Replica Refresh Strategies in a Database Cluster

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

Abstract

Relaxing replica freshness has been exploited in database clusters to optimize load balancing. However, in most approaches, refreshment is typically coupled with other functions such as routing or scheduling, which make it hard to analyze the impact of the refresh strategy itself on performance. In this paper, we propose to support routing-independent refresh strategies in a database cluster with mono-master lazy replication. First, we propose a model for capturing existing refresh strategies. Second, we describe the support of this model in Refresco, a middleware prototype for freshness-aware routing in database clusters. Third, we describe an experimental validation to test some typical strategies against different workloads. The results show that the choice of the best strategy depends not only on the workload, but also on the conflict rate between transactions and queries and on the level of freshness required by queries. Although there is no strategy that is best in all cases, we found that one strategy, ASAUL(0), is usually very good and could be used as default strategy.
Keywords: replication, database cluster, load balancing, refresh strategy.

1 Introduction

Database clusters provide a cost-effective alternative to parallel database systems, *i.e.* database systems on tightly-coupled multiprocessors. A database cluster [9, 8, 25, 26] is a cluster of PC servers, each running an off-the-shelf DBMS. A major difference with parallel database systems is the use of a "black-box" DBMS at each node. Since the DBMS source code is not necessarily available and cannot be changed to be "cluster-aware", parallel database system capabilities such as load balancing must be implemented via middleware.

The typical solution to obtain good load balancing in a database cluster is to replicate data at different nodes so that users can be served by any of the nodes. If the workload consists of (read-only) queries, then load balancing is relatively easy. However, if the workload includes (update) transactions in addition to queries, as in an Application Service Provider (ASP) [8], load balancing gets more difficult since replica consistency must be enforced.

Managing replication in database clusters has recently received much attention. As in distributed databases, replication can be eager (also called synchronous) or lazy (also called asynchronous). With eager replication, a transaction updates all replicas, thereby enforcing the mutual consistency of the replicas. By exploiting efficient group communication services provided by a cluster, eager replication can be made non blocking (unlike with distributed transactions) and scale up to large cluster sizes [12, 13, 24, 11]. Furthermore, it makes query load balancing easy.

With lazy replication, a transaction updates only one replica and the other replicas are updated (refreshed) later on by separate refresh transactions [21, 22]. By relaxing consistency, lazy replication can better provide transaction load balancing, in addition to query load balancing. Thus, depending on the consistency/performance requirements, eager and lazy replication can be both useful in database clusters.

Relaxing consistency using lazy replication has gained much attention [1, 2, 19, 29, 26, 15], even quite recently [10]. The main reason is that applications often tolerate to read data that is not perfectly consistent, and this can be exploited to improve performance. However, replica divergence must be controlled since refreshing replicas becomes more difficult as divergence increases. In [16], we addressed this problem in the context of a shared-nothing database cluster. We chose mono-master lazy replication because it is both simple and sufficient in many applications, *e.g.* ASP, where most of the conflicts occur between transactions and queries. Transactions are simply sent to a single master node while queries may be sent to any node. Because refresh transactions at slave nodes can be scheduled in the same order as the transactions at master nodes, queries always read consistent states, though maybe stale. Thus, with mono-master replication, the problem reduces to maintaining replica freshness. A replica 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 applied. Otherwise, the freshness level reflects the distance between the state of the replica at the slave node and that at the master node. By controlling freshness at a fine granularity level (relation or attribute), based on application requirements, we gained more flexibility for routing queries to slave nodes, thus improving load balancing.

To validate our approach, we built a middleware prototype called *Refresco* (*Routing Enhancer through FREShness COntrol*). In the implementation presented in [16], Refresco had only one refresh strategy which is *on-demand*: if the load balancer selects an underloaded node that is not fresh enough for an incoming query, it first sends refresh transactions to that node before sending the query. Such *routing-dependent* refresh strategy is locally optimal since the freshness level of some nodes may get lower and lower, thus increasing the cost of refreshment. For instance, when all nodes are busy and too stale, refreshment can take much time and hurt performance.

In this paper, we propose to support *routing-independent* refresh strategies that can maintain nodes at a reasonable level of freshness, independently of which queries are routed to them. Thus, refresh transactions can be triggered based on events other than routing. The events can be either internal, *e.g.* when a node is too stale, idle or little busy, or external, *e.g.* after some time from the last refreshment. There are several possible refresh strategies, each being best for a given workload and the level of freshness required by queries. For instance, if the workload is update-intensive and if queries are rare and require a perfect freshness, then it is better to refresh nodes frequently, *e.g.* as soon as possible, in order to take advantage of periods when nodes are query-free to refresh them. On the contrary, when the workload is query intensive but queries do not require high freshness, it is better to refresh only when necessary, in order to not overload nodes with unnecessary refreshment.

The general problem we address can be stated as follows: given a definition of a mixed workload of update transactions and queries, including the required level of freshness for queries, select the best routing-independent refresh strategy, *i.e.* the one that minimizes average query response time.

In most approaches to load balancing, re-

freshment is tightly-coupled with other issues such as scheduling and routing. This makes it difficult to analyze the impact of the refresh strategy itself. For example, refreshment in [26] is interleaved with query scheduling: it is activated by the scheduler, for instance if a node is too stale to fullfill the freshness requirement of any query in the scheduler input queue. Many refresh strategies have been proposed in the context of distributed databases, data warehouse and and database clusters. A popular strategy is to propagate updates from the source to the copies as soon as possible (ASAP), as in [3, 4, 6]. Another simple strategy is to refresh replicas periodically [5, 17] as in data warehouses [7]. Another strategy is to maintain the freshness level of replicas, by propagating updates only when a replica is too stale [28]. There are also mixed strategies. In [20], data sources push updates to cache nodes when their freshness is too low. However, cache nodes can also force refreshment if needed. In [15], an asynchronous Web cache maintains materialized views with an ASAP strategy while regular views are regenerated on demand. In all these approaches, refresh strategies are not chosen to be optimal with respect to the workload. In particular, refreshment cost is not taken into account in the routing strategy. There has been very few studies of refresh strategies and they are incomplete. For instance, they do not take into account the starting time of update propagation [27, 14] or only consider variations of ASAP [23].

This paper has three main contributions. First, we propose a model which allows describing and analyzing existing refresh strategies, independent of other load balancing issues. Second, we describe the support of this model in our Refresco prototype. Third, we describe an experimental validation based on a workload generator, to test some typical strategies against different workloads. The results show that the choice of the best strategy depends not only on the workload itself, but also on the conflict rate between transactions and queries and on the level of freshness required by queries. Although there is no strategy that is best in all cases, we found that one strategy, ASAUL(0), is usually very good and could be used as default strategy. Our prototype allows the DBA to select the best strategy according to the workload type generated by the application. It is thus compliant with the OGSA-DAI [18] definition of a Data Resource Manager providing flexible and transparent access.

The paper is organized as follows. Section 2 describes our database cluster architecture, with emphasis on load balancing and refreshment. Section 3 defines our model to describe refresh strategies. Section 4 defines a workload model which helps defining typical workloads for experimentations. Section 5 presents our experimental validation which compares the relative performance of typical refresh policies. Section 6 concludes.

2 Database Cluster Architecture

Figure 1 gives an overview of our database cluster architecture, which extends the architecture of [16] with a refresh manager, a cluster state manager and a scheduler. The database cluster system is a middleware layer between the clients and the database nodes: it receives requests from clients, sends them to nodes for processing, and receives back results which it forwards to clients.

This middleware preserves the autonomy of both applications and databases which can remain unchanged, as required in ASP [8] for instance. It receives requests from the applications through a standard JDBC interface. All additional information necessary for routing and refreshing is stored and managed sep-

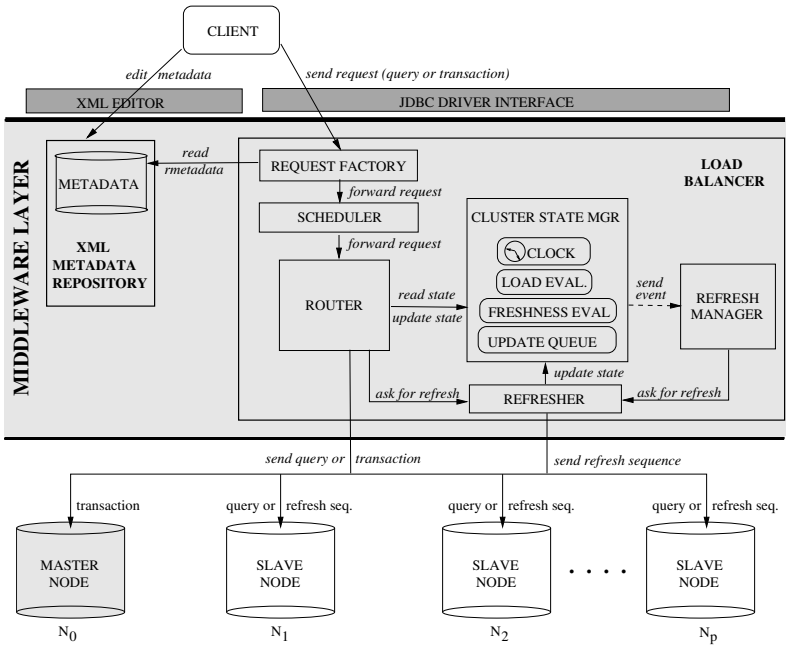


Figure 1: Mono-master replicated database architecture

arately of the requests.

We assume that the database is fully replicated: node N_0 is the *master node* which is used to perform transactions while nodes $N_1, N_2, \ldots, N_p$ are *slaves nodes* used for queries. The master node is not necessarily a single cluster node which could be a single point of failure and a bottleneck. It is an abstraction and can be composed of several cluster nodes coordinated by any eager replication protocol such as [13]. Slave nodes are only updated through *refresh transactions* which are sent sequentially, through *refresh sequences*, according to the serialization (commit) order on the master node. This guarantees the same serialization order on slave nodes.

To support general applications such as ASP, the access to the database is through stored procedures. Thus, clients requests are procedure calls. Metadata useful for the load balancer is specified through an XML editor and stored in a metadata repository. It includes for instance the default level of freshness required by a query. It also includes information about which part of the database is updated by the transactions and read by the queries, thus enabling the detection of potential conflicts between updates and queries. More precisely, each updating (resp. read-only) procedure defines a *transaction class* (resp. *query class*). A query class *potentially conflicts* with a transaction class if an instance of the transaction class may write data that an instance of the query class may read. We formally defined potential conflicts using conflict classes in [16].

The request factory enriches requests with metadata obtained from the repository and dynamic information provided by the clients (*e.g.* parameters for stored procedures). Then it sends the requests to the scheduler. The scheduler can implement different scheduling policies, such as Random, Earliest deadline, Minimal execution time first, and so on. In this paper, in order to focus on refresh policies, we use a simple FIFO scheduling: transactions are sent to the router in the same order they arrive to the scheduler.

Dynamic information such as transaction commit time on the master node, data freshness on slave nodes, estimated nodes load, is maintained by the cluster state manager. The information related to each transaction is maintained until every node has executed the corresponding refresh transaction, after which it is removed.

The router can implement different routing strategies, such as Random, (weighted) Round-robin, Shortest Queue first, Shortest execution first (SELF), and so on. Refresco implements a enhanced version of SELF which includes the estimated cost of refreshing a node on-demand. It works as follows. Upon receiving a transaction, it sends it to the master node. After the transaction committment on the master node, the results are sent to the client and the router updates the cluster state, so that the newly committed transaction is taken into account for computing node freshness. When the router receives a query Q, it computes a cost function for Q for every slave node and selects for execution the slave node which minimizes the cost function, thus minimizing the query response time. The cost of executing a query on a node is composed of the node's load plus an eventual cost of refreshing it before sending the query. The node's load is obtained from the load evaluation module which computes the sum of the remaining execution times of running requests on the node. The execution time of a request is estimated through a weighted mean algorithm based on the previous executions of the request. Depending on application needs, the router can be switched to perform *routing-dependent (on-demand) refreshment*. To this end, it asks the freshness evaluation module to compute, for every node, the corresponding minimum refresh sequence to make the slave node fresh enough for Q, and includes the cost of the possible execution of this sequence into the cost function. After the eventual on-demand refresh is performed by the refresher on the selected node, the router sends the query to this node and updates the cluster state. Since queries are only sent to slave nodes, they do not interfer with the transaction stream on the master node.

The refresh manager handles *routing-independent refreshment*. According to the refresh policy, it receives events coming from different part of the cluster state manager: load evaluation module, freshness evaluation module or external events such as time. It then triggers the selected routing-independent refresh policy which eventually asks the refresher module to perform refresh sequences. Whenever the refresher sends refresh sequences to a node, it updates the cluster state for further freshness evaluations by the corresponding module.

3 Modeling Refresh Strategies

In this section, we propose a model for defining various refresh strategies. It can be used as a basis to help a DBA specifying when, to which slave nodes, and how much to refresh. The refresh model is based on a freshness model which allows measuring the staleness of a slave node with respect to the master node.

3.1 Freshness Model

Freshness requirements are specified for *access atoms*, which represent portions of the database. Depending on the desired granularity, an access atom can be as large as the entire database or as small as a tuple value in a table. A *freshness atom* associated with an access atom a is 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 $\mu(a) \leq t$. The staleness of an access atom on a slave node is defined as the divergence between the value of a on the slave node and the value of a on the master node. The *freshness level* of a set of access

atoms $\{a^1, a^2, ..., a^n\}$ is defined as the logical conjunction of freshness atoms on a^i.

In [16] we introduced several freshness measures. For simplicity in this paper, we consider only measure *Age*. *Age*(a_N) denotes 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 N, *i.e.*

$$Age(a_N) = \begin{cases} Max(current\_time() \\ - T.commit\_time),\ T \in U(a_N) \\ 0 \text{ if } U(a_N) = \emptyset \end{cases}$$

where $U(a_N)$ is the set of transactions updating a and not yet propagated to slave node N.

The *freshness level of a query* Q is a freshness level on the set of access atoms read by Q. Users determine the access atoms of the query at the granularity they desire, and define a freshness atom for each access atom. A node N is fresh enough to satisfy Q if the freshness level of Q is satisfied on N. The *freshness level of a node* N is simply the freshness level on the entire database on N.

3.2 Refresh Model

We propose to capture refresh strategies with the model in Figure 2. A refresh strategy is described by the *triggering events* which raise its activation, the nodes where the refresh transactions are propagated and the number of transactions which are part of the refresh sequence. A refresh strategy may handle one or more triggering events, among:

- *Routing*(N, Q): a query Q is routed to node N.
- *Underloaded*$(N, limit)$: the load of node N decreases above the *limit* value.
- *Stale*$(N, \mu, limit)$: the freshness of node N for measure μ decreases above the *limit* value. In other words, the freshness level of node N for measure μ and threshold *limit* is no more satisfied. In this paper, since we only consider the *Age* measure, this parameter becomes implicit and the event can be simplified as *Stale*$(N, limit)$ which stands for *Stale*$(N, Age, limit)$
- *Update(T)*: a transaction T is sent to the master node.
- *Period(t)*: triggers every t seconds.

As soon as an event handled by the refresh manager is raised, the refresher computes a sequence of refresh transactions to propagate. Depending on the nature of the event, the refresh sequence is sent to a single slave node or broadcast to all slave nodes. For instance, *Routing*(N, Q) activates a refreshment only on slave node N while *Period(t)* activates a refreshment on all the slave nodes.

Finally, the refresh quantity of a strategy indicates how many refresh transactions are part of the refresh sequence. This value can be minimum, *i.e.* the minimum refresh sequence which brings a node to a certain freshness. The maximum value denotes a refresh sequence containing every transaction not yet propagated to the destination. Of course, the quantity may also be arbitrary (for instance, a fixed size).

We apply our refresh model to the following strategies:

3.2.1 On-Demand (OD)

The On-Demand strategy is triggered by a *Routing*(N) event. It sends a minimal refresh to node N to make it fresh enough for Q.

3.2.2 As Soon As Possible (ASAP).

The ASAP strategy is triggered by a *Update_sent(T)* event. It sends a maximal refresh sequence to all the slave nodes. As

Refresh Strategy ::= ({Event}, Destination, Quantity)
Event ::= *Routing(N,Q)*
| *Underloaded(N,limit)*
| *Stale(N,μ,limit)*
| *Update_sent(T)*
| *Period(t)*
Destination ::= Slave Node | All Slave Nodes
Quantity ::= Minimum | Maximum | Arbitrary

Figure 2: Refresh model

ASAP strategy maintains slave nodes perfectly fresh, the maximal refresh sequence is reduced to the transaction T which raised the event.

3.2.3 Periodic(t).

The Periodic(t) strategy is triggered by a *period*(t) event. It sends a maximum refresh sequence to all the slave nodes.

3.2.4 As Soon As Underloaded (ASAUL(limit)).

The ASAUL strategy is triggered by a *Underloaded(N,limit)* event. It sends a maximum refresh sequence to N.

3.2.5 As Soon As Too Stale (ASATS(limit)).

The ASATS strategy is triggered by a *Stale*($N, limit$) event. It sends a maximum refresh sequence to N.

3.2.6 Hybrid Strategies.

Refresh strategies can be combined to improve performance. Though a lot a combinations are possible, we focus here on the interaction between routing-dependent (On-Demand) and routing-independent strategies (all other strategies). Thus, for each routing-independent strategy, we derive an hybrid version which combines it with On-demand. Note however that combining On-demand with itself does not make sense and that ASAP is equivalent to its hybrid version (nodes are maintained perfectly fresh, thus on-demand refresh is never triggered).

The following table summarizes all these refresh strategies.

Strategy	Event	Dest	Qty
OD	*Routing*(N)	N	min
ASAP	*Update_sent(T)*	all	max
Periodic	*Period(t)*	all	max
ASAUL	*Underloaded(N,limit)*	N	max
ASATS	*Stale*($N, limit$)	N	max

4 Modeling Workloads

In this section, we propose a workload model that captures the main workload characteristics which impact refreshment. We use this model to define our experimental workloads for comparing refresh strategies.

4.1 Workload Model

Our workload model is shown in Figure 3.

Workload ::= (transaction workload, query workload, conflict rate)
Transaction workload ::= (nb of transaction clients, active phase, sleeping phase)
Query workload ::= (nb of query clients, waiting time, tolerated staleness)

Figure 3: Workload model

A workload is composed of several clients. Each client is either of type *transaction* or of type *query*, *i.e.* it only sends transactions or only queries. A workload is characterized by a transaction workload, a query workload and a conflict rate. A transaction workload is characterized by a number of transaction clients, an active phase duration and a sleeping phase duration. A query workload is characterized by a number of query clients, a waiting time and a tolerated staleness.

Clients of type transaction have a periodic behaviour. During a period, they are successively all active (they send transactions continuously) and then all sleeping. The *active phase* has a fixed duration while the *sleeping phase* is variable. The shorter the sleeping phase, the higher is the transaction load. The transition between the sleeping phase and the next active phase is smoothed, in order to be more realistic by avoiding all the transactions reaching the router exactly at the same time.

Clients of type query send queries sequentially, with a *waiting time* between two queries. The shorter the waiting time, the higher is the query load. In order to simplify, all the queries in a workload have the same *tolerated staleness*, which is the threshold of every query's freshness level. It describes the maximal staleness a data on a node can have for the query to be executed on it. For instance, a workload where queries require to read perfectly fresh data has a tolerated staleness equal to 0.

We define the *conflict rate* of a workload as the proportion of potential conflicts between transactions and queries. Let $\{TC_1, TC_2, \ldots, TC_n\}$ be the application set of transaction classes and $\{QC_1, QC_2, \ldots, QC_m\}$ the application set of query classes. The conflict rate (cr) of a workload is defined by the following formula :

$$cr = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} \alpha_j \times \text{conflict}(TC_i, QC_j)}{\sum_{j=1}^{m} \alpha_j}$$

where :

- conflict(TC_i, QC_j) is equal to 1 if the transaction class TC_i potentially conflicts (see Section) with the query class QC_j, otherwise it is equal to 0.
- α_j is the number of instances of the query class QC_j in the workload.

4.2 Experimental Workloads

We now use our workload model to define our experimental models which cover a wide range of applications. The number of transaction clients is fixed to 16, while the number of query clients is fixed to 256. Each workload has a total duration of 10000 time units (TU) [1]. We fix the active phase duration to 100 TU. We consider that a transaction load (tl) is high (resp. low) for a sleeping time equal to 50 TU (resp. 300 TU). A query load (ql) is considered as high (resp. low) for a waiting time equal to 0 TU (resp. 300 TU). This allows defining four basic workloads, according to the transaction load and the query load. For instance, a HIGH-LOW workload is composed of a high transaction load and a

[1] In order to get results independent of the underlying DBMS, all durations are described in terms of *time units (TU)* used for simulation.

low query load. All the basic workloads are parameterized with the conflict rate (cr) and a tolerated staleness for queries (ts). Thus, a workload is described as a tuple (tl, ql, cr, ts).

5 Experimental Validation

In this section, we describe our experimental validation where we compare the performance of various refresh strategies under different workloads. After describing our experimental setup, we study the performance of the basic refresh strategies with their hybrid version. Then we study the impact of conflict rates and of tolerated freshness on performance.

5.1 Experimental Setup

Our experimental validation is based on the enhanced version of the Refresco prototype, which is developed in Java 1.4.2. In order to get results independent of the underlying DBMS's behaviour, we simulated the execution of a request on a node, with 128 slave nodes. Each request is considered as a fixed-duration job: 100 TU for query classes and 5 TU for updates classes. We simulated database access using Simjava, a process-based discrete event simulation package in Java (see `http://www.dcs.ed.ac.uk/home/hase/simjava/`). We chose simulation because it makes it easier to vary the various parameters and compare strategies However, we also calibrated our simulator for database access using an implementation of our Refresco prototype on the 64-node cluster system of the Paris team at INRIA (`http://www.irisa.fr/paris/General/cluster.htm`) with PostgreSQL as underlying DBMS.

For each experiment, we first present the results obtained with a perfect freshness required (ts=0) and a varying conflict rate, then with a conflict rate of 1 (cr=1) and a varying tolerated staleness. As our aim is to speed up query execution, we choose the query mean response time of a workload (QMRT) as performance measure.

We study individually each strategy having a parameter: Periodic(t), ASAUL(limit) and ASATS(limit). This allowed selecting the most representative *strategy instance*, by selecting a "small value" (SV) and a "large value" (LV), represented in the following table.

Strategy	Parameter	SV (TU)	LV (TU)
Periodic	t	100	1000
ASAUL	limit	0	500
ASATS	limit	100	500

Note that Periodic(1) and ASATS(0) are *not* representative. Indeed, as they are each time unit or each time a node is not perfectly fresh, they are quite equivalent to ASAP.

5.2 Performance Comparisons of Basic and Hybrid Strategies

We now investigate the impact on performance of combining a basic refresh strategy with the On-Demand strategy. To this end, we use the four basic workloads by varying the conflict rate and the tolerated staleness and compare the performance of each strategy with its hybrid version.

Figure 4 shows the performance comparisons of each strategy with its hybrid version. We only give the results obtained for a low-low workload, with a conflict rate of 0.8 (cr=0.8) and no tolerated staleness (ts=0), but the conclusion is valid for all the workloads.

Strategy	basic QMRT	hybrid QMRT
ASAUL(0)	184	119
ASAUL(500)	125	124
ASAP	124	124
PERIODIC(100)	173	145
PERIODIC(1000)	629	250
ASATS(100)	173	149
ASATS(500)	638	214

Figure 4: Comparing each refresh strategy with its hybrid version

The results show that in every case, the hybrid version is at least as good as the non-hybrid version. In almost all cases, the hybrid version yields a substantial gain compared with the basic strategy (up to 300 % in Figure 4 for ASATS(500)). Of course, there is no gain for ASAP since its hybrid version is equivalent to its basic version. The performance gain is explained by the fact that the On-Demand strategy never performs unnecessary refreshment. It is only triggered when a node is not fresh enough for a query but lightly loaded so that, even with the cost of refreshing the node on-demand, it is the node which minimizes the query response time. In cases where few nodes are fresh enough for a query, this yields better load balancing. In cases where many nodes are fresh enough, the on-demand refreshment is not triggered, thus with no overhead.

Since they always dominate, we only consider hybrid strategies in the remaining of this section.

5.3 Impact of Conflict Rate on Performance

Figure 5 shows the performance (QMRT) of the various refresh strategies versus the conflict rate, with a perfect tolerated staleness (of 0). We omit workloads of type high-low and low-high, but they raise similar conclusions.

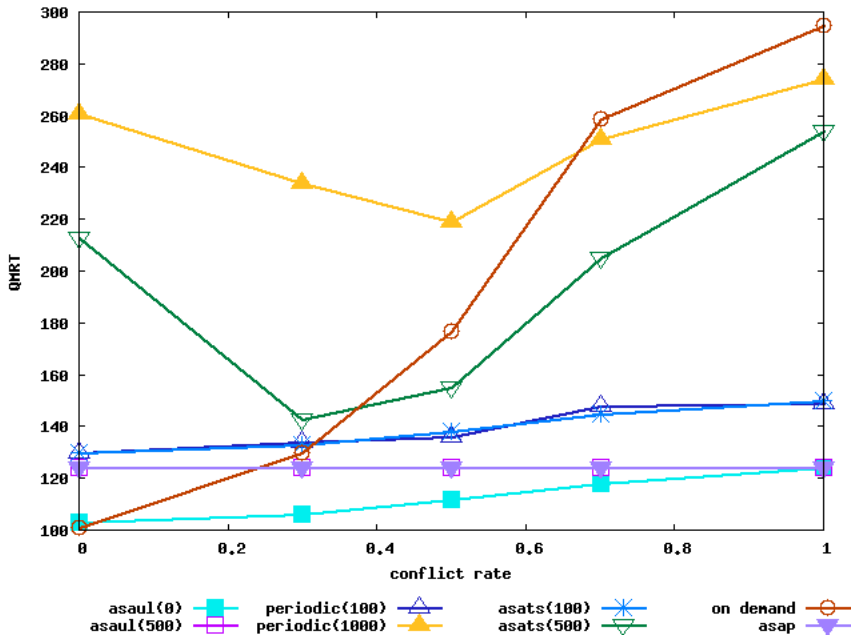
5.3.1 Light Workloads.

Figure 5(a) shows that, except for very small conflict rates, the best performance for light workloads is obtained with strategies that refresh frequently, *i.e.* maintain nodes (almost) always fresh. These strategies are ASAP (obviously) and ASAUL since nodes are idle very often. They trigger refreshment often but do not interfere much with queries because the refreh sequences are executed mostly during idle periods.

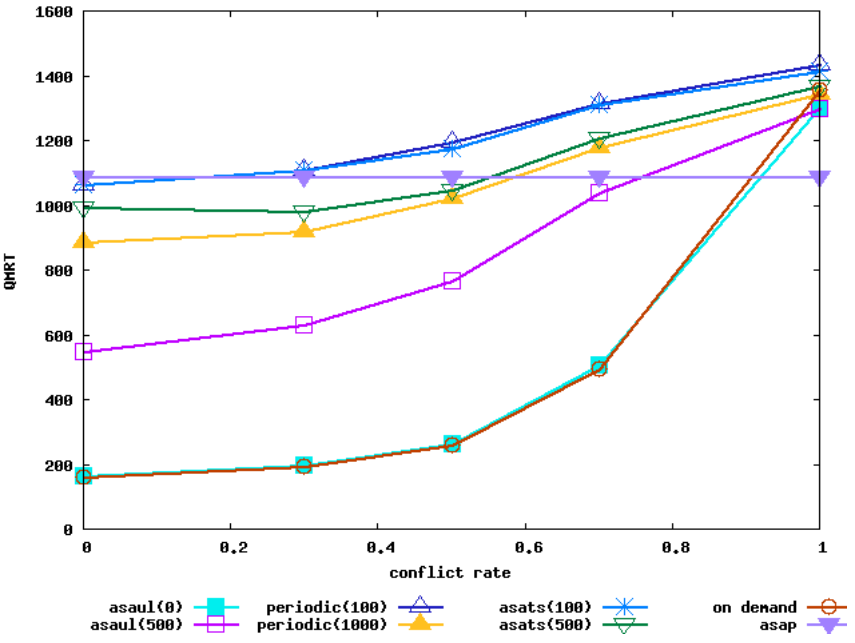
On the contrary, On-Demand performs rather poorly as soon as the conflict rate is exceeds 0.4. Indeed, since queries are rare, it is triggered rarely. Thus, each time a query is routed, the refresher must propagate many updates (since the last refresh) before executing the query. This increases response time significantly.

5.3.2 Heavy Workloads.

In Figure 5(b) , the behavior of the strategies is quite different from that in Figure 5(a). On-Demand yields the best performance in most cases (except when the conflit rate exceeds than 0.9) because refreshment is done often (the query frequency is high), but only when needed.



(a) light workloads: *(low,low,cr,0)*



(b) heavy workloads: *(high,high,cr,0)*

Figure 5: Performance comparisons with varying conflict rate (tolerated freshness=0)

In this context, ASAP is better only for very high conflict rates because it always refreshes. This is useless for smaller conflict rates where refresh is not frequently required. Similarly, Periodic and ASATS do not perform well. As they do not take into account the nodes load and perform maximum refresh sequences, they raise useless overhead when refreshing. We also observe that ASAUL(0) performs as On-Demand because nodes are never idle.

5.3.3 Conclusions.

When perfect freshness is required, we can draw the following conclusions. On-Demand is always the best strategy for heavy workloads, except when the conflict rate is very high, ASAP is the best strategy for light workloads except, for very small conflict rates, and in all cases when the conflict rate is very high. ASAUL(0) is the best overall strategy: for each workload type and conflict rate, it is always the best or "close to the best" (20 % in the worst case).

5.4 Impact on Tolerated Staleness on Performance

Figure 6 shows the performance (QMRT) of the various refresh strategies versus the tolerated staleness, with a high conflict rate (of 1). High-low and low-high workloads give results similar to high-high and are omitted.

A general observation is that, for all the strategies except ASAP, the results are better when the tolerated staleness is higher. Obviously, when queries do not require high freshness, there is a higher probability that a node is fresh enough for any query. Thus on-demand refresh is less necessary, which speeds up query execution. This is not the case for ASAP, since it does not require on-demand refresh. When the tolerated staleness is beyond a given value, performance does not change for most strategies. This is due to the fact that all the nodes are always fresh enough for queries and thus on-demand is no more triggered. Thus, refreshing nodes is useless for queries. This is obviously the case for Periodic, but also for ASATS. In fact, ASATS also behaves periodically in this context. This is due to the fact that transactions are performed periodically on the master node, thus the freshness on slave nodes always decreases at the same speed. More interesting is the case of ASAUL. For light workloads, ASAUL has also a periodic behaviour : when a node is idle or lightly loaded, ASAUL refreshes it and the node becomes busy. Thus, it is no more refreshed during a given duration and gets idle. Then ASAUL refreshes it, and so on. For heavy workloads, nodes are always busy and thus, as already mentionned, ASAUL is similar to On-Demand. Particularly, as nodes are never idle, ASAUL(0) performs quite the same as On-Demand. On-Demand is always sensitive to the tolerated staleness. As nodes are refreshed only when necessary, performance increases as tolerated stalenes increases.

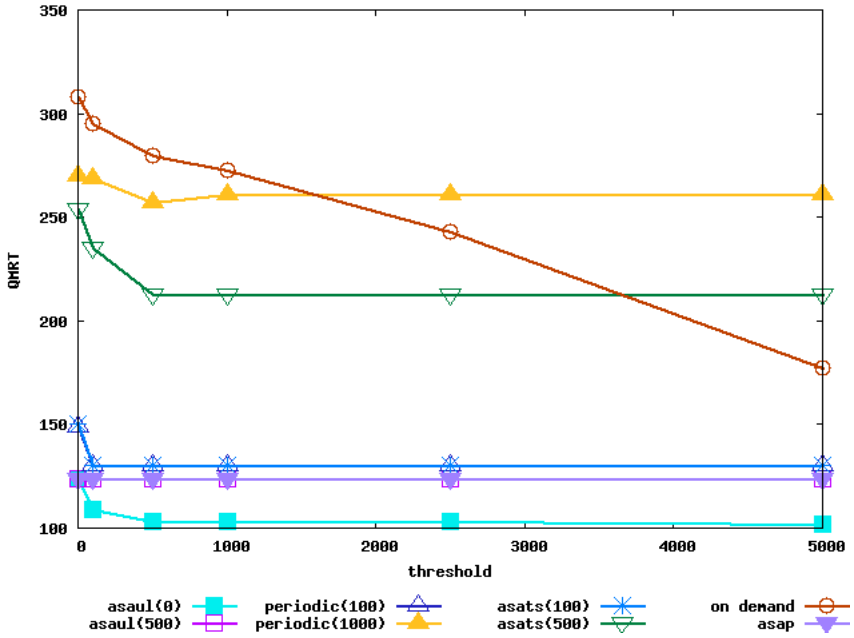
5.4.1 Light Workloads.

Figure 6(a) shows that On-Demand is outperformed by strategies which frequently refresh nodes and thus take advantage of nodes being frequently idle. Among them, ASAUL(0) is the best since it naturally adapts to idle node events.

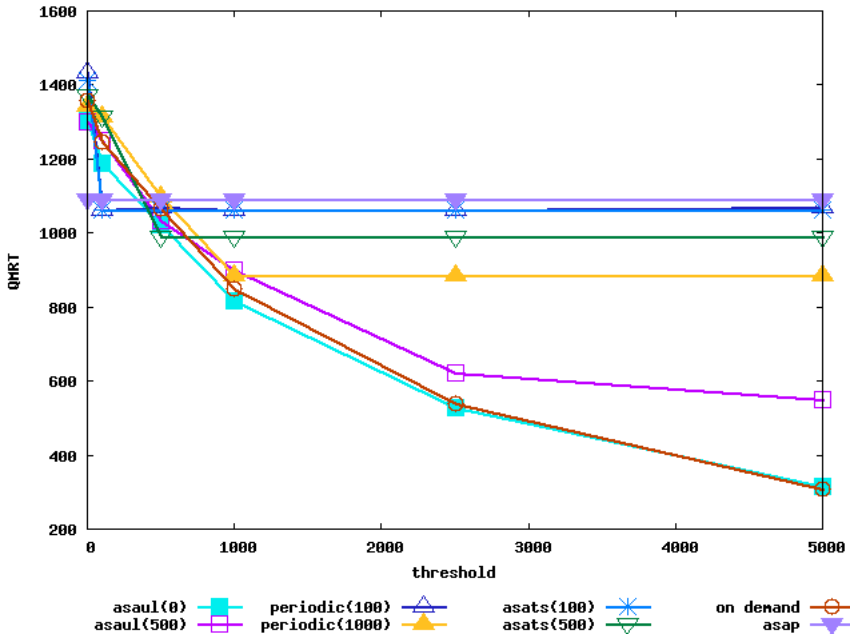
5.4.2 Heavy Workloads.

Figure 6(b) shows that On-Demand outperforms the other strategies whenever the tolerated staleness is above approximately 500 TU. In this case, the overhead due to the frequent refresh performed by other strategies is higher since nodes are never idle. It is moreover useless since queries do not require high freshness.

Again, since nodes are never idle, ASAUL(0) only triggers on-demand and thus



(a) light workloads: *(low,low,1,ts)*



(b) heavy workloads: *(high,high,1,ts)*

Figure 6: Comparing strategies for varying tolerated staleness and conflict rate 1

its performance is the same as On-Demand. When the tolerated staleness is below 500 TU, frequent refreshments are necessary, and thus ASAP is the best strategy.

5.4.3 Conclusions.

For workloads where the conflict rate is high, we can draw the following conclusions. On-Demand is always the best strategy for heavy workloads except when the tolerated staleness is small, ASAP is the best strategy for light workloads except when the tolerated staleness is very high. ASAUL(0) is the best overall strategy since it is equivalent to ASAP for light workloads, and to On-Demand for heavy workloads.

6 Conclusion

Relaxing replica freshness can be well exploited in database clusters to optimize load balancing. However, the refresh strategy requires special attention as the way refreshment is performed has strong impact on response time. In particular, it should be independent of other load balancing issues such as routing.

In this paper, we proposed a refresh model that allows capturing state-of-the-art refresh strategies in a database cluster with mono-master lazy replication. We distinguished between the routing-dependent (or on-demand) strategy, which is triggered by the router, and routing-independent strategies, which are triggered by other events, based on time-outs or on nodes state. We also proposed hybrid strategies, by mixing the basic strategies with the On-demand strategy.

We described the support of this model by extending the Refresco middleware prototype with a refresh manager which implements the refresh strategies described in the paper. The refresh manager is independent of other load balancing functions such as routing and scheduling. In our architecture, supporting hybrid strategies is straightforward, since they are simple conjunctions of basic strategies already implemented in the refresh manager (or in the router for On-Demand).

In order to test the different strategies against different application types, we proposed a workload model which captures the major parameters which impact performance: transaction and query loads, conflict rate between transactions and queries, and level of freshness required by queries on slave nodes.

We described an experimental validation to test some typical strategies against different workloads. An important observation of our experiments is that the hybrid strategies always outperform their basic counterpart. The experimental results show that the choice of the best strategy depends not only on the workload, but also on the conflict rate between transactions and queries and on the level of freshness required by queries. Although there is no strategy that is best in all cases, we found that one strategy (ASAUL(0)) is usually very good and could be used as default strategy.

Finally, the work presented in this paper can be seen as a first step toward a self-adaptable refresh strategy, which would combine different strategies by analysing on-line the incoming workload. According to the real-life applications dynamicity, our middleware should automatically adapt the refresh strategy to the current workload, using for instance machine-learning techniques.

References

[1] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. on Database Systems*, 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] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi isolation levels. In *ACM SIGMOD Int. Conf.*, 1995.

[4] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *ACM SIGMOD Int. Conf.*, pages 97–108, 1999.

[5] D. Carney, S. Lee, and S. Zdonik. Scalable application aware data freshening. In *IEEE Int. Conf. on Data Engineering*, 2002.

[6] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *IEEE Int. Conf. on Data Engineering*, pages 469–476, 1996.

[7] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *ACM SIGMOD Int. Conf.*, pages 469–480, 1996.

[8] 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.

[9] S. Gançarski, H. Naacke, and P. Valduriez. Load balancing of autonomous applications and databases in a cluster system. In *Workshop on Distributed Data and Structures (WDAS)*, 2002.

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

[11] R. Jiménez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Are quorums an alternative for database replication. *ACM Trans. on Database Systems*, 28(3):257–294, 2003.

[12] B. Kemme and G. Alonso. Don't be lazy be consistent : Postgres-r, a new way to implement database replication. In *Int. Conf. on Very Large Data Bases*, pages 134–143, 2000.

[13] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. on Database Systems*, 25(3):333–379, 2000.

[14] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An adaptive framework for tunable consistency and timeliness using replication. In *Int. Conf. on Dependable Systems and Networks*, pages 17–26, 2002.

[15] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *Int. Conf. on Very Large Data Bases*, pages 393–404, 2003.

[16] 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)*, pages 174–193, 2004.

[17] H. Liu, W.-K. Ng, and E.-P. Lim. Scheduling queries to improve the freshness of a website. *World Wide Web*, 8(1):61–90, 2005.

[18] S. Malaika, A. Eisenberg, and J. Melton. Standards for databases on the grid. *SIGMOD Rec.*, 32(3):92–100, 2003.

[19] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on Very Large Data Bases*, 2000.

[20] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on Very Large Data Bases*, 2000.

[21] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Int. Conf. on Very Large Data Bases*, 1999.

[22] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3):237–267, 2000.

[23] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3–4):305–318, 2000.

[24] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Int. Conf. on Distributed Computing (DISC'00)*, pages 315–329, 2000.

[25] U. Röhm, K. Böhm, and H.-J. Schek. Cache-aware query routing in a cluster of databases. In *IEEE Int. Conf. on Data Engineering*, 2001.

[26] 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 Very Large Data Bases*, 2002.

[27] Y. Saito and H. M. Levy. Optimistic replication for internet data services. In *Int. Symp. on Distributed Computing*, pages 297–314, 2000.

[28] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining coherency of dynamic data in cooperative repositories. In *Int. Conf. on Very Large Data Bases*, 1995.

[29] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Int. Conf. on Very Large Data Bases*, 2000.