Data Quality Management in a Database Cluster with Lazy Replication

Cécile Le Pape† - Stéphane Gançarski† - Patrick Valduriez‡
† *Firstname.Lastname@lip6.fr, LIP6, Paris, France*
‡*Patrick.Valduriez@inria.fr, INRIA and LINA, Nantes, France*

Abstract

We consider the use of a database cluster with lazy replication. In this context, controlling the quality of replicated data based on users' requirements is important to improve performance. However, existing approaches are limited to a particular aspect of data quality. In this paper, we propose a general model of data quality which makes the difference between "freshness" and "validity" of data. Data quality is expressed through divergence measures from the data with perfect quality. Users can thus specify the minimum level of quality for their queries. This information can be exploited to optimize query load balancing. We implemented our approach in our Refresco *prototype. 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.*

Keywords : Database Cluster, Middleware, Replication, Quality, Freshness, Validity.

1 Introduction

A database cluster [16] is a cluster of PC servers, each running an off-the-shelf "black-box" DBMS. Database clusters have recently gained much interest as they provide a cost-effective alternative to parallel database systems, such as Oracle Real Application Cluster. For instance, they can make new applications such as Application Service Providers economically viable [5]. In a database cluster, data replication is often used to increase both data availability and performance. In this context, controlling the quality of replicated data based on users' requirements is important to improve load balancing, and thus performance.

In a DBMS, the fundamental tool for maintaining data quality is transactions. However, the implementation of ACID transaction properties (e.g. two-phase locking) incurs a performance overhead. Thus, a typical solution is to distinguish between update transactions and (read-only) queries. ACID properties are needed for update transactions to avoid the introduction of inconsistencies in the database. However, queries (and users issuing thoses queries) may accept to relax consistency if they can obtain the results faster. Examples of such queries can be found in many decision-support applications, where, for instance, the inaccuracy on an individual data does not affect the overall results computed over thousands or millions of items. Thus, many approaches proposed to relax transactional properties, mainly isolation, for read-only queries. The main idea is to allow queries to read "dirty data", *i.e.* data written by transactions which have not yet been committed yet, in order to reduce response time.

With data replication, ensuring ACID properties is more complex since mutal copy consistency must be preserved, through *global isolation* [14]. Ensuring global isolation implies that a transaction modifying a data must update all its copies before any other transaction can access the data, property known as *1-copy serializability*. In other words, all the copies of a given data must be accessed in the same transaction order (or compatible order). This property can be ensured with *eager (or synchronous) replication*, where all the copies are updated within the initial transaction. However, eager replication tends to increase transaction latency because of the use of a distributed commit protocol. To reduce latency, *lazy (or asynchronous) replication* updates all the copies in separate transactions after the commitment of the initial transaction. However, lazy replication implies that, during a certain time, copies of the same data diverge : some have already the new value introduced by the initial transaction, others have not. This divergence refers to the notion of data *freshness*: the lower the divergence of a copy with respect to the other copies already updated, the fresher is the data copy. And users may accept to read stale data, *i.e.* data not perfectly fresh.

Although they correspond to different aspects, reading dirty data and reading stale data are very similar. In both cases, the value read differs from an "ideal" value, valid in the first case, perfectly fresh in the second case. In both cases, the divergence corresponds to a certain number of transactions, not yet committed in the first case, not

yet propagated in the second case. Several approaches have partially addressed these issues [8, 20, 19, 16]. However, most of them do not handle both dirty data and stale data. Furthermore, most of them typically require heavy modifications to the existing concurrency control mechanisms, which makes them inappropriate for database clusters with "black-box" DBMS. In summary, none of these solutions offers a user-friendly interface to let users specifying easily their requirements on data quality.

In this paper, we propose a general solution in the context of a database cluster with lazy replication. The main contributions are: (1) a data quality model where freshness and validity are treated in a uniform way, allowing users to specify the level of quality for their queries; (2) a database cluster middleware, independent of the underlying DBMS, that exploits quality requirements to perform query load balancing; (3) efficient algorithms for computing divergence measures used for routing queries and maintaining replicated data at a required level of quality; (4) performance results based on our *Refresco* prototype (of a database cluster middleware) that show that freshness control can help increase query throughput significantly.

The rest of the paper is organized as follows. Section 2 presents our data quality model, its use in different replication configurations, and several divergence measures useful in a database cluster. Section 3 discusses data quality management in our database cluster architecture. Section 4 describes performance evaluation using our Refresco prototype. Section 5 discusses related work. Finally, Section 6 concludes.

2 Data Quality Model

In this section, we define the notion of *data quality* in order to cope with different replication configurations. We define four basic quality measures. We also propose *quality formulas* for specifying more complex requirements.

2.1 Data Quality Definitions

The transaction model considers read-only transactions, called *queries* and other *update transactions*. As shown in Figure 1, a transaction starts in state"accepted", then "executed"[1], then "committed" or "aborted" and finally, reaches the state "notified".

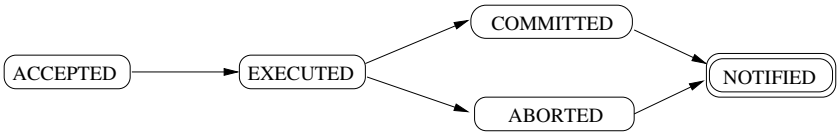

Figure 1: Transaction state chart

The states have the following meanings:

- *Accepted* : the user will be notified of the termination of the transaction and will get the results.
- *Executed* : the effects of the transaction are visible to other queries .
- *Committed* : the effects of the transaction are made durable. We assume that we know in which global order transactions are committed.
- *Aborted* : the effects of the transaction are undone.
- *Notified* : the user has been notified of the termination and has received the result (if any).

These states have different meanings depending on the replication configuration. In case of mono-master (*primary copy* [7]) replication, the master node is the only one to receive updates and then propagates them to the slave nodes. Hence, the local commitment of an update transaction on the master node implies the global commitment of the

[1]Failure handling is out of the scope of this paper

transaction. In case of multi-master replication (*primary group*), the global commitment requires communications between all the master nodes where the transaction is executed.

The data model distinguishes between a *logical data* a as seen by applications and its *physical copies* a_i as stored at cluster nodes. a may be defined at different levels of granularity:

Data ::= Database | Relation | Tuple | Attribute | Element

where Element represents an attribute value for a single tuple. The logical data defines a *reference state*, the state reached by a copy when it has received all the committed update transactions in the global order. Thus, a copy has *perfect quality* if in its reference state.

Property 1 *A copy has perfect quality if and only if*

- *it is* valid*, ie. it reflects the updates made only by update transactions committed in the global order,* and
- *it is* fresh*, ie. it reflects the updates made by all the committed update transactions.*

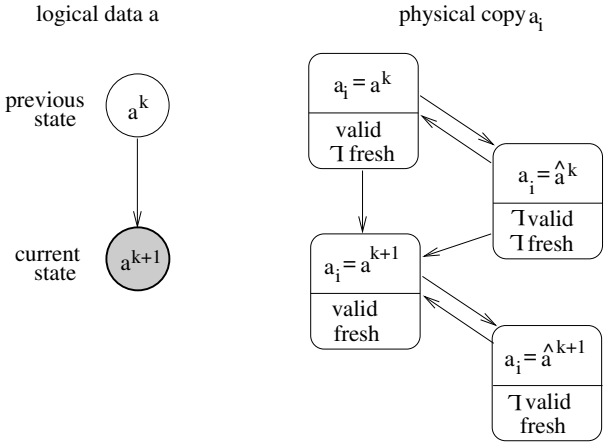

Figure 2: Logical and physical data state chart

With lazy replication, physical copies have not always perfect quality. The quality of a copy reflects the divergence between the copy state and the corresponding logical data state. Depending on the way transactions are committed and updates propagated, state transitions of a copy may differ. Possible states transitions are shown in Figure 2.

Assume a logical data a changing from state a^k to state a^{k+1} due to the global commitment of transaction T_k. a^{k+1} is now the reference state. Let a_i be a copy of a, initially in state a^k, fresh and valid. If no update is applied to a_i, a_i is no more fresh since it misses the effects of T_k but is still valid. Otherwise, a_i can have different state transitions according to the updates applied to it :

$a_i : a^k \rightarrow a^{k+1}$. T_k is performed on a_i which goes back to a perfect quality state.

$a_i : a^k \rightarrow \hat{a}^k$. One or several uncommitted transactions are performed on a_i which is thus no more valid and still not fresh since T_k has not been performed.

$a_i : a^{k+1} \rightarrow \hat{a}^{k+1}$. One or several uncommitted transactions are performed on a_i after T_k. a_i is thus fresh but not valid.

configuration	valid and ¬ fresh	¬ valid and fresh	¬ valid and ¬ fresh
single node / eager	impossible	scenario 1	impossible
lazy mono-master	scenario 2	scenario 1 on the master node	impossible
lazy multi-master	impossible	scenario 3	impossible
lazy multi-delegate	idem scenario 2	idem scenario 1	scenario 4

Figure 3: Possible data qualities according to configuration

A copy may be waiting for refresh or for commitment of update transactions already applied to it. We denote by *WaitRefresh(a_i)* the set of update transactions globally committed and waiting to be propagated to a_i, and by *WaitValid(a_i)* the set of update transactions already executed and waiting to be committed (or aborted). Finally, we denote by *Wait(a_i)* the set of update transactions waiting on a_i, *i.e.* *Wait(a_i) = WaitRefresh(a_i) ∪ WaitValid(a_i)*.

2.2 Replication Configurations

Data quality may apply to different replication configurations. We consider four configurations, which extend those of [7]. The first configuration combines non replicated databases and eager replication. The three other configurations correspond to different lazy replication modes and can be combined to form an hybrid configuration. Figure 3 shows the different possible states of a data according to the configuration.

In the remaining of this section, $R_i(a_k)$ (resp. $W_i(a_k)$) denotes that transaction T_i reads (resp. writes) data a_k. $W_i^*(a_k)$ denotes that the write operation initially applied to a copy of a is propagated to a_k. C_i (resp. A_i) denotes the local commit (resp. abort) of transaction T_i.

A replication configuration depends on the capabilities of the cluster nodes. We distinguish three types of nodes:

- a *master node* can execute both update transactions and queries. It participates to the global commitment of any update transaction.
- a *delegate node* can execute both update transactions and queries but does not participate to the global commitment of update transactions.
- a *slave node* can only execute queries. It is updated by propagating the updates made by transactions on other types of nodes.

The *single node and eager* configurations are presented together since they are equivalent in terms of data quality. In both cases, the local commitment of a transaction implies its global commitment. This implies that data is always fresh. However, dirty reads may happen if isolation is relaxed, such as in the following execution at a node k.

(scenario 1) node k : $R_1(a_k)\ W_1(a_k)\ R_2(a_k)\ C_1$

This scenario is referred to as "Read Uncommitted" in the ANSI isolation levels.

The *lazy mono-master* configuration, referred to as *primary master* in [7], is composed of one master node and several slave nodes. Update transactions are sent to the master node, slave nodes are updated through *refresh transactions*. On the master node, scenario 1 may occur. On the slave nodes, data is alway valid since updates are propagated through refresh transactions only when the corresponding update transaction is committed. However, it is not always fresh, as suggested by the following scenario:

(scenario 2) master node N_0 : $R_1(a_0)\ W_1(a_0)\ C_1$
slave node N_i : $R_2(a_i)\ W_1^*(a_i)$

When the read operation $R_2(a_i)$ is executed at node N_i, a_i is valid but not fresh since it misses the effects of the committed write $W_1(a_0)$ at the master node N_0. This scenario occurs in many applications where slave nodes are used as caches (or concrete views) of the master node.

The *lazy multi-master* configuration, or update anywhere, is composed of only master nodes, and is referred to as *lazy group* configuration in [7]. Update transactions can be sent to any node. A transaction can be committed only if all the nodes are ready to commit it. Thus, data is always fresh, but dirty reads are possible, such as in the following scenario:

(scenario 3) node i N_i : $R_1(a_i)\ W_1(a_i)\ R_2(a_i)$ | C_1
node j N_j : $W_1^*(a_j)$ | C_1

When the read operation $R_2(a_i)$ of transaction T_2 is executed at node N_i, a_i is not valid since it has read values written by T_1 which is not yet committed by both N_i and N_j.

The *multi-delegate* configuration, which also allows for update anywhere approach, is composed of one master node and several delegate nodes. Update transactions can be sent to any node, but only N_0 is allowed for validating them. At node N_0, scenario 1 may occur. Scenario 2 may occur as well in case a transaction sent to N_0 is waiting for propagation to other nodes. Furthermore, reading data that is neither fresh nor valid is also possible, as illustrated below:

(scenario 4) node N_0 (master) : $W_1(a_0)\ C_1$ $W_2^*(a_0)\ A_2$
node N_i (delegate): $W_2(a_i) R_3(a_i)$

When the read operation $R_3(a_i)$ of transaction T_3 is executed at node N_i, a_i is not valid since T_2 has not yet been committed. It is no more fresh since the write W_1 has been committed but has not yet been propagated to N_i.

2.3 Data Quality Measures

Several divergence measures have been proposed in the litterature [1, 3, 4, 18, 20]. We adapt them to fit with our definition of data quality and in our database cluster middleware. In our definitions, divergence (and thus data quality) can be expressed at different levels of granularity. We define four quality measures, *NbOp, NbElt, Age* and *Num* to compute the data quality of the physical copy a_i, *i.e.* the divergence between a and a_i.

•*NbOp* measures the number of update transactions waiting on data a_i, *i.e.* the cardinal of set *Wait(a_i)* defined in section 2.1.

$$NbOp(a_i) = \mid Wait(a_i) \mid$$

•*NbElt* measures the number of values not valid or not fresh contained in the data. If the data is a single element elt_i, then

$$NbElt(elt_i) = \begin{cases} 0 & \text{if } Wait(elt_i) = \varnothing \\ 1 & \text{else} \end{cases}$$

If the data is composed of several elements, *i.e.* if the data is a tuple, an attribute, a relation or the whole database, then

$$NbElt(a_i) = \sum_{elt_i \in a_i} NbElt(elt_i)$$

•*Age* measures the age of the oldest transaction waiting on the data.

$$Age(a_i) = Now() - Min\left(\{UpdateDate(t,a_i)\ ,\ t \in WaitValid(a_i)\} \cup \{CommitDate(t)\ ,\ t \in WaitRefresh(a_i)\}\right)$$

where *UpdateDate(t,a_i)* is the date when the data has been updated at node N_i and *CommitDate(t)* is the date of the global commitment of transaction t. The *Age* measure makes the assumption that nodes clocks are synchronized, which is reasonable in a cluster where nodes are interconnected through a fast network.

•*Num* is the euclidian distance between the respective values of a_i and a. It only makes sense for numerical attributes, and applies only for elements or for the set of elements of a same relation attribute (or column).

If the data is a single element elt_i, then

$$Num(elt_i) = \mid elt - elt_i \mid$$

where elt is the logical element corresponding to the physical copy elt_i.
If the data is the set of elements of a column, then

$$Num(a_i) = Max\{Num(elt_i), elt_i \in a_i\}$$

2.4 Data Quality Specification

Specifying data quality consists of bounding the allowed divergence. To this end, we first define the notion of *quality atom*:

Quality_atom ::= (Data, Quality_measure, Threshold, Mode)

The semantics of a quality atom is the following. The user chooses a *data* a, at the desired level of granularity, then chooses a *quality measure* m and a *thereshold* s, in order to bound the divergence for the data, *i.e.* $m(a) \leq s$. Finally, the *mode* allows specifying the nature of the measured divergence, freshness or validity.

Mode ::= Freshness | Validity

Then we define a *quality formula* as a logical combination of quality atoms:

Quality_formula ::= Quality_atom
| Quality_formula $\vee$ Quality_formula
| Quality_formula $\wedge$ Quality_formula

Combining several quality atoms in a quality formula is useful for specifying quality for different data at the same time, as well as for specifying complex quality definitions for one or several data. It is also useful for specifying the required data quality for the results of a query, by associating the formula with a query, as well as limiting the divergence of a node by associating the formula with a node.

3 Data Quality Management

In this section, we describe data quality management in a database cluster. First, we present our database cluster architecture. Then, we present in more details data quality evaluation.

3.1 Database Cluster Architecture

Figure 4 shows our database cluster architecture, which is based on [12]. Access to autonomous databases is provided by a middleware layer, through the JDBC interface. Applications send transactions to the middleware which selects the nodes for execution, according to the required data quality. The results are then sent back to applications. There are several advantages in a middleware solution : independence from the underlying local DBMSs; easy migration of existing databases; support of DBMS heterogenity, etc. For performance reasons, the different modules of the middleware can be replicated at several cluster nodes, provided that the metadata directory is shared by these nodes. This can be achieved by replicating the directory or using distributed shared memory [10].

We manage data quality with two different policies: *query-oriented* or *cluster-oriented*. In the query-oriented policy, data quality is associated with queries and is related with the data read by the query. In the cluster-oriented policy, data quality is associated with cluster nodes and gives a lower bound for the quality of the data stored in the node. The two policies can be combined: cluster nodes are maintained at a given quality level for all the queries but a query may specify whether it requires a higher quality. In all cases, the middleware handles update propagations in order to guarantee that user requirements are met. All necessary information is specified in XML without any modification to application code and stored in a *metadata repository*.

A major component in our middleware is the quality evaluation module used to evaluate the quality of a node based on its current state. If the quality is not sufficient, it computes a *upgrade plan*. If the required quality is related to freshness, the upgrade plan includes a sequence of update propagations to apply in order to reach the desired freshness. If the required quality is related to validity, the upgrade plan includes a mechanism ensuring that data

read is valid enough, either by forcing the serialization order at the node (if allowed by the underlying DBMS) or by forcing transactions to wait in order to avoid introducing invalid values.

The other modules work as follows. The transaction manager receives transactions from applications. If a transaction is a query, it retrieves the required data quality associated with the query. For each node able to execute the transaction (according to the node's type), it asks the quality evaluation module to compute the node's upgrade plan. In the best case, the node has already a sufficient quality and the upgrade plan is empty. Then, the router chooses the best node for executing the transaction, by evaluating for each node a cost function which takes into account the node's current load and the cost of the upgrade plan. The node manager maintains data quality on nodes. It retrieves the cluster-oriented quality policy from the metadata repository and triggers updgrade plans at the nodes to maintain their quality above the limit imposed by the policy. Communication and synchronization are managed by the execution module. It sends transactions and updgrade plans to nodes, waits for their validation and gets the results of local executions. Then it updates the information about cluster node states. Transactions are committed according to a global order.

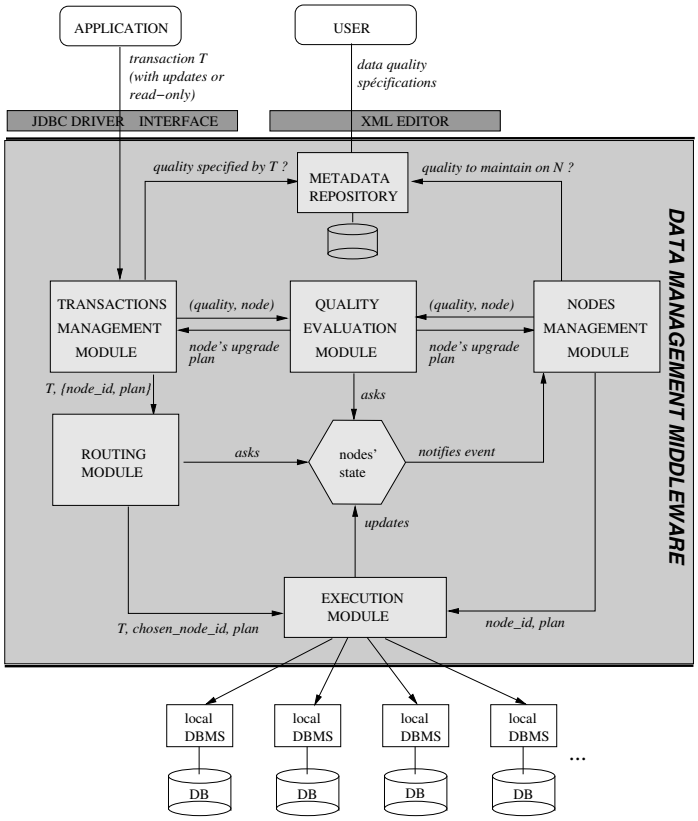

Figure 4: System architecture

3.2 Data Quality Evaluation

Data quality is evaluated based on which transactions have been sent and which have been executed at each node[2]. To know what are the effects of a transaction, we combine knowledge coming from parsing transaction code and

[2]Since local DBMS are considered black-box, we cannot use the knowledge available in the DBMS engine

dynamic knowledge obtained at execution time, return values coming from the JDBC driver and log records obtained by log sniffing [3]. The algorithm for data quality evaluation is shown in Figure 5. The quality of data a on node i for a measure m, given by the function *getQuality(a,m,i)*, is evaluated as the divergence between the value of a on i and the reference value of a. For each transaction t waiting on data copy a_i, (*i.e.* $t \in Wait(a_i)$), the algorithm updates the evaluated quality according to t's effects. For that, it computes the quantity of change made by t on a for measure m through function *evalModifs(t,a,m)*. Then it updates the corresponding divergence counter.

```
getQuality(a, m, i){
// a data
// m measure
// i node
W = Wait(a_i)
quality = 0
FORALL t in W DO
    quality = updateQuality(quality, m, evalModifs(t,a,m));
ENDFOR
return quality
```

Figure 5: Algorithm for evaluating data quality

Computing *Wait(a_i)* requires evaluating, among all the transactions waiting at node N_i, which ones are accessing data a. Obviously, if the granularity is the whole database, all the waiting transactions are concerned. If a has finer granularity, some transactions may not access a and thus are not concerned with a's quality. The simplest method for filtering transactions is parsing the transactions code. It is well adapted for relation and attribute granularity, since retrieving the corresponding information from transactions code is straightforward. For tuple granularity, it is not sufficient in the general case, and log sniffing is necessary, since it is not possible to know exactly *a priori* (from the transaction code) which tuple(s) will be updated, inserted or deleted.

Evaluating the modifications performed by a transaction t on data a depends on the considered measure m:

- *EvalModifs(t,a,NbOp)* is always equal to 1.
- *EvalModifs(t,a,NbElt) = ntt * nct*,

  where *ntt* (resp. *nct*) is the number of tuples (resp. columns) accessed by t. If t is a single statement transaction, *ntt* is the value returned by the JDBC driver after executing t. Otherwise, the information is obtained by log sniffing. *nct* is inferred during by parsing t's code.
- *EvalModifs(t,a,Age) = θ - dateSent(t,i)*,

  where θ is the date when the function is evaluated and *dateSent(t,i)* is the date when t is sent to node i. *dateSent(t,i)* is an under-estimation of the date when a is actually updated by t at node i. It guarantees that the computed quality is always an under-estimation of the actual quality, thus that the results given to applications are correct.
- *EvalModifs(t,a,Num)* is the most difficult measure to evaluate. It is obtained by combining code parsing and dynamic information (*ntt* or log sniffing). For instance:

  *EvalModifs("update R set attr=attr+10", R.attr, Num) = 10*ntt*

[3]For instance, in our implementation, we use the Oracle Logminer tool

4 Experimental Validation

In this section, we describe an experimental validation using our prototype. We present interesting results regarding the impact of data freshness and the level of granularity on query performance. Further details and results can be found in [12].

4.1 The Refresco Prototype

In order to validate our approach, we developped a prototype, called *Refresco*, (Routing Enhancer through FREShness COntrol). It has been developped in the Leg@net project[4], whose objective was to demonstrate the viability of the ASP model using a database cluster for pharmacy applications in France. Refresco implements the database cluster architecture and algorithms we proposed for for a mono-master configuration. The prototype is implemented in Java (jdk 1.4) and runs on a Linux 5 node cluster, each node running the Oracle 8i server. All nodes (Pentium IV 2Ghz, 512 Mb RAM) are interconnected by a switched 1 GBit/s Fast-Ethernet local area network. The database is fully replicated on four nodes. The middleware layer runs on a fifth node.

We generated the database according to the TPC-R benchmark [17] 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. One node of the cluster is dedicated to OLTP transactions (the master node) while queries are sent exclusively to the slave nodes. Each experiment has a duration of 20 minutes.

4.2 Impact of Freshness

In these experiments, we focus on how the freshness policy influences transaction and query performance. We use measures *Age*, *NbOp* and *NbElt* and the database granularity. We vary the freshness threshold from 0 to 1200s for measure *Age*, from 0 to 160000 transactions for measure *NbOp* and from 0 to 240000 tuples for measure *NbElt*. 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 would give the same results.

An interesting result of [12] shows that the update transaction throughput at the master node is not affected by the execution of queries. Thus, transactions are not slowed down by queries. This 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 update transactions, generated by front-office applications, are interactive.

Figure 6 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.

Other results show that the routing time used by the router to choose the best node is always negligible and decreases as the threshold increases since the router reaches faster the required freshness level. The time a query waits for refreshment 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 refreshment to fulfill the freshness requirements of queries. Of course, this means that each node becomes less and less fresh. However, outside of OLAP intensive periods, slave nodes are less busy and can be used for (possibly background) refreshment.

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

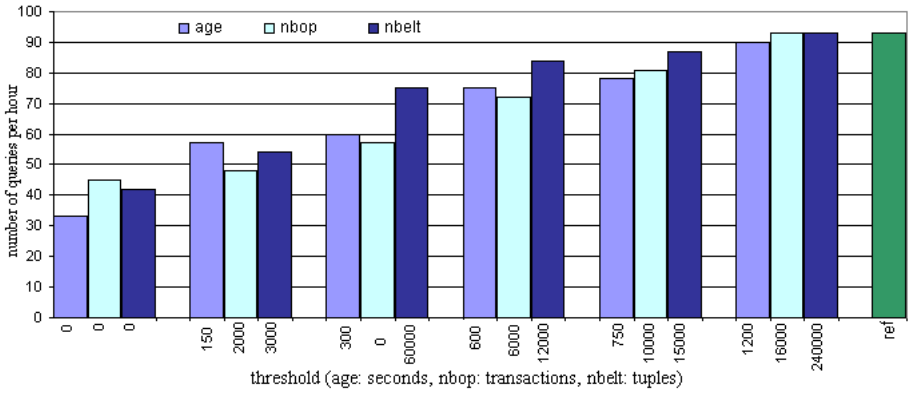

Figure 6: Influence of freshness on query throughput

4.3 Impact of Granularity

We now investigate how the granularity of freshness atoms impacts query performance. To this end, we define the *conflict rate* of a workload as the proportion of potential conflicts between transactions and queries. Thus, it only makes sense at granularity levels finer than the entire database (it is always equal to 1 at the database level). The same workload was run with measure *Age*, first at the database level, then at the relation level with a conflict rate of 0.15. The first results show that with such a low conflict rate, the mean query response time is divided by 4.

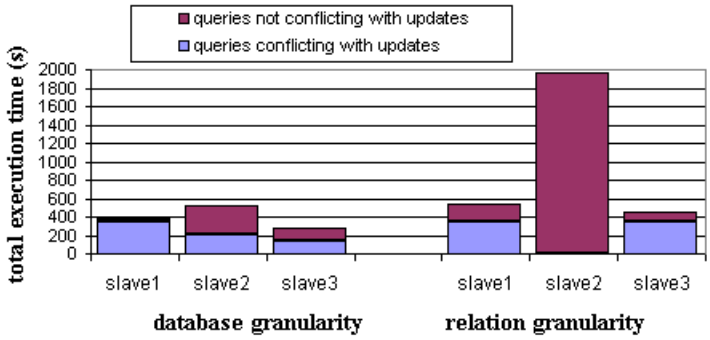

Figure 7: Load balancing of queries on slave nodes

Figure 7 shows, for a conflict rate of 0.15, how query execution is balanced at the slave nodes for database and relation granularities. At the database granularity level, queries are simply balanced at the slave nodes depending on the load, in a classical way. But even without conflict, queries must wait until their execution node is perfectly fresh 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 be refreshed. This explains why slave node 2 receives much more queries than other nodes. Since conflicting queries need refreshment, they require more resources so two nodes are used.

This locality-oriented phenomenon stems from the routing algorithm behavior, because refreshment cost is one

criterion. The nodes where conflicting queries have been executed get fresher than the nodes with only non conflicting queries. Thus, these fresh nodes are better candidates for the next conflicting queries.

5 Related work

There are several projects close to ours [1, 3, 13, 21, 6, 15, 16, 11]. However, they all have one or more of the following limitations: are specific to some kind of data (*e.g.* XML documents), model one kind of freshness, do not take updates into account, require substantial modification to the DBMS transaction manager, or do not model divergence at a granularity finer than the entire database. Among the most interesting approaches, we distinguish the ones which only handle data freshness and the ones which include control over invalid data.

Trading data freshness for performance has received much attention in the litterature. In the context of mono-master replication, the FAS prototype of ETH Zürich [16] shows substantial gains in query response time. However, their freshness model is very simple, with only one freshness measure, equivalent to our measure *Age*. Furthermore, they only consider one level of granularity for access atoms : the entire database. In the context of *data shipping*, where data is cached at the client side, relaxing freshness allows avoiding querying the remote server in case of cache miss. In [1], cached data remain available until a freshness threshold is reached. Different measures are proposed but they do not take into account data granularity. The Trapp Project at Stanford [13] adresses the problem of precision/performance trade-off by allowing users to specify precision constraints on queries. It focuses on optimizing the computation of aggregate queries over different cache sites by reducing the cost of wide-area network communication but requires a heavy modification of the local transaction managers. [9] proposes temporal guarantees on freshness of cached data, by adding the concept of consistency class : all the data of a same class belongs to the same snapshot. They add a new SQL clause to specify freshness requirements in queries, which requires a modification of the existing application code.

Querying invalid data has also received much attention. Introduced by [8], it led to the ANSI standard on isolation levels [2], some of them being implemented in the current DBMS products. However, if isolation levels allow for controling which phenomena users accept to occur, they do not provide a way to measure the divergence. More recent works have proposed divergence control including freshness and validity. Epsilon transactions [3] provide a nice theoretical framework for divergence control, with different consistency metrics. However, it requires to alter the DBMS transaction manager, since divergence control is done at the lock manager level. The TACT middleware layer [21, 22] 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 autonomous applications.

6 Conclusion

In a database cluster with lazy replication, controlling the quality of replicated data based on users' requirements is important to improve performance. In this paper, we proposed a general model of data quality where freshness and validity are treated in a uniform way. This model allows users to specify the level of quality for their queries with different levels of granularity and different divergence measures. For each measure, we provided an efficient evaluation method useful for routing queries and maintaining replicated data at a required level of quality;

We proposed a database cluster middleware that exploits quality requirements to perform query load balancing on autonomous databases. Our middleware solution has several advantages: independence from the underlying local DBMSs; easy migration of existing databases; support of DBMS heterogenity, etc.

We implemented our approach in our *Refresco* prototype. 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. Refresco has been first presented in [12]. This paper mainly differs from [12] by two major contributions: (1) it includes quality based on validity, not only on freshness, and (2), almost all the replication configurations are considered for defining the architecture and the quality measures, not only the mono-master replication.

Future work will include the definition of additional measures and the support for finer levels of granularity (e.g. tuple level). We will also port our Refresco prototype on a 64 node cluster to perfom larger-scale experiments. Finally, we will investigate support of multi-master and partial replication.

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] American National Standard for Information Systems - Database language - SQL, 1992.

[3] 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.

[4] R. Gallersdörfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *Int. Conf. on VLDB*, 1995.

[5] 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.

[6] 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.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Int. Conf. ACM SIGMOD*, 1996.

[8] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[9] H. Guo, P.-A. Larson, and J. Goldstein. Relaxed currency and consistency : How to say "good enough" in sql. In *Int. Conf. ACM SIGMOD*, 2004.

[10] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the Int. Symp. On Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[11] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *Int. Conf. on VLDB*, 2003.

[12] C. Le Pape, S. Gançarski, and P. Valduriez. Refresco : Improving query performance through freshness control in a database cluster. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, 2004.

[13] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on VLDB*, 2000.

[14] T. Özsu and P. Valduriez. Distributed and parallel database systems - technology and current state-of-the-art. *ACM Computing Surveys*, 28(1), 1996.

[15] 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.

[16] 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.

[17] Transaction Processing Performance Council. Tpc-r : a business reporting, decision support benchmark. http://www.tpc.org/tpcr/default.asp.

[18] K.-L. Wu and C. Pu. Divergence control algorithms for epsilon serializability. *Distributed and Parallel Databases*, 3(1), 1995.

[19] K.-L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Int. Conf. On Data Engineering (ICDE)*, 1992.

[20] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Int. Conf. on Operating Systems Design and Implementation*, 2000.

[21] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Int. Conf. on VLDB*, 2000.

[22] 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.