DBFarm: A Scalable Cluster for Multiple Databases

Christian Plattner[1], Gustavo Alonso[1], and M. Tamer Özsu[2]

[1]Department of Computer Science
ETH Zurich, Switzerland
{plattner,alonso}@inf.ethz.ch

[2]David R. Cheriton School of Computer Science
University of Waterloo, Canada
tozsu@uwaterloo.ca

Abstract. In many enterprise application integration scenarios, middleware has been instrumental in taking advantage of the flexibility and cost efficiency of clusters of computers. Web servers, application servers, platforms such as CORBA, J2EE or .NET, message brokers, and TP-Monitors, just to mention a few examples, are all forms of middleware that exploit and are built for distributed deployment. The one piece in the puzzle that largely remains a centralized solution is the database. There is, of course, much work done on scaling and parallelizing databases. In fact, several products support deployment on clusters. Clustered databases, however, place the emphasis on single applications and target very large databases. By contrast, the middleware platforms just mentioned use clustered deployment not only for scalability but also for efficiently supporting multiple concurrent applications. In this paper we tackle the problem of clustered deployment of a database engine for supporting multiple applications. In the database case, multiple applications imply multiple and different database instances being used concurrently. In the paper we show how to build such a system and demonstrate its ability to support up to 300 different databases without loss of performance.

Keywords: database clusters, scalability, replication, consistency.

1 Introduction

In many enterprise application integration scenarios, middleware has been instrumental in taking advantage of the flexibility and cost efficiency of clusters of computers. There exists a plethora of middleware solutions to create distributed deployments and to parallelize a wide range of application types. In addition, distributed deployment across a cluster of machines is most effective when the same platform can be used for concurrent applications. Thus, platforms such as J2EE or .NET are clearly designed to be used with multiple concurrent applications. In spite of this, the piece of the puzzle that still mostly remains a centralized solution is the database. Even though there is much work done in the area of cluster and parallel databases [20], the emphasis is always on improving access to a single database. This is not a trivial distinction. The problem with single instance optimizations is that they often conflict with the goal of supporting multiple database instances. For instance, crucial to be able to exploit clusters is the ability to freely move resources from machine to machine as needed and allocate more or less machines to different applications as load fluctuates. In the single database case, the clients simply connect to that database. In a cluster based solution, clients should see a single image of the system even when databases are dynamically moved around. Similarly, efficient use of the resources of the cluster imply that database instances might

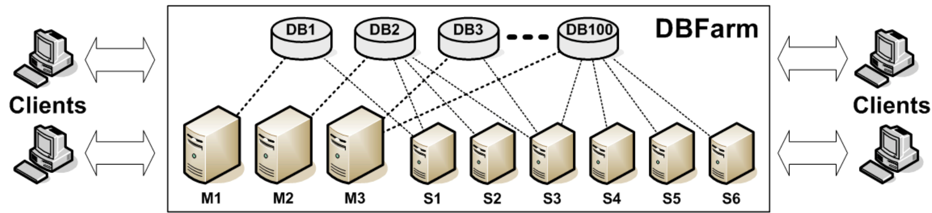


Fig. 1. Managing 100 Databases (DBx) on a DBFarm using 3 Master (Mx) and 6 Satellite (Sx) Servers. Each Database is installed on one Master and may be replicated on different Satellites.

share computing nodes. Care must be taken that work on one instance does not negatively affect work on a different instance. Finally, clients should always see a consistent state but consistency needs to be accomplished without limiting the scalability of the cluster and without introducing unacceptable overhead.

These constraints point out to the need for some form of middleware based solution since database optimizations mostly apply to single instances and somebody has to coordinate the access to multiple, independent instances. The challenge in building such a middleware based solution is that it must be very light weight so as not to limit scalability. Handling, for instance, 100 databases each running 100 transactions per second does not allow to spend too much time on each transaction (and our goal is to scale well beyond that). Yet, that same middleware must guarantee consistency and a single system image.

In this paper we describe the architecture and implementation of *DBFarm*, a cluster based database server implemented as a thin middleware layer that can be used to support several hundred database instances. DBFarm is based on using two kinds of database servers: master database servers (see Figure 1) and satellites. Master servers can be made highly reliable by using specialized hardware, RAID systems, hot stand-by techniques, and sophisticated back-up strategies. This provides the necessary reliability and does it in a way that the resources are shared among all databases. Scalability is then provided by a cluster of unreliable satellite machines where copies of the individual databases are placed. A key aspect of DBFarm is that the users of the databases are not aware of the fact that they may be working with an unreliable copy rather than working on the master database. DBFarm ensures that they see a consistent state at all times. Another key feature of DBFarm is that the load distribution between the master databases and the satellite copies is done based on the read/write characteristics of the operations requested by the users. Writes are performed at the master databases, reads at the copies. This workload distribution allows to significantly reduce the load at the masters (and thus, be able to support a larger number of databases on the same machines) while providing a high level of parallelization for the satellites (and, thus, the basis for high scalability). The distribution also reflects the characteristics of most database loads where updates tend to be small operations with high locality while read-only transactions typically cause much more I/O (to retrieve data and indexes) and computation overhead (to perform operations such as joins, order, averages, or group by).

The feasibility and advantages of this approach are demonstrated through an extensive set of experiments. We show how DBFarm provides more stable and scalable performance (in both response time and throughput) for a set of up to 300 TPC-W data-

bases than a stand alone database server installation. We also show the performance of DBFarm using the RUBBoS benchmark (much larger databases with a more complex load) and discuss how to assign priorities to individual databases so that they get a better performance than others.

In terms of applications, DBFarm can be used in a wide variety of settings. It can be used to turn a cluster of machines into a database service that is provided within a LAN setting (a company, a university), thereby localizing the maintenance and administration of the databases and the machines on which they run. It can also be used to implement database services as part of an Application Service Provider where small or medium companies place their databases at a provider's DBFarm. The sharing of resources implicit in DBFarm makes this an efficient solution and its simple scalability enables the support of a large number of users.

Our work exploits a novel form of replication and load distribution that is well suited to many modern applications such as web servers. Unlike many existing solutions, it provides consistency while remaining light weight. DBFarm does not involve complex software or hardware layers (e.g., specialized communication infrastructures, shared disks) or requiring the modification of the application (e.g., special clustering of the data or submission of complete transactions). Our approach also includes innovative algorithms for transaction routing across a database cluster that avoid many of the limitations of current database replication solutions. Finally, DBFarm provides a highly scalable clustering solution for databases.

2 Architecture

2.1 Load Separation

In order to provide scalability in databases, the load must be distributed across a number of machines. This is a well know problem in replicated, distributed and parallel databases. In DBFarm the added complication is that the load separation must happen on a per database instance basis while still maintaining consistency. There are many ways to implement load separation in single instance settings. [2, 5] use versioning and concurrency control at the middleware level to route transactions to the appropriate machines. [17] relies on clients to produce a well balanced load over a cluster, different cluster nodes are then synchronized using group communication primitives. [13, 18] assumes the database schema has been pre-partitioned into so called conflict classes, which are then used for load separation. [1] requires clients to specify the freshness level of the data and directs transactions to different nodes according to the freshness of the nodes (i.e., how up-to-date they are). This approach, however, already assumes that clients do not want to see the most up-to-date state and, thus, represents a relaxation of concurrency.

Unfortunately, none of these methods is feasible in the context of a multi instance cluster. Duplicating concurrency control at the middleware layer would result in a prohibitive overhead per transaction (aside of a complex maintenance issue since the middleware would have to be aware and maintain versioning and schema information for hundreds of instances). Group communication as a way to distribute load and maintain consistency is also out of the question because of the overhead of membership (each instance would have its own group membership) and the implicit synchronization of

cluster nodes. Similarly, the use of conflict classes requires to parse the incoming transaction load to identify the conflict class being accessed which will immediately put a limit on the number of transactions the middleware can route per second.

The load separation approach used in DBFarm is a direct consequence of these restrictions. Rather than relying on schema or data partition information, we simply distinguish between read-only and update transactions. Update transactions are those that perform an update in the database (can nevertheless contain many read operations as well). Read-only transactions are those that do not result in a state change at the database. Such a load separation can be efficiently implemented at the middleware level without having to parse the statements of the transactions and without the need to maintain information on the schema of each database instance in the cluster. Such a separation also has important advantages. The update transactions determine the state of each instance and define what is consistent data. We only need this stream of transactions to determine correctness and consistency. Read-only transactions are used to provide scalability by re-routing them to different cluster nodes in the system. This is based on the nature of many database benchmarks (e.g., TPC-W or RUBBoS) that are used as representative loads and where the load is clearly dominated by read-only transactions (at least 50% of the transactions in most database benchmarks and typically far more complex than update transactions).

Load separation based on read/write transactions results in scalability that is limited only by the proportion of reads and writes in the load. Thus, DBFarm will not provide any scalability to a database with 100% updates (but such load also results in no scalability for any replicated solution -not just DBFarm- and would severely tax existing parallel database engines). However, in the common case that read-only transactions dominate the load, DBFarm provides significant scalability.

2.2 Master and Satellite Servers

As a direct result of the load separation technique used, DBFarm adopts a per instance primary copy, lazy propagation strategy. The primary copy of an instance is called the master database. Master databases are always located on master servers (*masters*). A master server processes all the update transactions of the hosted master databases. As such, it is always up-to-date. Each master database is responsible for maintaining its internal consistency using its own concurrency control and recovery mechanisms. Copies of the master databases (so called *satellite copies*) are placed in *satellite* servers. Satellites are used exclusively for executing read-only transactions. Hence, what is a consistent state is always dictated by the master servers. As a direct result, recovery of failed satellites and spawning of new database copies is not a complex issue in DBFarm (unlike other approaches where the transfer of state for creating or removing a copy involves complex synchronization operations [12, 17]). In addition, the master database of an instance is the fall back solution. If all copies fail, the master server allows clients to continue working on a consistent version of the managed data (albeit with a reduction in performance).

Generally, we assume the masters to be large computers with enough resources (main memory and disk capacity) to accommodate a large number of databases (see the experimental section for details). We also assume the masters to be highly available based on either software or hardware techniques. The capacity of a master will

determine how many different databases can be supported by that master and how high the overall transaction load can be. DBFarm reaches saturation once all masters reach the limit of update transaction streams they can process. At that point, and unless they submit exclusively read-only transactions, clients cannot increase the rate at which they submit transactions since the speed at which the update part of the load is processed is determined by the masters which, at saturation, have no more available capacity. Scalability can, however, be easily increased by adding more masters and redistributing the databases across these machines.

Satellites in turn do not need to be as powerful as the masters, nor need they to be very reliable. There can be an arbitrary number of database copies in each satellite. Scalability for a single master database is provided by having copies in multiple satellites. The limit in scalability, aside of the limit imposed by the proportion of updates in the load, is reached when each concurrent read-only transaction executes in its own satellite server. Beyond that, additional copies will remain idle. Such an extreme degree of distribution can nevertheless be employed in a cluster without loss of efficiency by placing copies of different instances on the same satellite. Since satellites might be shared by different database copies, no resources are wasted because each copy is executing a single read-only transaction. On the other extreme, in DBFarm it is possible for an instance to be centralized and without any copy. In such a case, the master processes both the update and the read-only transaction streams for that database.

An advantage of the approach taken in DBFarm is that it provides a solution for realistic database applications. The master servers host normal databases. User defined functions, triggers, stored procedures, etc. can all be placed at the masters and do not need to be duplicated at the satellites. Most existing database replication solutions assume such features are either not used (e.g., when concurrency control or versioning happens at the middleware level) or require that they are replicated across all copies and behave deterministically at all cluster nodes - something not entirely feasible in the case of triggers in most existing database engines (e.g., when update propagation is done with group communication).

2.3 Transaction Scheduling in DBFarm

While the load separation approach of DBFarm enables scalability and makes sure that the master databases are always up-to-date, it does not by itself guarantee consistent views when copies are being accessed. Therefore, as long as no other measures are taken, the consistency from the client point of view will depend on how the transactional load generated by a client is split into read-only and update streams and how these streams are forwarded to the different database machines in the cluster.

From a client's perspective, what matters is what has been called *strong session serializability* [8]. Briefly explained, strong session serializability requires that a client always sees its own updates. In other words, a read-only transaction from a given client must see not only a consistent state but one that contains all committed updates of that client. This prevents the client from experiencing *travel-in-time* effects where suddenly a query returns correct but stale data. Although in principle strong session serializability would be enough, the approach we take in DBFarm goes a step beyond and makes sure that a read-only transaction executed at a copy will see all updates executed at the master database up to the point in time the read-only transaction has arrived at the system. In

doing so, DBFarm effectively becomes transparent to the clients[1] since they will always read exactly the same they would have read using a single database server.

For simplicity in the explanations, and without loss of generality, we describe the details of the DBFarm transaction scheduling with a single master database server. As clients communicate only with masters, they are not aware of any satellites. Therefore, incoming read-only transactions need to be forwarded by the master to the satellites in a way that consistency is guaranteed. Update transactions for a given database instance are executed locally on the master database. The result is, conceptually, a series of consistent states of the database each of which contains the updates of all the transactions committed up to that point. The master takes advantage of this conceptual ordering by capturing the writesets of update transactions in commit order. A writeset is a precise representation of what has been changed (i.e., the tuple id and the new value). The master then uses this commit order to forward the writesets to all needed satellites using FIFO queues. Satellites then apply these writesets in the same order they are received. It is easy to see and formally prove that, if the execution of transactions at the master database was correct, then the application of changes to a copy in the commit order established by the master guarantees that the copy will go through the same sequence of consistent states as the master database and will eventually reach the same correct state as the master database.

Conceptually, the way this is done in a DBFarm is as follows. For every successful committed update transaction, the master sends back to the client a commit acknowledgment message. The latest sent commit acknowledgment therefore reflects the oldest state that any client should see when sending the next transaction. Hence, if DBFarm sends a commit acknowledgment to a client that executed transaction T_k, then all later incoming read-only transactions from any client must observe a state of the database that includes the effects of writeset WS_k (the writeset that includes the changes done by transaction T_k). Therefore, once a request for a read-only transaction arrives at the master, the master can deduce (by keeping track of the sent commit acknowledgment messages) the minimum state of the database that the read-only transaction must observe. Of course, the management of tracking commit acknowledgment messages and assuring consistency for read-only transactions must be done by DBFarm for each database separately. A possible approach to ensure that a read-only transaction sees no stale data is to block it on the master and then only forward its operations to the target satellite once it has reported the successful application of all needed writesets. However, this approach has several disadvantages: first, there is additional logic, communication and complexity; second, the master must implement transaction queuing; third, overhead is introduced due to the round-trip times of the involved messages and, as a result, read-only transactions may be unnecessarily blocked. In practice, the way the system works achieves the same result but without blocking transactions longer than needed and without imposing additional load on the master. The solution is based on a *tagging mechanism*. Every time an update transaction commits, the master not only extracts the writeset, but also atomically creates an increasing number which gets shipped together with the writeset. We call this number the *change number* for database DB (denoted by

[1] Of course, by giving up transparency and using less strict forms of consistency, the system could offer even more scale-out for read-only transactions.

Algorithm 1: Master Transaction Handling

```
 1: INPUT DB: Database Name
 2: INPUT T: Incoming Transaction
 3: INPUT M: Mode of T, M ∈ {Read-Only, Update}
 4: if M == 'Update' then
 5:     /* T must be executed on the master */
 6:     Start a local update transaction in database DB
 7:     for all Incoming statements S in T do
 8:         if S == 'ROLLBACK' then
 9:             Abort the local transaction
10:             Send abort response back to client
11:             RETURN
12:         end if
13:         if S == 'COMMIT' then
14:             ENTER CRITICAL SECTION
15:             Commit the local update transaction
16:             if Commit operation failed then
17:                 LEAVE CRITICAL SECTION
18:                 Abort the local update transaction
19:                 Send error response back to client
20:                 RETURN
21:             end if
22:             Determine T's commit number CN
23:             Set CN(DB) := CN
24:             LEAVE CRITICAL SECTION
25:             Send successful commit reply to client
26:             Retrieve T's encoded writeset WS from DB
27:             Forward (DB, WS, CN) to all satellites that
                host a copy of database DB
28:             RETURN
29:         end if
30:         Execute S locally
31:         if Execution of S fails then
32:             Abort the local transaction
33:             Send error response back to client
34:             RETURN
35:         end if
36:         Send result of S back to client
37:     end for
38: end if
39: /* T is Read-Only, try to execute on a satellite */
40: if ∃ satellite N with a copy of DB then
41:     Use load balancing to select a satellite N
42:     MIN_CN := CN(DB)
43:     Forward (DB, T, MIN_CN) to N
44:     Relay all incoming statements S in T to N
45:     RETURN
46: end if
47: /* Need to execute T locally */
48: Start a local read-only transaction in database DB
49: for all Incoming statements S in T do
50:     if S ∈ {COMMIT, ROLLBACK} then
51:         Commit the local read-only transaction
52:         Send response back to client
53:         RETURN
54:     end if
55:     Execute S locally
56:     if Execution of S fails then
57:         Abort the local read-only transaction
58:         Send error response back to client
59:         RETURN
60:     end if
61:     Send result of S back to client
62: end for
```

Algorithm 2: Satellite Transaction Handling

```
 1: INPUT DB: Database Name
 2: INPUT T: Incoming Transaction
 3: INPUT CN_MIN: Min. committed State needed to exe-
    cute T
 4: Wait until CN(DB) ≥ CN_MIN
 5: /* Execute T locally in read-only mode */
 6: Start a local read-only transaction in database DB
 7: for all Incoming statements S in T do
 8:     if S ∈ {COMMIT, ROLLBACK} then
 9:         Commit the local read-only transaction
10:         Send response back to client
11:         RETURN
12:     end if
13:     Execute S locally
14:     if Execution of S fails then
15:         Abort the local read-only transaction
16:         Send error response back to client
17:         RETURN
18:     end if
19:     Send result of S back to client
20: end for
```

Algorithm 3: Satellite Writeset Application

```
 1: INPUT DB: Database Name
 2: INPUT WS: Writeset
 3: INPUT CN: Committed State produced by WS
 4: Turn WS into a set of SQL statements
 5: Wait until CN(DB) == (CN - 1)
 6: Start a local update transaction in database DB
 7: Apply the produced SQL statements
 8: Commit the local transaction
 9: if Application of WS failed then
10:     Report ERROR to the master
11:     Disable further processing for database DB
12:     RETURN
13: end if
14: Set CN(DB) := CN
```

$CN(DB)$). Note that the change number is no longer client specific but applies to all update transactions executed on the database. When re-routing read-only transactions to satellite nodes, the master *tags* the begin operation of such transactions with the current change number of the respective database. This number is also maintained for the copies of database DB: if a satellite applies a writeset to a copy, then the copy is known to have achieved the writeset's assigned change number.

A satellite that receives a tagged read-only transaction will only start executing it after it has applied the needed writesets. This is how we make sure a read-only transaction sees all updates performed up to the point it starts executing, not only those from

that particular client. The blocking of read-only transactions is therefore delegated to the satellites, where it can be efficiently handled.

It is important to note that other than checking that every read-only transaction observes the newest consistent state, no concurrency control is needed outside the databases and scheduling therefore has a low overhead in our approach. This is in contrast to existing replication strategies that require additional concurrency control and versioning mechanisms outside the databases [2, 6, 18].

The details of the algorithms used by DBFarm to handle transactions on the master and satellite servers, as well as writeset handling, are given in algorithms 1, 2 and 3.

Algorithm 1 describes the routing on a master. Lines 4-38 handle update transactions. These are executed locally on the master, statement by statement, until the client either decides to *rollback* (abort the transaction, in line 8) or to commit (line 13). In case of a rollback, the transaction can simply be aborted on the master. No further action is required. If the client wishes to commit, then the master commits the transaction on the master database DB, extracts the encoded (compressed) writeset and forwards it to all satellites that keep a copy of the database. Also, the master atomically updates the $CN(DB)$ value. The handling of read-only transactions is described in lines 39-62. Basically, such transactions are always tagged with $CN(DB)$ and re-routed if possible. If no copies are available, then the transaction has to be executed on the master.

Algorithms 1 and 2 describe the handling of read-only transactions and writesets on the satellites. Line 4 in algorithm 2 is where we make sure that read-only transactions never observe stale data. Transactions re-routed to satellites are always started in read-only mode. If a client inside a declared read-only transaction tries to update database elements, then the underlying database will automatically abort the transaction.

3 Implementation

The current implementation of DBFarm runs on top of PostgreSQL 8.1. The core part of DBFarm are the *adapters*, which are distributed middleware components used to integrate the concepts of the previous section (see Figure 2). As the adapters make up the main component of our implementation we are going to describe them first.

3.1 The Adapter Approach

Clients access the DBFarm by establishing a connection to an adapter at a master server. If the requested database instance is not locally hosted as a master database, then the connection is transparently forwarded to the correct master. Database copies hosted on satellites are not directly accessible by the clients. In Figure 2 the adapter on the master (the *master adapter*) intercepts all incoming client connections and re-routes read-only transactions to satellites. The actual routing is more fine grained, since clients never send transactions as a block. The master adapter therefore needs to inspect the data stream on each client connection and handle operations accordingly. To be able to identify read-only transactions the master adapter assumes that clients use the standard SQL mechanisms to either declare the whole session as read-only or decorate the begin operation of submitted read-only transactions with the *READ ONLY* attribute. In Java clients, for example, this can be forced by executing the *Connection.setReadOnly()* method. If a client does not give any information, then the adapter assumes that the client is starting an update transaction.

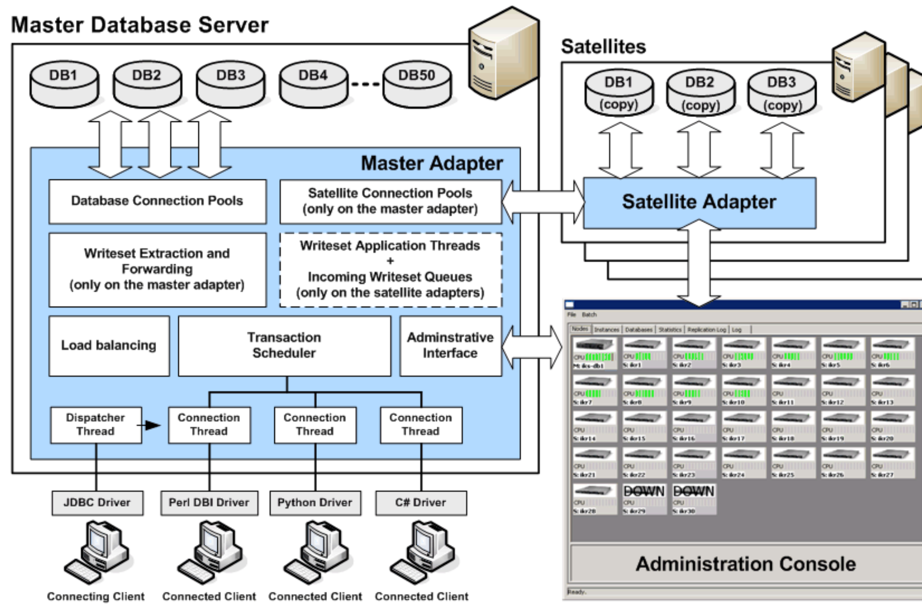


Fig. 2. Adapters in an example DBFarm with one Master Server and a set of Satellites.

The master adapter also extracts the latest changes produced by each update operation (the *writesets*) from the master databases and sends them to the corresponding satellite adapters. In contrast to [6] we use writesets rather than the original SQL update statements since it has been shown that they are a more efficient way to propagate changes in replicated systems [14]. The writesets consist of a compact, encoded description of the tuples that need to be inserted, changed or deleted. At the target adapter, they are translated and executed as a minimal set of SQL statements. The extraction of the writeset for a given transaction occurs after the transaction commits and it is done for the entire transaction. Thus, when the master adapter propagates changes, it does so for all the changes of a given transaction.

Adapters on the satellites (*satellite adapters*) receive operations from read-only transactions and execute them on the locally installed copies making sure that consistency is preserved. To maintain consistency, satellite adapters constantly apply the received writesets and respect the tags at the beginning of re-routed read-only transactions.

Upon startup, each adapter configures and starts the local PostgreSQL installation (each PostgreSQL installation typically contains several master databases or satellite copies). All needed PostgreSQL configuration files are then dynamically generated so that the PostgreSQL database software only binds itself to the local *loopback* network interface. As a result, all PostgreSQL installations are not accessible by clients directly from the network. After the local PostgreSQL installation is up, the adapter connects to it and scans for installed databases. As a last step, the adapter starts its own listener on the external network interface. However, unless an adapter has been further configured by the administration console, it denies all client requests - until then it is simply not aware if it is running as a master or satellite. Information about the overall DBFarm

configuration is at this stage centralized and provided by the administration console. Only after a master has been informed by the administration console about its mode and about other masters and the available satellite adapters (and the database copies that they host) it can establish all needed writeset and transaction re-routing connections and is then ready to process client transactions.

Currently, only the master adapters distribute read-only transactions across the satellites, for load-balancing purposes a round-robin assignment is used. At a later point of time, we plan to explore more sophisticated assignments to improve overall performance. The need for more sophisticated assignments arises from the fact that some read-only transactions may take a long time to execute (hours in some cases). If the scheduling would take load into consideration, it would do a better job distributing the incoming transactions. Nevertheless, for the purposes of demonstrating the characteristics of DBFarm, round robin scheduling suffices.

For efficiency reasons, connections from the master to other adapters are organized in pools: for each local database and for each known peer adapter there is a connection pool. The reason for using a pool for each local database (instead of one pool for the whole PostgreSQL installation) lies in a limitation of the PostgreSQL on-wire protocol: one cannot switch the selected schema and database user after a connection has been established and authenticated. Connections to other adapters are mainly used for two purposes: first, to send tagged read-only transactions and second, to stream writesets to the satellites. Connections are generated lazily; once a connection is no longer used, it will be put back into its pool. Pooled connections that are not used for a certain period of time will be closed and removed.

The adapters have been implemented as a thin layer of Java software. The software for the master and satellite adapters is the same, however, depending on its configuration an adapter either acts as the master or a satellite. The advantage of having identical adapters at both master and satellites is that it will eventually also allow us to move a master to one of the satellite machines. This makes DBFarm more dynamic but also changes the properties of the resulting architecture since the satellites are, in principle, unreliable. For reasons of space we do not further pursue such an approach in this paper but use the idea to emphasize the flexibility that the DBFarm architecture provides.

3.2 Assuring Consistency

Following up on our previous work on consistent database replication [22, 23, 24], we use snapshot isolation (SI) [3, 25] as correctness criteria for the master and satellite databases. Common database products that make use of SI are Oracle, PostgreSQL and Microsoft SQL Server 2005. SI is used to prevent complex read operations from conflicting with updates and viceversa. The way it works is by giving every transaction a snapshot of the database at the time it starts (the snapshot contains all committed changes up to that point). Since each transaction works on a different snapshot, conflicts between concurrent reads and writes do not occur. In the original definition of SI, the check for conflicts between transactions that perform updates is only done at commit time: if concurrent transactions try to modify a common item, the *first committer wins rule* is applied (the first one to commit succeeds, all others will be aborted). However, real implementations all rely on more efficient, incremental conflict detection methods. Read-only transactions are not checked for conflicts. SI avoids the four extended ANSI

SQL phenomena as described in [3] (which is a prerequisite for an implementation of a SERIALIZABLE isolation level). However, one has to be aware that this is not the same as the classic definition of conflict serializability, e.g., as given in [4]. Fortunately, this does not impose problems in real applications, e.g., [11] has shown that transactions can be re-structured so that running them in SI based databases leads to serializable executions.

Using SI makes it relatively simple to implement consistency requirements by DB-Farm to provide clients with a consistent view. Since a read-only transaction is executed in a copy using SI and the copy will provide consistent snapshots, a read-only transaction will always read a snapshot that has existed in the master database. This has important practical advantages since it allows a copy to constantly keep applying updates without having to abort or interfere with concurrently running read-only transactions from the clients.

3.3 PostgreSQL Frontend

In the current implementation of DBFarm a master adapter, as seen from the client side, looks like a normal PostgreSQL installation (the adapter listens on TCP port 5432). We have implemented server and client-side support for the low-level PostgreSQL protocol. The server side is used to implement the PostgreSQL frontend end, the client side is used to communicate with the locally installed PostgreSQL databases. When routing transactions between different adapters, the adapters use a slightly extended variant of the PostgreSQL client/server protocol (e.g., it is possible to switch the database schema/user for the current connection and transactions can be tagged with commit numbers; in addition, the mechanism for transportation of writesets was added).

Since master adapters implement the standard PostgreSQL server interface, DB-Farm can be used by a plethora of application types and platforms: C, C++, Java, Perl, Phyton, .NET (with the PostgreSQL ADO provider), etc. - actually every application that is designed to be used with PostgreSQL. In fact, it can be used with already existing applications without requiring any changes as long as those applications use the standard PostgreSQL interface. This is in sharp contrast to other replication proposals, where clients need to be changed (or need special drivers) to be able to use the database system (e.g., [2, 6, 17]).

3.4 Writeset Extraction

We have implemented and tested different variants of writeset extraction. Currently, DBFarm supports two approaches. The basic, generic approach, is similar to what is being done in other systems (e.g., [6]): we simply collect all DML (data modification language) statements of transactions in the master adapter and use them as writeset. This method was only used for testing, since it has many open problems. For instance, one cannot handle updates that have been produced by triggers, since those are not visible to the adapter. Also, there are problems with statements that instruct the database to insert function based values into tuples (e.g., random numbers or the current time in milliseconds, which will, obviously, not lead to the same result when being executed on different cluster nodes).

The second approach is based on triggers: we have implemented a shared library which can be loaded into PostgreSQL at run-time. The library contains functions which

will then be assigned by the master adapters as triggers to all tables that need replication. Whenever there is a change on a table, then our trigger function will capture the new values - no matter if the change was directly provoked by the user or by a stored procedure inside the database. The writeset is then simply collected in memory. At the end of a transaction a master adapter can then call another function in the shared library to extract the writeset. This is very fast, since writeset collection does not involve any disk accesses. Our implementation is also able to capture schema changes due to DDL (data definition language) commands (e.g., table and index creation) and to produce special writesets which lead to the corresponding changes of the database schema on the copies.

To keep different database copies synchronized, other replication systems often replay the server's complete redo-log on the replicas - e.g., [29] have implemented this for PostgreSQL. Unfortunately, such an approach only works for very simple setups, where each replicated PostgreSQL installation has the same content and page layout. This considerably reduces the flexibility. In DBFarm, the source and destination PostgreSQL installations may contain different sets of databases, and therefore we need to extract and apply writesets per database. Currently, we are working on an approach where it is possible to extract the redo-information for a subset of the databases in a PostgreSQL installation.

3.5 Administration Console

The administration console has been implemented as a platform independent graphical Java application. It is used to remotely start, stop and configure the adapters. Furthermore it helps to inspect the state of each cluster node. All communication between the administration console and the DBFarm cluster nodes is encrypted. What we require is that each node runs an OpenSSH [19] daemon. To be able to use the SSH-2 protocol from within Java, we use our own open source, pure Java SSH-2 client library [21].

4 Performance Evaluation

In general our approach makes no restrictions on how resources can be shared across different databases. However, in this paper we only evaluate the performance of static setups where a single powerful server hosts all master databases and a set of smaller, less reliable satellites are used to host the read-only copies. We present the results from our experiments involving a DBFarm deployment that uses 360 customer databases on a master database server and up to 30 additional satellite machines to offer improved performance for clients.

To produce realistic measurements, we used database setups based on two different standard benchmarks: TPC-W (as defined by the Transaction Processing Council [28]) and RUBBoS (defined by the Object Web Consortium [26]). The TPC-W benchmark models customers that access an online book store, while RUBBoS models a bulletin board similar to the Slashdot website [27]. For TPC-W we use the default *shopping mix* workload which consists of 80% read-only interactions. The workload defined by the RUBBoS benchmark consists of 85% read-only interactions.

We installed 300 TPC-W databases (using scaling factors 100/10,000, which results in 497 MB per database) on the master server, as well as 60 RUBBoS databases (using the *extended data set*, which results in 2,440 MB disk consumption per database).

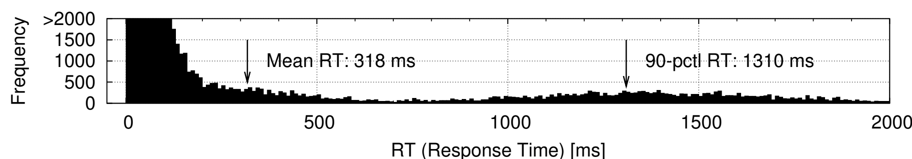


Fig. 3. A Section from an example Histogram.

The denoted sizes include all the diskspace needed for a given single database (e.g., including index files). The overall disk space occupied by these databases on the master machine exceeds 417 GB. All databases were reasonably configured with indexes.

The master database server is a dual Intel(R) Xeon CPU 3.0 GHz machine with 4 GB RAM and an attached RAID-5 (ICP-Vortex GDT8586RZ PCI controller with 5 Hitachi HDS722525VLAT80 SATA 250 GB disks) which results in 931 GB of available space (we use a XFS partition which spawns the whole RAID-5), running Fedora Core 4 (2.6.13-1.1532smp). The thirty satellite machines have dual AMD Opteron(tm) 250 processors (2.4 GHz), 4 GB RAM and a Hitachi HDS722512VLAT80 disk (120 GB). These machines run Red Hat Enterprise Linux AS release 4 (2.6.9-11.ELsmp). All machines are connected with 100MBit links over a local area network (all machines are attached to the same ethernet switch). The adapter software was run with the Java-Blackdown 1.4.2-02 JVM. We used an unmodified version of PostgreSQL 8.1 for all experiments.

To measure the performance of our setups we use a Java based loadclient software that is able to reproduce the database loads that are generated by the TPC-W and RUBBoS benchmarks. One has to emphasize that we are not running the entire benchmarks, but only the database part to stress DBFarm (e.g., a full TPC-W implementation would also have to measure the performance of the used web- and application servers).

The loadclient uses worker threads to simulate a number of clients. On startup, the loadclient generates a pool of connections to the target databases on the master. If the number of workers is less than the number of target databases, then for each database one connection is put into the pool. Otherwise, the loadclient generates connections to the target databases in a round-robin fashion, until the amount of connections is as large as the set of worker threads. Also, for each connection per target database there is a state machine that dictates the next transaction type to be executed. Each worker thread, in an endless loop, randomly chooses a connection from the pool and executes a transaction according to the connection's state machine. After the transaction has been executed, the worker puts the connection back into the pool. It is important to note that between the executions of the different transactions the workers use no thinking time - each worker is intended to stress the tested setup as hard as possible.

When benchmarking the system, the loadclient uses varying numbers of workers. Whenever the number of workers changes, the loadclient uses a warm up time of several minutes until the system is stable. Then, a benchmarking phase of two minutes follows. During the benchmarking phase, the loadclient measures the response times for all executed transactions. At the end of a run it reports the mean response time as well as the 90-percentile response time. The data points in the following figures represent such runs.

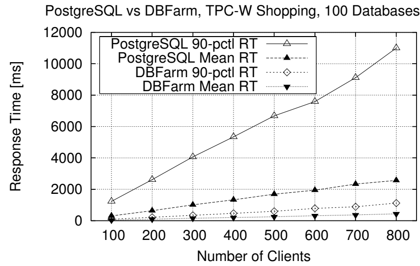


Fig. 4. TPC-W Results for 100 Databases.

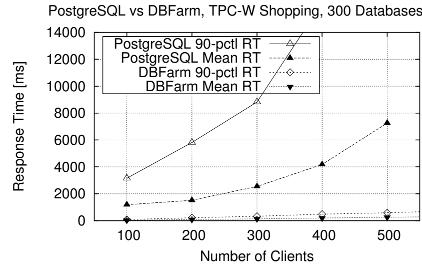


Fig. 5. TPC-W Results for 300 Databases.

To be able to calculate the 90-percentile response time, the loadclient keeps an internal histogram for each experiment. Please refer to Figure 3 for an example. The figure only shows a section of the overall collected historical data - internally the loadclient keeps track of all measured response times with a 1 ms resolution over the range from 0 to 20 seconds.

4.1 Part A: Handling Many Concurrent Databases

In the following experiments we show that DBFarm is able to handle situations where many databases are being accessed concurrently. We use a set of satellites to execute expensive read-only transactions, and therefore we can reduce the number of page fetches on the master. At the same time, more resources are available for update transactions on the master. In the experiments we use a simple satellite setup - for each database on the master we created only one satellite copy. These copies were then evenly spread over the available satellites.

Results for TPC-W: In these experiments we compared the achievable performance for a large amount of TPC-W databases that are accessed concurrently. First, we used the loadclient to stress the master alone. Then, we put the DBFarm system in place and measured the performance again.

In the first experimental round we used 100 concurrent TPC-W databases (Figure 4). One can observe that the master server already is at its limits with 300 concurrent TPC-W shopping workers, since a mean response time of 1 second and a 90-percentile response time of almost 4 seconds is probably not acceptable for most interactive applications.

By applying the same load to DBFarm with 10 attached satellites (each containing 10 database copies), one can observe that the system is able to scale-up to a much higher number of concurrent clients while at the same time giving acceptable response times. These results are particularly telling since there is only one copy of each master database. Performance could be improved even more by adding more satellite machines and having 2 copies for each master database.

With the DBFarm setup, each satellite hosts 10 TPC-W database copies. This makes up a data set of about 5 GB that has to be handled by each satellite. Due to the fact that the TPC-W workloads mainly access hot-spot data (e.g., queries for the best seller books in the store), the 4 GB of main memory on each satellite is sufficient to keep the number of disk accesses low. Also the update transactions that appear in the TPC-W

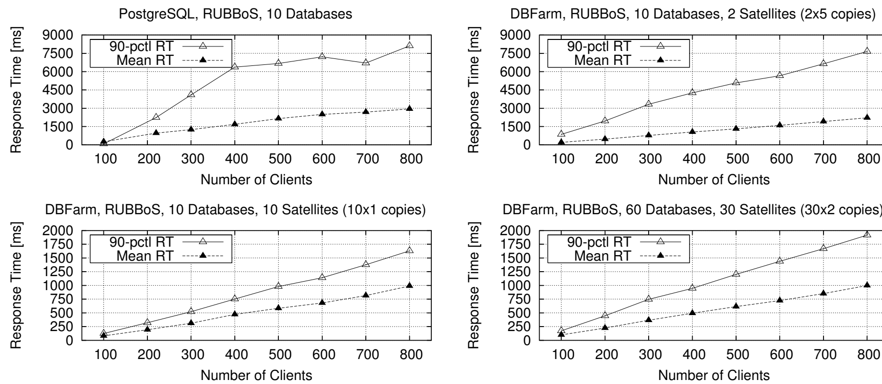


Fig. 6. RUBBoS Results.

workload (mainly operating on the customer's shopping cart and placing new orders) need only on a small fraction of the data in each database. Therefore, the master server in the DBFarm setup (executing only update-transactions) can easily handle update-transactions for all accessed databases with the in-memory buffer cache. Most of its disk accesses are related to writing the latest changes to disk - this is also true for the satellites, which, after the warm-up phase, mainly access their disks to commit the latest received writesets.

Encouraged by the good results for 100 concurrent databases we also tried to handle 300 TPC-W databases. The results are given in Figure 5. Clearly, without the DBFarm approach the load of 300 concurrently accessed TPC-W databases is too much for our master server, the response times are not acceptable. Due to the fact that the machine is mainly doing disk I/O, the results are rather unstable - performance is dictated by the RAID-5 controller and seek times of the attached disks.

The same experiment over a DBFarm setup with the same 300 databases using thirty satellites (each holding 10 database copies) shows that DBFarm is able to handle the load and to offer acceptable response times for such a scenario.

Results for RUBBoS: The RUBBoS databases are not only larger (each database is over 2 GB) than the benchmarked TPC-W databases, but the used workload is also more complex, as the resulting transactions not only use hot-spot data but also touch a wide range of tuples inside the databases.

As before, we first measured the performance of the master server alone. The results are given in Figure 6. The results show that the master machine alone cannot handle many RUBBoS databases concurrently, already 10 databases lead to performance problems as shown by the large 90-percentile results.

By looking at the results for the DBFarm setup, one can observe that it is crucial that the number of copies on each satellite does not exceed a certain threshold. In the experiment where we used only two satellites (each containing 5 RUBBoS database copies) the performance improvement over a single master machine is insignificant. This is due to the fact that DBFarm has a similar problem as a single master server: the working set of 5 databases is too big for the available memory, and therefore the

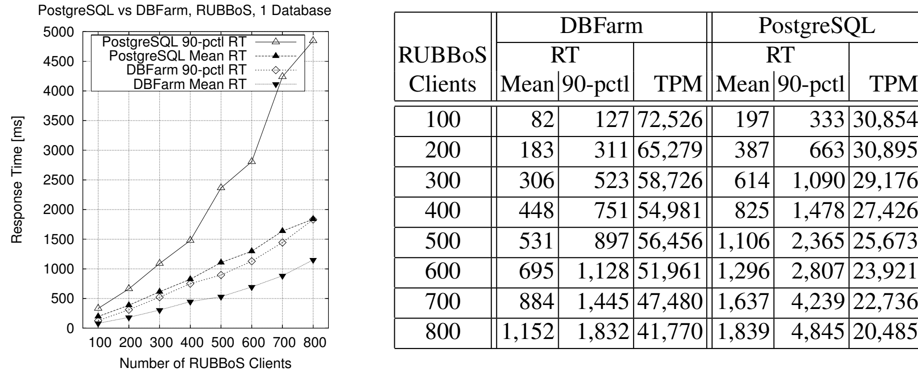
throughput on the satellites is limited by the available disk I/O-bandwidth. One can interpret the result as having moved the bottleneck from the master database server to the satellites. This may look like a waste of resources, but one should keep in mind that the overall setup has improved: by taking away load from the master, there is more available capacity for other concurrently accessed databases. We will point out this feature in the next section. To verify that the satellites are really the bottleneck, we then tested the same workload on a DBFarm setup with 10 satellites, therefore having only 1 RUBBoS copy per satellite. One can observe how the performance significantly improves over a setup with only 2 satellites.

In a last experiment set we tried to handle 60 concurrent RUBBoS databases with DBFarm. We used a setup with 30 satellites, each holding two database copies. It was impossible to perform the same experiment with a single master server, as the machine was stuck with disk-I/O and no stable results could be achieved (the throughput never got higher than a few transactions per second). The results in Figure 6 show that, again, DBFarm can handle such a scenario. Interestingly, the achieved performance is slightly lower than for the experiment based on 10 copies on 10 satellites. There are two reasons: first, by having two RUBBoS copies on each satellite, the buffer cache of the satellites is not big enough two hold both databases in memory. However, this is only a minor problem, as could be verified by observing the number of disk reads on the satellites during the experiment. Second, with 60 concurrent RUBBoS databases, the master server is becoming a bottleneck, since the data needed for each update transaction is not always in memory (the machine was performing more read operations than in the 10 databases experiment). One can learn the following from this experiment: when using a system like DBFarm, it is very important to optimize the structure of update-transactions: one has to try to keep the number of read operations (e.g., select operations or index scans for update statements) small, otherwise the master databases become the bottleneck of the system. This can, e.g., be achieved by introducing appropriate indexes specific to master databases.

4.2 Part B: Scaleout for selected Databases

In the preceding experiments we used database copies on satellites to extend the capacity of a master database server. In all setups, we used no more than one copy per database on the master server. We could show that with such a system setup we can handle bursts over a set of databases.

In the last experiment we show how a single RUBBoS database can benefit from the DBFarm approach. For instance, one could think of having a high priority customer database that needs a certain guaranteed response time since there may be a service level contract with the customer. The approach to solve the problem is to assign a set of satellites, each holding exactly and exclusively one copy of the customer's database on the master server. In this way, the customer's read-only transactions can be load-balanced over different satellites which are at the same time guaranteed not to be affected by other customers. Again, there are two measurements: first we measured the performance of a pure PostgreSQL installation on the master server, then we measured the performance of the DBFarm setup. However, this time the load for the DBFarm was made much harder: to make things more interesting, in parallel to the RUBBoS load 200 TPC-W databases were also loaded by 100 worker threads with the shopping-mix. Copies of the



RUBBoS Clients	DBFarm			PostgreSQL		
	RT			RT		
	Mean	90-pctl	TPM	Mean	90-pctl	TPM
100	82	127	72,526	197	333	30,854
200	183	311	65,279	387	663	30,895
300	306	523	58,726	614	1,090	29,176
400	448	751	54,981	825	1,478	27,426
500	531	897	56,456	1,106	2,365	25,673
600	695	1,128	51,961	1,296	2,807	23,921
700	884	1,445	47,480	1,637	4,239	22,736
800	1,152	1,832	41,770	1,839	4,845	20,485

Fig. 7. Detailed Scale-Out Results for the RUBBoS Database. Note that DBFarm had to handle simultaneously 100 clients that randomly accessed 200 TPC-W databases (not included in TPM).

200 TPC-W databases were located on 20 separate satellites (each holding 10 copies). In case of the RUBBoS database, we used 3 satellites each holding exactly one copy. The results for the two experiments are given in Figure 7. Clearly one can observe that the high priority RUBBoS customer database is performing much better than with the single server setup - even though DBFarm has to concurrently deal with 200 TPC-W databases. The detailed results show that the throughput (given in *TPM*, transactions per minute) for the RUBBoS database has more than doubled.

5 Related Work

DBFarm builds upon the ideas developed in several previous projects in our group [15, 16, 22, 23] as well as on a wealth of related work on middleware based database replication. In [22, 23] we presented a system for replicating single database instances using snapshot isolation. The current version of DBFarm uses that implementation for providing snapshot isolation consistency to the clients.

On the theoretical side, [8] has extensively studied the problem of session consistency as a more meaningful correctness criterion for replicated databases than standard 1-copy-serializability. Their algorithms are targeted at offering serializability for a single, fully replicated database and they have so far only simulated the algorithms they describe. DBFarm offers a stronger notion of consistency (not only one's own updates but all updates until a certain timestamp) using mechanisms that should not result in a loss of performance when compared to those presented in [8]. [9] proposes *generalized snapshot isolation*, a technique for replicated databases where readers may use older snapshots. Again, our system offers scale-out without giving up consistent views for all clients. In our current implementation, we also use snapshot isolation as a concurrency control mechanism. [11] investigated research in the serializability aspects of snapshot isolation. The consistency guarantees of systems that allow the use of other concurrency control mechanisms in parallel to snapshot isolation have been investigated in [10], these results directly apply to our system, as we are able to mix different concurrency control mechanisms on the different database nodes.

In terms of implemented systems, [1] applies the techniques presented in [22] to provide a travel-in-time feature where clients can requests older snapshots. Although

this technique can easily be implemented in DBFarm, the goal of DBFarm is to support full consistency and On Line Transaction Processing (OLTP) loads ([1] uses TPC-R as benchmark, a data mining load). Note that once consistency is relaxed, scalability can be significantly increased (and, in fact, the concept of scalability changes since clients are accessing historical rather than actual data). The work described in [2] centers around a technique called distributed versioning. The key idea is to use a centralized middleware based scheduler which does bookkeeping of *versions* of tables in all the replicas. Every transaction that updates a table increases the corresponding version number. At the beginning of every transaction, clients have to inform the scheduler about the tables they are going to access. The scheduler then uses this information to assign versions of tables to the transactions. Our time tagging of transactions resembles the per table versioning of [2] but ours introduces clearly less overhead as it does not require any parsing of statements nor schema information at the middleware layer. C-JDBC [6], an open source database cluster middleware, has been primarily designed for fault tolerance. To be able to access a C-JDBC cluster, clients need to use a special Java JDBC driver. The system implements variants of the *Read-One Write-All* approach with consistency guaranteed through table level locking at the middleware level. The backend databases are accessed over JDBC, so the system can be used with different database implementations, they only need to provide a JDBC interface. The downside of this approach is the need for duplicating logic from the backend databases into the middleware, since JDBC does not supply mechanisms to achieve a fine grained control over an attached database. One example for this is *locking*, which, again, has to be done at the middleware level by parsing the incoming statements and then doing table-level locking. Another example is the writesets, which are not supported by the JDBC standard, so the middleware has to broadcast SQL update statements to all replicas to keep them in-sync. Also, when encountering peaks of updates, this leads to a situation where every backend database has to evaluate the same update statements. To circumvent these scalability problems, C-JDBC offers also the partition of the data on the backend replicas in various ways (called *RAIDb-levels*, in analogy to the RAID concept). However, static partitions of data restrict the queries that can be executed at every node. Like the solution in [2], C-JDBC cannot be used in the context of DBFarm because of the overhead it introduces at the middleware level it does not scale to hundreds of database instances. [7] presents a replication architecture based on partial replication and refresh transactions. To offer consistent views for readers, the system relies on the ordering properties of global FIFO multicast of the underlying network and as well as on maximum message delivery times.

There are also a number of systems that use group communication to implement single instance database replication [15,16,17]. These systems do not consider the problem of load balancing (they assume clients distribute themselves evenly across all copies) and impose severe restrictions on the transactional load. For instance, they require that transactions are submitted as a single block since the system can only reason about complete transactions. This is in contrast to DBFarm where clients can submit transactions statement by statement as it is done in most database applications. From the point of view of clustered databases with multiple instances, the biggest drawback of group communication based replication is the high overhead of group communication itself.

With hundreds of database instances and several copies of each, the number of messages to be handled by the group communication system can be very high. Also, maintaining a membership group for each instance is very expensive and limits the flexibility in allocating copies to satellites. Since these system also adopt an update everywhere approach, each database copy must also duplicate application logic in addition to data (triggers, user defined functions, etc.). In the context of DBFarm this is simply not practical. Finally, group communication primitives rely on all nodes involved making suitable progress at roughly the same pace. In DBFarm, where a node may contain a potentially large amount of database instances, such forced synchronization will make it impossible for the system to scale. The approach proposed in [13] and [18] where load is partitioned using conflict classes is also not feasible in the context of multiple instances.

Oracle RAC (Real Application Clusters) is a commercial clustering solution that also uses snapshot isolation. It relies on the use of special hardware (all nodes in the database cluster need access to a set of shared disks) or the use of special network file systems. Therefore, unlike our approach, the system cannot easily be installed on a set of commodity servers.

6 Conclusions

This paper presents the architecture and implementation of DBFarm, a multi-instance database cluster solution that can handle hundreds of client databases concurrently. Additionally, it supports controlled scale-out for selected customer databases. DBFarm offers consistency at all times, to the clients it looks like an ordinary database server. There is no need to change any client code to be able to use the system. Our light weight adapter approach offers many advantages over classic middleware based replication solutions. Our experiments show that the approach is feasible and that the system can efficiently schedule transactions for relatively large amounts of customer databases while offering good performance for large sets of concurrent clients.

Our future work will concentrate on the dynamic aspects of the system. By allocating satellites and establishing database copies as demand requires, we plan to build an autonomic database service provider.

References

1. F. Akal, C. Türker, H.-J. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 565–576.
2. C. Amza, A. L. Cox, and W. Zwaenepoel. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 230–241.
3. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1–10, May 1995.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
5. E. Cecchet. C-JDBC: a Middleware Framework for Database Clustering. IEEE Data Engineering Bulletin, Vol. 27, No. 2, June 2004.
6. E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference, FREENIX Track*, pages 9–18, 2004.

7. C. Coulon, E. Pacitti, and P. Valduriez. Consistency Management for Partial Replication in a High Performance Database Cluster. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS 2005), Fuduoka, Japan, July 20-22, 2005*.
8. K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004), 30 March - 2 April 2004, Boston, MA, USA*, pages 424–435.
9. S. Elnikety, F. Pedone, and W. Zwaenepoel. Database Replication Using Generalized Snapshot Isolation. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*.
10. A. Fekete. Allocating Isolation Levels to Transactions. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 206–215, June 2005.
11. A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.
12. R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness. In *21st IEEE Int. Conf. on Reliable Distributed Systems (SRDS 2002), Oct. 2002, Osaka, Japan*, pages 150–159.
13. R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *IEEE 22nd Int. Conf. on Distributed Computing Systems, ICDCS'02, Vienna, Austria*, pages 477–484, July 2002.
14. B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Diss. ETH No. 13864, Dept. of Computer Science, Swiss Federal Institute of Technology Zurich, 2000.
15. B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Databases, 2000*.
16. B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.
17. Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430.
18. J. M. Milan-Franco, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive Distributed Middleware for Data Replication. In *Middleware 2004, ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, Canada, October 18-22, Proceedings*, 2004.
19. OpenSSH. A free version of the SSH protocol suite. http://www.openssh.org/.
20. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.
21. C. Plattner. The Ganymed SSH-2 Library. http://www.ganymed.ethz.ch/ssh2.
22. C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Middleware 2004, 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-22, Proceedings*, 2004.
23. C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with Satellite Databases. *Accepted for Publication in the VLDB Journal*, 2006. http://www.iks.inf.ethz.ch/publications/satellites.html.
24. C. Plattner, A. Wapf, and G. Alonso. Searching in Time. In *SIGMOD '06, 2006 ACM SIGMOD International Conference on Management of Data, June 27-29, 2006, Chicago, Illinois, USA*, 2006.
25. R. Schenkel and G. Weikum. Integrating Snapshot Isolation into Transactional Federation. In *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*.
26. The ObjectWeb Consortium. RUBBoS: Bulletin Board Benchmark. http://jmob.objectweb.org/rubbos.html.
27. The Slashdot Homepage. http://slashdot.org/.
28. The Transaction Processing Performance Council. TPC-W, a Transactional Web E-Commerce Benchmark. TPC-C, an On-line Transaction Processing Benchmark. http://www.tpc.org.
29. S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 422–433.