Replica Refresh Strategies in a Database Cluster

Cécile Le Pape[1] and Stéphane Gançarski[1]

Laboratoire d'Informatique de Paris 6, Paris, France
email : Firstname.Lastname@lip6.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 makes 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 freshness level required by queries. Although there is no strategy that is best in all cases, we found that one strategy is usually good and could be used as default strategy. This work was partially financed through the ANR-ARA Respire project.
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 [10,21,22] is a cluster of PC servers, each running an off-the-shelf DBMS. With a large number of servers, it can reach high performances, and thus can be used as a basic block for building Grid environments, by grouping several database clusters distributed in a large scale network such as the Internet. 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 it is the case in Grid environments, load balancing gets more difficult since replica consistency must be enforced. With lazy replication, a transaction updates only one replica and the other replicas are updated (refreshed) later on by separate refresh transactions [19]. By relaxing consistency, lazy replication can provide flexible transaction load balancing, in addition to query load balancing.

Relaxing consistency using lazy replication has gained much attention [1,2,18,25,22,14], even quite recently [11]. 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 [15], 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 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.

In most approaches to load balancing, refreshment 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 [22] 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. Furthermore, they do not use routing-dependent refresh: when no node is fresh enough for a query, the query execution is delayed, without guarantee on the query liveness. Many refresh strategies have been proposed in the context of distributed databases, data warehouse 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 [5,16] 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 [24]. There are also mixed strategies. In [18], data sources push updates to cache nodes when their freshness is too low. However, cache nodes can also force refreshment if needed. In [14], 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 [23,13] or only consider variations of ASAP [20].

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, As Soon As Underloaded or ASAUL(0), is usually very good and could be used as default strategy. Our prototype allows to select the best strategy according to the workload type generated by the application. It is thus compliant with the OGSA-DAI [17] definition of a Data Resource Manager providing flexible and transparent access for Grid applications.

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

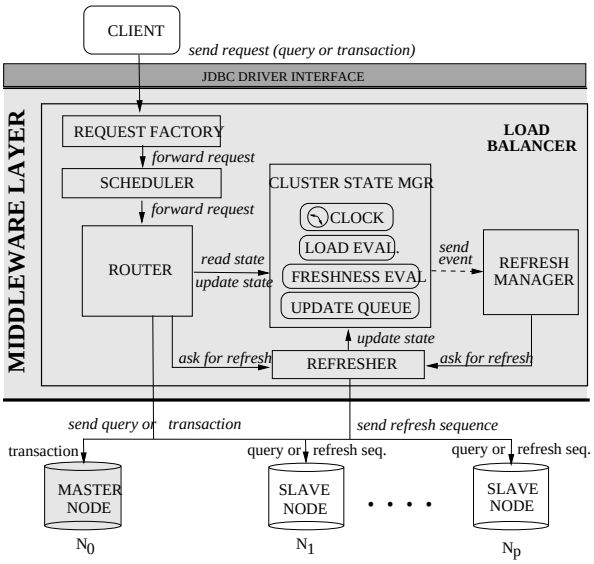


Fig. 1. Mono-master replicated database architecture

Figure 1 gives an overview of our database cluster architecture. It preserves the autonomy of both applications and databases which can remain unchanged, which is important for Grid applications which require sites autonomy. It receives requests from the applications through a standard JDBC interface. All additional information necessary for routing and refreshing is stored and managed separately 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 [12]. 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. Access to the database is through stored procedures. 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 [15].

The request factory enriches requests wih metadata such as parameters for stored procedures and required freshness for a query. Then it sends the requests to a FIFO 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 implements an enhanced version of SELF (Shortest Execution Length First). 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 interfere 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 parts 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.

3 Modeling Refresh Strategies

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 $\mu(a) \leq t$. If we note a_i the copy of access atom a on the slave node N_i, the staleness of a_i is computed by $\mu(a_i)$ and represents the divergence between the value of a_i (on the slave node) and the value of a_0 (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 [15] 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. 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.

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 gets a value less than or equal to the *limit* value.
- *Stale*$(N, \mu, limit)$: the freshness of node N for measure μ decreases below 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_sent(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, which we implemented and compared, since they are the most popular in the literature.

- **On-Demand (OD).** On-Demand strategy is triggered by event *Routing*(N). It sends a minimal refresh to node N to make it fresh enough for Q.
- **As Soon As Possible (ASAP).** ASAP strategy is triggered by a *Update_sent(T)* event. It sends a maximal refresh sequence to all the slave

nodes. As ASAP strategy maintains slave nodes perfectly fresh, the refresh sequence is reduced to the transaction T which raised the event.
- **Periodic(t).** The Periodic(t) strategy is triggered by a *period*(t) event. It sends a maximum refresh sequence to all the slave nodes.
- **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.
- **As Soon As Too Stale (ASATS(limit)).** ASATS strategy is triggered by event *Stale*($N, limit$). It sends a maximum refresh sequence to N.

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. We ran several experiments (not shown here for space limitations) to compare each basic strategy with its hybrid version. They showed that hybrid strategies always outperform basic strategies because they never trigger unnecessary refreshments. Therefore in the following, we study only hybrid strategies. In order to simplify the presentation, we use the same name as the basic strategy, since there is no ambiguity.

4 Experimental Validation

In this section, we compare the performance of hybrid refresh strategies under different workloads. After describing our experimental setup and workloads, we study the impact of conflict rate and of tolerated freshness on performance.

4.1 Experimental Setup and Workload

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, 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. 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. In this case, for typical transactions and queries, the value of a Time Unit (TU) is approximately 10 ms.

Our main objective is to provide a relative comparison of the refresh strategies. Therefore, we strive to keep the workload model simple, with a definition of the main parameters that impact refreshment. Note that our

objective is not to capture all possible workloads which would require a much more complex workload model and is beyond the scope of this paper.

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. 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 TU. Each request is considered as a fixed-duration job: 100 TU for queries and 5 TU for transactions. We consider that a transaction load (*tl*) is low (respt. high) when the transaction clients are active 1/4 (respt. 2/3) of the time. A query load (*ql*) is low (respt. high) when queries clients wait 300 TU (respt. 0 TU) between two queries. All the workloads are parameterized with the conflict rate (cr) and a tolerated staleness for queries (ts). 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 :

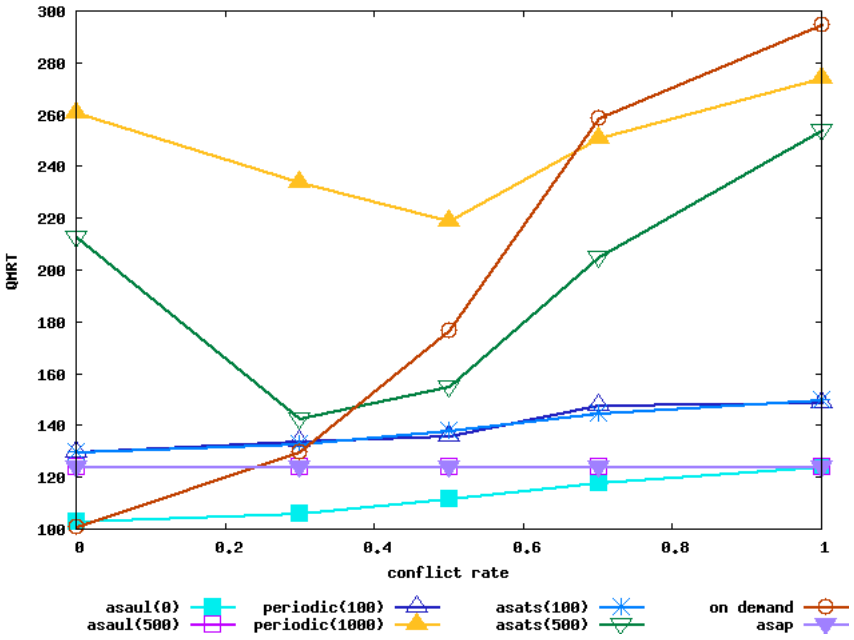
$$cr = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} \alpha_j \times \text{conflict}(TC_i, QC_j)}{\sum_{j=1}^{m} \alpha_j}$$

where conflict(TC_i, QC_j) is equal to 1 if the transaction class TC_i potentially conflicts (see Section 2) with the query class QC_j, otherwise it is equal to 0 and α_j is the number of instances of the query class QC_j in the workload. 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 is 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. Thus, a workload is described as a tuple (tl, ql, cr, ts). Not all the parameters do impact on all the strategies. For instance, the ASAP strategy, which propagates immediately any transaction sent to the master node, is not sensitive to the cr and ts parameters.

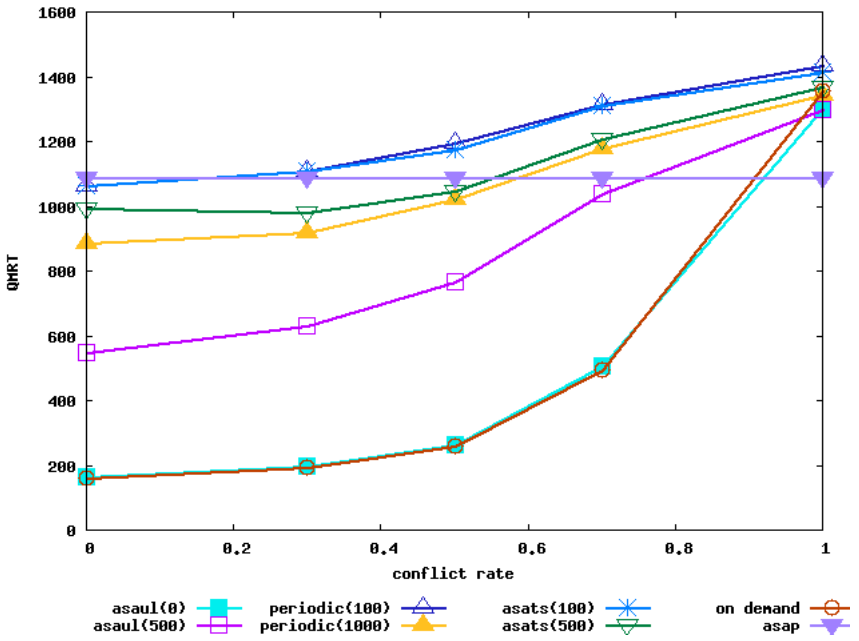
4.2 Impact of Conflict Rate on Performance

Figure 2 shows the query mean response time (QMRT, average of the observed response times of queries during the experiment) of the various refresh strategies versus the conflict rate. As we focus on the conflict rate, there is no tolerated staleness (ts is fixed to 0), which is the worst case for performances. We omit workloads of type (high,low,cr,ts) and (low,high,cr,ts), but they yield similar conclusions.

Light Workloads. Figure 2(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



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



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

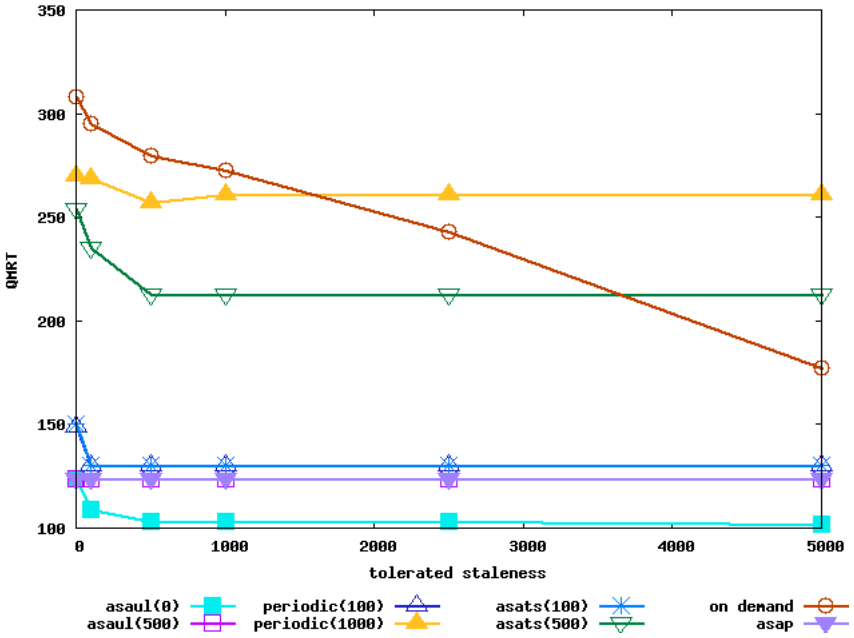
Fig. 2. Performance comparisons with varying conflict rate (tolerated staleness=0)

because the refresh sequences are executed mostly during idle periods. In this context, ASAUL(0) is better than ASAP since it refreshes exactly during idle periods while ASAP may trigger refreshments during non-idle periods, even if such periods are rare. On the contrary, On-Demand performs rather poorly as soon as the conflict rate 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.

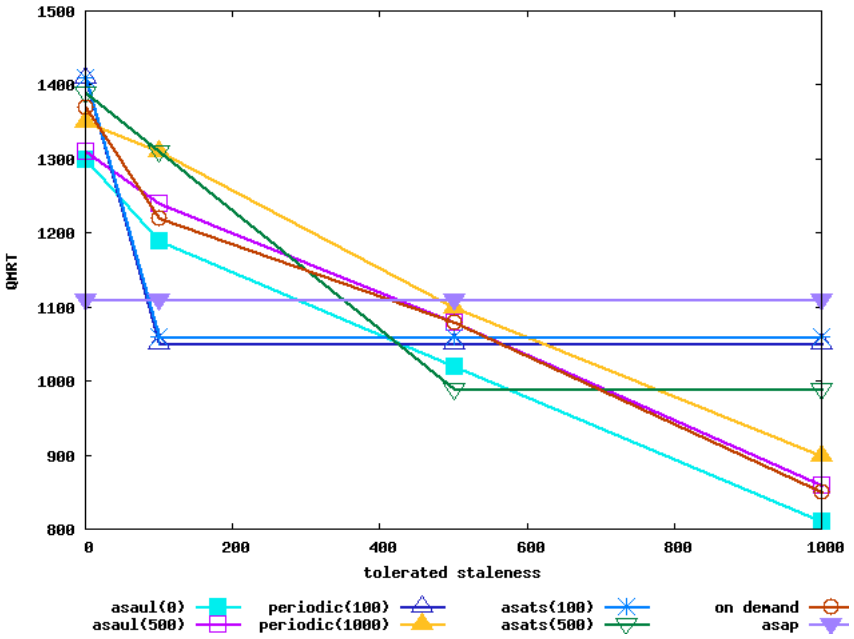
Heavy Workloads. In Figure 2(b) , the behavior of the strategies is quite different from that in Figure 2(a). On-Demand yields the best performance in most cases (except when the conflit rate exceeds 0.9) because refreshment is done often (the query frequency is high), but only when needed. In this context, 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.

4.3 Impact of Tolerated Staleness on Performance

Figure 3 shows the performance (QMRT) of the various refresh strategies versus the tolerated staleness. As we focus here on the tolerated staleness, the conflict rate is fixed to 1, which is the worst case w.r.t to performances. High-low and low-high workloads give results similar to high-high and thus are omitted. A general observation is that, for all 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. For light workloads, ASAUL has also a periodic behavior: when a node is idle or lightly loaded, ASAUL refreshes it and the node becomes busy. Thus, it is no longer 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 mentioned, ASAUL is similar to On-Demand. In particular, 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 staleness increases.



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



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

Fig. 3. Comparing strategies for varying tolerated staleness and conflict rate 1

Light Workloads. Figure 3(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.

Heavy Workloads. Figure 3(b) shows that when the tolerated staleness is below 100 TU, ASAP is the best strategy, since frequent refreshments are necessary. From 100 up to 500, PERIODIC(100) and ASATS(100), which behave equally, are the best strategies. When the tolerated staleness is over 500, the overhead due to frequent refreshment is higher since nodes are never idle. Furthermore, it is useless since queries do not require high freshness. In this case, ASAUL(0) is the best strategy since it naturally adapts to idle node events. For the sake of clarity, values for a tolerated staleness over 1000 are not represesented. They remain constant for all the strategies, except for ASAUL(500) which still decreases down to 550 for a tolerated staleness of 5000, and for ASAUL(0) and On-Demand which have almost equal performance (since nodes are never idle, ASAUL(0) only triggers on-demand) and decrease down to 300, thus being the best strategies.

5 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 (among others) 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 for the workload types we defined. As a future work, we plan to continue testing strategies against other workload types, using a richer workload model. For instance, we can assign different freshness levels for different queries in the same workload, or we can vary the ratio query/transaction in a workload, and so on. We also plan to integrate the refresh strategies into the multi-master approach presented in [9], as we suggested in [8]. 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.

Our approach currently works on a database cluster. As mentionned in the introduction, database clusters are good candidates to build large scale Grid environments. However, this implies that we must adress some new issues to cope with Grid application requirements. The first issue is the heterogeneity of the sources. Our approach handles any relational data sources, through the use of SQL procedure and a standard JDBC driver. We must adapt it to non-relational data sources, for instance XML documents, for instance using a mediator/wrapper approach. The second issue is fault-tolerance : we must distribute our middleware over several nodes, using for instance a shared memory layer, to prevent it from being a single point of failure. Finally, we must also adapt our system to large scale distribution, by modifying the cost function used for load balancing, in order to take into account the different latencies between different sites.

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, C. Le Pape, and H. Naacke. Fine-grained refresh strategies for managing replication in database clusters. In *VLDB Wshp. on Design, Implementation and Deployment of Database Replication*, pages 47–54, 2005.
9. S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. The leganet system: Freshness-aware transaction routing in a database cluster. *Information Systems*, To appear.
10. 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.
11. 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.
12. 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.
13. 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.
14. A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *Int. Conf. on VLDB*, pages 393–404, 2003.
15. 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.
16. 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.
17. S. Malaika, A. Eisenberg, and J. Melton. Standards for databases on the grid. *SIGMOD Rec.*, 32(3):92–100, 2003.
18. C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Int. Conf. on VLDB*, 2000.
19. E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Int. Conf. on VLDB*, 1999.
20. E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3–4):305–318, 2000.
21. 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.
22. 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.
23. Y. Saito and H. M. Levy. Optimistic replication for internet data services. In *Int. Symp. on Distributed Computing*, pages 297–314, 2000.
24. S. Shah, K. Ramamritham, and P. Shenoy. Maintaining coherency of dynamic data in cooperative repositories. In *Int. Conf. on VLDB*, 1995.
25. H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Int. Conf. on VLDB*, 2000.