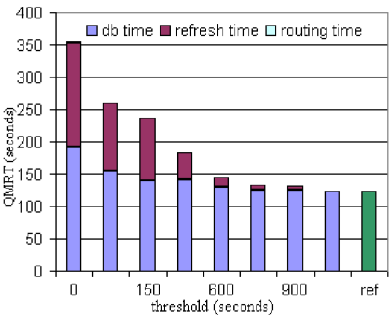
In these experiments, we focus on how the freshness policy influences transaction and query performance. We use measures *Age*, *Order* and *Card*. We ran these experiments on 4 nodes using the database granularity. We vary the freshness threshold from 0 to 1200s for measure *Age*, from 0 to 160000 transactions for measure *Order* and from 0 to 240000 tuples for measure *Card*. Maximum limits for the threshold are defined according to the experiment duration (20 minutes). Over this limit, freshness thresholds become so high that even the most obsolete slave node would be fresh enough to satisfy the query. Any higher threshold will give the same results.

Figure 4(a) shows that varying freshness does not impact transaction throughput on the master node, as one could expect. More interesting, it also shows that transaction mean response time is almost the same than the reference time, when no queries are sent to slave nodes. This mean that transactions are not slowed down by queries. This result is a direct consequence of choosing a mono master configuration where the master node does not perform queries. Though obvious, this result is important if we remember that in our context, we must guarantee that transactions, generated by front-office applications, are interactive.

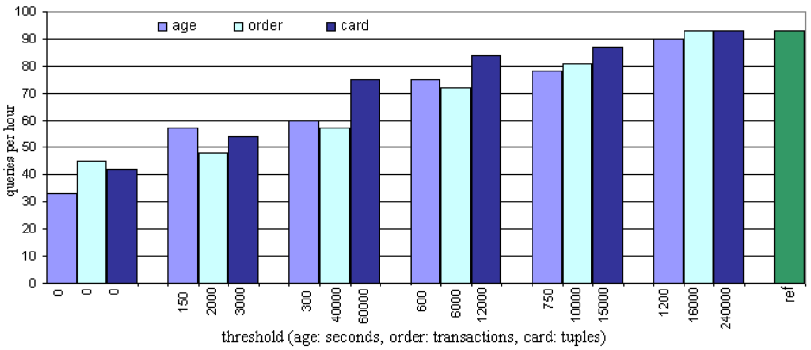
Figure 4(b) shows that relaxing data freshness improves query throughput significantly. For instance, with a freshness threshold of 300s for measure *Age* (*i.e.* data may be out-of-date since at most 5 minutes), twice as many queries are performed within the same time as when the freshness threshold is 0 (*i.e.* data must be perfectly fresh). The query throughput is 70% percent as good as the reference throughput, obtained when no transaction is applied on the master node (last column on the right). This is important in pharmacy applications where statistics on stocks may be computed on-line but are usually acceptable even if computed with data stale since hours or even days.



(a) Influence of freshness (measure *Age*) on transaction mean response time



(c) Influence of freshness (measure *Age*) on query response time



(b) Influence of freshness on query throughput

Fig. 4.

Figure 4(c) shows how relaxing data freshness decreases query response time. For instance, with a freshness threshold of 300s (measure *Age*), the user obtains the query results 50% faster compared to a freshness threshold of 0. Results for other measures are very similar and we omit them to keep the figure readable. The decomposition of response time in routing time, refresh time and database time helps explaining these results. First, the database time decreases with respect to the threshold. This is purely due to experimental conditions: queries wait less time for refresh, thus more queries are sent to each node during the same time and the local DBMS remains less idle. With a more intensive workload and the same number of nodes, the local DBMS would be overloaded and the database time would no longer decrease. Second, we see that the routing time used by the router to choose the best node is negligible[3]. In fact, it decreases as the threshold increases since the router reaches faster the required freshness level.

[3] It is even too small to be seen on the figure

Third, the time a query waits for refresh also decreases with respect to the threshold and can be considered negligible with a threshold greater than 600s, *i.e.* half of the experiment total time. This is explained by the fact that with a larger threshold, nodes need less refresh to fulfill the freshness requirements of queries. Of course, this means that each node becomes less and less fresh and there will be a higher price to pay to refresh it sooner or later during the lifecycle of the node's database. However, in typical ASP applications, the OLTP activity is more regular than the OLAP activity which can increase at specific times. Thus, outside of OLAP intensive periods, slave nodes are less busy and can be used for (possibly background) refreshment. This shows that, for normal use cases, the overhead induced by our middleware, i.e. routing time + refresh time, remains acceptable and can be considered negligible when users accept to read data stale since a reasonable time.

5.4 Impact of Granularity

We now investigate how the granularity of freshness atoms impacts query performance. The freshness threshold is 0, *i.e.* queries access perfectly fresh data. We built three different workloads with different conflict rates. The conflict rate of a workload is defined as the proportion of potential conflicts between transactions and queries. Thus, it is always equal to 1 at the database level. At the relation level, the three workloads have the following conflict rates: 0.15, 0.50 and 0.80. We ran the three workloads on four nodes with measure *Age*. Each workload was run first at the database level, then at the relation level.

Figure 5 shows that the query mean response time is from 16% (conflict rate of 0.8) to 70% (conflict rate of 0.15) better when the freshness requirements are specified at the relation level. At the database level, every query must wait until its execution node is perfectly fresh. At the relation level, the router knows when a query asks for data belonging to a relation which has not yet been updated on the master node. Hence, queries without conflicts do not have to wait for refresh (see section 5.3). The benefits depend on the conflict rate since the more queries conflict with transactions, the more slave nodes must be refreshed before query execution.

Figure 6 shows for conflict rate of 0.15 how queries are balanced on the slave nodes at the database and relation levels. We model the quantity of work done on a slave node as the total execution time of all the queries executed on the node. We distinguish between queries conflicting (resp. non conflicting) with transactions at the relation level. At the database level, queries are simply balanced on the slave nodes depending on the load, in a classical way. But even queries without conflict must wait until their execution node is perfectly fresh because the router cannot detect it since their freshness is specified at the database level. At the relation level, slave nodes appear to get specialized: node 2 gets non conflicting queries while other queries are balanced between node 1 and node 3. Queries without conflict are executed without waiting because they need not refresh. Since conflicting queries need refresh, they require more resources so two nodes are used. The percentage of slave nodes used by conflicting queries

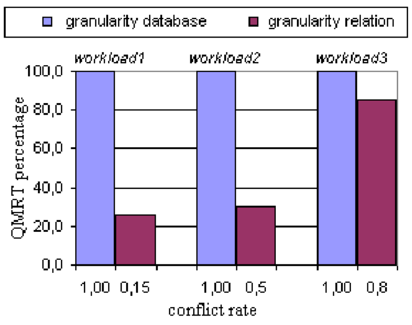


Fig. 5. Influence of granularity on query mean response time

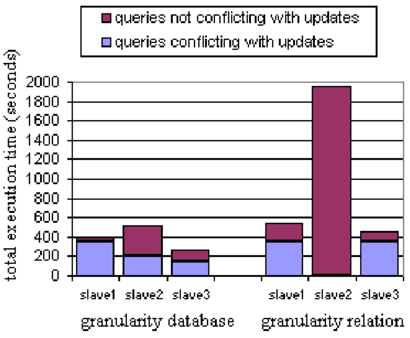


Fig. 6. Load balancing of queries on slave nodes

decreases with the conflict rate. This locality-oriented phenomenon stems from the routing algorithm behavior, because refreshment cost is one criterion. Nodes where conflicting queries have been executed are fresher than nodes with only queries without conflict. Thus, these fresh nodes are better candidates for the next conflicting queries because their refreshment cost is low. As time goes on, the freshness divergence of slave nodes increases and this phenomenon is amplified. This behavior of our router satisfyes the requirements of applications where many queries asks for data which are not updated very often. It is particularly efficient when the conflict rate is low. In the pharmacy application case, it is true for instance for queries computing incompatibilities among drugs sold to the same customers. The table which contains such information is only updated whenever a new product is put on the market, which is less than one time per day.

5.5 Impact of the Number of Cluster Nodes

We now focus on the impact of the number of cluster nodes on performance. We want to demonstrate the benefits we can expect, even with a small number of nodes in order to keep the cost of hosting an application reasonable. The same experiment (freshness measure is *Age*, threshold is 600 and granularity is the database) has been executed successively on one up to four nodes. Other measures and thresholds give similar results and are omitted due to space limitations.

Figure 7 shows that with only 2 slave nodes, the query mean response time is twice better than with only one node. The explanation is simply that the router balances the queries between the two slave nodes. Adding another node decreases significantly the mean response time. This is obtained by a large gain in database time. It appears that the refresh time increases with the number of nodes, but remains acceptable (less than 10%). This is mainly due to the fact that when many nodes are used, each one receives less queries in average.

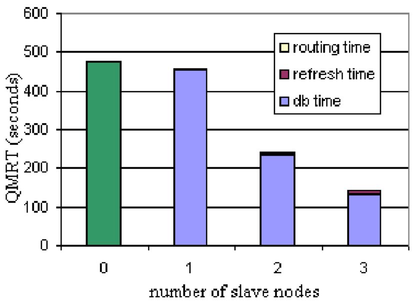


Fig. 7. Influence of number of nodes on query response time

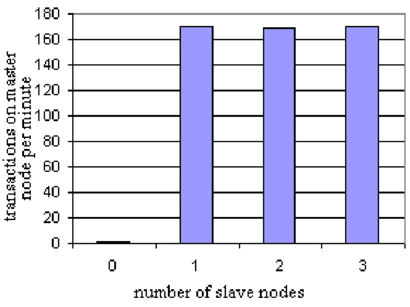


Fig. 8. Influence of number of nodes on transaction throughput

Whenever a query is sent to such a node, it takes less advantage from the refresh already performed on the node for preceding queries.

Figure 8 justifies our choice of dedicating the master node to transactions, taking into account that our workload is rather transaction intensive. The first column on the left shows that the transaction throughput would be dramatically poor if queries where all sent to the master node. With more than one node, we can route queries only on slave nodes and the number of nodes does not impact the transaction throughput, since slave nodes are refreshed in a lazy mode.

6 Related Work

There are several interesting projects related to our work. The PowerDB project at ETH Zurich deals with the coordination of cluster nodes in order to provide a consistent view to the clients. Their authors give a specific solution to XML document management in [5] and to cache evaluation for OLAP queries in [8], without taking updates into account. More recently, they addressed issues similar to the ones addressed in this paper [9]. With a similar architecture, they show how trading freshness for query performance leads to substantial gains in query response time and make a nice comparison of various refresh strategies. However, their freshness model is very simple, with only one freshness measure, equivalent to our measure *Age*. Furthermore, they do not model conflict classes to detect potential conflicts, *i.e.* they only consider one level of granularity for access atoms : the entire database. As mentionned in Section 4.3, their routing is independent of their refresh strategy and they do not take into account, as we do, cases when the refreshment cost is lower than sending the query to a node fresh enough but very loaded. Furthermore, they model freshness as the ratio between the commit time of the last transaction propagated on a slave node and the commit time of the most recent update transaction on the master node. This definition does not reflect any real-world measure. It is also difficult to interpret, except when freshness is equal to 1, since it depends on the clock origin. The Trapp Project at

Stanford [7] adresses the problem of precision/performance trade-off, but focuses on optimizing the computation of aggregate queries by reducing the cost of wide-area network communications. The TACT middleware layer [12] implements the continous consistency model. However, reads and writes are mediated individually, not at the transaction level, which is not appropriate for the management of legacy database application in an ASP cluster. Quasi-copies [1] can be seen as materialized views with limited inconsistency, but the fressness model is not as complete as ours, and it is not clear how queries coming from legacy applications may be seamlessly integrated into their system. Epsilon transactions [2] provide a nice theoretical framework for divergence control, with different consistency metrics. However, it requires to alter the concurrency control, since divergence control is done at the lock manager level. Thus, it hurts the DBMS autonomy constraint.

7 Conclusion

In this paper, we addressed the problem of query performance in a database cluster with optimistic replication. Based on the observation that many queries do not need to access perfectly fresh data, which is the case in our ASP context with pharmacy applications, we strived to exploit user requirements on data freshness to improve query performance.

Assuming mono-master replication, we proposed a freshness model for users to specify the required freshness level for queries. The model is flexible since it allows users defining composite freshness formulas, with different freshness measures and at different levels of granularity. We proposed 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. Our refresh strategy is embedded in the load balancing process: a node is selected to execute a query based on its current load as well as on the cost of refreshing it enough to comply with the query freshness requirements.

To validate our approach, we developped the *Refresco* prototype on LIP6's cluster running Oracle 8i under Linux. Through experimentation with the TPC-R benchmark, we showed that significant benefits can be obtained by relaxing freshness with a reasonable threshold, whatever the freshness measure and even with few nodes. We also showed that the overhead induced by computing nodes' freshness is negligible in the routing process. Finally, we showed the major impact of granularity levels on load balancing when defining conflict classes. It appears that, if freshness requirements are defined at a fine level of granularity (*e.g.* relation), our routing strategy is self-adaptable. It routes queries that read update-intensive data to some nodes which remain always fresh, while queries that read data with low update frequency are routed to other nodes which can remain stale longer. This yields significant gains in response time for workloads where the conflict rate is low (*e.g.* a 70% gain for a conflict rate of 0.15). Our choice of mono-master replication was motivated by its simplicity advantage (to maintain copy consistency) and by the fact that it is sufficient to many applications like in our ASP context. However, a remaining issue is that the master node is a single point of failure and a potential bottleneck for heavy transactional workloads. A solution is multi-master replication which provides high-availability and allows for transaction parallelism (using several master nodes). But multi-master replication is more involved since parallel updates may produce inconsistent copies. In [4], we introduced a preventive solution to this problem. The preventive replication method provides strong consistency without the overhead of synchronous replication, by exploiting the cluster's high speed network. Thus, to exploit the solution proposed in this paper with multi-master replication, we can use preventive replication between masters and optimistic replication between each master and its slaves.

There are several interesting directions for future work. First, we want to investigate other freshness measures, such as the euclidian distance for numerical data. We also want to study the impact on performance induced by finer levels of granularity such as tuple or relation subset. It is not clear yet if the added overhead for metadata management will be amortized by performance gains. Second, we plan to improve our refresh strategy. As mentionned in Section 5.3, our approach fits well with OLAP intensive sessions of limited duration so that refreshment may be performed during idle periods. In order to limit staleness of some nodes, we plan to include autonomous refresh capabilities in our system, for instance, active rules implemented through triggers. Finally, despite our purpose was to demonstrate that the ASP mode is viable with few nodes dedicated to each application, we want to how our approach scales up with the number of nodes by running experiments on larger clusters.

References

1. R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM TODS*, 15(3):359–384, 1990.
2. D. Barbará and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB Journal*, 3(3):325–353, 1994.
3. R. Gallersdörfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *Int. Conf. on VLDB*, 1995.
4. S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. Parallel processing with autonomous databases in a cluster system. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, 2002.
5. T. Grabs, K. Böhm, and H.-J. Schek. Scalable distributed query and update service implementations for XML document elements. In *Workshop on Research Issues in Data Engineering*, 2001.
6. A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *Int. Conf. on VLDB*, 2003.
7. C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on VLDB*, 2000.
8. U. Röhm, K. Böhm, and H.-J. Schek. Cache-aware query routing in a cluster of databases. In *Int. Conf. On Data Engineering (ICDE)*, 2001.

9. U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In *Int. Conf. on VLDB*, 2002.
10. Transaction Processing Performance Council. Tpc-r : a business reporting, decision support benchmark. http://www.tpc.org/tpcr/default.asp.
11. K.-L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Int. Conf. On Data Engineering (ICDE)*, 1992.
12. H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS*, 20(3):239–282, 2002.