The VLDB Journal
DOI 10.1007/s00778-006-0026-x

REGULAR PAPER

Extending DBMSs with satellite databases

Christian Plattner · Gustavo Alonso · M. Tamer Özsu

Received: 17 July 2005

Abstract In this paper, we propose an extensible architecture for database engines where satellite databases are used to scale out and implement additional functionality for a centralized database engine. The architecture uses a middleware layer that offers consistent views and a single system image over a cluster of machines with database engines. One of these engines acts as a master copy while the others are read-only snapshots which we call satellites. The satellites are lightweight DBMSs used for scalability and to provide functionality difficult or expensive to implement in the main engine. Our approach also supports the dynamic creation of satellites to be able to autonomously adapt to varying loads. The paper presents the architecture, discusses the research problems it raises, and validates its feasibility with extensive experimental results.

Keywords Satellite databases · Snapshot isolation · Extending database functionality · Dynamic satellite creation

Part of this work was done while Tamer Özsu was a visiting guest at ETH Zurich.

C. Plattner (✉) · G. Alonso
ETH Zurich, 8092 Zurich, Switzerland
e-mail: plattner@inf.ethz.ch

G. Alonso
e-mail: alonso@inf.ethz.ch

M. Tamer Özsu
David R. Cheriton School of Computer Science,
University of Waterloo, Waterloo, Canada
e-mail: tozsu@uwaterloo.ca

1 Introduction

Databases are facing important challenges in terms of functionality, scalability, and extensibility. This is particularly true for many new applications: genomics, sensor networks, mobile objects, etc. From our experience with data repositories for astrophysics [33], we find that a big obstacle in using databases within the information infrastructure for these novel applications is that it is difficult to extend a database engine. Also, many of these new applications require a degree of scalability difficult to provide with centralized engines.

To address this problem, in this paper we describe Ganymed, a novel architecture for extending and opening up database engines. Ganymed is based on what we call *satellite databases*. The idea is to extend a centralized database engine with additional processing capacity (by offloading queries to satellites) or additional functionality (implemented as external data blades in the satellites) while maintaining a single system image and avoiding adding extra load to the main database engine.

1.1 Extensibility

An example of a Ganymed-based system is shown in Fig. 1. The figure shows several lightweight satellite DBMSs that contain data from two independent master DBMSs. Each satellite is assigned to exactly one master. These satellites are then used to extend the masters for performance purposes (by replicating data on the satellites; in the paper we discuss how to do this using full replication) or for functionality purposes (by implementing functionality at the satellites that is not available at the masters; we provide results for skyline queries and text keyword search). Any given satellite is

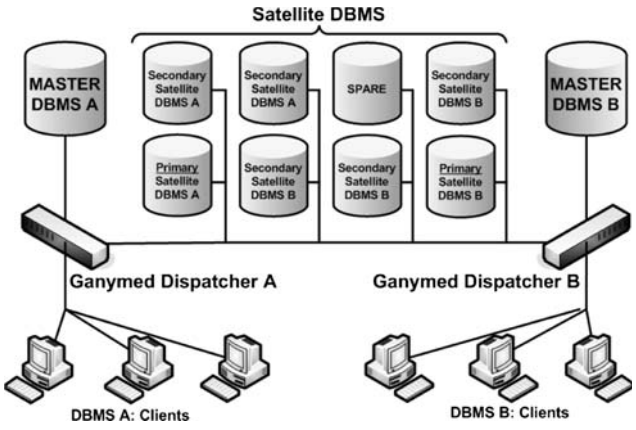

Fig. 1 Example of a Ganymed system configuration

kept consistent with its master at all times, using a lazy replication scheme. Transaction dispatchers make sure that clients always see a consistent state [we use snapshot isolation (SI) as correctness criteria] and that the system enforces strong consistency guarantees (a client always sees its own updates and it is guaranteed to work with the latest snapshot available). Finally, satellites do not necessarily need to be permanently attached to a master. In the paper we explore how to create satellites dynamically (by using spare machines in a pool) to react to changes in load or load patterns. This opens up the possibility of offering database storage as a remote service as part of a data grid.

1.2 Consistency

A key difference between our system and existing replication solutions (open source and commercial) is that clients always see correct data according to SI [34]. SI, implemented in, e.g., Oracle [27], PostgreSQL [29], and Microsoft SQL Server 2005 [25], avoids the ANSI SQL phenomena P1–P4 [7]. It also eliminates conflicts between reads and writes. Read-only transactions get a snapshot of the database (a version) as of the time the transaction starts. From then on, they run unaffected by writers.

For correctness purposes, we differentiate between *update* (there is at least one update operation in the transaction) and *read-only* transactions (or queries). Transactions do not have to be sent by clients as a block (unlike in, e.g., [19,26]), they can be processed statement by statement. Update transactions are forwarded to the master. Read-only transactions will, whenever possible, be sent to one of the satellites and executed with SI. The same snapshot (on the same satellite) is used for all statements in the transaction, thereby ensuring that a client's read-only transactions always see a consistent state of the database over their whole execution time.

Queries are executed only after the assigned satellite has applied all the master's updates up to the time the transaction starts. Hence, a client always sees its own updates and all the snapshots it sees are consistent in time. To enforce this requirement on the satellites, we resort to transaction *tagging*. Obviously, if updates do not get applied fast enough on the satellites, then readers must be delayed until the necessary snapshot becomes available.

1.3 Validation

The system described in the paper has been fully implemented. We describe here only a proof of concept implementation that demonstrates the feasibility of the idea. We do not claim that ours is the most optimal implementation. We also do not claim that we have solved all problems and issues that arise in this context. Rather, the objective is to show the potential of our architecture by analyzing their behavior under a variety of contexts and applications.

For the experiments performed, there is an overwhelming number of design variables to consider. There are also many product-specific tools, interfaces, and solutions that would not only significantly boost the performance of our system but also detract from its generality. Consequently, in the paper we use only generic solutions instead of product-specific optimizations. This is of particular relevance in regard to update extraction from the master, application of updates at the satellites, and the dynamic creation of copies, all of which can be performed more efficiently by using product-specific solutions.

Also note that we discuss two potential applications: full replication satellites and specialized satellites. Implementing full replication satellites across heterogeneous databases is very complex. However, most of the techniques necessary to implement full replication satellites also apply to specialized satellites. We use full replication satellites as a worst case scenario for analyzing our system. The lessons learned from this experience help in implementing specialized satellites which may prove to be the most interesting application. The advantage of specialized satellites is that most of the difficulties involved in creating and operating satellites disappear. Thus, specialized satellites may not only be more useful, they are also easier to implement. Yet, the purpose of the paper is to explore the entire design space so we include the full replication experiments to show the capabilities and limitations of the idea we propose.

Note that none of the middleware-based replication solutions recently proposed ([5,10,23,26]) can cope with partial replication, let alone specialized satellites.

1.4 Contributions

In this paper, we propose a novel replication architecture to extend databases. We also propose novel execution strategies for some applications over this architecture. The main contributions of the architecture and the paper are:

- The use of satellites acting as external data blades for database engines, in homogeneous and heterogeneous settings.
- We show how satellites can used to extend database engines with conventional replication and with *specialized functionality that may not be available in the original master DBMS*. This external data blade approach allows the addition of extensions without modification of the original database engine and without affecting its performance.
- We provide clients with a consistent view of the database using a technique that we have introduced in an earlier paper [28]. We are not aware of any implemented solution, open source or commercial, that provides the consistency levels we provide (much less with the performance our system exhibits).
- We support dynamic creation of satellites. This operation is intrinsically very expensive for full replication but can be used to great effect with specialized satellites that only need a fraction of the master's data to be created.

1.5 Paper organization

The rest of this paper is organized as follows: first, we describe the Ganymed system architecture (Sect. 2). This is followed by a description of the details of SI and the system's underlying transaction routing algorithm (Sect. 3). In the following Sects. 4 and 5 we describe the problems that have to be handled in heterogeneous setups. Then, in Sect. 6, we describe the experimental evaluation of our system.

To demonstrate the more advanced possibilities of the system, namely, specialized satellites, we then show in Sect. 7 how we used PostgreSQL satellites to extend an Oracle master with a skyline query facility [9,11]. As another possibility, we also show how we used the Ganymed system to implement a full text keyword search based on satellite databases. The objective of these experiments is to show how satellites can be used to implement new operators that require heavy computation without affecting the master. In Sect. 8 we then explore the behavior of the system when satellites are dynamically created. We include experiments on the cost of dynamic satellite creation (as well as a discussion on how this is done) and experiments on dynamic skyline satellites. The paper ends with a discussion of related work (Sect. 9) and conclusions.

2 System architecture

2.1 Design criteria

One of the main objectives of Ganymed is to offer database extensibility while maintaining a single system image. Clients should not be aware of load balancing, multiple copies, failing satellites, or consistency issues. We also do not want to make use of relaxed consistency models – even though we internally use a lazy replication approach, clients must always see a consistent state. Unlike existing solutions (e.g., [5,10,35]) that rely on middleware for implementing database replication, we neither depend on group communication nor implement complex versioning and concurrency control at the middleware layer.

This point is a key design criterion that sets Ganymed apart from existing systems. The objective is to have a very thin middleware layer that nevertheless offers consistency and a single system image. Unlike [23,35], we do not want to use group communication to be able to provide scalability and fast reaction to failures. Avoiding group communication also reduces the footprint of the system and the overall response time. As recent work shows [23], group communication is also no guarantee for consistency. Although not stated explicitly, the design in [23] offers a trade-off between consistency – clients must always access the same copy – and single system image – at which point clients are not guaranteed to see consistent snapshots. We also want to avoid duplicating database functionality at the middleware level. Ganymed performs neither high level concurrency control (used in [10], which typically implies a significant reduction in concurrency since it is done at the level of table locking) nor SQL parsing and version control (used in [5], both expensive operations for real loads and a bottleneck once the number of versions starts to increase and advanced functionality like materialized views is involved). The thin middleware layer is a design objective that needs to be emphasized as the redundancy is not just a matter of footprint or efficiency. We are not aware of any proposed solution that duplicates database functionality (be it locking, concurrency control, or SQL parsing) that can support real database

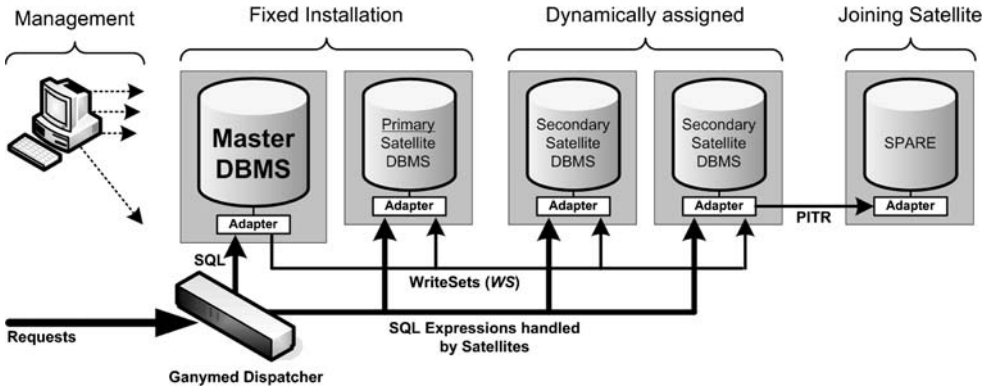

Fig. 2 Data flows between the dispatcher, the master and the satellites

engines. The problem of these designs is that they assume that the middleware layer can control everything happening inside the database engine. This is not a correct assumption (concurrency control affects more than just tables, e.g., recovery procedures, indexes). For these approaches to work correctly, functionality such as triggers, user-defined functions and views would have to be disabled or the concurrency control at the middleware level would have to work at an extremely conservative level. In the same spirit, Ganymed imposes no data organization, structuring of the load, or particular arrangements of the schema (unlike, e.g., [18]).

In terms of the DBMSs that the architecture should support, the objective is flexibility and, thus, we do not rely on engine specific functionality. The design we propose does not rely on the existence of special features or modifications to the underlying DBMS.

2.2 Overview

The system works by routing transactions through a dispatcher over a set of backend databases. For a given dispatcher, the backends consist of one master and a set of satellites. The objective is to use the satellites to extend the master according to two principles: clients see a consistent database at all times and the master can take over if the satellites cannot deal with a query.

The latter point is crucial to understanding the design of the system. In the worst case, our system behaves as a single database: the master. When in doubt, the dispatcher routes the traffic to the master. We also rely on the master to provide industrial strength (e.g., crash recovery and fault tolerance). The idea is that the satellites extend the functionality or capacity of the master but neither replace it nor implement redundant functionality. This same principle applies to the problem of replicating triggers, user-defined functions, etc. Our system is not meant to extend that functionality. Thus, transactions that involve triggers or user-defined functions are simply sent to the master for execution there.

A basic assumption we make is that we can achieve a perfect partition of the load between master and satellites. However, unlike previous work [10,26], we do not require the data to be manually partitioned across nodes. For the purposes of this paper, the loads we consider involve full replication and specialized functionality (skyline queries and keyword search). For fully replicated satellites, the master executes all write operations while the satellites execute only queries (read-only transactions). In the case of specialized functionality the satellites execute skyline queries and keyword searches, all other transactions are done at the master. We also assume that queries can be answered within a single satellite.

2.3 Main components

The main components of the system are as follows (see Fig. 2). The *dispatcher* is responsible for routing transactions to the master and satellites. It acts as front end for clients. The system is controlled and administered from a *management console*. Communication with the backend DBMSs always takes place through *adapters*, which are thin layers of software installed on the DBMS machines.

In terms of database machines, we consider three types: masters, primary satellites, and secondary satellites. Primary satellites are optional and used for dynamic creation of satellites. The purpose of primary satellites is to be able to create new satellites without hitting the master for all the data necessary to spawn a new satellite. When implementing dynamic satellites, there is always one primary satellite attached to the master. Secondary satellites are those created dynamically.[1]

[1] In Fig. 2, PITR stands for the technique we use in our prototype to dynamically create satellites. Please refer to Sect. 8.

The current Ganymed prototype, implemented in Java, does not support multiple, in parallel working dispatchers, yet it is not vulnerable to failures of the dispatcher. If a Ganymed dispatcher fails, it can immediately be replaced by a standby dispatcher. The decision for a dispatcher to be replaced by a backup has to be made by the manager component. The manager component, running on a dedicated machine, constantly monitors the system. The manager component is also responsible for reconfigurations. It is used, e.g., by the database administrator to add and remove replicas. Interaction with the manager component takes place through a graphical interface. In the following sections, we describe each component in more detail.

2.4 Transaction dispatcher

The transaction dispatcher (see Fig. 3) acts as the gateway to services offered by the master database and the satellite engines. As its interface we have implemented the PostgreSQL protocol specification.[2] This is a significant departure from existing replication projects (e.g., [5,6,10,26,28]) that rely on proprietary interfaces with limited support for programming languages and OS platforms.

The dispatcher is mainly in charge of routing transactions across the system. Routing has three main components: *assigning transactions* to the correct backend DBMS, *load balancing* if more than one DBMS is able to answer a request and *transaction tagging* to enforce consistent views of the available data.

However, before a client can send transactions, it first needs to authenticate the current session. The dispatcher has no internal user database, rather it relies on the adapters to verify locally whether the given credentials are correct. On the satellites, for simplicity, we assume that the same users exist as on the master DBMS. Access control to tables is therefore always performed locally by the backend DBMS that is executing a transaction.

In an authenticated session, assigning transactions involves deciding where to execute a given transaction. Typically, updates go to the master, queries to the satellites. Since the dispatcher needs to know if a transaction is of type update or read only, the application code has to communicate this.[3]

If an application programmer forgets to declare a read-only transaction, consistency is still guaranteed, but reduced performance will result due to the increased load on the master replica. On the satellites, all transactions will always be executed in read-only mode. Therefore, if a programmer erroneously marks an update transaction as read only, it will be aborted by the satellite's DBMS upon the execution of the first update statement.

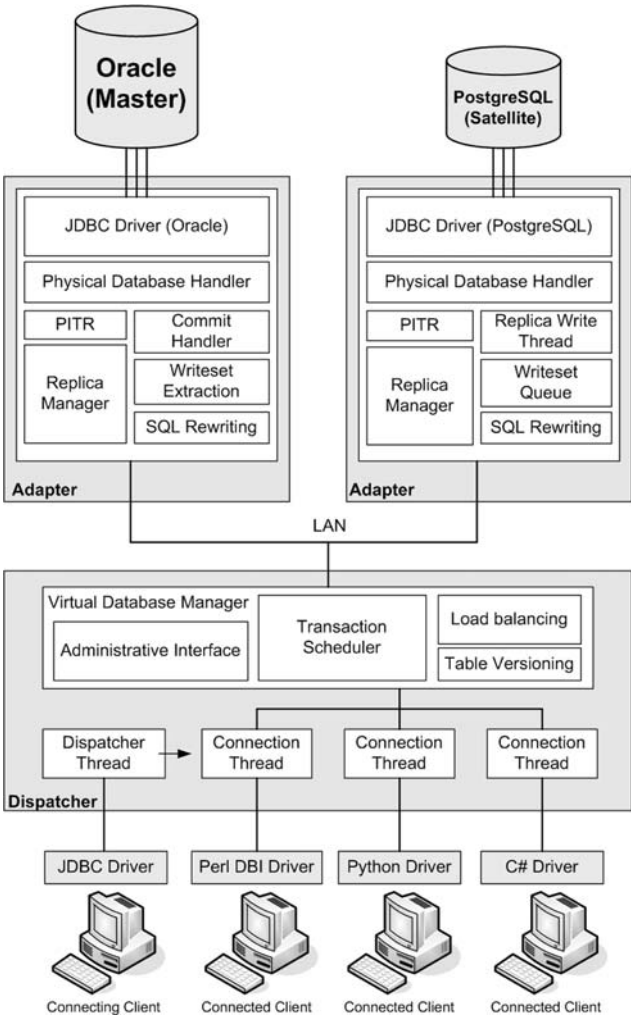

Fig. 3 Example of a dispatcher that is connected to two adapters running on an Oracle master and a PostgreSQL satellite

In the case of multiple equivalent satellites, the dispatcher also performs load balancing as a part of the routing step. Different policies can be selected by the administrator, currently *round-robin*, *least-pending-requests-first,* or *least-loaded* are offered. The latter is determined based upon load information in the meta data sent by the adapters that is piggy backed on query results (see Sect. 2.7).

As explained above, satellites always contain consistent snapshots of the master. Transactions that work on

[2] Therefore, the dispatcher can be accessed from any platform/language for which a PostgreSQL driver has been written [e.g., C#, C/C++, Java (JDBC), Perl, Python, etc.].

[3] For example, a Java client would have to use the standard JDBC *Connection.setReadonly()* method.

the master can use whatever isolation levels are offered by that database. On the satellites, however, queries are answered by using SI. To be able to produce fresh snapshots, the satellites consume *writesets* [21,28] from the master. Each produced writeset relates to an update transaction and contains all the changes performed by that transaction. Writesets are applied on the satellites in the order of the corresponding commit operations on the master, thereby ensuring that the satellites converge to the same state as the master. The application of writesets is done under SI: updates from the master and queries from the clients do not interfere in the satellites.

The adapter on the master is responsible for assigning increasing numbers (in the order of successful commit operations) to all produced writesets. Every time a transaction commits on the master, the dispatcher is informed, along with the successful `COMMIT` reply, about the number of the produced writeset. The number n of the highest produced writeset WS_n so far is then used by the dispatcher to *tag* queries when they start, before they are sent to the satellites. When a satellite starts working on a query, it must be able to assign a snapshot at least as fresh as WS_n. If such a snapshot cannot be produced, then the start of the transaction is delayed until all needed writesets have been applied. Note that transactions that are sent to the master are never tagged, since they always see the latest changes.

Since only a small amount of state information must be kept by a Ganymed dispatcher, it is even possible to construct parallel working dispatchers. This helps to improve the overall fault tolerance. In contrast to traditional eager systems, where every replica has its own scheduler that is aware of the global state, the exchange of status information between a small number of RSI-PC dispatchers can be done very efficiently. Even in the case that all dispatchers fail, it is possible to reconstruct the overall database state: a replacement dispatcher can be used and its state initialized by inspecting all available replicas.

In the case of failing satellites, a Ganymed dispatcher simply ignores them until they have been repaired by an administrator. However, in the case of a failing master, things are a little bit more complicated. By just electing a new master the problem is only halfway solved. The dispatcher must also make sure that no updates from committed transactions get lost, thereby guaranteeing ACID durability. This objective can be achieved by only sending commit notifications to clients after the writesets of update transactions have successfully been applied on a certain, user-defined amount of replicas.

2.5 Master DBMS

The master DBMS, typically a commercial engine, is used to handle all update transactions in the system. We also rely on the master for availability, consistency, and persistence. Since all update traffic in the system is handled by the master, all conflict handling and deadlock detection are done there.

A critical point in our design is update extraction from the master. This is a well-known problem in database replication that has been approached in many different ways. In general, there are three options: propagating the SQL statements from the dispatcher, using triggers to extract the changes, and reading the changes from the log. Log analysis is commonly used in commercial solutions. Triggers are preferred in open source replication systems [13]. SQL propagation is mostly used in research systems [6,19,26]. There is also the option of using product-specific tools (Oracle, e.g., has numerous interfaces that could be used for this purpose).

For generality, we use trigger-based extraction and SQL propagation (which we call the *generic* approach, see Sect. 4). Nevertheless, both introduce problems of their own. Triggers are expensive and lower the performance of the master. SQL propagation does not affect the master but has problems with non-deterministic statements and creates SQL compatibility problems across heterogeneous engines. In terms of update extraction, there is really no perfect solution, and focusing on a single type of master rather than aiming for generality will always yield a better solution. Since, as already indicated, the purpose is to have a proof of concept implementation, the two options we explore provide enough information about the behavior of the system. We leave the evaluation of other options and tailoring to specific engines to future work.

2.6 Satellites

The main role of the satellites is to extend the master. The satellites have to be able to appear and disappear at any time without causing data loss. There might be work lost but the assumption in our system is that any data at the satellites can be recreated from the master.

Satellites apply incoming writesets to an FIFO order. For specialized satellites (e.g., with aggregations or combinations of the master's tables), writesets have to be transformed before they are applied (e.g., combining source columns into new tables, or performing aggregation by combining writesets with the tablespace data). In general, the transformation is based on two parameters: the incoming writeset WS_k and the satellite's latest snapshot S_{k-1} (see Sect. 3.1).

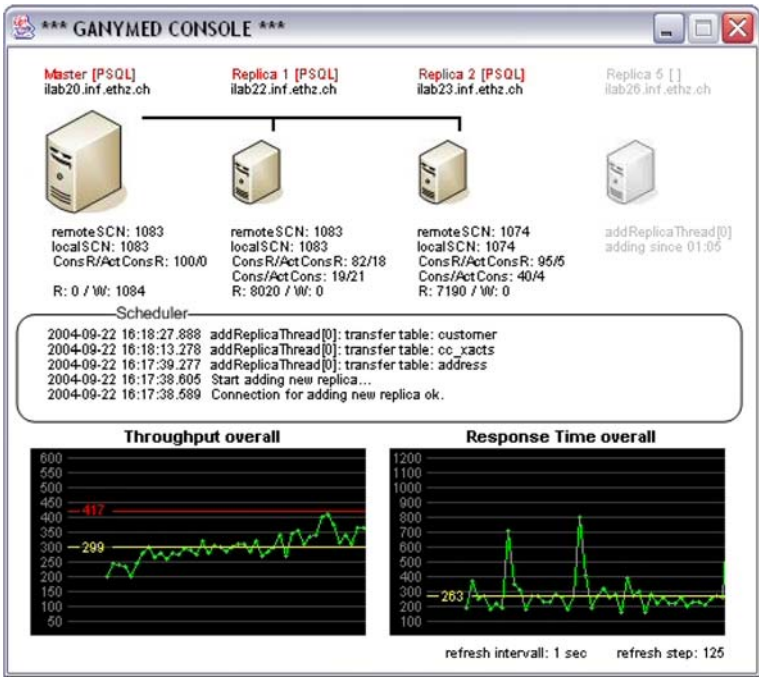

Fig. 4 The management console

2.7 Adapters

Adapters (see Figs. 2, 3) are used for gluing the different components of the system together. The dispatcher and the management console never talk directly to any backend DBMS; communication is always with an adapter. Adapters form a common abstraction of database engines and offer services like writeset handling and load estimation. Apart from these tasks, adapters also send information to the dispatcher (piggyback on the query results) on CPU and I/O load,[4] latest applied writeset, `COMMIT` numbers, etc. This information is also used by the management console, to decide if the system has the appropriate number of satellites to handle the current load.

Adapters are also used to solve problems that arise in heterogeneous environments, mainly SQL dialect differences between engines. For the experiments in this paper, we have implemented translation modules for adapters, which can, to a certain extent, dynamically translate queries from the master's SQL dialect to the one used by the satellite on which the adapter is running. As an example, these translators help to execute Oracle *TOP-N* queries based on the *ROWNUM* construct on a PostgreSQL satellite database which in turn offers the *LIMIT* clause. In general, such query rewriting is a typical $O(n^2)$ problem for n database engines. Nevertheless, note that query rewriting is necessary only for full replication satellites but not for specialized satellites. The assumption we make is that if the notion of satellite databases takes hold and full replication satellites are implemented, there will be enough motivation to develop such query rewriters, at least for the more important database engines.

2.8 Management console

The manager console (see Fig. 4 for a screenshot) is responsible for monitoring the Ganymed system. On the one hand, it includes a permanently running process which monitors the load of the system and the failure of components. On the other hand, it is used by the administrator to perform configuration changes.

While replica failure can directly be handled by the dispatcher (failed satellites are simply discarded, failed masters are replaced by a satellite, if possible), the failure of a dispatcher is more critical. In the event that a dispatcher fails, the monitor process in the manager console will detect this and is responsible for starting a backup dispatcher.[5] Assuming fail stop behavior, the connections between the failing dispatcher and all replicas will be closed, all running transactions will be aborted by the assigned replicas. The manager will then inspect all replicas, elect a master and configure the new dispatcher

[4] The actual measurement of CPU and I/O usage does not introduce additional load as we simply collect the needed statistics from the Linux proc file system, similar to the *vmstat* utility.

[5] As part of our prototype implementation we have also created a modified PostgreSQL JDBC driver that will detect such failures and try to find a working dispatcher according to its configuration.

so that transaction processing can continue. The inspection of a replica involves the detection of the last applied writeset, which can be done by the same software implementing the writeset extraction.

The manager console is also used by administrators that need to change the set of attached replicas to a dispatcher, or need to reactivate disabled replicas. While the removal of a replica is a relatively simple task, the attachment or re-enabling of a replica is a more challenging task. Syncing-in a replica is actually performed by copying the state of a running replica to the new one. At the dispatcher level, the writeset queue of the source replica is also duplicated and assigned to the destination replica. Since the copying process uses a SERIALIZABLE read-only transaction on the source replica, there is no need for shutting down this replica during the duplicating process. The new replica cannot be used to serve transactions until the whole copying process is over. Its writeset queue, which grows during the copy process, will be applied as soon as the copying has finished. Although from the viewpoint of performance this is not optimal, in the current prototype the whole copying process is done by the dispatcher under the control of the manager console.

Besides its role as the central administrative interface, the management console hosts a monitoring component that watches the whole setup, and performs, if necessary, reconfiguration according to the policy set by the administrator. Based on those policies, the monitor component can take machines out of a spare machine pool and assign them as a secondary satellite to a primary, or it can remove and put them back into the pool. In our prototype, policies are implemented as Java plugins.[6]

3 Consistency guarantees in ganymed

As indicated, Ganymed uses SI as the correctness criterium for replica management. In this section, we give a short introduction to SI and then extend the notion of SI to apply it to replicated databases. The result is the transaction routing algorithm we use in our system: Replicated SI with primary copy (RSI-PC).

3.1 Snapshot isolation

Snapshot isolation [7,32] is a multiversion concurrency control mechanism used in databases. Popular database engines[7] that are based on SI include Oracle [27] and PostgreSQL [29]. One of the most important properties of SI is the fact that readers are never blocked by writers, similar to the multiversion protocol in [8]. This property is a big win in comparison to systems that use *two phase locking* (2PL), where many non-conflicting updaters may be blocked by a single reader. SI completely avoids the four extended ANSI SQL phenomena P0–P3 described in [7], nevertheless it does not guarantee serializability. As shown in [16,15] this is not a problem in real applications, since transaction programs can be arranged in ways so that any concurrent execution of the resulting transactions is equivalent to a serialized execution.

For the purposes of this paper, we will work with the following definition of SI (slightly more formalized than the description in [7]):

SI: A transaction T_i that is executed under SI gets assigned a start timestamp $start(T_i)$ which reflects the starting time. This timestamp is used to define a snapshot S_i for transaction T_i. The snapshot S_i consists of the latest committed values of all objects of the database at the time $start(T_i)$. Every read operation issued by transaction T_i on a database object x is mapped to a read of the version of x which is included in the snapshot S_i. Updated values by write operations of T_i (which make up the *writeset* WS_i of T_i) are also integrated into the snapshot S_i, so that they can be read again if the transaction accesses updated data. Updates issued by transactions that did not commit before $start(T_i)$ are invisible to the transaction T_i. When transaction T_i tries to commit, it gets assigned a commit timestamp $commit(T_i)$, which has to be larger than any other existing start timestamp or commit timestamp. Transaction T_i can only successfully commit if there exists no other committed transaction T_k having a commit timestamp $commit(T_k)$ in the interval $\{start(T_i), commit(T_i)\}$ and $WS_k \bigcap WS_i \neq \{\}$. If such a committed transaction T_k exists, then T_i has to be aborted (this is called the *first-committer-wins* rule, which is used to prevent lost updates). If no such transaction exists, then T_i can commit (WS_i gets applied to the database) and its updates are visible to transactions which have a start timestamp which is larger than $commit(T_i)$.

A sample execution of transactions running on a database offering SI is given in Fig. 5. The symbols B, C and

[6] The following is a basic example of the logic contained in such a policy: "If the average CPU load on primary satellite X is more than 90% during more than 10 s, then start adding a new secondary satellite to X. If the load on X is less than 20% during more than 10 s, then remove a secondary satellite."

[7] The new Microsoft SQL Server 2005 [25] implements a hybrid approach, offering SI along with traditional concurrency control. However, for the SI part of this paper we focus on pure SI DBMSs.

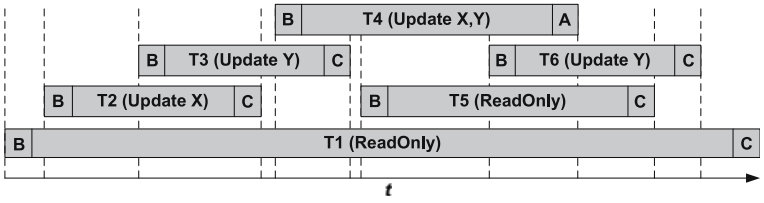

Fig. 5 An example of concurrent transactions running under SI

A refer to the *begin*, *commit* and *abort* of a transaction. The long running transaction T_1 is of type *readonly*, i.e., its writeset is empty: $WS_1 = \{\}$. T_1 neither will be blocked by any other transaction, nor will it block other transactions. Updates from concurrent updaters (like T_2, T_3, T_4, and T_6) are invisible to T_1. T_2 will update the database element X, it does not conflict with any other transaction. T_3 updates Y, it does not see the changes made by T_2, since it started while T_2 was still running. T_4 updates X and Y. Conforming to the first-committer-wins rule it cannot commit, since its writeset overlaps with that from T_3 and T_3 committed while T_4 was running. The transaction manager has therefore to abort T_4. T_5 is readonly and sees the changes made by T_2 and T_3. T_6 can successfully update Y. Due to the fact that T_4 did not commit, the overlapping writesets of T_6 and T_4 do not impose a conflict.

As will be shown in the next section, practical systems handle the comparison of writesets and the first-committer-wins rule in a different, more efficient way. Both, Oracle and PostgreSQL, offer two different ANSI SQL isolation levels for transactions: SERIALIZABLE and READ COMMITTED. An extended discussion regarding ANSI SQL isolation levels is given in [7].

3.1.1 The SERIALIZABLE isolation level

Oracle and PostgreSQL implement a variant of the SI algorithm for transactions that run in the isolation level SERIALIZABLE. Writesets are not compared at transaction commit time, instead this process is done progressively by using row level write locks. When a transaction T_i running in isolation level SERIALIZABLE tries to modify a row in a table that was modified by a concurrent transaction T_k which has already committed, then the current update operation of T_i gets aborted immediately. Unlike PostgreSQL, which then aborts the whole transaction T_i, Oracle is a little bit more flexible: it allows the user to decide if he wants to commit the work done so far or if he wants to proceed with other operations in T_i. If T_k is concurrent but not committed yet, then both products behave the same: they block transaction T_i until T_k commits or aborts. If T_k commits, then the same procedure gets involved as described before, if however T_k aborts, then the update operation of T_i can proceed. The blocking of a transaction due to a potential update conflict can of course lead to deadlocks, which must be resolved by the database by aborting transactions.

3.1.2 The READ COMMITTED isolation level

Both databases offer also a slightly less strict isolation level called READ COMMITTED, which is based on a variant of SI. READ COMMITTED is the default isolation level for both products. The main difference to SERIALIZABLE is the implementation of the snapshot: a transaction running in this isolation mode gets a new snapshot for every issued SQL statement. The handling of conflicting operations is also different than in the SERIALIZABLE isolation level. If a transaction T_i running in READ COMMITTED mode tries to update a row which was already updated by a concurrent transaction T_k, then T_i gets blocked until T_k has either committed or aborted. If T_k commits, then T_i's update statement gets reevaluated again, since the updated row possibly does not match a used selection predicate anymore. READ COMMITTED avoids phenomena P0 and P1, but is vulnerable to P2 and P3 (fuzzy read and phantom).

3.2 RSI-PC

For simplicity in the presentation and without loss of generality, we will describe RSI-PC, the dispatcher's transaction routing algorithm, assuming that the objective is scalability through replication for a single master DBMS. When the satellite databases implement specialized functionality, the routing of the load at the middleware layer will take place according to other criteria (such as keywords in the queries, tables being accessed, etc.).

The RSI-PC algorithm works by routing transactions through a middleware dispatcher over a set of backend databases. There are two types of backend databases: one master, and a set of satellites. From the client view, the middleware behaves like an ordinary database. Inconsistencies between master and satellite databases are hidden by the dispatcher. The dispatcher differentiates between *update* and *read-only* transactions. Note

that "update transaction" does not necessarily mean a transaction consisting of pure update statements, update transactions may contain queries as well. Transactions do not have to be sent by clients in a single block (unlike in, e.g., [19,26]), they are processed by the dispatcher statement by statement. However, the client must specify at the beginning of each transaction whether it is a read only or an update. Otherwise they are handled as update transactions by the dispatcher.

In Ganymed, update transactions are always forwarded to the master and executed with the isolation level specified by the client (e.g., in the case of Oracle or PostgreSQL this is either READ COMMITTED or SERIALIZABLE). Read-only transactions will, whenever possible, be sent to one of the satellites and executed with SI in the SERIALIZABLE mode. Therefore, the same snapshot (on the same satellite) will be used for all statements in the transaction, ensuring that a client's read-only transactions see always a consistent state of the database over their whole execution time. To keep the satellites in sync with the master, so-called *writeset extraction* is used [19]. Every update transaction's writeset (containing the set of changed objects on the master) is extracted from the master and then sent to FIFO queues on the satellites. Writesets are applied on the satellites in the order of the corresponding commit operations on the master, thereby guaranteeing that the satellites converge to the same state as the master. The application of writesets is done under SI: readers and writers do not interfere. This speeds up both the reading while the satellite is being updated, and the propagation of changes – as their application is not slowed down by concurrent readers.

3.3 Properties of RSI-PC

If required, RSI-PC can be used to provide different consistency levels. This has no influence on update transactions, since these are always sent to the master and their consistency is dictated by the master, not by our system. For instance, if a client is not interested in the latest changes produced by concurrent updates, then it can run its read-only transactions with *session consistency*. This is very similar to the concept of *strong session 1SR* introduced in [14]. On this consistency level, the dispatcher assigns the client's read-only transactions a snapshot which contains all the writesets produced by that client, though it is not guaranteed that the snapshot contains the latest values produced by other clients concurrently updating the master. By default, however, all read-only transactions are executed with *full consistency*. This means that read-only transactions are always executed on a satellite that has all the updates performed up to the time the transaction starts. The actual selection of a satellite is implementation specific (e.g., round-robin, least-pending-requests-first, etc.). Obviously, if writesets do not get applied fast enough on the satellites, then readers might be delayed. If the master supports SI and has spare capacity, the dispatcher routes readers to the master for added performance.

Another possible consistency level is based on the age of snapshots, similar to [31]. The age of a snapshot is the difference between the time of the latest commit on the master and the time the snapshot's underlying commit operation occurred. A client requesting *time consistency* always gets a snapshot which is not older than a certain age specified by the client. Time consistency and session consistency can be combined: read-only snapshots then always contain a client's own updates and are guaranteed not to be older than a certain age.

Due to its simplicity, there is no risk of a dispatcher implementing the RSI-PC algorithm becoming the bottleneck in the system. In contrast to other middleware based transaction schedulers, like the ones used in [6,10], this algorithm does not involve any SQL statement parsing or concurrency control operations. Also, no row or table level locking is done at the dispatcher level. The detection of conflicts, which by definition of SI can only happen during updates, is left to the database running on the master replica. Moreover, (unlike [10,18]), RSI-PC does not make any assumptions about the transactional load, the data partition, organization of the schema, or answerable and unanswerable queries.

3.4 Fine grained routing of single queries

For the common case of read-only transactions in auto commit mode (i.e., single queries), our algorithm is more selective. Upon arrival of such a query, the dispatcher parses the SQL SELECT statement to identify the tables that the query will read. If this is possible (e.g., no function invocations with unknown side effects or dependencies are detected), it then chooses a satellite where all the necessary tables are up to date. This optimization is based on the observation that, in some cases, it is perfectly legal to read from an older snapshot. One only has to make sure that the subset of data read in the older snapshot matches the one in the actual requested snapshot.

Parsing of single queries also makes it possible to implement partial replication. In the case of ordinary read-only transactions, it is not possible to determine in advance which objects will be touched during a transaction. Therefore, snapshots always have to be created on satellites which hold all objects. For single queries this is different: the required tables can be identified and a

snapshot can be created on a satellite which holds only those tables. This opens the possibility of dynamically creating satellites that hold only hot spot tables and even materialized views.

4 Update extraction

Unfortunately, writeset extraction, which is needed on the master replica, is not a standard feature of database systems. Therefore, additional programming effort is needed for every attached type of master. Currently, our prototype supports the following databases:

PostgreSQL: This open source DBMS can be used both as master and satellite. PostgreSQL is very flexible, it supports the loading of additional functionality during runtime. Our writeset support consists of a shared library written in C which can be loaded at runtime. The library enables the efficient extraction of writesets based on capturing triggers; All changes to registered tables, be it changes through ordinary SQL statements, user-defined triggers or user-defined functions and stored procedures, will be captured. The extracted writesets are table row based, they do not contain full disk blocks. This ensures that they can be applied on a replica which uses another low level disk block layout than the master replica.

Oracle: Similar to PostgreSQL, we have implemented a trigger-based capturing mechanism. It consists of a set of Java classes that are loaded into the Oracle internal JVM. They can be accessed by executing calls through any of the available standard Oracle interfaces (e.g., OCI). The software is able to capture the changes of user executed DML (data manipulation language) statements, functions and stored procedures. A limitation of our current approach is the inability to reliably capture changes as a result of user defined triggers. To the best of our knowledge it is not possible to specify the execution order of triggers in Oracle. It is thus not guaranteed that our capturing triggers are always the last ones to be executed. Hence, changes could happen after the capturing triggers. A more robust approach would be to use the Oracle change data capture feature [27].

DB2: We have developed support for DB2 based on loadable Java classes, similar to the approach chosen for Oracle. As before, we use triggers to capture changes and a set of user-defined functions which can be accessed through the standard DB2 JDBC driver to extract writesets. Due to the different ways in which DB2 handles the JVM on different platforms, the code works on Windows but not on DB2 for Linux. A more complete implementation would be based on using DB2 log extraction. Due to the incompleteness of our DB2 support for Linux, and to maintain fairness in the comparisons, we used the generic approach, described next, to connect to DB2.

4.1 Generic capturing of updates

Our generic approach is similar to that used on replication systems, e.g., [5,10]. We parse a transaction's incoming statements at the middleware and add all statements that possibly modify the database to its writeset. Of course, this type of writeset extraction (actually more a form of writeset collection) is not unproblematic: the execution of SELECT statements with side effects (e.g., calling functions that change rows in tables) is not allowed. The distribution of the original DML statements instead of extracted writesets also leads to a variety of difficulties. As an example take the following statement:

```
INSERT INTO customer (c_id, c_first_login)
 VALUES (19763, current_timestamp)
```

The execution of this statement on different replicas might lead to different results. Therefore, in the generic approach, the middleware replaces occurrences of *current_timestamp*, *current_date*, *current_time*, *sysdate*, *random* and similar expressions with a value before forwarding the statement. Our current prototype for use with TPC-W does this by search-and-replace. A full blown implementation would have to be able to parse all incoming DML statements into tokens and then apply the substitution step. In the case of queries, an alternative would be to send specialized statements to the master and forward to the satellites only those that do not involve any transformation. As mentioned before, our system provides at least the functionality and performance of the master. If the application is willing to help, then the satellites can be used with the corresponding gains. This is not different from clients providing hints to the optimizer, or writing queries optimized for an existing system. Although it can be kept completely transparent, the maximum gains will be obtained when queries are written taking our system into account. It should nevertheless be noted that even though the generic approach has obvious limitations, it is still suitable for many common simple database applications which do not rely on database triggers or (user-defined) functions with side effects. This problem does not appear either if the satellite databases implement functionality that is different from that of the master.

Table 1 Overhead of capturing triggers

PostgreSQL	Insert	126%
	Update	120%
	Delete	122%
Oracle	Insert	154%
	Update	161%
	Delete	160%

4.2 Trigger overhead

Our two extensions for PostgreSQL and Oracle make use of capturing triggers. Therefore, the question arises as to how much overhead these mechanisms add. To find out, we measured the difference of performing changes in the databases with and without the capturing software. For both systems, we made measurements where we determined the overhead for *each single update statement*. Three types of transactions were tested: one containing an INSERT operation, one with an UPDATE and one with a DELETE statement. No other load was present to obtain stable results.

Table 1 shows that the overhead per update statement can be as high as 61%. These figures, however, need to be prorated with the overall load as the overhead is *per update statement*. In the next section, we show that the proportion of update transactions in TPC-W is never more than 25% of the total load, and they do not solely consist of update statements. Hence, the actual trigger overhead is much less (see the experiments). In addition, our system reduces the load at the master, thereby leaving more capacity to execute updates and the associated triggers in an efficient manner.

4.3 Update propagation

Ganymed applies writesets on the satellites in the commit order at the master. Similar to the writeset extraction problem, there is no general interface which could be used to efficiently get notified of the exact commit order of concurrent commit operations. Since our initial intention is to prove the concept of satellite databases, we currently use a brute force approach: commits for update transactions are sent in a serialized order from the master adapter to the master database. However, this can be done more efficiently, since all involved components in this time critical process reside on the same machine.

5 SQL compatibility

Every DBMS available today features extensions and modifications to the SQL standard. Moreover, even if all DBMSs would understand the same dialect of SQL, it would not be guaranteed that the optimal formulation of a query on one DBMS would also lead to an optimal execution on another DBMS. To illustrate these problems, we use two examples from TPC-W.

5.1 TOP-N queries

In the TPC-W browsing workload (see later section), a *TOP-N* query is used in the *New Products Web Interaction*. The query itself returns a list of the 50 most recently released books in a given category. One possible way to implement this in Oracle is as follows:

```
SELECT * FROM (
 SELECT i_id, i_title, a_fname, a_lname
 FROM item, author
 WHERE i_a_id = a_id
  AND i_subject = 'COOKING'
 ORDER BY i_pub_date DESC, i_title
) WHERE ROWNUM <= 50
```

The TOP-N behavior is implemented by surrounding the actual query with a "`SELECT * FROM (...) WHERE ROWNUM <= n`" construct. The database then not only returns at most *n* rows, but the query planner can also use that knowledge to optimize the execution of the inner query.

Unfortunately, since *ROWNUM* is a virtual column specific to Oracle, the query does not work on PostgreSQL or DB2. However, it can easily be modified to be run on those systems: in the case of PostgreSQL, taking the inner query and adding a `LIMIT 50` clause to the end of the query leads to the same result. For DB2, the approach is very similar, the corresponding clause that has to be appended is called `FETCH 50 ROWS ONLY`.

5.2 Rewriting as optimization

As a second example, we look at the query that is used in TPC-W for the "Best Sellers Web Interaction". Amongst the 3,333 most recent orders, the query performs a TOP-50 search to list a category's most popular books based on the quantity sold. Again, we show the SQL code that could be used with an Oracle database:

```
SELECT * FROM (
 SELECT i_id, i_title, a_fname, a_lname,
  SUM(ol_qty) AS orderkey
 FROM item, author, order_line
 WHERE i_id = ol_i_id AND i_a_id = a_id
  AND ol_o_id >
   (SELECT MAX(o_id)-3333 FROM orders)
  AND i_subject = 'CHILDREN'
```

```
 GROUP BY i_id, i_title, a_fname,
  a_lname
 ORDER BY orderkey DESC
) WHERE ROWNUM <= 50
```

This query has the same problem as the previous TOP-N query. However, at least in the case of PostgreSQL, replacing the ROWNUM construct with a LIMIT clause is not enough: although the query works, the performance would suffer. The reason is the use of the `MAX` operator in the `SELECT MAX(o_id)-3333 FROM orders` subquery. Many versions of PostgreSQL cannot make use of indexes to efficiently execute this subquery due to the implementation of aggregating operators. When specified using the `MAX` operator, the subquery leads to a sequential scan on the (huge) orders table. To enable the use of an index set on the *o_id* column of the orders table, the subquery must be rewritten as follows:

```
SELECT o_id-3333 FROM orders
 ORDER BY o_id DESC LIMIT 1
```

At first glance, sorting the orders table in descending order looks inefficient. However, the use of the *LIMIT* 1 clause leads to an efficient execution plan which simply looks up the highest *o_id* in the orders table by peeking at the index set on the *o_id* column[8].

5.3 Rewriting SQL on the fly

These last two examples illustrate the problems of heterogeneous setups. Since we assume that the client software should not be changed (or only in a very narrow manner), obviously the incoming queries need to be rewritten by the middleware (more exactly, in the adapters). We propose two ways to perform the translation step:

Generic rewriting: Depending on the DBMS (Oracle, DB2, ...) a query was originally formulated for, and the effective place of execution (a satellite with possibly a different DBMS), a set of specific modules in the middleware could rewrite the query. There are two ways to structure this: one is defining a common intermediate representation and set of in- and out- modules. The other possibility is to have direct translation modules for every supported pair of master/satellite DBMS. These solutions need no support from the client application programmer, but are hard to develop from the perspective of the middleware designer. Yet, if the idea of satellite databases becomes widely used, such modules will certainly be incrementally developed.

Application-specific plug-ins: If the query diversity is very limited (the generic rewriting approach would be too much overhead), or if a certain generic rewriting approach for a set of master/satellites is not available, it would make sense to allow the administrator to be able to add custom rewriting rules. At the cost of additional maintenance, a more flexible and possibly more efficient integration of applications and new master/satellite DBMS pairs would be possible. Of course, the two approaches can also be mixed: for some translations the generic modules could be used, for others the system could call user specified translation plug-ins.

Our current prototype is mainly used with TPC-W databases, and so the query load is well known. Since the rules needed for TPC-W query rewriting are rather simple, we have chosen to use specific plug-ins which can be loaded into the adapters. The plug-ins use string replacement which only works for our TPC-W loads but with negligible overhead.

6 Experimental evaluation

To verify the validity of our approach we performed several tests. We start with a worst case scenario intended to provide a lower bound on what can be done with satellite databases: satellites that contain full replicas of a master database. This is a worst case scenario because of the amount of data involved (the entire database) and the issues created by heterogeneity (of the engines and the dialects of SQL involved). Yet, the experiments show that satellite databases can be used to provide a significant degree of scalability in a wide range of applications.

6.1 Experiments performed

First, we did extensive scalability measurements for homogeneous setups consisting only of fully replicated PostgreSQL databases. We used a load generator that simulates the transaction traffic of a TPC-W application server. Additionally, we tested the behavior of Ganymed in scenarios where database replicas fail. The failure of both, satellites and masters, was investigated.

In a later section, we present measurements for more complex setups, namely, heterogeneous Ganymed systems implementing full replication for Oracle and DB2 masters with attached PostgreSQL satellites. Again, we measured the achievable performance by using different TPC-W loads.

[8] In newer PostgreSQL versions (8.0+) this particular problem has been resolved. If the database detects the use of the `MAX` operator as explained, it will rewrite the query execution plan using the `LIMIT` operator.

6.2 The TPC-W load

The TPC benchmark W (TPC-W) is a transactional web benchmark from the Transaction Processing Council [12]. TPC-W defines an internet commerce environment that resembles real world, business oriented, transactional web applications. The benchmark also defines different types of workloads which are intended to stress different components in such applications [namely, multiple on-line browser sessions, dynamic page generation with database access, update of consistent web objects, simultaneous execution of multiple transaction types, a backend database with many tables with a variety of sizes and relationships, transaction integrity (ACID) and contention on data access and update]. The workloads are as follows: primarily shopping (WIPS), browsing (WIPSb) and web-based ordering (WIPSo). The difference between the different workloads is the ratio of browse to buy: WIPSb consists of 95% read-only interactions, for WIPS the ratio is 80% and for WIPSo the ratio is 50%. WIPS, being the primary workload, is considered the most representative one.

For the evaluation of Ganymed we generated database transaction traces with a running TPC-W installation. We used an open source implementation [24] that had to be modified to support a PostgreSQL backend database. Although the TPC-W specification allows the use of loose consistency models, this was not used since our implementation is based on strong consistency. The TPC-W scaling parameters were chosen as follows: 10,000 items, 288,000 customers and the number of EBs was set to 100.

Traces were then generated for the three different TPC-W workloads: *shopping mix* (a trace file based on the WIPS workload), *browsing mix* (based on WIPSb) and *ordering mix* (based on WIPSo). Each trace consists of 50,000 consecutive transactions.

6.3 The load generator

The load generator is Java based. Once started, it loads a trace file into memory and starts parallel database connections using the configured JDBC driver. After all connections are established, transactions are read from the in-memory tracefile and then fed into the database. Once a transaction has finished on a connection, the load generator assigns the next available transaction from the trace to the connection.

For the length of the given measurement interval, the number of processed transactions and the average response time of transactions are measured. Also, to enable the creation of histograms, the current number of processed transactions and their status (committed/aborted) are recorded every second.

6.4 Experimental setup

For the experiments, a pool of machines were used to host the different parts of the Ganymed system. For every component (load generator, dispatcher, databases (master and satellites)) of the system, a dedicated machine was used. All machines were connected through a 100 MBit Ethernet LAN. Java software was run with the Blackdown-1.4.2-rc1 Java 2 platform. All machines were running Linux with a 2.4.22 kernel. The load generator ran on a dual Pentium-III 600 MHz machine with 512 MB of RAM. The dispatcher and the webservice ran on dual Pentium-IV machines with 3 GHz and 1 GB of main memory. Using identical machines (Dual AMD Athlon 1400 MHz CPU, 1 GB RAM, 80 GB IBM Deskstar harddisk) we installed several PostgreSQL (8.0.0), one Oracle (10G) and one DB2 (8.1) database. For the full replication experiments, each database contained the same set of TPC-W tables (scaling factor 100/10K [12]). All databases were configured using reasonable settings and indexes; however, we do not claim that the databases were configured in the most optimal manner: in the case of Oracle we used automatic memory management and enabled the cost-based optimizer. DB2 was configured using the graphical configuration advisor. Additional optimizations were done manually. The PostgreSQL satellites were configured to not use fsync, which enables fast application of writesets. The disabling of the fsync on the satellites is not problematic (in terms of durability), since the failure of a satellite would involve a re-sync anyway and durability is guaranteed by the master. Before starting any experiment, all databases were always reset to an initial state. Also, in the case of PostgreSQL, the *VACUUM FULL ANALYZE* command was executed. This ensured that every experiment started from the same state.

6.5 Homogeneous full replication

First, we describe our experimental results that result from a Ganymed system used to implement homogeneous full replication with a set of PostgreSQL databases. We did extensive scalability measurements by comparing the performance of different multiple satellite configurations with a single PostgreSQL instance. Second, we describe the behavior of Ganymed in scenarios where databases fail. The failure of both satellites and masters was investigated.

6.5.1 Performance and scalability

The first part of the evaluation analyzes performance and scalability. The Ganymed prototype was compared with a reference system consisting of a single PostgreSQL instance. We measured the performance of the Ganymed dispatcher in different configurations, from 0 up to 5 satellites. This gives a total of seven experimental setups (called PGSQL and SAT-n, $0 \leq n \leq 5$), each setup was tested with the three different TPC-W traces.

The load generator was then attached to the database (either the single instance database or the dispatcher, depending on the experiment). During a measurement interval of 100 s, a trace was then fed into the system over 100 parallel client connections and at the same time average throughput and response times were measured. All transactions, read only and updates, were executed in the SERIALIZABLE mode. Every experiment was repeated until a sufficient, small standard deviation was reached.

Figure 6 shows the results for the achieved throughput (transactions per second) and average transaction response times, respectively. The ratio of aborted transactions was below 0.5% for all experiments.

Figure 7 shows two example histograms for the TPC-W ordering mix workload: on the left side the reference system, on the right side SAT-5. The sharp drop in performance in the SAT-5 histogram is due to multiple PostgreSQL replicas that did checkpointing of the WAL (*write ahead log*) at the same time. The replicas were configured to perform this process at least every 300 s; this is the default for PostgreSQL.

Based on the graphs, we can prove the lightweight structure of the Ganymed prototype. In a relay configuration, where only one replica is attached to the Ganymed dispatcher, the achieved performance is almost identical to the PostgreSQL reference system. The performance of the setup with two replicas, where one replica is used for updates and the other for read-only transactions, is comparable to the single replica setup. This clearly reflects the fact that the heavy part of the TPC-W loads consists of complex read-only queries. In the case of the write intensive TPC-W ordering mix, a two replica setup is slightly slower than the single replica setup. In the setups where more than two replicas are used, the performance compared to the reference system could be significantly improved. A close look at the response times chart shows that they converge. This is due to the RSI-PC algorithm which uses parallelism for different transactions, but no intra-parallelism for single transactions. A SAT-5 system, for example, would have the same performance as a SAT-1 system when used only by a single client.

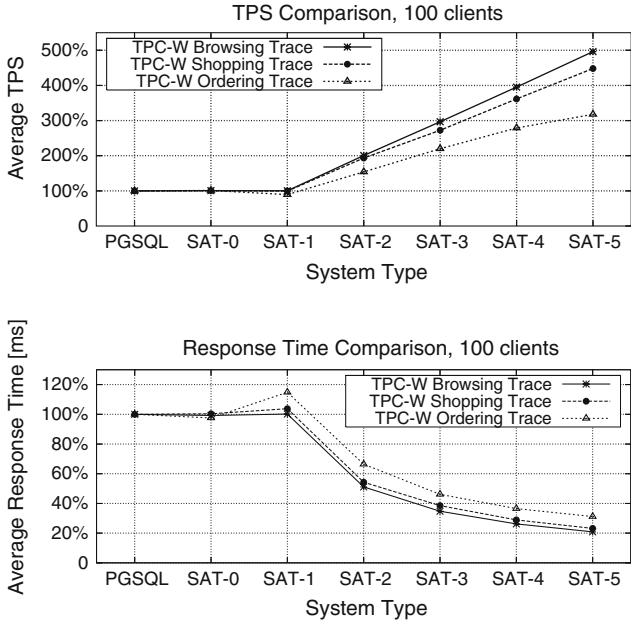

Fig. 6 Ganymed performance for TPC-W mixes

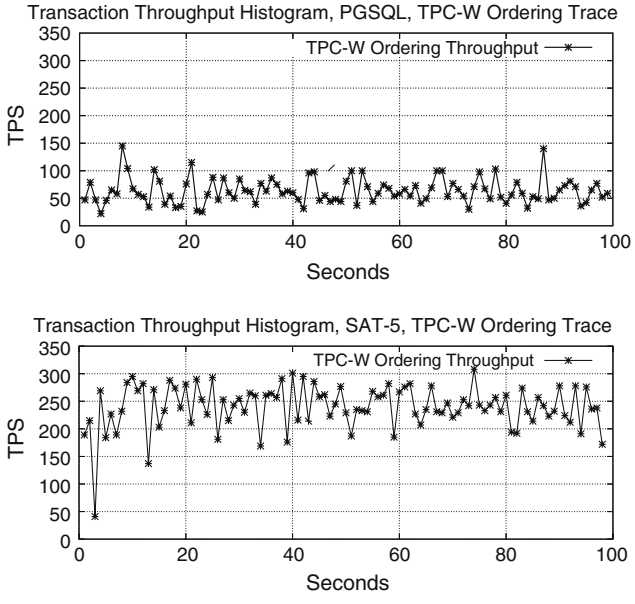

Fig. 7 Example histograms for the TPC-W ordering mix

One can summarize that in almost all cases a nearly linear scale-out was achieved. These experiments show that the Ganymed dispatcher was able to attain an impressive increase in throughput and reduction of transaction latency while maintaining the strongest possible consistency level.

It must be noted that in our setup all databases were identical. By having more specialized index structures

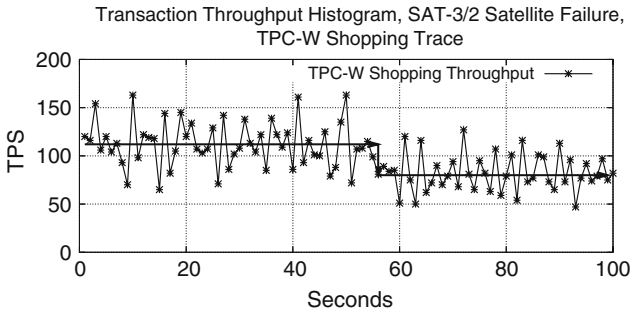

Fig. 8 Ganymed reacting to a satellite failure

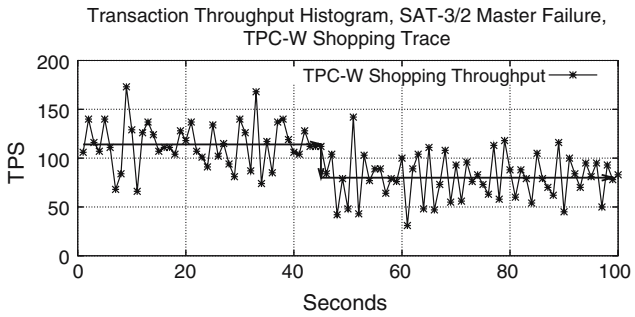

Fig. 9 Ganymed reacting to a master failure

on the satellites the execution of read-only transactions could be optimized even more. We are exploiting this option as part of future work.

6.5.2 Reaction to a failing satellite replica

In this experiment, the dispatcher's reaction to a failing satellite replica was investigated. A SAT-3 system was configured and the load generator was attached with the TPC-W shopping mix trace.

After the experiment was run for a while, one of the the satellite replicas was stopped, by killing the PostgreSQL processes with a kill (*SIGKILL*) signal. It must be emphasized that this is different from the usage of a *SIGTERM* signal, since in that case the PostgreSQL software would have had a chance to catch the signal and shutdown gracefully.

Figure 8 shows the generated histogram for this experiment. In second 56, a satellite replica was killed as described above. The failure of the satelite replica led to an abort rate of 39 read-only transactions in second 56; otherwise no transaction was aborted in this run. The arrows in the graph show the change of the average transaction throughput per second. Clearly, the system's performance degraded to that of a SAT-2 setup. As can be seen from the graph, the system recovered immediately. Transactions running on the failing replica were aborted, but otherwise the system continued working normally. This is a consequence of the lightweight structure of the Ganymed dispatcher approach: if a replica fails, no costly consensus protocols have to be executed. The system just continues working with the remaining replicas.

6.5.3 Reaction to a failing master replica

In the last experiment, the dispatcher's behavior in the case of a failing master replica was investigated. As in the previous experiment, the basic configuration was a SAT-3 system fed with a TPC-W shopping mix trace. Again, a SIGKILL signal was used to stop PostgreSQL on the master replica.

Figure 9 shows the resulting histogram for this experiment. Transaction processing is normal until in second 45 the master replica stops working. The immediate move of the master role to a satellite replica leaves a SAT-2 configuration with one master and two satellite replicas. The failure of the master led to an abort of two update transactions; no other transactions were aborted during the experiment. As before, the arrows in the graph show the change of the average transaction throughput per second.

This experiment shows that Ganymed is also capable of handling failing master replicas. The system reacts by reassigning the master role to a different, still working satellite. It is important to note that the reaction to failing replicas can be done by the dispatcher without intervention from the manager console. Even with a failed or otherwise unavailable manager console the dispatcher can still disable failed replicas and, if needed, move the master role autonomously.

6.6 Heterogeneous full replication

In this section, we discuss several experiments performed using Oracle and DB2 master databases. In both cases we attached a dispatcher and a varying set of PostgreSQL satellite databases. Again, we used the three TPC-W workloads to measure the performance. However, this time we also added a fourth load which is used to demonstrate the capability of the heterogeneous systems to handle peaks of read-only transactions: 200 concurrent clients were attached to the system, having 100 clients sending TPC-W ordering transactions and 100 clients sending read-only transactions. The read-only transaction load was created by taking a TPC-W browsing trace and eliminating all update transactions.

For all experiments, the dispatcher was configured to execute read-only transactions with full consistency. We did not make use of loose consistency levels or partial

replication possibilities (as these benefit our approach). The assignment of read-only transactions to satellites was done using a *least-pending-requests-first* strategy. Since Oracle offers SI, we also assigned readers (using isolation level *read only*) to the master when readers would have been otherwise blocked (due to the unavailability of the requested snapshot on the satellites). In the case of DB2 the master could not be used to offer SI for readers. Therefore, in the worst case, read-only transactions were delayed until a suitable snapshot on a satellite was ready.

To get an idea of the design space, we tested Oracle with update extraction using triggers (table level triggers) and using the generic approach (SQL propagation). DB2 was tested only with the generic approach, since we have not implemented a trigger-based approach yet.

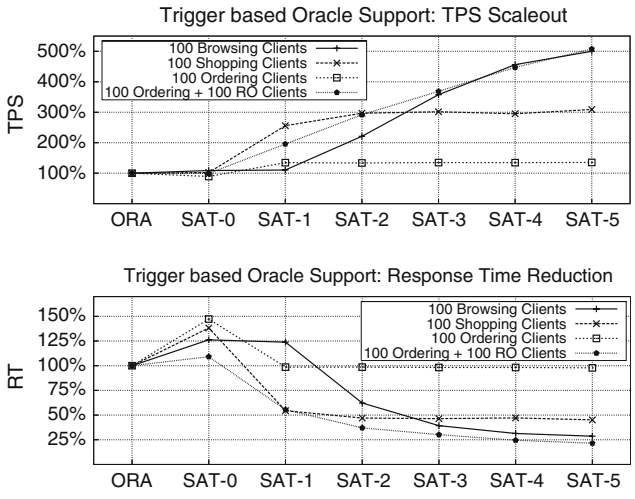

Fig. 10 Measurement results for Oracle with the trigger-based approach

6.6.1 Oracle using triggers

In our first series of experiments we tested a dispatcher that uses triggers to do writeset extraction on Oracle. First we measured performance by attaching the load generator directly to Oracle without triggers. Then, we attached the dispatcher and installed the writeset extraction features on the Oracle master for the remaining measurements. The results, relative throughput in terms of transactions per second (TPS) and response times, are shown in Fig. 10. For both graphs the first experiment, labeled ORA, represents Oracle without triggers and without our system. The second column, SAT-0, shows the performance when putting the dispatcher between the load generator and the master but no satellite databases. The next columns show the results of having attached additional 1–5 PostgreSQL satellite databases to the dispatcher (SAT-1 until SAT-5).

As the graphs show, the system provides substantial scalability both in throughput and response time for loads with enough read operations. In addition, using the master for blocked readers has interesting consequences. While in the browsing workload readers almost never have to be delayed, the shopping workload produces many readers that need a snapshot which is not yet available on the satellites. As a result, the shopping workload can be scaled better for small numbers of replicas, since the resources on the master are better utilized. Second, the trigger approach and commit order serialization adds a certain overhead. Thus, if clients execute many update transactions (as in the ordering workload) scale-out is reduced and performance is dominated by the master. However, scale-out is only reduced from the viewpoint of clients that execute updates: the mixed workload is able to achieve a very good scale-out, even though the number of clients is doubled in comparison to the ordering workload. The system is therefore able to handle peaks of readers.

6.6.2 Oracle using the generic approach

Since our used TPC-W workload does not use any stored procedures, it can also be used with the generic writeset extraction. This eliminates the overhead of the capturing triggers. In the second experiment series we tested such a configuration, again with Oracle. The results are shown in Fig. 11.

Obviously, as can be seen by comparing ORA and SAT-0, the overhead of the dispatcher approach is now very small. However, commit order detection is still in place, which has an impact on update transactions. Again, the graphs show that the system is able to scale well for read dominated loads.

6.6.3 DB2 using the generic approach

Using DB2 with generic writeset capturing we performed the same experiments as before. However, some modifications had to be introduced. Since DB2 offers no SI, the system was configured to block such transactions until the snapshot became available. Also, our first trial experiments with DB2 suffered from the same problem as those with Oracle, that is to say, serialized commits lead to limited scale-out for update intensive workloads. In the case of DB2, we then optimized the database engine configuration to speed-up commits.

The results based on DB2 are shown in Fig. 12. As before, our system offers very good scalability. One can also observe that the optimized commit at the master

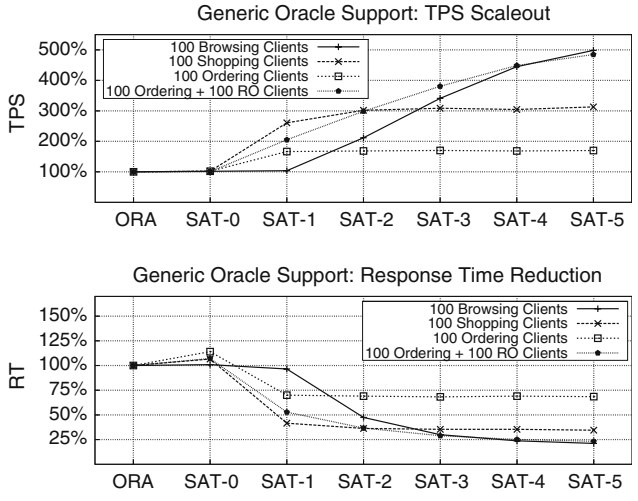

Fig. 11 Measurement results for Oracle with the generic approach

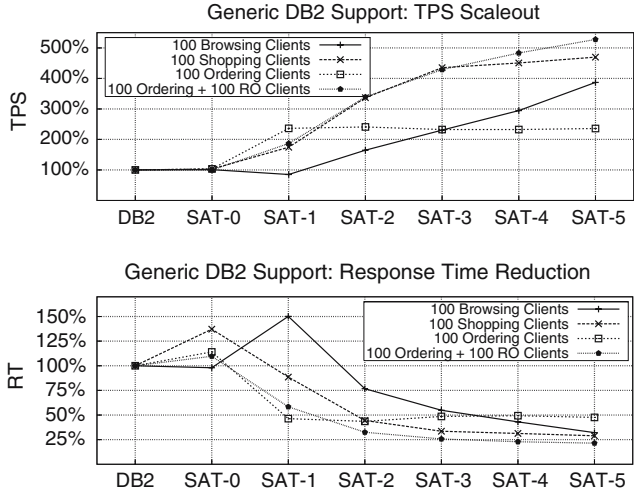

Fig. 12 Measurement results for DB2 with the generic approach

pays off: the performance of the shopping workload approaches that of the mixed workload. An interesting point is the scale-out of the browsing workload, which is less than expected. The reason is the slightly different set of indexes and queries which we used on the DB2 and PostgreSQL engines. As a consequence, PostgreSQL could more efficiently handle the read-only transactions in the shopping and order workloads, while DB2 performed better when executing the (more complex) mix of read-only transactions in the browsing workload. Since the objective of the experiments was to prove the feasibility of the approach, we did not proceed any further with optimizations and tuning on both master and satellites.

6.6.4 Interpretation of heterogeneous results

The main conclusion from these experiments is that satellites can provide significant scalability (both in terms of throughput and response time) as long as there is enough reads in the load. The differences in behavior between the three set-ups are due to low level details of the engines involved. For instance, with Oracle we need to commit transactions serially (only the commit, not the execution of the operations) to determine the write-set sequence. This causes a bottleneck at the master that lowers the scalability for the shopping and ordering traces (the ones with most updates). We also execute queries at the master when there are many updates, thereby also increasing the load at the master and hence enhancing the bottleneck effect for traces with high update ratios. In DB2 we have a more efficient mechanism for determining commit order and we never execute queries on the master. This leads to better scalability for the shopping and ordering traces. As another example, the browsing trace scales worse with DB2 as master than with Oracle as master. The reason is the slightly different set of indexes and queries which we used on the DB2 and PostgreSQL engines. As a consequence, PostgreSQL could more efficiently handle the read-only transactions in the shopping and ordering workloads, while DB2 performed better when executing the (more complex) mix of read-only transactions in the browsing workload.

When the overhead is dominated by queries (browsing mix), assigning readers to a single satellite does not yield any performance gain. Two satellites are needed to increase the read capacity in the system – this is also the case for Oracle, since the master is used only for *blocked* queries.

For all set-ups, even when there is a significant update load at the master, the system provides good scalability. This can be seen in the mixed workload (100 ordering + 100 read-only clients) which has twice as many clients as in the other workloads. These results are encouraging in that they demonstrate the potential of satellite databases. They also point out to the many issues involved in implementing full replication in satellites. One such issue is the trade off between loading the master or not loading it (the Oracle set-up achieves slightly better response times although the throughput is reduced due to the bottleneck effect mentioned). Other issues that are beyond the scope of this paper include: SQL incompatibilities, overhead of update extraction, and engine-specific optimizations. Most of these problems are to a large extent engineering issues that need to be resolved in a product-specific manner. This is why we see these results as a form of lower bound in terms of what can

be achieved rather than as an absolute statement of the performance of full replication satellites.

7 Specialized satellites

The third set of experiments explore the performance of specialized satellites. The objective here is to show the true potential of satellite databases in a setting that turns out to be much easier to implement and less constrained than full replication satellites. We show satellites implementing skyline queries as an extension to an Oracle master and satellites implementing an extra table for keyword search as an extension to a DB2 master.

7.1 Skyline satellites

As a first example of specialized satellites, we use PostgreSQL satellites to extend an Oracle master with skyline queries [9,11]. The objective is to show how satellites can be used to implement new operators that require heavy computation without affecting the master (unlike, e.g., data blades). Our implementation of the skyline operator for PostgreSQL databases is based on the *Block Nested Loop* (BNL) algorithm [9]. Clients can execute queries of the following form:

```
SELECT ... FROM ... WHERE ...
    GROUP BY ... HAVING ... ORDER BY ..
    SKYLINE OF c1 [MIN|MAX] {, cN [MIN|MAX]}
```

These statements are distinctive and can easily be detected by the dispatcher, which forwards them to the satellites. Any other update or query is sent to the master. The routing policy in the dispatcher is *least-pending-requests-first*.

For the experiments, we used Oracle as master and 1 to 8 PostgreSQL satellites (only replicating the *orders* and *order_line* tables). We performed two experiments, one with no updates at the master and one with updates at the master (50 *NewOrder* transactions per second from the TPC-W benchmark). We measured throughput and response times for various numbers of skyline clients. All results are given relatively to the performance of a single client connected to a system with a single satellite (we cannot compare with a master since the master does not support the skyline operator). Each client constantly sends read-only transactions containing the following skyline query to the system, with no thinking time between transactions:

```
SELECT o_id, o_total, SUM (ol_qty) AS qty_sum
     FROM orders, order_line
     WHERE o_id = ol_o_id AND ol_o_id >
          (SELECT o_id-500 FROM orders
```

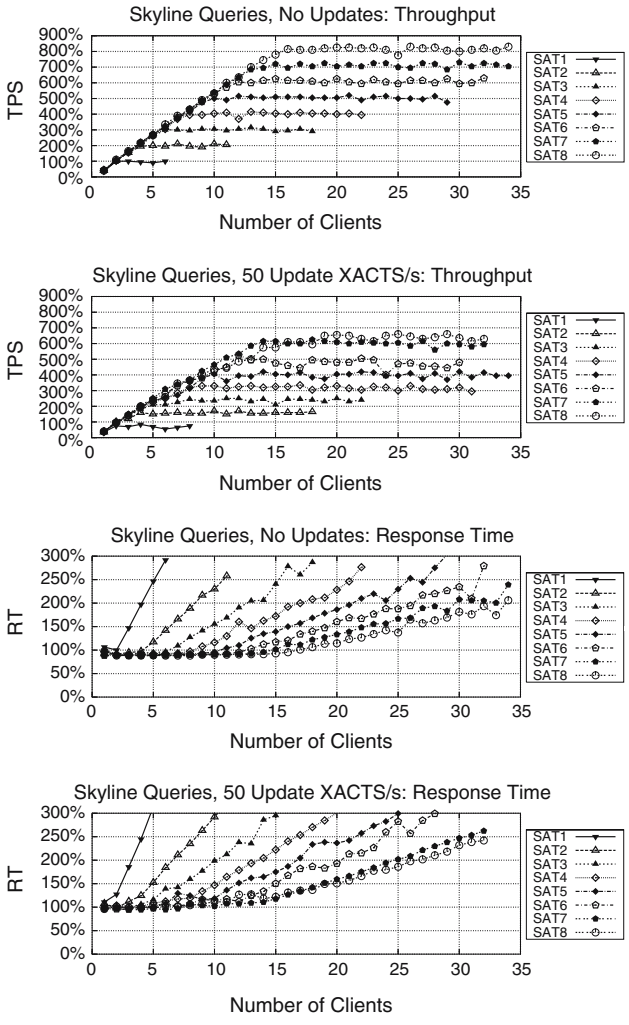

Fig. 13 Skyline performance results

```
          ORDER BY o_id DESC LIMIT 1)
GROUP BY o_id, o_total ORDER BY o_total
SKYLINE OF o_total MAX qty_sum MIN
```

7.1.1 Results: skyline satellites

The results are shown in Fig. 13 (the axis represent scalability in percentage versus number of clients submitting skyline queries, the lines in each graph correspond to the different number of satellites). Overall, adding additional satellites supports more clients at a higher throughput. It also improves the response time as clients are added, thereby contributing to the overall scalability of the resulting system. One can also observe that the system starts saturating when the number of clients is twice the number of satellites – which corresponds to the two CPUs present in each satellite machine. This is a consequence of the BNL-based skyline operator which is very CPU intensive – a client that constantly sends

queries is able to almost saturate one CPU. This also explains why the response time increases linearly with the number of clients. The important aspect here is that, as the experiments show, we have extended an Oracle database with scalable support for skyline queries with little overhead on the original master.

7.2 Keyword search satellites

As a second example of specialized satellites, we use a PostgreSQL-based full text search that extends the TPC-W schema deployed in a DB2 master. The objective here is to show how satellites can be used to add new tables (or, e.g., materialized views, indexes, etc.) to an existing database without inflicting the cost of maintaining those tables on the master.

The keyword search facility consists of a keyword table (holding 312,772 triples consisting of a keyword, book id and weight) and an index to speed-up look-ups. The keywords are generated by extracting words from the `i_desc` field in the item table. The keyword search facility is not static: we assigned weights to the keywords which correlate with the last 3,333 orders in the order_line table.

The challenge is how to keep the keyword table up-to-date. In terms of the load, maintaining the table has two consequences: first, the read-load increases due to the need to read the data to compute the table. Second, there is an additional load caused by the logic that computes the table.

The approach we have tested is one in which the table is re-built periodically on the satellite by additional Java code that is hooked into the adapter. It scans the descriptions of all books in the item table and builds a keyword table with regard to the latest orders. This is, of course, not the most efficient solution to this problem but it is quite representative of a wide range of applications that use such external programs to maintain derived data.

7.3 Results: keyword search satellites

Figure 14 shows the results of this experiment with two sets of measurements: throughput (solid columns) and build time (shaded columns).

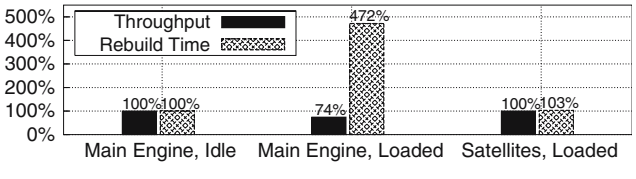

Fig. 14 Rebuilding the keyword table

For throughput, the base line is DB2 running 100 TPC-W shopping clients (100% performance). The middle solid column (main engine, loaded) shows what happens to the throughput of those 100 clients when the table is rebuilt: it drops to 74%. The rightmost solid column shows the throughput for a DB2 master with one PostgreSQL satellite that is used to store the table (and replicate the item and order_line tables) and keep it up-to-date. Throughput goes back to 100% since the master does not see the overhead of updating the table.

For build time, we use as base line the time it takes to rebuild the table in DB2 when DB2 is not loaded (30 s, taken as 100% in the figure). When we try to rebuild the table but DB2 is loaded with the 100 TPC-W shopping clients, the rebuild time is almost five times longer than when DB2 is idle (140 s, or 472% of the base line). Once the rebuild happens in the satellite, the rebuild time goes down to 103% even with the master loaded with the 100 TPC-W shopping clients.

7.4 Discussion: specialized satellites

The two experiments show that specialized satellites can be used to extend an existing master database without affecting performance at the master. This new functionality can be, e.g., operators that are not supported at the master (like the skyline queries) or additional tables derived from already existing data (like the keyword search). In both cases, the experiments show that satellites can easily support such extensions in a scalable, non-intrusive manner.

Overall, it is in this type of applications where satellites may prove most valuable. On the one hand, the complex issues around full replication satellites are not present in specialized satellites, thereby simplifying the implementation. On the other hand, the fact that our satellites are open source databases offers an open platform in which to implement new functionality. This is particularly important for many scientific applications. The only downside is that satellites are static and may not always be needed. The obvious next step is to be able to add such specialized satellites dynamically.

8 Dynamic creation of satellites

In this section, we describe our approach to dynamically create satellites. First, we describe the basic available options. Then, we describe the differences between the so-called logical and physical copy approach. Next, we describe our PITR (point-in-time recovery) based physical copy approach that we use in Ganymed. We also

provide several experiments that show the flexibility of our PITR-based prototype.

8.1 Basic options

We have identified three basic techniques that could be used to dynamically create satellites; they can actually be combined (all of them similar to crash recovery procedures):

Copy: To create a satellite, one can simply copy the latest snapshot from the master. If another satellite is already available, one can copy the data from the satellite instead. Also, during the copy process, the writeset queue of the new satellite is already being populated. However, these writesets can only be applied after the satellite has successfully finished the copying process. When adding the first satellite to a master, this is the only possible approach. If the master does not support SI (as DB2), then this means that updates have to be blocked until the initial satellite has been set up (for instance by using table level locks). However, this happens only once, since further satellites can be generated by taking a snapshot from existing satellites.

Writeset replay: This approach is feasible if we have an old checkpoint of the master. For instance, this could be useful when a satellite is taken out of a Ganymed setup and then later added again. If the contained data are not too stale, then the middleware can apply just the writesets which are needed to bring it to the latest state. Of course, this involves keeping a history of writesets – either on the master or on the primary satellite. The maximum acceptable staleness is therefore dependant upon the space assigned for keeping the history.

Hybrid: A more fine grained approach is to decide for each table object which of the two above approaches is more appropriate. Note that the availability of the writeset replay approach for a certain table does not imply that the copying approach has to be disregarded. For example, in the case of a small table, copying is probably cheaper than applying a large set of updates.

Even though the writeset reply and hybrid approaches offer interesting possibilities, we have so far only implemented the copy approach in our prototype.

8.2 Logical or physical copy

There are two options for implementing the copy approach, one has to consider two options: logical copy and physical copy. By logical copy we refer to the mechanism of extracting and importing data using queries, while physical copy refers to directly transferring DBMS table space files from machine to machine. In our prototype we use a variant of the physical copy approach to spawn new satellites given an existing primary satellite.

The big advantage of the logical copy approach is high flexibility. Table data can be replicated between different types of DBMSs through well-defined interfaces, e.g., using a DBMS's import/export tools or JDBC.[9] Using logical copies, partial replication is straightforward (one could replicate, e.g., only the customer table or European customers). However, the price for this flexibility is speed. First, copying at the logical level is more expensive, and second, indexes have to be rebuilt after the corresponding table data have been copied.

The main advantage of the physical copy approach is speed. Throughput is only limited either by disk controllers or available network bandwidth. Also, indexes do not have to be rebuilt, they are just copied the same way as the table data. The downside of the approach is the limited use in heterogeneous setups. Yet, when using the same DBMS type across machines, physical copy is often only possible by copying the full DBMS installation, even though only a subset of the tables is needed. Therefore, enhancing a running replica with additional tables is not straightforward. Another problem arises when the source machine does not allow to copy the tablespaces while the DBMS is online. Then, depending on the capabilities of the used DBMS software, it might be that the source DBMS has to be stopped (and therefore its writesets have to be queued), as long as the copying process is ongoing. This leads to a temporary decrease in performance, since one less satellite can work on read-only transactions. Also, the copied data cannot be accessed until all data have been transferred, since only then the new DBMS replica can be started up. This is in contrast to the incremental nature of the logical copy approach: there, as soon as pending writesets have been applied, consistent access to tables is possible – even though indexes and other tables may not yet be ready.

To demonstrate the difference in terms of performance of the two alternatives, we have performed measurements based on the replication of the `order_line` table defined in TPC-W. The table has three indexes and was populated in two versions: *small* and *big* (which corresponds to the TPC-W scaling factors 10,000 and 100,000). The measurements were done using two PostgreSQL satellites, the results are shown in Table 2.

As expected, the physical copy approach is faster, even though much more data have to be transferred. Yet, the logical copy approach allows to create a satellite in minutes, which is already acceptable in many

[9] Octopus [30], for example, is a popular transfer tool based on JDBC.

Table 2 Logical versus physical copy performance for the TPC-W order_line table

	Logical copy			Physical copy		
	Small	Big	$\frac{\text{Big}}{\text{Small}}$	Small	Big	$\frac{\text{Big}}{\text{Small}}$
Tuples	777,992	7,774,921	9.99	777,992	7,774,921	9.99
Transfer data size bytes	65,136,692	658,239,572	10.11	95,608,832	953,860,096	9.98
Data Copy Time s	22.86	234.80	10.27	8.96	83.39	9.31
Data Transfer Speed MB/s	2.72	2.67	0.98	10.18	10.91	1.07
Sum of Indexes Size bytes	45,506,560	453,607,424	9.97	45,506,560	453,607,424	9.97
Indexes Creation/CopyTime s	10.58	358.94	33.93	4.67	41.49	8.88
Indexes Transfer Speed MB/s	NA	NA	NA	9.29	10.43	1.12
Total time s	33.44	593.74	17.76	13.63	124.88	9.16

applications. One could further reduce this cost by using checkpoints and the hybrid approach discussed before. Note, however, the result for the index recreation time using logical copies: even though the ratio of tuples is approximately 10:1, the ratio of index recreation times is about 34:1. This is due to the involved sorting algorithm which is needed to rebuild the indexes.

8.3 Physical copy: The PITR approach

In our prototype we create new satellites by physically copying an existing satellite to a fresh, empty satellite. As already said, the main advantage over a logical copying process (e.g., *dump-and-restore*) is the time saved by not having to re-create any data structures (like indexes). We assume that the master has one *primary* satellite permanently attached to it. In heterogenous setups, it is the job of the administrator to create an initial primary satellite when setting up a Ganymed system. New, *secondary* satellites are then created from the primary satellite, not from the master. Reading the needed data from the primary satellite allows us to minimize the impact on the master and removes the problems created by the heterogeneity between master and satellites.

The main problem we face during the creation of new satellites is the initialization phase. We need to get a copy of the primary, install it in the secondary, apply any additional changes that may have occurred in the meantime, and start the secondary. And all this without interrupting the clients that are operating on the system. The solution we propose is based on a technique called PITR (*point-in-time-recovery*). The PITR support in PostgreSQL (only available starting with the beta releases of version 8) allows the creation of a satellite by using the REDO log files from the primary. We extend this technique so that the rest of the transactions can be obtained from the adapter that resides at the master.

Adding a dynamic satellite works in several stages. First, an copy of the primary installation is pushed into the secondary. Then, a marker transaction is submitted, which becomes part of the REDO log files at the primary. It marks those transactions executed between the time the copy of the primary was made and the time the marker was submitted (i.e., the time it took to place the copy on the secondary). In the next step, the REDO log files from the primary are copied to the new satellite and PITR is used to recover up to the marker transaction. What is left to apply are the transactions that were executed after the marker transaction. This is done by the adapter of the new satellite, which reads the contents written by the marker transaction. There, among other things, it finds the number of the latest applied writeset. It then uses this information to ask the adapter at the master for the missing writesets. Since the master keeps a (limited) history of writesets, the fresh satellite can ask for slightly older writesets. If the time between the marker transaction and the startup of the adapter is too long, the needed writesets may not exist on the master anymore and the whole operation fails. The length of the history kept at the master is a parameter that can be set at configuration time and largely depends on what type of satellites are being created.

Therefore, the history on the master has to be kept not too small, and the last steps (copying of the latest REDO files, PITR recovery and startup of the adapter) shall be as efficient as possible. The main task of the process, namely, the copying of the datafiles, is not time critical; however, it can lead to increased amounts of REDO information depending on the current update-load and the total copying time.

8.4 Results: dynamic satellites

We have performed two sets of experiments to evaluate the dynamic creation of satellites. In a first experiment, we measure the time it takes to create a new satellite from a given idle primary. We use two different sizes for the primary: a small TPC-W database (scaling factors 100/10 K, with a database size of 631 MB) and a bigger one (100/1 M, requiring 1.47 GB). The results are

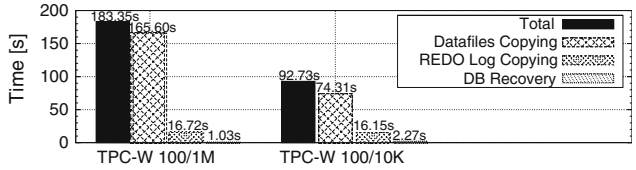

Fig. 15 Replication times with PITR

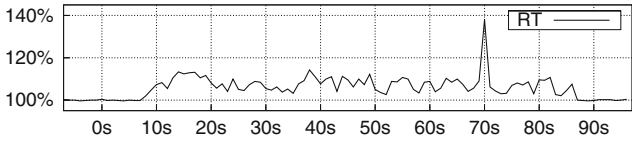

Fig. 16 Effect of replication on source machine

shown in Fig. 15: we can create a new satellite for the large primary in about 3 min., for the small primary the process only takes 1.5 min. As the times for each phase of the satellite creation indicate, the largest cost consists of the physical copying of the database. This time is directly determined by the available network bandwidth.

In a second experiment, we evaluated the overhead caused at the primary by dynamic satellite creation. To measure this, we loaded a primary satellite with a constant load of 50 TPC-W *promotion Related* transactions per second and measured the response times at the primary during the creation of a new satellite. Fig. 16 shows the results. As can be seen, the response times degrade for a short time (data copying took place from seconds 0 to 89) but quickly revert to normal once the copy has been made.

8.5 Experiment details: dynamic skyline satellites

Once we have established that satellites can be dynamically created, we need to look at the effects of dynamic creation over specialized satellites. For this purpose we ran an experiment with dynamic skyline satellites (characteristics identical to the ones described above).

In the experiment, an Oracle master is extended with a PostgreSQL primary satellite configured to offer the skyline facility over a *subset* of the TPC-W data (*order* and *order_line* tables). The system was loaded with a varying number of skyline query clients over time. Additionally, the master database was loaded with 50 TPC-W *NewOrder* update transactions per second, hence, each running satellite also had to consume 50 writesets per second. The management console was configured with a pool of spare satellite machines which it activates according to the policy described in the footnote in Sect. 2.8 (the exact nature of the policy is not relevant for the purposes of this paper). Response times were measured relative to the results of a single client working on the primary satellite. Note that our current implementation can only create one secondary satellite per primary satellite at a time.

8.6 Results: dynamic skyline satellites

The results are shown in Fig. 17. The figure shows how the number of clients varies over time (with two peaks of 12 clients), the number of satellites spawned by the management console in response to the load, and the actual response time observed by the clients (on average over all clients). In the first peak that has a slow increase of clients, the system can respond well by gradually increasing the number of satellites so that the response time slowly converges to the initial one with only one client. In the second peak, there is a surge of clients that saturate the system. As a result, the copying of the source tables is slower than before. Nevertheless, the system eventually manages to create enough satellites to cope with the new load.

As shown, the dynamic process not only spawns new satellites, it also removes the ones no longer needed.

8.7 Discussion: dynamic satellites

These experiments demonstrate that satellites can be created dynamically. The time it takes to create new satellites is dominated by the size of the data to be copied, the available network bandwidth, and the load at the primary. Again, this favors specialized satellites that do not need to copy as much data as full replication satellites. Together with the results we show for static specialized satellites, the conclusion is that satellites can be used to great advantage to extend the functionality of existing database engines and to do so dynamically, on demand. This ability opens up several interesting applications for satellites such as database services within a data grid, dynamic database clusters and autonomic scalability by dynamically spawning specialized satellites.

As in the case of full replication, the basic procedure described here can be optimized in many different ways. For instance, the machines used for creating satellites could already store an old version of the database. Then dynamic creation would only involve replaying the log. Also, many modern computer clusters have several networks, thereby providing more bandwidth. The slow reaction observed when new satellites are created from a saturated system can be avoided by using a more aggressive policy for creating satellites. Also, for very large increases in load, several satellites could be created simultaneously, rather than one at a time. We leave these optimizations for future work.

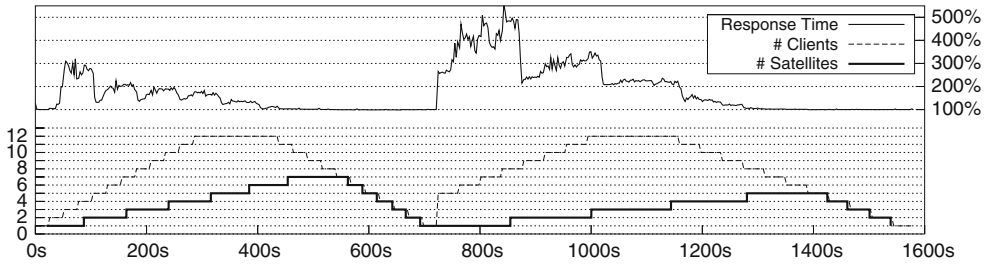

Fig. 17 Autonomic adaptation to a varying number of skyline clients

9 Related work

Work in Ganymed has been mainly influenced by C-JDBC [10], an open source database cluster middleware based on a JDBC driver. C-JDBC is meant mainly for fault tolerance purposes and it imposes several limitations in the type of load that can be run. It offers variants of the *Read-One Write-All* approach with consistency guaranteed through table level locking. C-JDBC differs from Ganymed in that it duplicates database functionality at the middleware level, including locking and writeset extraction and propagation.

Daffodil Replicator [13] is an open source tool that allows data extraction from several database engines and replication in heterogeneous engines. It is based on triggers and stored procedures but does not provide any degree of consistency to the client. Clients must also connect exclusively either to the master or to a replica and therefore do not see a single database.

Distributed versioning and conflict aware scheduling is an approach introduced in [5,6]. The key idea is to use a middleware-based scheduler which accepts transactions from clients and routes them to a set of replicas. Consistency is maintained through the 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. Similar to the C-JDBC approach, SQL statements have to be parsed at the middleware level for locking purposes.

Group communication has been proposed for use in replicated database systems [3,2,20]; however only a few working prototypes are available. Postgres-R and Postgres-R(SI) [21,35] are implementations of such a system, based on modified versions of PostgreSQL (v6.4 and v7.2). The advantage of these approaches is the avoidance of centralized components. Unfortunately, in the case of bursts of update traffic, this becomes a disadvantage, since the system is busy resolving conflicts between the replicas. In the worst case, such systems are slower than a single instance database and throughput increases at the cost of a higher response time. A solution to the problem of high conflict rates in group communication systems is the partition of the load [18]. In this approach, although all replicas hold the complete data set, update transactions cannot be executed on every replica. Clients have to predeclare for every transaction which elements in the database will be updated (so called *conflict classes*). Depending on this set of conflict classes, a so-called *compound conflict class* can be deduced. Every possible compound conflict class is statically assigned to a replica, replicas are said to act as *master site* for assigned compound conflict classes. Incoming update transactions are broadcasted to all replicas using group communication, leading to a total order. Each replica decides then if it is the master site for a given transaction. Master sites execute transactions, other sites just install the resulting writesets, using the derived total order. Recently, the work has been extended to deal with autonomous adaption to changing workloads [26].

In contrast to this work [6,10,26], Ganymed does not involve table level locking for updates, nor does it force the application programmer to pre-declare the structure of each transaction or to send transactions as full blocks. Furthermore, Ganymed supports partial replication, something that would be very difficult to do with the systems mentioned. Much of such existing work applies exclusively to full replication [5,23,26,31,35].

Recent work has demonstrated the possibility of caching data at the edge of the network [1,4,17,22] from a centralized master database. This work, however, concentrates on caching at the database engine level and it is very engine specific. Some of these approaches are not limited to static cache structures; they can react to changes in the load and adapt the amount of data kept in the cache. To local application servers, the caches look like an ordinary database system. In exchange with decreased response times, full consistency has to be given up.

In terms of the theoretical framework for Ganymed, the basis for consistency in the system is the use of snapshot isolation [7]. SI has been identified as a more

flexible consistency criteria for distribution and federation [32]. It has also been studied in conjunction with atomic broadcast as a way to replicate data within a database engine [19]. This work was the first to introduce the optimization of propagating only writesets rather than entire SQL statements. The first use and characterization of snapshot isolation for database replication at the middleware level is our own work [28]. In that early system, SI was used as a way to utilize lazy replication (group communication approaches use eager replication and, thus, enforce tight coupling of all databases) and to eliminate the need for heavy processing at the middleware level (be it versioning, locking, or atomic broadcast functionality). The level of consistency introduced by Ganymed (clients always see their own updates and a consistent snapshot) was independently formalized as strong session consistency in [14], although those authors emphasize serializability. Nevertheless, note that Ganymed provides a stronger level of consistency in that clients do not only get strong session consistency, they get the latest available snapshot. More recently, [23] have proposed a middleware-based replication tool that also uses SI. Conceptually, the system they describe is similar to the one described in [28]. The differences arise from the use of group communication and the avoidance of a central component in [23] and the corresponding limitations that this introduces: load balancing is left to the clients, partial replication is not feasible, and the middleware layer is limited to the scalability of the underlying atomic broadcast implementation. More importantly, unless clients always access the same replica, [23] does not provide strong session consistency as clients are not guaranteed to see their own updates. Clients are not even guaranteed to see increasing snapshots as time travel effects are possible when consecutive queries from the same client are executed on different replicas that are not equally up-to-date.

A possible extension of our system is to implement a solution similar to that proposed in [31], where clients can request to see a snapshot not older than a given time bound. In our system, this can be achieved by using satellites that lag behind in terms of snapshots.

10 Conclusions

This paper has presented the design, implementation, and evaluation of a database extension platform (Ganymed). The core of the platform is what we call satellite databases: a way to extend and scale out existing database engines. Satellite databases are lightweight databases that implement either copies of the master database or specialized functionality that is not available at the master database. In the paper, we describe the architecture of the system and its implementation. We also provide extensive experiments that prove the feasibility of the idea by showing how satellite databases perform in a wide variety of settings: full replication, specialized functionality (skyline queries and keyword search), dynamic creation of satellites, and the overall performance of the system.

Acknowledgements Many people have contributed valuable ideas to the system described in this paper. We are grateful to Roger Rüegg, Stefan Nägeli, Mario Tadey and Thomas Gloor for their help in developing and benchmarking different parts of the system. We would also like to thank Donald Kossmann for his help with skyline queries; Dieter Gawlick for his input on Oracle; C. Mohan and Bruce Lindsay for their input on DB2 and other IBM replication tools; Adam Bosworth for new ideas on how to use satellites; and Mike Franklin and the database group at UC Berkeley for many discussions on the use and possibilities of satellite databases. Many thanks also go to Willy Zwaenepoel who gave us feedback when we came up with the initial idea for RSI-PC.

References

1. Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald, B.: Cache tables: paving the way for an adaptive database cache. In: Proceedings of the 29th International Conference on Very Large Data Bases. Berlin, Germany, September 9–12, 2003
2. Amir, Y., Tutu, C.: From Total Order to Database Replication. Technical report, CNDS (2002). http://www.citeseer.nj.nec.com/amir02from.html
3. Amir, Y., Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Ciarfella, P.: The totem single-ring ordering and membership protocol. ACM Trans. Comput. Syst. **13**(4), 311–342 (1995). http://www.citeseer.nj.nec.com/amir95totem.html
4. Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: DBProxy: A dynamic data cache for web applications. In: Proceedings of the 19th International Conference on Data Engineering, Bangalore, India, 5–8, March 2003
5. Amza, C., Cox, A.L., Zwaenepoel, W.: 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), pp. 230–241 (2005)
6. Amza, C., Cox, A.L., Zwaenepoel, W.: Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In: Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16–20, Proceedings (2003)
7. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of the SIGMOD International Conference on Management of Data, pp. 1–10 (1995)
8. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley, Reading (1987)
9. Borzsonyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: IEEE Conference on Data Engineering, pp 421–430. Heidelberg, Germany (2001)

10. Cecchet, E.: C-JDBC: a middleware framework for database clustering. IEEE Data Engineering Bulletin, vol. 27, no. 2 (2004)
11. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: ICDE, pp 717–816 (2003)
12. Council, T.T.P.P.: TPC-W, a transactional web e-commerce benchmark. http://www.tpc.org/tpcw/
13. Daffodil Replicator: http://sourceforge.net/projects/daffodil-replica
14. Daudjee, K., Salem, K.: Lazy database replication with ordering guarantees. In: Proceedings of the 20th international conference on data engineering (ICDE 2004), Boston, MA, USA, pp 424–435, 30 March – 2 April 2004
15. Fekete, A.D.: Serialisability and snapshot isolation. In: Proceedings of the Australian Database Conference, pp. 210–210 (1999)
16. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Sasha, D.: Making snapshot isolation serializable. http://www.cs.umb.edu/isotest/snaptest/snaptest.pdf
17. Härder, T., Bühmann, A.: Query processing in constraint-based database caches. IEEE Data Engineering Bulletin, vol. 27, no. 2 (2004)
18. Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B., Alonso, G.: Improving the scalability of fault-tolerant database clusters. In: IEEE 22nd International Conference on Distributed Computing Systems, ICDCS'02, Vienna, Austria, pp. 477–484 (2002)
19. Kemme, B.: Database replication for clusters of workstations. Ph.D. thesis, Diss. ETH No. 13864, Department of Computer Science, Swiss Federal Institute of Technology Zurich (2000)
20. Kemme, B.: Implementing database replication based on group communication. In: Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo 2002), Bertinoro, Italy (2002). http://www.citeseer.nj.nec.com/pu91replica.html
21. Kemme, B., Alonso, G.: 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
22. Larson, P.A., Goldstein, J., Zhou, J.: Transparent mid-tier database caching in SQL server. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp 661–661. ACM Press, (2003). DOI http://www.doi.acm.org/10.1145/872757.872848
23. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: SIGMOD'05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 419–430 (2005)
24. Lipasti, M.H.: Java TPC-W Implementation Distribution of Prof. Lipasti's Fall 1999 ECE 902 Course. http://www.ece.wisc.edu/pharm/tpcw.shtml
25. Microsoft SQL Server 2005: http://www.microsoft.com/sql/2005
26. Milan-Franco, J.M., Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B.: Adaptive distributed middleware for data replication. In: Middleware 2004, ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, Canada, October 18–22, Proceedings (2004)
27. Oracle Database Documentation Library: Oracle Streams Concepts and Administration, Oracle Database Advanced Replication, 10g Release 1 (10.1) (2003). http://www.otn.oracle.com
28. Plattner, C., Alonso, G.: Ganymed: Scalable replication for transactional web applications. In: Middleware 2004, 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18–22, Proceedings (2004)
29. PostgreSQL Global Development Group. http://www.postgresql.org
30. Project, T.E.O.: Octopus, a simple Java-based Extraction, Transformation, and Loading (ETL) tool. http://www.octopus.objectweb.org/
31. Röhm, U., Böhm, K., Schek, H., Schuldt, H.: FAS – A freshness-sensitive coordination middleware for a cluster of OLAP components. In: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China, pp. 754–765 (2002)
32. Schenkel, R., Weikum, G.: Integrating snapshot isolation into transactional federation. In: Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, Proceedings, 6–8 September 2000
33. Stolte, E., von Praun, C., Alonso, G., Gross, T.R.: Scientific data repositories: designing for a moving target. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 349–360 (2003)
34. Weikum, G., Vossen, G.: Transactional information systems. Morgan Kaufmann, San Francisco (2002)
35. Wu, S., Kemme, B.: 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), pp. 422–433