Available online at www.sciencedirect.com
ScienceDirect
Information Systems 32 (2007) 320–343
www.elsevier.com/locate/infosys

The leganet system: Freshness-aware transaction routing in a database cluster

Stéphane Gançarski[a], Hubert Naacke[a], Esther Pacitti[b], Patrick Valduriez[b,*]

[a]*LIP6, University Paris 6, France*
[b]*INRIA and LINA, University of Nantes, France*

Received 17 February 2004; received in revised form 14 September 2005; accepted 20 September 2005

Recommended by: B. Kemme

Abstract

We consider the use of a database cluster for Application Service Provider (ASP). In the ASP context, applications and databases can be update-intensive and must remain autonomous. In this paper, we describe the Leganet system which performs freshness-aware transaction routing in a database cluster. We use multi-master replication and relaxed replica freshness to increase load balancing. Our transaction routing takes into account freshness requirements of queries at the relation level and uses a cost function that takes into account the cluster load and the cost to refresh replicas to the required level. We implemented the Leganet prototype on an 11-node Linux cluster running Oracle8i. Using experimentation and emulation up to 128 nodes, our validation based on the TPC-C benchmark demonstrates the performance benefits of our approach.

Keywords: Database cluster; Transaction routing; Load balancing; Replication; Freshness; Performance

1. Introduction

Database clusters now provide a cost-effective alternative to parallel database systems. A *database cluster* [1] is a cluster of PC servers, each running an off-the-shelf DBMS. A major difference with parallel database systems implemented on PC clusters [2], e.g., Oracle Real Application Cluster, is the use of a "black-box" DBMS at each node which avoids expensive data migration. However, since the DBMS source code is not necessarily available and cannot be changed or extended to be "cluster-aware", additional capabilities like parallel query processing must be implemented via middleware. Database clusters make new businesses like Application Service Provider (ASP) economically viable. In the ASP model, customers' applications and databases (including data and DBMS) are hosted at the provider site and need be available, typically through the Internet, as efficiently as if they were local to the customer site. Thus, the challenge for a provider is to fully exploit the

*Corresponding author. Tel.: +33 2 51 12 58 24; fax: +33 2 51 12 58 97.
E-mail addresses: Stephane.Gancarski@lip6.fr (S. Gançarski), Hubert.Naacke@lip6.fr (H. Naacke), Esther.Pacitti@univ-nantes.fr (E. Pacitti), Patrick.Valduriez@inria.fr (P. Valduriez).

0306-4379/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2005.09.004

cluster's parallelism and load balancing capabilities to obtain a good cost/performance ratio. The typical solution to obtain good load balancing in a database cluster is to replicate applications and data at different nodes so that users can be served by any of the nodes depending on the current load. This also provides high-availability since, in the event of a node failure, other nodes can still do the work. This solution has been successfully used by Web search engines using high-volume server farms (e.g., Google). However, Web search engines are typically read-intensive which makes it easier to exploit parallelism.

In the ASP context, the problem is far more difficult. First, applications and databases must remain *autonomous*, i.e., remain unchanged when moved to the provider site's cluster and remain under the control of the customers as if they were local, using the same DBMS. Preserving autonomy is critical to avoid the high costs and problems associated with code modification. Second, applications can be update-intensive and the use of replication can create consistency problems [3,4]. For instance, two users at different nodes could generate conflicting updates to the same data, thereby producing an inconsistent database. This is because consistency control is done at each node through its local DBMS. The main solution readily available to enforce global consistency is to use a parallel database system such as Oracle Real Application Cluster or DB2 Parallel Edition. If the customer's DBMS is from a different vendor, this requires heavy migration (for rewriting customer applications and converting databases). Furthermore, this hurts the autonomy of applications and databases which must be under the control of the parallel database system.

In this paper, we describe a new solution for routing transactions in a database cluster which addresses these problems. This work has been done in the context of the Leg@Net project sponsored by the RNTL[1] whose objective was to demonstrate the viability of the ASP model for legacy (pharmacy) applications in France. Our solution exploits a replicated database organization. The main idea is to allow the system administrator to control the tradeoff between database consistency and performance when placing applications and databases onto cluster nodes. Databases and applications are replicated at multiple nodes to increase access performance. Application requirements are captured (at compile time) and stored in a shared directory used (at run time) to allocate cluster nodes to user requests. Depending on the users' requirements, we can control database consistency at the cluster level. For instance, if an application is read-only or the required consistency is weak, then it is easy to execute multiple requests in parallel at different nodes. But if an application is update-intensive and requires strong consistency (e.g., integrity constraint satisfaction), an extreme solution is to run it at a single node and trade performance for consistency.

There are important cases where consistency can be relaxed. With lazy replication [5], transactions can be locally committed and different replicas may get different values. Replica divergence remains until reconciliation. Meanwhile, the divergence must be controlled for at least two reasons. First, since synchronization consists in producing a single history from several diverging ones, the higher the divergence is, the more difficult the reconciliation. The second reason is that read-only applications may tolerate reading inconsistent data. In this case, inconsistency reflects a divergence between the values actually read and the values that should have been read in ACID mode.

In most approaches (including ours), consistency reduces to freshness: update transactions are globally serialized over the different cluster nodes, so that whenever a query is sent to a given node, it reads a consistent state of the database. In this paper, global consistency is achieved by ensuring that conflicting transactions are executed at each node in the same relative order. However, the consistent state may not be the latest one, since update transactions may be running at other nodes. Then, the *data freshness* of a node reflects the difference between the data state of the node and the state it would have if all the running transactions had already been applied to that node.

In this paper, we describe the design and implementation of the Leganet system which performs freshness-aware transaction routing in a database cluster. We use multi-master replication and relaxed replica freshness to increase load balancing. The Leganet architecture, initially proposed in [6], preserves database and application autonomy using non-intrusive techniques that work independently of any DBMS. The main

[1] www.industrie.gouv.fr/rntl/AAP2001/Fiches_Resume/LEG@NET.htm, between LIP6, Prologue Software and ASP-Line.

contribution of this paper is a transaction router which takes into account freshness requirements of queries at the relation level to improve load balancing. This router uses a cost function that takes into account not only the cluster load in terms of concurrently executing transactions and queries, but also the estimated time to refresh replicas to the level required by incoming queries. Using the Leganet prototype implemented on an 11-node cluster running Oracle8i and using emulation up to 128 nodes, our validation based on the TPC-C OLTP benchmark [7] demonstrates the performance benefits of our approach.

This paper is organized as follows. Section 2 provides a motivating example for transaction routing with freshness control. Section 3 introduces the basic concepts and assumptions regarding our replication model and freshness model. Section 4 describes the architecture of our database cluster, focusing on the transaction router. Section 5 presents the strategies for transaction routing with freshness control, with the cost functions used by those strategies. Section 6 describes the Leganet prototype. Section 7 gives a performance evaluation using a combination of experimentation and emulation. Section 8 compares our approach with related work. Section 9 concludes.

2. Motivations

To illustrate how transaction routing with freshness control can improve load balancing, let us consider a simple example (derived from the TPC-C benchmark). We use the relations Stock (item, quantity, threshold) and Order (item, quantity).

Transaction D decreases the stock quantity of item *id* by q.

procedure $D(id, q)$:
update Stock **set** quantity = quantity–q
where item = *id*;

Query Q that checks for the stocks to renew:

select item **from** Stock
where quantity < threshold

Finally, transaction O increases the ordered quantity of item *id* by q.

procedure O(id, q):
update Order **set** quantity = quantity + q
where item = id;

Since D and O do not access the same relation, it is easy for the system to detect that they do not potentially conflict. In other words, any D transaction is commutative with any O transaction. In this example, for simplicity, we measure transaction's changes and data's staleness as a number of modified tuples, and the cost of executing a transaction (resp. query) is 1 (resp. 4). Let us assume that both D and O change 1 tuple and that Q tolerates a staleness of 2 (i.e., Q can be executed on a replica that has not yet received 2 D transactions executed on other replicas). Let us consider a sale application that executes a sequence of D and O transactions, until reaching the state depicted in Fig. 1. $D11$ and $D21$ have been routed to node N_1, thus its load is 2. $O21$ and $O11$ have been routed to N_2, thus its load is 2. As $D20$ and $D10$ have already been executed on N_1 but not yet propagated to N_2, and D accesses a relation read by Q, the staleness of N_2 with respect to Q is 2. It is 0 on N_1, since all the executed transactions on N_2 do not conflict with Q. As shown in the figure, the transactions are balanced over the two nodes without needing any refreshment, since D transactions do not conflict with O transactions. Without conflict detection, all the transactions would have been sent to the same (perfectly fresh) node or refresh transactions would have been sent to both nodes to maintain them perfectly fresh for the next transactions. In both cases, this increases transaction latency. Of course, O (resp. D) transactions need be propagated to N_1 (resp. N_2), but this can be done later on, when N_1 (resp. N_2) is less loaded. Let

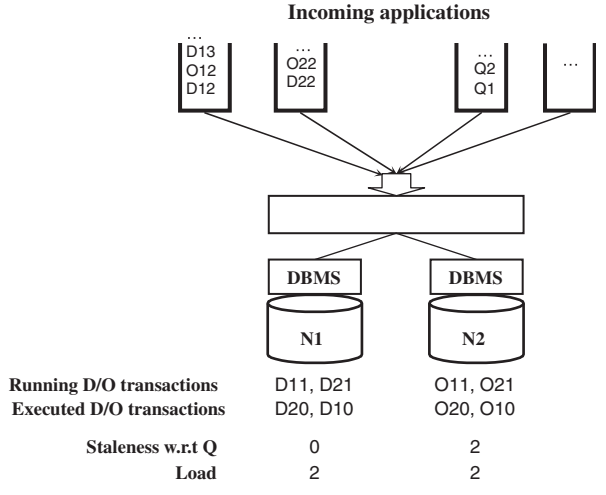

Fig. 1. Routing example.

us assume now that a user issues query $Q1$. Node N_1 is fresh enough for the query (i.e., with respect to relation Stock) and its load is 2. Thus the cost of executing $Q1$ on N_1 is 2. It is 4 on N_2, since the load of N_2 is 2 and the cost to refresh it for Q is 2, for propagating $D20$ and $D10$. Thus, $Q1$ is routed to N_1, the load of which becomes 6 (2 plus 4 for $Q1$). Now assume a user issues query $Q2$. Since the cost is now 6 on N_1 and still 4 on N_2, $Q2$ is routed to N_2, which gives the best response time despite the cost of refreshment. A strategy which does not take into account the possibility of refreshing a node before sending it a transaction would choose (and thus overload) N_1, which is the only node fresh enough at routing time. This example illustrates the importance of considering the processing cost of refreshing a node for making the routing decision.

3. Basic concepts

In this section, we introduce the basic concepts and assumptions regarding metadata, our replication model and freshness model.

3.1. Metadata

Our system acts as a middleware layer between black-box applications and black-box DBMSs. Thus, it needs some information, i.e., metadata, about transactions to route updates and queries to the appropriate nodes. Transaction's metadata essentially capture the potential effects of a transaction: the transaction type (update or read-only), the tables accessed (read or written) by the transaction, the number of tuples potentially involved (inserted/updated/deleted) by the transactions. Potential effects of a transaction represent all the effects that the transaction can produce. Thus, the actual effects of a transaction when executed on a node are always a subset of the potential effects.

Let T be a transaction, $Write_T$ is the write set of relations potentially updated by T and $Read_T$ is the read set of relations potentially read by T. $Write_T$ and $Read_T$ may be easily obtained by parsing the transaction code. They are defined at the relation level for simplicity. Defining the write set and read set at the tuple level is not practical in our ASP context where applications typically call stored procedures with parameter values so we cannot know in advance which tuples will be updated.

A transaction's changes typically involve relations. Thus, we define the quantity of change at the relation level. Let R_i be a relation, and T an update transaction which modifies R_i. We denote by $Change(T, R_i)$ the maximum number of tuples which T may modify in R_i. $Change(T, R_i)$ is an upper-bound of the number of tuples that T may update in R_i. It may be obtained in several ways: by parsing the transaction code; using statistics about R_i in the database catalog; or by sniffing the log after T has been executed on its initial node. If T is a single statement transaction, the number of modified tuples can also be obtained as the transaction return value. For simplicity, we assume that this upper bound is known by the application and stored as metadata. We define the changes of T as

$$Change(T) = \{(R_i, Change(T, R_i)) | R_i \in Write_T\}.$$

Obviously, over-estimating $Change(T, R_i)$ may lead to process useless refreshment, since the system would assume that the nodes on which T has not been propagated are staler than they actually are. Useless refreshment may affect the performance of queries that tolerate "medium" staleness with respect to T's change. Queries that tolerate very small staleness are not affected by inaccuracies in $Change(T, R_i)$, since they require nodes to be refreshed anyway. Queries that tolerate very high staleness are not affected since they do not require nodes to be refreshed. If the application is not able to supply good estimates of the upper bound, then the only viable solution is to get the value of $Change(T, R_i)$ on-line, through log-sniffing or using triggers.

3.2. Replication model

Our replication model specifies the way databases are replicated in our cluster system and how we handle transactions. We assume a single database composed of relations $R_1, R_2, \ldots R_n$ that is fully replicated at nodes $N^1, N^2, \ldots, N^m$. The local copy of R_i at node N^j is denoted by R_i^j and is managed by the local DBMS, without any specific requirement. We use a lazy multi-master (or update everywhere) replication scheme. Each node can be updated by any incoming transaction and is called the *initial node* of the transaction. Other nodes are later refreshed by propagating the transaction through refresh transactions.

We distinguish between three kinds of transactions:

- *Update transactions* are composed of one or several SQL statements which update the

database. Update transactions can be seen as stored procedures, so the metadata associated with the transaction (see Section 3.1) can be known when the transaction's execution is started.

- *Refresh transactions* are used to propagate update transactions to the other nodes for refreshment. A refresh transaction can be made by either replaying the initial transaction or by propagating its effects to the database as a sequence of write operations. The former solution is straightforward but may incur redundant computation. The latter may be more efficient in some cases (e.g., many tuples read, few tuples updated) but requires log sniffing at the initial node for extracting write operations. Sniffing the log of a black-box DBMS is doable but quite complex. Furthermore, the performance gain may be quickly offset by the cost of sniffing the log, typically organized as a sequential file. Thus, in this paper, we choose the former solution, i.e., refresh transactions replay the initial transaction.
- *Queries* are read-only transactions, and thus need not be refreshed.

Let us note that, because we assume a single replicated database, we do not need to deal with distributed transactions, i.e., each incoming transaction can be entirely executed at a single node.

In a multi-master replicated database, the mutual consistency of the database can be compromised by conflicting transactions executing at different master nodes. A solution that preserves mutual consistency is to enforce one-copy serializability [8]. In lazy multi-master replication, one-copy serializability can be obtained by ensuring that conflicting transactions are executed at each node in the same relative order [9,10]. In this paper, we use a weaker form of one-copy serializability, which makes a distinction between update transactions and queries. Update transactions are executed at database nodes in compatible orders, thus producing mutually consistent states on all database replicas. Queries are sent to any node that is fresh enough with respect to the query requirement. This implies that a query can read different database states according to the node it is sent to. However, since queries are not distributed, they always read a consistent (though stale) state. To achieve this, we maintain a graph, called *global precedence order graph*, which keeps track of the *conflict dependencies* among active transactions, i.e., the transactions currently running in the system but not yet committed. It is based on the notion of *potential conflict*: an incoming transaction potentially conflicts with a running transaction if they potentially access at least one relation in common, and at least one of the transactions performs a write on that relation. We define global precedence and global precedence order graph as follows.

Definition 1 (*global precedence*). Let T and T' be two active transactions, we say that T' *globally precedes* T, denoted by $T \prec T'$, if

- T and T' potentially conflict, i.e.,

$$Write_T \cap (Write_{T'} \cup Read_{T'}) \neq \varnothing \quad \text{or} \quad Write_{T'} \cap (Write_T \cup Read_T) \neq \varnothing, \text{ and}$$

- When T arrives in the system, T' is already running at least on one node.

Definition 2 (*global precedence order graph*). The global precedence order graph is a couple $(T, \prec)$ where

- T is the set of active transactions (each node in the graph represents an active transaction), and
- $\prec$ is the *global precedence order* among transactions (each edge in the graph represents a global precedence between the two related transactions).

Note that, following Definition 1, the global precedence order graph is acyclic. We now define refresh sequences, which are built according to this global order, in order to propagate transactions to a given node.

Definition 3 (*refresh sequence*). Let T be a transaction and N a node. $S = \{T_1, T_2, T_3, \ldots, T_k\}$ is a refresh sequence for T on N if:

- $\forall i \in (1,2,\ldots,k)$, T_i has not yet been executed on N, and,
- $\forall i, \forall j \in (1,2,\ldots,k)$, if $T_i \prec T_j$ then $i > j$ (the sequence order is compatible with the global order), and,
- If T is an update transaction, then $\forall i \in (1,2,\ldots,k)$ $(T_i \prec T)$.

Refresh sequences are executed sequentially. This ensures that, whenever a transaction is executed on a node, all the preceding transactions have already

been executed on that node. Since transactions are not distributed, a transaction always reads a consistent, possibly stale, state of the database. As the global order is not a total order, transactions can be executed in different orders, each of them compatible with the global order, on different nodes. This gives more flexibility for load balancing. For instance, assume that transaction T_1, then T_2, then T_3 arrives in a cluster composed of two nodes, N_1 and N_2. T_3 potentially conflicts with both T_1 and T_2, T_2 does not conflict with T_1. T_1 is sent to node N_1, then, as T_2 does not conflict with T_1, it can be sent to node N_2. The global order is still empty. Then, when T_3 arrives, as it potentially conflicts with both T_1 and T_2 and both T_1 and T_2 are already running, the global order now contains $T_3 \prec T_1$ and $T_3 \prec T_2$. Thus, T_3 can been executed on N_1 after the refresh sequence (T_2). It can also be executed on N_2 after the refresh sequence (T_1). Note that the sequential orders on N_1 (T_1; T_2; T_3) and on N_2 (T_2; T_1; T_3) are different but both compatible with the global order.

3.3. Freshness model

Our freshness model defines the concept of replica's staleness. Based on this definition, we can define node refreshment, or synchronization, and transaction execution plans that are freshness-aware.

With our replication scheme, several replicas at different cluster nodes can have different states because they have not yet reached the latest consistent database state,[2] i.e., the state obtained after correct execution of all transactions received. Furthermore, transactions (updates or queries) submitted to the cluster system may have specific freshness requirements. Intuitively, the freshness of a replica captures the quantity of changes (made to other replicas) which have not yet been applied to it. This quantity of changes is referred to as import-limit in epsilon transactions [11]. If the quantity of changes is zero, we say that the replica has maximum freshness, i.e., has the latest consistent state. However, freshness is not a concept easy to use, since its value is not defined for perfectly fresh database states. Thus, we use the opposite concept of *staleness*, which is always defined and is equal to 0 for perfectly fresh database states. The *staleness of a* relation *replica* R_i^j can then be captured by the quantity of change which has been made to the other replicas of R_i but R_i^j. Let TR_i^j be the set of transactions which have modified at least one of these other replicas:

$$TR_i^j = \{T_k | \text{Change}(T_k, R_i) > 0 \text{ and } \exists N^1, T_k \text{ has been executed on } N^1 \text{ and } T_k \text{ has not been executed on } N^j\}.$$

Definition 4 (*staleness of a relation replica*). The *staleness of a* relation *replica* R_i^j is defined as the sum of all the changes made by TR_i^j, i.e.,

$$Staleness(R_i^j) = \sum\nolimits_k (Change(T_k, R_i) | Tk \in TR_i^j).$$

Thus, we can define the *staleness of a cluster node* N^j as the following vector:

$$Staleness(N^j) = \{(R_i^j, Staleness(R_i^j)) | i = 1, n\}.$$

The tolerated staleness of a transaction expresses the maximum staleness that the transaction accepts when reading data on a node. It is based on the tolerated staleness of all the relations accessed by the transaction.

Definition 5 (*tolerated staleness of a transaction*). We define the tolerated staleness of T as

$$ToleratedStaleness(T) = \{(R_i, toleratedStaleness(R_i, T) | R_i \in Read_T\}.$$

where $toleratedStaleness(R_i, T)$ is a user-defined positive value, which expresses the maximal tolerated staleness required by the user for her transaction T, when reading relation R_i. If T is an update transaction, this value is always equal to 0.

In other words, update transactions do not tolerate staleness, in order to ensure that conflicting transactions are executed at each node in the same relative order. Defining a tolerated staleness for update transactions is not necessary, since always equal to 0. However, it allows treating update transactions and queries homogeneously, and simplifies the definitions of the routing and refreshment algorithms.

The tolerated staleness associated with a transaction allows sending a transaction to a node even if it is not perfectly fresh. However, the node must be fresh enough with respect to the transaction requirements. For an update transaction, this means that all the transactions which precede it must be

[2] A replica can be in a consistent state which is not necessarily the latest state of some other replicas.

executed on the node before it starts. For a query, this means that the node must have received enough refresh transactions so that its data is close enough to the data at the nodes where the transactions have already been executed. To compute how many and which refresh transactions must be sent to a node to make it fresh enough for a query, we define the *Synch* property. This property states that a refresh sequence S is sufficient to make a node N^j fresh enough *w.r.t.* the tolerated staleness of a transaction T. The *MinSynch* property states that S is not only sufficient to refresh N^j, but also does not contain any unnecessary refreshment, and thus, minimizes the amount of work for synchronization.

Definition 6 (*minimum synchronizing sequence*). Let S be a refresh sequence of T on N^j, we say that S synchronizes N^j for T, denoted by $Synch(T, N^j, S)$, if, after executing all the transactions in S at N^j, N^j's staleness is less than or equal to the tolerated staleness of Q, i.e.,

$$Synch(T, N^j, S) = true \text{ if } \forall R_i \in Read_T, \sum_{T_l \in TR_i^j - S} Change(T_l, R_i) \leqslant toleratedStaleness(R_i, T) = false \text{ otherwise.}$$

We say that S is a *minimum synchronizing sequence for T on* N^j, denoted by $MinSynch(T, N^j, S)$, if S synchronizes N^j for T and no transaction is useless in S, i.e.,

$$MinSynch(T, N^j, S) = true \text{ if } Synch(T, N^j, S) \text{ and } \forall T_k \in S,\ (Synch(T, N^j, S - \{T_k\})) = false \text{ otherwise.}$$

Based on the definition of *MinSynch*, we can now define the concept of transaction execution plan needed to specify where and how to execute a transaction in the cluster. Given a transaction T, its *transaction execution plan* (*TEP*) specifies the node N^j to process T and the minimal refreshment to perform in order to reach the database state required by the user as *toleratedStaleness* (R_i, T).

Definition 7 (*transaction execution plan*). Let T a transaction to execute, we define a *transaction execution plan for T*, denoted $TEP(T)$, as

$$TEP(T) = (N^j, S) \text{ such that } MinSynch(T, N^j, S).$$

Note that there is no reason for S to be unique. S is minimal with respect to set inclusion: if any transaction is dropped from S, then $MinSynch(T, N^j, S) = false$. This does not imply that S is minimal with respect to the number of transactions it contains, or with respect to the number of tuples changed by the transactions it contains. In Section 5, we will describe how to compute a possible S.

4. Database cluster architecture

In this section, we introduce the architecture for processing user requests coming, for instance, from the Internet, into our cluster system and discuss our solution for placing applications, DBMS and databases in the system. Then, we describe in detail the architecture of the transaction router which is the main focus of this paper.

4.1. Cluster system architecture

The general processing of a user request is as follows. First, the request is authenticated and authorized using a shared directory which captures information about users and applications. The directory is also used to route requests to nodes. If successful, the user gets a connection to the application (possibly after instantiation) at some node which can then connect to a DBMS at some, possibly different, node and issue transactions for retrieving and updating database data.

We consider a cluster system with similar nodes, each having one or more processors, main memory (RAM) and disk. We assume applications typically written in a programming language like C++ or Java making DBMS calls to stored procedures using a standard interface like ODBC or JDBC. In this paper, we use a lazy multi-master replication scheme, which is the most general as it provides for both application and database access parallelism. Based on these choices, we propose the cluster system architecture in Fig. 2. Applications, databases and DBMS can be replicated at different nodes without any change. Besides the directory, we add three new modules which can be implemented at any node. The *application load balancer* simply routes user requests to the application node that has the lowest load. The *transaction router* intercepts DBMS procedure calls (in JDBC) from the applications, and for each one, generates a TEP, based

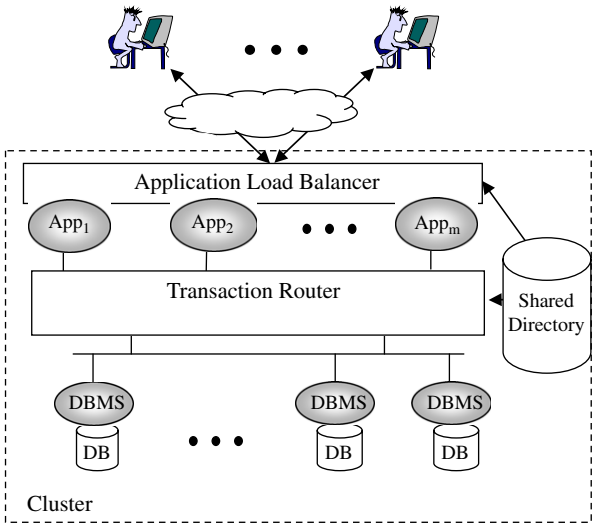

Fig. 2. Leganet cluster system architecture.

on application and user freshness requirements obtained from the shared directory. Finally, it triggers transaction execution (to execute stored procedures) at the best nodes, using run-time information on nodes' load. When necessary, it also triggers refresh transactions in order to make some nodes fresher for executing subsequent transactions (update or queries). There may be several instances of the transaction router, in order to scale up with the number of DBMS nodes, each one accessing the shared directory.

4.2. Router architecture

Typically, an application is a program that connects to a DBMS to process a transaction. During execution, the application may process several transactions but it is not aware of transaction load balancing opportunities offered by the cluster. For that reason, we design a router in charge of transaction load balancing. The general processing of a transaction is as follows. First, the transaction executes at a cluster node. When the transaction calls a DBMS, the call is trapped to be redirected to the router. The router chooses a DBMS node to process the transaction, and sends the result back to the application. The router seamlessly integrates with existing applications and their DBMS, to take advantage of the cluster computing power and parallelism.

The router architecture is designed for preserving applications and databases' autonomy. To preserve database autonomy, the cluster system exploits a shared-nothing architecture among DBMS nodes, so that each DBMS node accesses data on separate disks. Our router architecture is motivated by the following requirements:

(a) The router must be aware of which node is able to process a transaction. Thus, it must know if a node is fresh enough to meet the transaction requirements. In case the node is not fresh enough, the router must know which refresh transactions must be sent to make it fresh enough.
(b) If several nodes can process a transaction, the router should choose the one that yields the most efficient execution.
(c) Because of replication, the router must propagate deferred refresh transactions to replicas. This synchronization work should interfere as little as possible with the actual transaction processing, to reduce overhead.

The router architecture is shown in Fig. 3, with its two main modules: TEP generation module and synchronization module.

In the current Leganet system, the router is centralized (running at one node). However, a centralized router can obviously be a single point of failure. And for very large cluster configurations, it may also become a performance bottleneck. The last problem can be simply solved by using a powerful router node, e.g., a 4-processor node. However, the first problem is more involved and

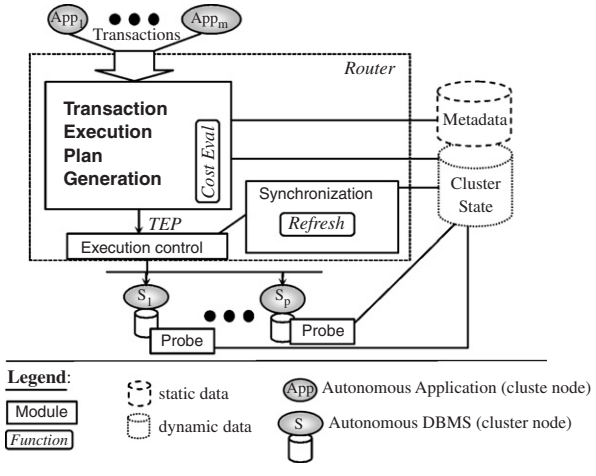

Fig. 3. Router architecture.

suggests replicating the router at two or more nodes. To enable each router node to perform transaction routing in parallel, the directory must thus be available at each node. The directory static metadata (e.g., tolerated staleness) could be simply replicated and periodically refreshed. However, the directory dynamic metadata regarding the cluster state (e.g., nodes' load, transactions to be propagated, etc.) may change frequently and need to be up-to-date at each router node. Thus, lazy replication is not appropriate for such dynamic metadata. Instead, we could use a more efficient, eager replication technique tailored to cluster systems to maintain strong consistency of such data, e.g., [12]. Another solution would be to have the directory virtually shared by all router nodes using distributed shared memory (DSM) software [13]. By providing support for shared data structures, DSM software simplifies parallel programming and can provide good scalability. However, a more detailed study of these two solutions is beyond the scope of this paper and subject to future work.

4.3. TEP generation module

Upon receiving a transaction T, the TEP generation module first accesses the metadata stored in the shared directory. If T is a query it finds the maximal staleness tolerated for processing T, assuming that high staleness will yield more parallel access to replicas, thus increasing overall performance. If T is an update transaction, it computes the effect of processing T at a node, i.e., computes the $Change(T)$ vector defined in Section 3.1 and uses it to compute the replica's staleness. To perform load balancing, the TEP generation module chooses a replica (i.e., a node) to process a transaction with high efficiency. It considers all the nodes as potential candidates. If a candidate node is too stale, it computes the set of transactions to propagate to that node in order to meet the transaction freshness requirements. To choose an efficient node, the TEP generation module uses a cost model to estimate transaction processing time. The cost model takes as input the cluster state which describes the current cluster load including which transaction is at which node and the freshness of each node, as well as the node's load obtained through load probes.

We consider each incoming transaction individually, with the objective of minimizing its response time. A possible optimization would be to globally consider all the transactions waiting for execution and reorder them in the incoming queue (scheduling). However, this is much more complex and not always better, since the gain on response time can be offset by the overhead needed to reorder the incoming queue. Thus, we do not consider this optimization.

4.4. Synchronization module

The synchronization module finds the appropriate timings to propagate transactions to other replicas. To avoid useless synchronization, an obvious solution is to lazily propagate as late as possible, only when needed by a transaction. But late synchronization is not always optimal as it may increase transaction latency. For instance, consider a period of incoming transactions which do not require any freshness. During that period, the nodes' freshness decreases continuously. Then, if the next incoming transaction requires some freshness, the amount of synchronization to perform can be high and may slow down the transaction. Forecasting that such a transaction will eventually occur gives the opportunity to prepare the cluster by starting synchronization work earlier. Thus, the synchronization module propagates a transaction at a node depending on two conditions: (i) if the node's staleness is above a given limit (late synchronization) or (ii) if the node's load is below a given limit (early synchronization). The values of the node's staleness and node load's limit depend on the application and workload.

5. Transaction routing with freshness control

In this section, we describe how transactions are routed in order to improve performance. First, we present the routing and refresh strategies. Then, we introduce the cost function used by these strategies.

5.1. Routing algorithms

We propose two routing strategies, each well-suited to different application needs. The first strategy (Section 5.1.1) makes no assumption about the workload. The second strategy (Section 5.1.2) favors update transactions to deal with OLTP workloads. Both strategies are cost-based and use late synchronization, thus they take into account the cost of refreshing a node before sending it a transaction. For a fixed workload, the routing complexity is $O(m) \times O(p)$, where m is the number

of nodes and p the number of active transactions in the system. $O(p)$ corresponds to computing the minimum refresh sequence to make a node fresh enough for an incoming transaction. More precisely, it is always less than p, and thus remains always small. This complexity makes our approach scalable. Finally, we show how early synchronization could be integrated within the routing process.

5.1.1. Cost-based only strategy

The cost-based only strategy (CB) simply evaluates, for each node, the cost of refreshing the node enough (if necessary) to meet the transaction freshness requirements as well as the cost of executing the transaction itself. Then it chooses the node which minimizes the cost. This strategy is described in Fig. 4 by algorithm CBroute.

Algorithm CBroute iterates over all the nodes in *NSet* to find out the node that minimizes the cost of executing T, taking into account the necessary synchronization to make the node fresh enough with respect to the tolerated staleness of T. Function *computeMinSynch* returns a minimal sequence of refresh transactions sufficient to make N^i fresh enough for T, i.e., a sequence of transactions S such that $MinSynch(T, N^i, S)$. It iterates over the refresh transactions waiting for being propagated to N^i and decrements the node staleness according to the refresh transaction changes. It stops when the node staleness satisfies T's tolerated staleness (always equal to 0 for update transactions). In order to ensure global consistency, refresh transactions are inserted in the refresh sequence according to the global serialization order: whenever a refresh transaction is inserted, all its predecessors not yet executed on the node are also inserted, in the appropriate order, so that the sequence order is compatible with the global precedence order (see Section 3.2). Function *costEval* evaluates the cost of executing the transaction execution plan $TEP(T) = (S, N^i)$. This cost function is detailed in Section 5.2. Finally, function $executeTEP(T,S, N^k)$ executes the TEP which minimizes the cost, i.e., it first executes all the transactions in S on N^k, and then, if T is an update transaction, T is inserted in the global precedence order graph as a child of all the transactions in S with no child. Finally, it executes T itself on N^k. It also updates the cluster state: all the transactions in S are dropped from the set of transactions waiting to be refreshed on N^k. When a transaction has been executed on all the nodes, it is removed from the global precedence order graph.

5.1.2. Cost-based with bounded response time strategy

The CB strategy works well for applications where there is no difference between update transactions and queries. However, most applications in the ASP context are OLTP-oriented: update transactions represent front-office procedures (e.g., a drug sale in a pharmacy) which must be executed as fast as possible while queries represent back-office procedures (e.g., computing statistics for marketing purposes). Thus, the main requirement for such applications is to guarantee that the response time of update transactions never increases beyond an acceptable limit. Our main objective is first to ensure that queries do not slow down the transaction throughput. Once we reach this objective, the secondary objective is to reduce the response time of these queries. Therefore, the optimization objective is to: *ensure that update transactions' response time is below a given limit, and minimize query response time without interfering with running update transactions*. To this end, we propose a second strategy with bounded response time (BRT) which dynamically separates the nodes responsible for update transaction processing from

```
CBroute(T: transaction, NSet: set of cluster nodes) {
      minCost = ∞
      foreach Ni ∈ NSet
            S' = computeMinSynch(T, Ni)
            cost = costEval(T, S', Ni )
            if cost < minCost
                  minCost = cost
                  k = i
                  S = S'
      executeTEP(T,S, Nk)
      }
```

Fig. 4. Algorithm for cost-based only (CB) strategy.

the nodes responsible for query processing. Because the number of concurrent incoming transactions is not constant over time, the algorithm dynamically assigns a node either for update transaction or query processing. Let NS be the set of all cluster nodes, NS_T be the set of nodes dedicated to update transactions, NS_Q be the set of nodes dedicated to queries (with $NS = NS_Q \cup NS_T$ and $NS_Q \cap NS_T = \varnothing$), and $T_{\max}$ be the maximum allowed response time for an update transaction. NS_Q, NS_T and $T_{\max}$ are global variables stored in the shared directory. To ensure suitable usage of cluster nodes, our algorithm dynamically dedicates the minimal set of nodes to update transactions, the remaining nodes being available for long running queries. Whenever no node in NS_T is able to handle an update transaction within $T_{\max}$, the nodes in NS_Q are considered. If one of them can execute T within $T_{\max}$, it is removed from NS_Q and added to NS_T. If no node is able to execute T within $T_{\max}$, the node which minimizes the cost is chosen. Nodes are removed from NS_T and added to NS_Q when, after executing an update transaction, they become idle, i.e., their current load is below a given limit. Queries are treated by the cost-based algorithm CBroute in Fig. 4, over the nodes assigned to queries.

As algorithm BRTroute favors update transactions, it may happen that, temporarily, no node is assigned for queries, i.e., $NS_Q = \varnothing$, until a node becomes idle and added to NS_Q. In this case, an error is raised and the query execution is delayed (Fig. 5).

5.1.3. Early synchronization

Early synchronization is important to avoid increasing the response time of some transactions because of nodes being too stale. When the router receives an incoming transaction T, it chooses a node. Then, if the node requires refreshment, the router first sends the required refresh transactions to the node before sending T. Therefore, the refreshing time may greatly increase the overall transaction response time. To avoid this situation, the router may propagate transactions in advance to prepare some nodes for subsequent incoming transactions (early synchronization). To minimize the overhead, it operates only on idle nodes, i.e., nodes with a load under a given threshold, using algorithm RefreshIdleNode in Fig. 6. In order to use the *computeMinSynch*() function, we use a virtual transaction T_{all} which potentially conflicts with all the possible transactions. This ensures that the refresh sequence S will bring all the relations to a perfect freshness. As we focus in this paper on late synchronization, we will leave experimentation with early synchronization as future work.

```
BRTroute(T) {
      minCost = ∞, routed = false
      if T is a query
            if NS_Q = ∅
                  raise_error("No node available for queries, retry later")
            else CBroute(T, NS_Q)
      else
            foreach N^i ∈ NS_T
                  if not routed
                        S = computeMinSynch(T, N^i)
                        cost = costEval(T, S, N^i)
                        if cost < T_max
                              executeTEP(T, S, N^i)
                              routed = true
                  else if load(N^i) = 0
                        NS_Q = NS_Q ∪{N^i}, NS_T = NS_T -{N^i}
            if not routed
                  foreach N^i ∈ NS_Q
                        S' = computeMinSynch(T, N^i)
                        cost = costEval(T, S', N^i)
                        if cost < minCost
                              minCost = cost
                              k = i
                              S = S'
                  NS_Q = NS_Q - {N^k}, NS_T = NS_T ∪{N^k}
                  executeTEP(T, S, N^k)
}
```

Fig. 5. Algorithm for bounded response time (BRT) strategy.

```
RefreshIdleNode {
      Whenever load(N^k) < threshold
            S = computeMinSynch(T_all, N^k)
            execute_tep(∅, S, N^k)
}
```

Fig. 6. Algorithm for early synchronization.

5.2. Cost function

In this section, we show how we compute the cost of a TEP, which corresponds to the *costEval* function call in the algorithms of Section 5.1. The cost of a TEP is that of processing the transaction at a node including the necessary refreshment. In our context, because of the application autonomy requirement, we do not know the details about the read and write operations of a transaction. Therefore, we assume that system resource consumption (CPU, I/O) is uniformly distributed during the transaction processing time. Let *load*(N) represent the DBMS load at node N. Such information is based on operating system load whose value increases with the number of concurrent transactions and aggregates CPU and I/O load (in our implementation, it is obtained by the load_average() Linux probe). *load*(N) is always greater than 1 and is always close to the number of concurrent transactions running at N. However, the current node's load is not sufficient for load balancing, since we must also estimate the load of executing a TEP in order to estimate the response time of a transaction. Let *avgTime*(T, N) be the average time of processing T at N. *avgTime* is a moving average based on previous executions of T on N. It is initialized by a default value obtained by running T on an unloaded node. Let Δt be the elapsed time for T at N, we define *rt*(T, N), the remaining time for T at N as

$$rt(T,N) = avgTime(T,N) - \frac{\Delta t}{load(N)}.$$

Then, given RT_N the set of running transactions at node N, we define *procTime*(T, N) based on node load and elapsed time of running transactions at N as

$$procTime(T,N) = avgTime(T,N) + \sum_{T_i \in RT_N} \min(avgTime(T,N), rt(T_i,N)).$$

The time interval, during which T is running concurrently with another transaction of RT_N, is the minimum among the execution time of T and the remaining execution time of the concurrent transaction. Let us now define the function *costEval*(T,S,N) to process a TEP, taking into account the time to process T at N, along with its necessary synchronization S:

$$costEval(T,S,N) = procTime(T,N) + synchTime(S,N),$$

where

$$synchTime(S,N) = \sum_{S_i \in S} procTime(S_i,N).$$

6. Implementation

We implemented the Leganet prototype on an 11-node Linux cluster running Oracle8i. However, we use standards like LDAP and JDBC, so the main part of our prototype is independent of the target environment. In this section, we briefly describe our current implementation.

6.1. Transaction router

The transaction router is implemented in Java. It acts as a JDBC server for the application, preserving the application autonomy through the JDBC standard interface. Inter-process communication between the application and the load balancer uses RmiJdbc open source software.[3] To reduce contention, the router takes advantage of the multi-threading capabilities of Java based on Linux's native threads. For each incoming transaction, the router delegates TEP generation and execution to a distinct thread. The router sends transactions for execution to DBMS nodes through JDBC drivers provided by the DBMS vendors. To reduce latency when executing transactions, the router maintains a pool of connections to all cluster nodes.

6.2. Leganet GUI

The user interface has a Web-based architecture shown in Fig. 7. A Web server controls the Leganet services: transaction router, node probes, and application workload. We benefit from the browser capabilities to display dynamic graphical content. The prototype activity can be visualized in two complementary ways. First, we can visualize graphically (through SQL queries + JFreeChart) a trace database produced by the prototype during execution. Second, we can follow the prototype activity through a monitoring panel that is dynamically updated by run-time events generated by the

[3] www.objectweb.org/rmijdbc.

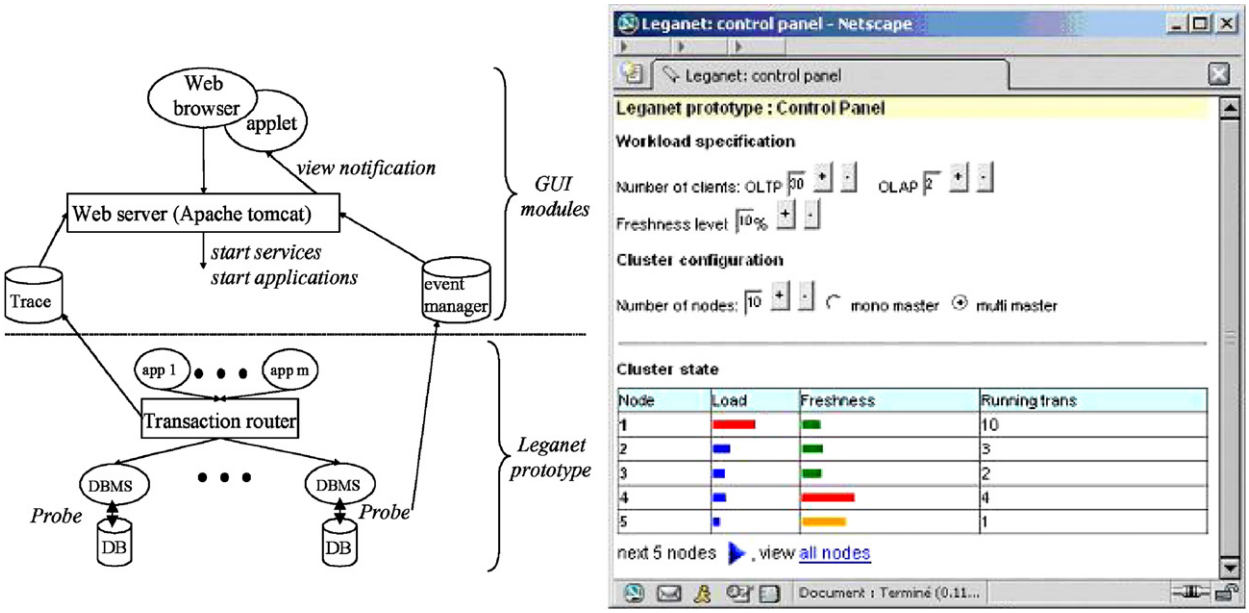

Fig. 7. GUI architecture and control panel.

prototype (using DHTML functionalities and Javascript for applet/browser communication).

Using the control panel, we can select a predefined scenario or specify scenario characteristics. Then, we run the scenario and visualize the evolving node load, the number of transactions per node and the node freshness. After execution, we visualize various reports and charts that are dynamically generated, e.g., the transactions response time versus the freshness level, or the transaction response time at each node.

7. Performance evaluation

In this section, we evaluate the performance of our approach through experimentation and emulation. We first describe the experimental environment for transaction routing. Then, we compare the efficiency of our cost function with that of two baseline cost functions. Then, we study the performance of our two cost-based routing strategies. Then, we evaluate the performance speed up when increasing the number of cluster nodes. Finally, we analyze the impact of tolerating staleness on the system throughput.

7.1. Experimental environment

We have implemented all the router modules in Java. The router acts as a JDBC server for the application, preserving the application autonomy through the JDBC standard interface. Inter-process communication between the application and the router uses the RMI standard. The cluster has 11 nodes (Pentium IV 2 Ghz, 512 Mb RAM) connected by a 1 Gb/s network. To ease experimentation, we use one node as router node and all other 10 nodes as database nodes, each hosting a database replica. Given the relatively small number of database nodes, one router node is sufficient. All database nodes run the Oracle 8i DBMS under Linux. We have implemented the *load*(*node*) function (see Section 5.2), used for cost estimation, by calling the OS level *load_average* function which returns the mean load (CPU and I/O) observed during the last minute. We do not use the instantaneous load since it is too instable and differs too much from the average load in the time interval (a few seconds in our context) of processing a transaction.

To measure the router performance for typical OLTP applications, we have implemented the main parts of the TPC-C benchmark [7]. The TPC-C database (2 Gb of data within 9 relations, the *Stock* relation has 1 million items and the *OrderLine* relation has 3 million tuples) is replicated at each DBMS node. TPC-C transactions are implemented as stored procedures. In the following experiments, we consider the *new-order* update transaction, and the *order-status* query. We developed a workload generator as a set of application terminals sending

transactions simultaneously. During an experiment, the number of application terminals remains constant. A terminal sends either front-office transactions (*new-order*) or back-office queries (*order-status*). The standalone response, which is the time to process a transaction alone at a node, is 0.32 s for transactions and 10 s for queries.

We divide transactions into distinct classes. Within the same class, transactions interfere and access the same data subsets of relations *Order* and *OrderLine*. However, transactions from two different classes do not interfere and access disjoint subsets of relations *Order* and *OrderLine*. Each terminal delivers transactions randomly chosen among all classes. To exploit parallel transaction execution, the number of classes is set to twice the number of cluster nodes.

We use the following setup. First, except for the experiments of Section 7.5 which investigate the impact of freshness control on performance, both update transactions and queries do not tolerate any staleness. Second, the workload is high enough to keep the system fully busy during the experiments. The workload is composed of t terminals. Each terminal sends a sequence of transactions or queries interleaved with a randomly distributed think time (with a mean of half the transaction standalone time). In order to measure the maximum throughput that the router can support, we choose t such that all the cluster resources are fully used during the entire experiment, i.e., increasing t would not increase throughput since the system is fully loaded. As a consequence, there is duality between the results obtained in response time and in throughput. Since the number of terminals remains constant during an experiment, the response time is always inversely proportional to the throughput. Thus, in the following, we only analyze the results related to throughput. Results related to response time are only given as a confirmation and lead to the same observation and conclusions.

7.2. Comparison of costEval function with baseline cost functions

In this section, we compare our *costEval* function with two baseline cost functions:

- $basicLoad = \sum_{T_i \in RT_N} rt(T_i, N)$ which represents the estimated cost of processing the outstanding transaction/query load at each node N (see Section 5.2).
- $basicSynch = SynchTime(T,S,N)$ which represents the cost of synchronizing node N for transaction T through refresh sequence S (see Section 5.2).

The idea behind including the node's freshness in the cost function is that we can achieve better performance by reducing the total amount of synchronization performed during an experiment. Thus, routing a transaction to the freshest node (i.e., with the least work to synchronize) would minimize the total amount of work. But it may increase node staleness, thus yielding under utilization of the cluster, and low performance, which justifies including the node's load as well.

In order to experimentally prove this assumption, we investigate whether choosing either the freshest node or the least loaded node would be as efficient as our *costEval* function. We run the router with the *CBroute* algorithm (see Section 5.1.1) which calls *costEval, basicLoad* or *basicSynch*, respectively, and compare the results. We set up a cluster with 10 nodes. The workload is composed of 20 transaction terminals. The number of query terminals is varied from 0 to 100. A terminal accesses any part of the database with the same probability (the transaction class is randomly and uniformly assigned). Thus, the number of terminals accessing the same part at the same time (concurrency rate) is 5%. Queries do not tolerate staleness. We run three experiments, one for each cost function. We report the average transaction response time in Fig. 8 (Fig. 8b is a focus of 8a).

The first observation is that *costEval* is the best overall cost function, whatever the number of query terminals. For very light workloads (1 or 2 terminals), *costEval* performs as *basicSynch*, due to the fact that nodes are lightly loaded. For higher workloads, *basicSynch* is outperformed by the two other functions. This is because *basicSynch* does not take into account the nodes (high) load: when a node is too stale, it is never selected and thus remains idle. On the contrary, for heavy workloads (more than 50 query terminals), *costEval* performs as *basicLoad*. This is due to the fact that, when the load is high, nodes are always working and thus, they are kept almost perfectly fresh by synchronization. Hence, the cost of synchronization is negligible with respect to the node's load. Between 2 and 50 terminals, *costEval* is much better than the two other cost functions, since it combines the advantages of both.

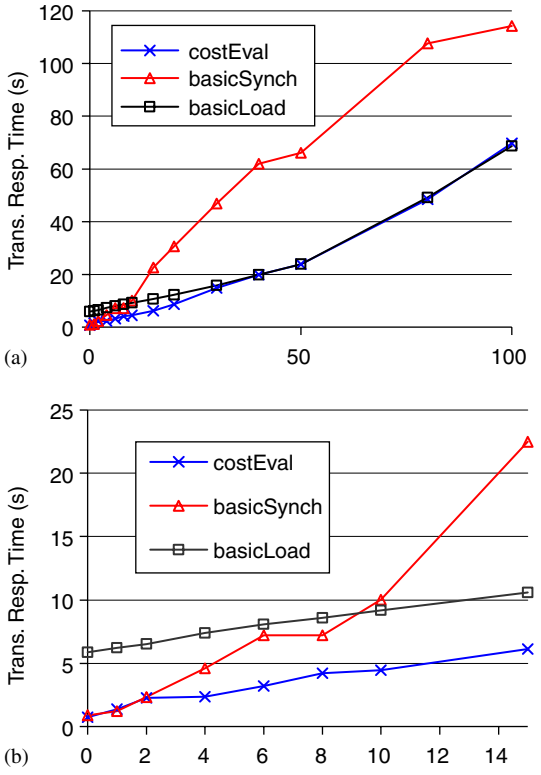

Fig. 8. Transaction response time vs. number of query terminals: (a) 0 to 100 query terminals; (b) 0 to 14 query terminals.

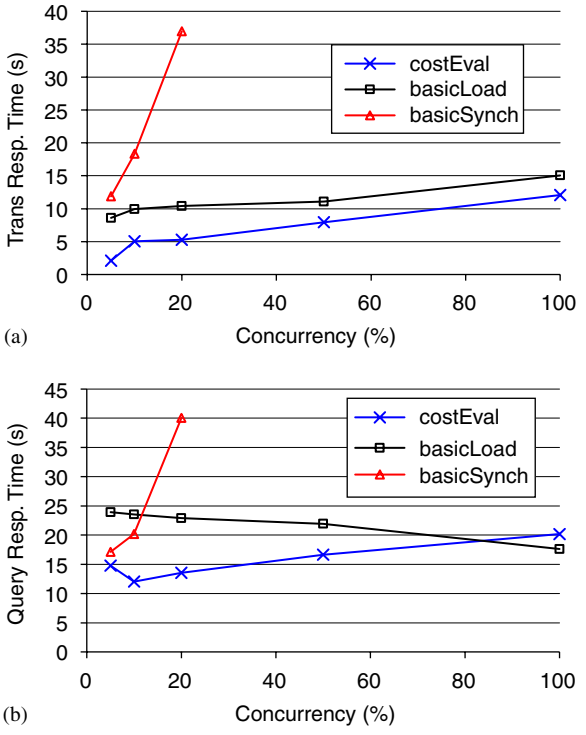

Fig. 9. Transaction and query response time vs. data access skew.

In the next experiment, we evaluate the impact of the concurrency rate on the routing performance. The number of terminals is fixed. We slightly modify the workload to increase concurrent data access: concurrency rate is varied from 5% (i.e., 2 out of 40 terminals) to 100%. We report update transaction and query response times in Fig. 9.

Fig. 9 shows that *basicSynch* is not adapted for highly concurrent workloads. This is not surprising since with a higher concurrency rate, the probability that a node is never selected is higher. We also see that *costEval* outperforms *basicLoad* by a factor of 4 for low concurrent data access. The benefit is slowly decreasing but still remains above 20% for the update transaction response time in case of a fully concurrent workload. Concerning the query response time, *costEval* is outperformed by *basicLoad* when the concurrency rate is extremely high. However, this case is not realistic with a fine-grained conflict detection. In fact, workloads with a concurrency rate higher than 50% are extremely rare. Under this value, *costEval* always outperforms *basicLoad* by more than 30%. This illustrates the advantage of using a cost function that takes into account node load as well as synchronization to run highly skewed workloads.

7.3. Comparison of routing algorithms

In this section, we compare the performance of the two routing strategies proposed in Section 5. The CB only strategy routes every incoming transaction to the node that minimizes the *costEval* function. The cost based with BRT strategy dynamically dedicates a subset of nodes to ensure an upper bound ($T_{\max}$) for transaction response time, and routes queries to remaining nodes that minimize the *costEval* function. The goal is to study cluster resource allocation, among queries and transactions, depending on the routing strategy and to compare their respective performance in terms of update transaction and query throughput.

The workload is composed of 20 transaction terminals and 20 query terminals. With the CB strategy, we get an average response time of 3.76 s for update transactions and 240 s for queries. We measure the corresponding throughputs of 303 transactions per minute (TPM) and 4.81 queries per minute (QPM), respectively. We now compare

this result with the performance of the BRT strategy when T_{max} varies from 0.67 to 6.67 s. Since BRT dynamically dedicates nodes to either update transactions or queries, we first observe the number of nodes dynamically dedicated to update transactions. As T_{max} increases, update transactions need fewer resources to be executed within T_{max} and the number of dedicated nodes decreases. Thus, the number of nodes available for queries increases. With T_{max} under 0.67 s, all the 10 database nodes are needed to perform update transactions within the time limit. With T_{max} over 6.67 s, one node is sufficient. Table 1 shows, for each possible number of nodes *NbNodes*, the minimal value of T_{max} leading to dedicating *NbNodes* nodes to update transactions (i.e., T_{max} such that $NbNodes = |NS_T|$ of Section 5.1.2).

Fig. 10 shows the response time and throughput for update transactions and queries with the BRT strategy, compared with the CB strategy. T_{max} varies within the interval [0.67, 6.67]. Outside this interval, as the number of nodes dynamically allocated to update transactions is constant, response times remain constant as well. As T_{max} is only relevant for the BRT strategy, the CB strategy is represented by a horizontal line.

An expected result with BRT is that, when increasing T_{max}, the response time of update transactions increases linearly. Therefore, since fewer nodes are dedicated to update transactions, queries have more resources for execution and their response time decreases.

When T_{max} is small, we observe in Fig. 10a that BRT outperforms CB for T_{max} under a given value, 3.9 in our experimental conditions. However, for T_{max} under this value, we observe in Fig. 10c that CB has better query response time than BRT. This is a general result because BRT, with a small T_{max}, acts in favor of transactions at the expense of suboptimal routing for queries. In order to provide a transaction response time below T_{max}, the algorithm does not release idle nodes for queries too quickly to anticipate forthcoming transactions. This slows down query processing, but is necessary to keep most of the transactions under T_{max}. In all our experiments, we obtain at least 91% of transactions having their response time below T_{max}. This complies with the TPC-C specification for new-order transactions which states that 90% must have a response time less than T_{max}.

Table 1
Minimal T_{max} needed to dedicate NbNode to update transactions

NbNode	10	9	8	7	6	5	4	3	2	1
T_{max}	0.67	0.74	0.83	0.95	1.11	1.33	1.67	2.22	3.33	6.67

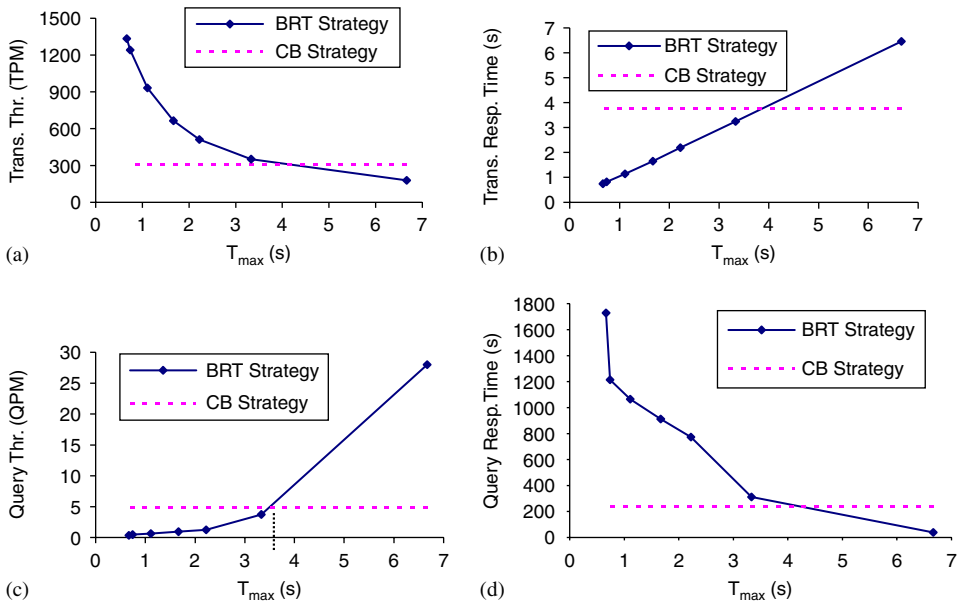

Fig. 10. Throughput and response time for update transactions and queries with CB and BRT strategy: (a) transaction throughput; (b) transaction response time; (c) query throughput; (d) query response time.

When T_{max} increases, we observe in Fig. 10a and c that the two strategies behave differently. Above a given value, 3.5 in our experiments, BRT is no more efficient for update transactions. This result is obvious since a large T_{max} implies that the system can execute update transactions with a large delay, and thus uses only a few nodes for update transactions. As more nodes are available, queries perform faster and BRT outperforms CB for queries, at the expense of increased update transaction time.

In conclusion, CB provides the best overall performance, thus it should be used as soon as it meets the application needs. BRT should be used only in cases where the application requires favoring the execution of update transactions (resp. queries), using a small T_{max} (resp. large T_{max}).

7.4. Speed up experiments

In this section, we study the performance speed up of our transaction routing when increasing the number of cluster nodes. Since CB provides the best overall performance, we ignore the BRT strategy.

To experiment with more than 11 nodes (up to 128 nodes), we consider 8 cluster nodes, each of them running a node emulator that acts as a set of 16 DBMS nodes. This emulates up to 128 nodes. The emulator supports the same interface as the other nodes. Thus, our router can connect to the emulator as if it was connected to real DBMS nodes. This lets us measure whether our router remains efficient with a growing number of nodes and application terminals, or whether one router becomes a bottleneck. We calibrated the emulator so that it matches the real throughput we get for 4 to 10 nodes without emulation.

The goal is to investigate linear speed-up, i.e., if doubling the number of nodes and application terminals results in doubling the throughput. The workload depends on the number of nodes n. It is composed of $2n$ transaction terminals and $2n$ query terminals. Fig. 11a shows transaction throughput versus the number of nodes. It shows that the throughput scales almost linearly from 4 up to 32 nodes, the difference with a linear growth being less than 16%. This validates our solution for medium-scale clusters and shows the good quality of our research prototype.

Between 32 and 96 nodes, performance increases slightly less but the increase is still good. Beyond 96 nodes, performance no longer improves mainly because of synchronization. The more transactions are executed, the more synchronization is required for queries. Speed up experiments lead to the same conclusions for queries (linear increasing throughput until 32 nodes, stable response time), so they are omitted here.

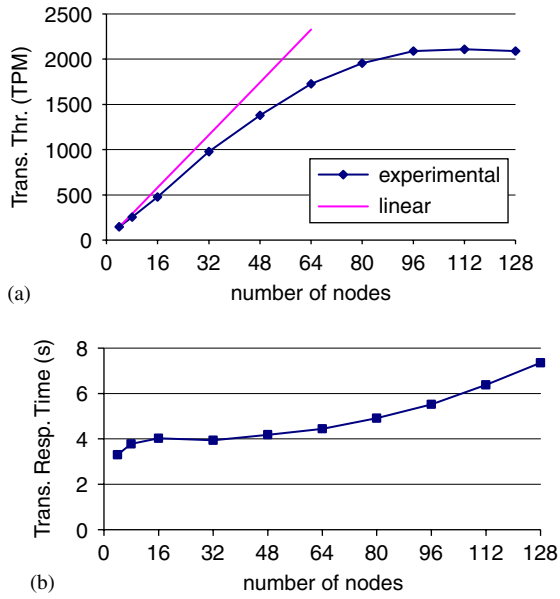

Fig. 11. Transaction throughput and response time vs. number of nodes: (a) transaction throughput vs. number of nodes; (b) transaction response time vs. number of nodes.

7.5. Influence of tolerated freshness

In this section, we study the impact of the tolerated staleness on performance. We measure how much the tolerated staleness of queries can improve the response time of both queries and update transactions. We choose a medium size setup: 10 nodes and 40 terminals (20 query terminals and 20 transaction terminals). We vary the tolerated staleness of queries. Fig. 12 shows the average throughput and response time of queries and transactions versus their tolerated staleness (expressed as a number of tuples).

The results in Fig. 12 show that increasing the tolerated staleness improves significantly the query throughput, reaching more than five times the initial throughput, i.e., when queries do not tolerate any staleness. The transaction throughput is also increased by a factor of 2. This is mainly due to the fact that increasing the tolerated staleness gives more flexibility to load balancing and postpones the

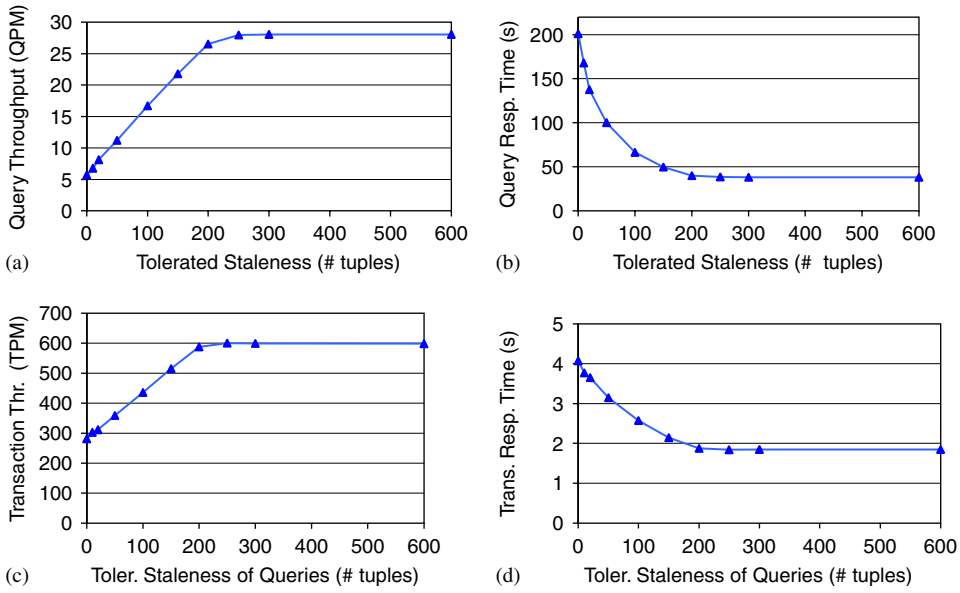

Fig. 12. Throughput and response time of queries and transactions vs. the tolerated staleness: (a) query throughput; (b) query response time; (c) transaction throughput; (d) transaction response time.

synchronization work. This result is important for applications where a tolerated staleness of few hundreds of missing tuples is acceptable for queries (e.g., statistical queries over millions of tuples). We note that, for a staleness value greater than 450, the throughput does not increase. This is because the system reaches the maximum throughput: the tolerated staleness reaches the value for which the synchronization would have started immediately after the end of the experiment.

7.6. Concluding remarks

We now summarize the main observations and conclusions obtained from the experiments with the Leganet prototype. First, our cost function is more accurate than simpler cost functions. We compared the efficiency of our cost function with that of two baseline cost functions, one that considers the nodes' outstanding load and another that considers the nodes' freshness. The results showed that using our cost function yields better load balancing and performance for all kinds of workloads and realistic rates of concurrent data access. Second, CB outperforms BRT in the general case. BRT should be preferred only in some cases, when update transactions are more important than queries. This is the case for instance in pharmacy applications where updates are generated by front-office applications and thus are interactive, while queries are generated by back-office applications which tolerate a greater latency. Third, our approach scales almost linearly for medium size clusters (up to 32 nodes). For larger clusters, the scaling remains acceptable but is no more linear. One solution to overcome this limitation would be to reduce synchronization time using asymmetric synchronization, i.e., sending the modified tuples obtained at the initial node of a transaction instead of replaying the whole transaction. However, as mentioned in Section 3.1, this solution is not straightforward using black-box DBMSs, since it implies log sniffing which cost may be prohibitive in the general case.[4] Fourth, relaxing freshness has a great impact on transaction processing performance (up to a factor 5), for both updates and queries. This gain is obtained by better balancing due to greater flexibility and by reducing synchronization since nodes need less refreshment.

8. Related work

There has been extensive work on exploring the trade-offs between data consistency, transaction processing performance and scalability in database systems. The main areas of work related to ours are:

[4]Our experimentation with Oracle's Logminer tool showed that reading the log takes at least 0.34 s.

replication and load balancing in database systems, relaxed consistency models, replication and load balancing in database clusters.

8.1. Replication and load balancing in database systems

Replication has long been used in database systems to improve both data availability and performance. However, the major problem of data replication is to manage the mutual consistency of the replicas in the presence of updates [8]. The basic solution that enforces strong replica consistency is eager (also called synchronous) replication, typically using the Read-One-Write All—ROWA protocol [3]. Whenever a transaction updates a replica, it also updates all other replicas (using a distributed transaction), thereby enforcing the mutual consistency of the replicas. However, the atomic commitment of the distributed transaction typically relies on the two-phase commit (2PC) protocol [3] which is known to be blocking (i.e., does not deal well with nodes' failures) and has poor scale up.

The alternative approach to eager replication is lazy (also called asynchronous) replication [8,14] whereby a transaction only updates one replica, the updates to the other replicas being propagated later on using separate refresh transactions. Lazy replication trades consistency for performance. In the case of single master copies (for each replica, there is a single master node that can accept update transactions), lazy replication can yield good performance [15] and achieve mutual consistency [4]. However, failure of a master node hurts data availability. Lazy multi-master replication (also called update anywhere) is supported by most commercial database systems because it yields better data availability [16]. However, mutual consistency can be compromised as a result of different master nodes executing conflicting updates. The typical solution is optimistic and provides for periodic conflict detection and semi-automatic replica reconciliation. In our work, we also exploit lazy multi-master replication but we avoid conflicts by exploiting the commutativity between transactions.

Database replication can improve performance by increasing data access locality which reduces the number of network accesses. It can also improve performance by load balancing data accesses across the database replicas, each at a different node. The simplest way to perform load balancing is using the ROWA protocol which reads data from any node, e.g., randomly chosen, and writes to all nodes. Most replicated database systems use a simple strategy for load balancing. The Mariposa distributed data manager [17] uses a more sophisticated strategy based on lazy replication with rule-based conflict resolution and a micro-economic model for load balancing, whereby a Mariposa broker uses a distributed advertising service to select nodes that want to bid on queries. Using a micro-economic paradigm is suited for very large networks of autonomous, competing nodes.

8.2. Relaxed consistency models

The earliest form of relaxed consistency is snapshot isolation [18] whereby a transaction sees only the latest database snapshot, as produced by all transactions committed before it starts. A transaction can commit only if its write set does not intersect with those of current transactions. Otherwise it aborts. Snapshot isolation is popular because it is simple and increases performance by never blocking or aborting read-only transactions. Recent work has focused on using snapshot isolation to improve the performance of read-only transactions in replicated databases. The RSI-PC [19] algorithm is a primary copy solution which separates update transactions from read-only transactions Update transactions are always routed to a main replica, whereas read-only transactions are handled by any of the remaining replicas, which act as read-only copies. Postgres-R(SI) [20] proposes a smart solution that does not require declaring transaction properties in advance. It uses the replication algorithm of [12] which must be implemented inside the DBMS. The experiments are based on a 10-node cluster. SI-Rep [21] provides a solution similar to Postgres-R(SI) on top of PostgreSQL which needs the write set of a transaction before its commitment. Write sets can be obtained by either extending the DBMS, thus compromising DBMS autonomy, or using triggers.

With lazy replication, consistency is relaxed until refreshment or reconciliation. There has been much interesting work on relaxed consistency models for controlling the divergence between replicas according to user requirements. Non-isolated queries are also useful in non-replicated environments [18]. The specification of inconsistency for queries has been widely studied in the literature, and may be divided in two dimensions, temporal and spatial [22]. An example of temporal dimension is found in

quasi-copies [23], where a cached (image) copy may be read-accessed according to temporal conditions, such as an allowable delay between the last update of the copy and the last update of the master copy. The spatial dimension consists of allowing a given "quantity of changes" between the values read-accessed and the effective values stored at the same time. This quantity of changes may be for instance the number of data items changed, the number of updates performed or the absolute value of the update. In the continuous consistency model [24], both the temporal dimension (staleness) and the spatial dimension (numerical error and order error) are controlled. Each node propagates its writes to other nodes, so that each node maintains a predefined level of consistency for each dimension. Then each query can be sent to a node having a satisfying level of consistency (*w.r.t.* the query) in order to optimize load balancing

The Trapp project [25] addresses the problem of precision/performance trade-off in the context of wide area networks. The focus is on numeric computation of aggregation queries with the objective of minimizing communication costs. The TACT middleware [24] implements the continuous consistency model. Although additional messages are used to limit divergence, a substantial gain in performance may be obtained if users accept a small error rate. However, read and write operations are mediated individually: an operation is blocked until consistency requirements can be guaranteed. This implies monitoring at the server level, and it is not clear if it allows installation of a legacy application in a database cluster. In the quasi-copy caching approach [23], four consistency conditions are defined. Quasi-copies can be seen as materialized views with limited inconsistency. However, they only support single-master replication. Epsilon transactions [11] provide a nice theoretical framework for dealing with divergence control. As in the continuous consistency model [24], they allow different consistency metrics to give answers to queries with bounded imprecision. However, the implementation of epsilon transactions requires to significantly alter the concurrency control, since each lock request must read or write an additional counter to decide whether the lock is compatible with the required level of consistency. Recent work on relaxed currency [26] goes one step further in allowing users to explicitly specify fine-grained currency and consistency constraints (in SQL) and in providing well-defined semantics for such constraints so they can be enforced by a DBMS query processor. However, compared to our freshness model and its implementation, none of these models addresses the issue of leaving databases and applications autonomous and unchanged.

8.3. Replication in database clusters

In the context of database clusters, recent work on replication has dealt with the issues of scalability (to achieve high-performance with large numbers of nodes) and, to a lesser extent, autonomy (to exploit black-box DBMS). A major result is that, by exploiting efficient group communication services, eager replication (which provides strong consistency) can be made non-blocking and can scale up to large cluster sizes. The seminal paper on this is [12]. Its eager multi-master replication algorithm significantly reduces the number of messages exchanged to commit transactions compared to 2PC. Furthermore, it is non-blocking: when a failed node is detected, all its transactions propagated to other nodes are aborted. It exploits group communication services to guarantee that messages are delivered at each node according to some ordering criteria. The implementation within the PostGRES DBMS shows scale up to 15 nodes. In [27], the authors provide a wider range of experiments and show through emulation that the algorithm scales up well to 100 nodes. However, the proposed implementation which combines concurrency control with group communication primitives hurts DBMS autonomy. Furthermore, because update transactions are executed by all nodes, there is no opportunity for transaction load balancing, only query load balancing. An extension to this work [28] further improves scalability and avoids redundant transaction execution. By propagating updates to other replicas, it enables to perform transaction load balancing. However, it works only for single-master copies. In [29], it is shown that this solution can be implemented outside a DBMS, and thus support DBMS autonomy. The experiments show scale up to 15 nodes. In [30], the same authors show, through performance analysis, that eager replication can scale up much better than quorums, yet being much simpler to implement. The result is significant as quorums are often suggested to reduce the overhead of scale replication. However, the analytical framework they propose, though elegant, is not adapted to analyze how our lazy replication scheme scales up. Indeed, as it is designed for

comparing quorums and eager replication, it does not take into account the cost of refreshing stale nodes. Thus, if we use this framework for analyzing our approach, either refresh transactions are not taken into account and the system appears to scale perfectly, which is not true, or they are taken into account and the system appears to behave like eager replication, which is neither true.

Lazy replication can also be used in the context of database clusters to provide strong consistency. A lazy multi-master solution that achieves strong consistency is preventive replication [10]. Instead of using atomic broadcast as in eager group-based replication, preventive replication uses FIFO reliable multicast which is a weaker constraint. It deals with stored procedures and is able to deal with autonomous databases. Its implementation on top of the PostgreSQL DBMS shows good scale up in a 32-node cluster. Another lazy replication solution that provides strong consistency in database clusters is presented in [9]. To avoid conflicts, it makes the scheduler conflict-aware at the table level. However, it does not support stored procedures as in our solution.

There are also middleware solutions that support data replication and achieve DBMS autonomy. C-JDBC [31,32] is a database clustering middleware which emphasizes flexibility and adaptation to the application needs. It provides different replication algorithms with support for various configurations such as full replication or partial replication. However, the algorithms are based on optimistic transaction-level schedulers with deadlock-detection and thus do not support strong consistency as we do. Middle-R [33] is another middleware for transparent database replication which focuses on dynamic adaptation to failures and workload variations. It uses a synchronous replication algorithm to enforce one-copy-serializability but only supports master–slave configurations. ESCADA [34] is another middleware that builds on the Database State Machine model which we discussed above. All these middleware solutions focus on replication and fault-tolerance.

8.4. Load balancing in database clusters

Load balancing in database clusters using replication has recently received much attention. One of the most popular projects is PowerDB at ETH Zurich. Its solution fits well for some applications, such as XML document management [35] or read-intensive OLAP queries [36]. For interactive Web-based information systems, the scheduling algorithm of [37] is tailored for low-complexity read only queries, and requires detailed information about every query operator. However, none of these solutions addresses the problem of seamless integration of legacy applications.

In [38], the authors propose strategies for freshness-aware query scheduling which provide one-copy serializability in a database cluster with mono-master replication. In particular, their one-idle strategy yields good cluster utilization. When a query comes in, it is routed to the least loaded node that is fresh enough. However, if no node is fresh enough, the query simply waits. More generally, they do not consider the case where refreshing an idle node would yield better performance than an overloaded fresh node. Our cost-based routing is general enough to handle these important cases. Furthermore, their freshness model has only one level of granularity: the entire database. Whenever a transaction updates a relation R at a given node, all the other nodes become stale, even for other transactions which are not reading R. In our approach, the level of granularity is the relation: when a query reads a relation at a node, the local copy freshness is computed using only the running transactions which are updating the same relation at other nodes. This makes our freshness computation more accurate, thus increasing query throughput.

There are also middleware solutions for database clusters that focus on load balancing. In [39], the authors describe a middleware for data replication that adjusts to changes in the load submitted to the different replicas and to the type of workload. They propose a novel strategy which combines load-balancing techniques with feedback-driven adjustments of the number of concurrent transactions. The proposed solution is shown to provide high throughput, good scalability, and low response times for changing loads and workloads with little overhead. In [40], the authors describe a middleware for scaling and availability of dynamic content sites using a cluster of Web servers and database engines. They show that replication with relaxed consistency is key for scalability while the actual choice of the load balancing strategy is less important.

A cooperative caching mechanism is proposed in [41] and is shown to yield a high hit rate for heterogeneous clusters. In [42], a scheme is proposed for scheduling disk requests that takes advantage of the ability of high-level functions to operate directly

at individual disk drives. Both techniques could enhance our solution at the system level, but would compromise database autonomy. The dynamically ordered scheduling strategy proposed in [43] distributes CPU and disk load of OLAP queries in the context of parallel data warehouses with a specific data allocation scheme [44]. Affinity-based routing [45] is another scheduling technique that partitions transactions into affinity groups to avoid contention and thus improve load balancing. However, these scheduling techniques are designed for a shared-disk architecture and are not suited for shared-nothing cluster architectures.

Extensive work has also been done for cluster load balancing at the system level or for Internet services. The Neptune Project [46] proposes a framework for scheduling service requests with quality constraints (such as maximum response time), but does not deal with data freshness. It also proposes and validates routing strategies based on random polling which is well adapted for short requests (less than 100 ms) but does not work for longer transactions. The GMS project [47] uses global information to optimize page replacement and pre-fetching decisions over the cluster. However, it mainly addresses specific Internet applications such as the Porcupine mail server.

Workload allocation in distributed transaction processing systems is surveyed in [48]. In [49], a classification of transaction-routing algorithms for shared-nothing transaction processing systems is proposed. Our routing algorithm fits in the categories named Single Router, Dynamic Algorithms based on Routing History, because our cost model relies on response time history, and Goal Oriented because we consider data freshness as a requirement to reach. We note that none of the surveyed algorithms considers transaction routing with freshness control.

9. Conclusion

In this paper, we described the Leganet system which performs freshness-aware transaction routing in a database cluster. To optimize load balancing, we use lazy multi-master database replication with freshness control, and strive to capitalize on the work on relaxing freshness for higher performance. The Leganet system preserves database and application autonomy using non-intrusive techniques that work independently of any DBMS.

The main contribution of this paper is a transaction router which takes into account freshness requirements of queries at the relation level to improve load balancing. It uses a cost function that takes into account not only the cluster load in terms of concurrently executing transactions and queries, but also the estimated time to refresh replicas to the level required by incoming queries. The model to estimate replica freshness estimates the freshness of databases updated by autonomous applications at the level of relations, which is accurate enough to improve transaction routing. It works with multi-master replication which provides the highest opportunities for transaction load balancing.

We also proposed two CB routing strategies that improve load balancing. The first routing strategy (CB) assesses the synchronization cost to respect the tolerated staleness by queries and transactions and chooses the node with minimal cost. The second strategy (BRT) is a variant with a parameter, $T_{\max}$, which represents the maximum response time users can accept for update transactions. It dedicates as many cluster nodes as necessary to ensure that updates are executed in less than $T_{\max}$, and uses the remaining nodes for processing queries.

We implemented our solution on an 11-node cluster running Oracle 8i under Linux. We used this implementation for initial performance experiments and to calibrate an emulation model that deals with larger cluster configurations (up to 128 nodes). First, we showed that, compared with two baseline cost functions (one based on the nodes' current load and the other based on the nodes' freshness), our cost function yields better load balancing and performance. Second, the experiments showed that CB outperforms BRT in the general case and that BRT should be preferred only when update transactions are more important than queries. Third, our approach scales very well (almost linearly) for clusters up to 32 nodes and has good scale up until 96 nodes. Finally, we showed that relaxing freshness has a great impact on transaction processing performance (up to a factor 5), for both updates and queries, thanks to better load balancing and reduced node synchronization.

In this paper, we have made the simplifying assumption of full replication for concentrating on the problem of freshness-aware transaction routing. However, we could extend our approach to deal with partial replication, with a mix of partitioned relations (typically the largest relations) and replicated relations over a subset of the cluster nodes as

in [1]. Although this approach does not violate database autonomy, it would require some careful database design.

Another improvement we are investigating is to use asymmetric synchronization, i.e., sending the modified tuples obtained at the initial node of a transaction instead of replaying the whole transaction. As explained in Section 7.6, this solution is not straightforward using black-box DBMSs, since it implies log sniffing. Our experimentation with Oracle's Logminer tool showed that reading the log takes at least 0.34 s, thus we must study carefully in which conditions asymmetric synchronization may be used.

Finally, the current Leganet system uses a centralized router which can obviously be a single point of failure and a performance bottleneck. A solution to this problem suggests replicating the router and its metadata at two or more nodes. Maintaining the consistency of the replicated metadata is an interesting issue, for instance, using either eager replication or distributed shared memory software, and the subject of future work.

References

[1] U. Röhm, K. Böhm, H.-J. Schek, OLAP query routing and physical design in a database cluster, International Conference on Extending Database Technology (EDBT'00), Springer, Konstanz, Germany, 2000, pp. 254–268.

[2] P. Valduriez, Parallel database systems: open problems and new issues, Distributed and Parallel Databases 1 (2) (1993) 137–165.

[3] T. Özsu, P. Valduriez, Principles of Distributed Database Systems, 2nd ed, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[4] E. Pacitti, P. Minet, E. Simon, Replica consistency in lazy master replicated databases, Distributed and Parallel Databases 9 (3) (2001) 237–267.

[5] E. Pacitti, O. Dedieu, Algorithms for optimistic replication on the web, Journal of the Brazilian Computing Society 8 (2) (2002) 7–11.

[6] S. Gançarski, H. Naacke, E. Pacitti, P. Valduriez, Parallel processing with autonomous databases in a cluster system, International Conference of Cooperative Information Systems (CoopIS'02), Irvine, California, 2002, pp. 410–428.

[7] Transaction Processing Performance Council. TPC Benchmark C, Rev 5.1, www.tpc.org/tpcc/.2002.

[8] P. Bernstein, E. Newcomer, Principles of Transaction Processing, Morgan Kaufmann, 1997.

[9] C. Amza, A. Cox, W. Zwaenepoel, Conflict-aware scheduling for dynamic content applications. USENIX Symposium on Internet Technologies and Systems (USITS'03), Seattle, Washington, 2003.

[10] E. Pacitti, T. Özsu, C. Coulon, Preventive multi-master replication in a cluster of autonomous databases, International Conference on Parallel Processing (Euro-Par'03), Klagenfurt, Austria, 2003, pp. 318–327.

[11] K. L. Wu, P. S Yu, C. Pu, Divergence control for epsilon-serializability, IEEE International Conference on Data Engineering (ICDE'92), Tempe, Arizona, 1992, pp. 506–515.

[12] B. Kemme, G. Alonso, Don't be lazy be consistent: Postgres-R. A new way to implement Database Replication, International Conference on Very Large Databases (VLDB'00), Cairo, Egypt, 2000, pp. 134–143.

[13] C. Amza, A. Cox, S. Dwarkadas, P.J. Keleher, H. Liu, R. Rajamony, W. Yu, W. Zwaenepoel, TreadMarks: shared memory computing on networks of workstations, IEEE Computer 29 (2) (1996) 18–28.

[14] T.A. Anderson, Y. Breitbart, H.F. Kort, A. Wool, Replication, consistency, and practicality: are these mutually exclusive? ACM SIGMOD, International Conference on Management of Data (SIGMOD'98), 1998, Seattle, Washington, pp. 484–495.

[15] E. Pacitti, P. Minet, E. Simon, Fast algorithms for maintaining replica consistency in lazy master replicated Databases, International Conference on Very Large Databases (VLDB'99), Edinburgh, Scotland, 1999, pp. 126–137.

[16] D. Stacey, Replication: DB2, Oracle, or Sybase?, ACM SIGMOD Record 24 (4) (1995) 95–101.

[17] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, A. Yu, Data replication in mariposa, IEEE International Conference on Data Engineering (ICDE'96), New Orleans, Louisiana, 1996, pp. 485–494.

[18] H. Berenson, P. Bernstein, J. Gray, J. Melton, E.J. O'Neil, P.E. O'Neil, A Critique of ANSI SQL Isolation Levels. ACM SIGMOD, International Conference on Management of Data (SIGMOD'95), 1995, San Jose, California, pp. 1–10.

[19] C. Plattner, G. Alonso, Ganymed: scalable replication for transactional web applicationism, International Middleware Conference (Middleware'04), Toronto, Canada, 2004, pp. 155–174.

[20] S. Wu, B. Kemme, Postgres-R(SI): combining replica control with concurrency control based on snapshot isolation, IEEE International Conference on Data Engineering (ICDE'05), Tokyo, 2005, pp. 422–433.

[21] Y. Lin, B. Kemme, M. Patino-Martinez, R. Jimenez-Peris, Middleware based data replication providing snapshot isolation, ACM SIGMOD International Conference on Management of Data. Baltimore, USA, 2005, pp. 419–430.

[22] A. Sheth, M. Rusinkiewicz, Management of interdependent data: specifying dependency and consistency requirements, Workshop on the Management of Replicated Data, Houston, Texas, IEEE Computer Society, Silver Spring, MD, 1990, pp. 133–136.

[23] R. Alonso, D. Barbará, H. Garcia-Molina, Data caching issues in an information retrieval system, ACM Transactions on Database Systems, 15 (3) (1990) 359–384.

[24] H. Yu, A. Vahdat, Efficient numerical error bounding for replicated network services, International Conference on Very Large Databases (VLDB'00), Cairo, Egypt, 2000, pp. 123–133.

[25] C. Olston, J. Widom, Offering a precision-performance tradeoff for aggregation queries over replicated data, International Conference on Very Large Databases (VLDB'00), Cairo, Egypt, 2000, 14–155.

[26] H. Guo, P.-A. Larson, R. Ramakrishnan, J. Goldstein, Relaxed currency and consistency: how to say "Good

Enough'', SQL. ACM SIGMOD International Conference on Management of Data (SIGMOD'04), Paris, 2004, pp. 815–826.
[27] B. Kemme, G. Alonso, A new approach to developing and implementing eager database replication protocols, ACM Transactions on Database Systems 25 (3) (2000) 333–379.
[28] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, G. Alonso, Scalable replication in database clusters, International Conference on Distributed Computing (DISC'00), Toledo, Spain, 2000, pp. 315–329.
[29] R. Jiménez-Peris, M. Patino-Martinez, B. Kemme, G. Alonso, Improving the scalability of fault-tolerant Database clusters: early Results, IEEE International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria, 2002, pp. 477–484.
[30] R. Jiménez-Peris, M. Patino-Martinez, B. Kemme, G. Alonso, Are quorums an alternative for database replication, ACM Transactioins on Database Systems 28 (3) (2003) 257–294.
[31] E. Cecchet, J. Marguerite, W. Zwaenepoel, Partial replication: achieving scalability in redundant arrays of inexpensive databases, International Conference on Principles of Distributed Systems (OPODIS'03), 2003, La Martinique, France, pp. 58–70.
[32] E. Cecchet, J. Marguerite, W. Zwaenepoel, C-JDBC: flexible database clustering middleware, USENIX Annual Technical Conference (USENIX'04), Boston, MA, 2004.
[33] R. Jiménez-Peris, M. Patino-Martinez, G. Alonso. Non-intrusive, parallel recovery of replicated data, IEEE Symposium on Reliable Distributed Systems (SRDS'02), Osaka, Japan, 2002, pp. 150–159.
[34] ESCADA, Fault Tolerance Scalable Distributed Databases, http://gsd.di.uminho.pt/ESCADA/escada.html.
[35] T. Grabs, K. Böhm, H.-J. Schek. Scalable distributed query and update service implementations for XML document elements. IEEE RIDE International Workshop on Document Management for Data Intensive Business and Scientific Applications (RIDE'01), Heidelberg, Germany, 2001, pp. 35–42.
[36] U. Röhm, K. Böhm, H.-J. Schek. Cache-aware query routing in a cluster of databases, IEEE International Conference on Data Engineering (ICDE'01), Heidelberg, Germany, 2001, pp. 641–650.
[37] F. Waas, M.L. Kersten, Memory-aware query routing in interactive web-based information systems, British National Conference on Databases (BNCOD'01), Chilton, UK, Lecture Notes in Computer Science, vol. 2097, Springer, 2001, pp. 168–184.
[38] U. Röhm, K. Böhm, H.-J. Schek, H. Schuldt, FAS–A freshness-sensitive coordination middleware for a cluster of OLAP components, International Conference on Very Large Databases (VLDB), Hong Kong, China, 2002, pp. 754–765.
[39] J.M. Milan-Franco, R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, Adaptive middleware for data replication, International Middleware Conference (Middleware'04), Toronto, Canada, 2004, pp. 175–194.
[40] C. Amza, A. Cox, W. Zwaenepoel, Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites, International Middleware Conference (Middleware'03), Rio de Janeiro, 2003, pp. 282–304.
[41] G. Chen, C.-L. Wang, F.C.M. Lau, Building a scalable web server with global object space support on heterogeneous clusters, IEEE International Conference on Cluster Computing (CLUSTER'01), Newport Beach, California, 2001, pp. 313–321.
[42] E. Riedel, C. Faloutsos, G. R. Ganger, D. Nagle, Data mining in an OLTP system (Nearly) for free, ACM SIGMOD International Conference on Management of Data (SIGMOD'00), Dallas, Texas, 2000, pp. 13–21.
[43] H. Märtens, E. Rahm, T. Stöhr. Dynamic query scheduling in parallel data warehouses, International Conference on Parallel Processing (Euro-Par'02), Paderborn, Germany, 2002, pp. 321–331.
[44] T. Stöhr, H. Märtens, E. Rahm, Multi-dimensional database allocation for parallel data warehouses, International Conference on Very Large Databases (VLDB'00), Cairo, Egypt, 2000, pp. 273–284.
[45] P.S. Yu, D.W. Cornell, D.M. Dias, B.R. Iyer, On affinity based routing in multi-system data sharing, International Conference on Very Large Databases (VLDB'86), Kyoto, Japan, 1986, pp. 249–256.
[46] K. Shen, H. Tang, T. Yang, L. Chu, Integrated resource management for cluster-based Internet services, USENIX Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, MA, 2002.
[47] G. M. Voelker, E.J. Anderson, T. Kimbrel, M.J. Feeley, J.S. Chase, A.R. Karlin, H.M. Levy, Implementing Cooperative Prefetching and Caching in a Global Memory System, ACM SIGMETRICS Conference on Performance Measurement, Modeling, and Evaluation (SIGMETRICS'98), Madison, WI, 1998, pp. 33–43.
[48] E. Rahm, A framework for workload allocation in distributed transaction processing systems, Journal of Systems and Software 18 (2) (1992) 171–190.
[49] C.N. Nikolaou, M. Marazakis, G. Georgiannakis, Transaction routing for distributed OLTP systems: survey and recent results, Information Science 97 (1,2) (1997) 45–82.