Pangea: An Eager Database Replication Middleware guaranteeing Snapshot Isolation without Modification of Database Servers

Takeshi Mishima
The University of Tokyo
mishima@hal.rcast.u-tokyo.ac.jp

Hiroshi Nakamura
The University of Tokyo
nakamura@hal.rcast.u-tokyo.ac.jp

ABSTRACT

Recently, several middleware-based approaches have been proposed. If we implement all functionalities of database replication only in a middleware layer, we can avoid the high cost of modifying existing database servers or scratch-building. However, it is a big challenge to propose middleware which can enhance performance and scalability without modification of database servers because the restriction may cause extra overhead. Unfortunately, many existing middleware-based approaches suffer from several shortcomings, i.e., some cause a hidden deadlock, some provide only table-level locking, some rely on total order communication tools, and others need to modify existing database servers.

In this paper, we propose Pangea, a new eager database replication middleware guaranteeing snapshot isolation that solves the drawbacks of existing middleware by exploiting the property of the *first updater wins* rule. We have implemented the prototype of Pangea on top of PostgreSQL servers without modification. An advantage of Pangea is that it uses less than 2000 lines of C code. Our experimental results with the TPC-W benchmark reveal that, compared to an existing middleware guaranteeing snapshot isolation without modification of database servers, Pangea provides better performance in terms of throughput and scalability.

1. INTRODUCTION

Many database researchers may think of database replication as a solved problem; however, there are few practical solutions which meet customers' expectations [8]. Traditionally, most database replication approaches need modifying existing database servers or scratch-building. The strong point of the approaches is that they may introduce only a minimal overhead. However, the modification or scratch-building is too expensive, since the code of database servers is large and complex. This is one of the reasons that most of the approaches were not implemented as a practical prototype and were evaluated only as simulation based studies.

Recently, several middleware-based approaches have been proposed [20, 17, 16, 3, 18, 19, 10, 14, 15, 4, 9, 12]. If we implement all functionalities of database replication only in a middleware layer, we can avoid the high cost of the modification or scratch-building. Moreover, the middleware offers flexibility, which does not depend on a particular vendor, permits heterogeneous settings and can be maintained independently of database servers. These benefits have motivated several researchers to study various replication middlewares. However, it is a big challenge to propose a middleware that can enhance performance and scalability without modification of database servers because the restriction may cause extra overhead.

Database replication approaches can be categorized into *eager* and *lazy* replication schemes [13]. Eager replication keeps all database servers (replicas) exactly synchronized by updating all the replicas as part of one atomic transaction. Lazy replication asynchronously propagates replica updates to other replicas after the updating transaction commits.

Most existing middleware-based approaches [20, 17, 16, 3, 18, 19, 10] belong to lazy replication. This is due to the fact that eager replication generally offers less performance and scalability than lazy replication [13]. However, under the condition that we use off-the-shelf database servers without modification, the fact does not necessarily hold, i.e., maintaining consistency of lazy replication may produce non-trivial overhead.

We focus on eager replication because we think the simplicity of eager replication is suitable for the aim that replication functionalities produce little overhead even if all of them are implemented only in a middleware layer. Furthermore, we adopt snapshot isolation (SI) [5] as a correctness criterion because SI is not only provided by practical database servers such as Oracle, SQL Server and PostgreSQL but also widely used, increasing performance by executing transactions concurrently.

Existing eager replication middlewares have some problems. First, if the execution order of conflicting write operations on one replica differs from that on another replica, a "hidden" deadlock can occur, which lies across replicas and prevents the middleware from receiving responses from all replicas due to blocking the latter operation[1]. To avoid this "hidden" deadlock, the middleware must make all replicas execute the conflicting write operations in the same order. Some existing eager middlewares [9, 12] do not have this mechanism. Some middlewares [14, 15] can avoid this "hidden" deadlock by using group communication. However,

[1]In eager replication, a middleware makes progress after receiving all responses as stated in Sec. 3.1.

VLDB '09, August 24-28, 2009, Lyon, France

the middlewares not only suffer from inevitable overhead by the group communication but also need to modify database servers. Other middleware [4] assigns a unique version number to each request and sends the requests to all replicas in the same order of the version number sequentially. Although the middleware avoids the deadlock, the sequential execution by managing version numbers leads to table-level locking. This coarse grain control causes degradation of throughput.

Second, most existing eager middlewares [4, 9, 15, 12] do not guarantee SI. SI-Rep [14] guarantees SI but checks write-write conflict at commit time. This is not compatible with the first updater wins rule [11] used by practical database servers such as Oracle and PostgreSQL.

In this paper, we tackle the problems and propose Pangea, a novel eager database replication middleware guaranteeing SI without modification of database servers. Our main contributions are as follows:

- We propose a new correctness criterion for eager replicated database systems called *global snapshot isolation (GSI)*. GSI not only guarantees SI to clients but also maintains consistency between database servers.
- We propose a novel concurrency control which creates snapshots in a synchronized fashion across replicas, and regulates the order of conflicting write operations by delegating a designated "leader" replica to detect conflicting write operations with the property of the first-updater-wins rule. The regulation allows us to avoid the hidden deadlock. To the best of our knowledge, this paper is the first to show the effective exploitation of the first updater wins rule.
- We propose Pangea, a novel eager database replication middleware guaranteeing GSI with the key control. Compared with existing eager replication middlewares, Pangea provides the highest concurrency because the tuple-level locking allows Pangea to execute non-conflicting write operations concurrently. Furthermore, compared with existing lazy replication middlewares, Pangea prevents a particular replica from overloading because read operations can go to any replica, even if executed by an update transaction.
- Pangea has been easily implemented using PostgreSQL servers without modification due to the simplicity of our control. An advantage of Pangea is that it uses less than 2000 lines of C code.
- The experiments using the TPC-W benchmark show that, compared to an existing middleware guaranteeing SI without modification of database servers, Pangea provides better performance in terms of throughput and scalability.

The remainder of this paper is organized as follows. Section 2 introduces the transaction model and the important feature of SI. In Section 3, we develop a formal characterization of SI in eager replicated systems and define GSI. In Section 4, we propose Pangea, a new eager database replication middleware guaranteeing SI without modification of database servers. Section 5 presents experimental results. Section 6 talks about related work. Section 7 concludes the paper.

2. DATABASE MODEL

In this section, we introduce the formal apparatus that is necessary to reason about SI.

2.1 Transaction

A *transaction* T_i is a sequence, i.e., total order $<$, of read and write operations on data items and starts with a begin operation b_i. The subscript i identifies a particular transaction and distinguishes it from other transactions. $r_i(x)$ represents a read operation by T_i on data item x and $w_i(x)$ means a write operation of x. We use c_i and a_i to denote T_i's commit and abort operation, respectively. We assume a new snapshot is created for T_i when s_i called a *snapshot operation* is executed. In the practical database servers such as Oracle, SQL Server and PostgreSQL, a new snapshot is created for T_i just before the first $r_i(x)$ or $w_i(x)$ is executed. It is convenient for reasoning about SI to use s_i and this does not interfere the result.

2.2 Versioned Data Item

To reason about the execution of operations under SI, we need to represent a versioned data item because SI is one of multi-version concurrency control. For each data item x, we denote the versions of x by x_i, x_j, ..., where the subscript is the index of the transaction that wrote the version. Thus, each write operation is always of the form $w_i(x_i)$, where the version subscript equals the transaction subscript. The notation $r_i(x_j)$ represents a read operation by transaction T_i that returns the version of the data item x_j written earlier by transaction T_j. We assume that all data items start with zero versions, x_0, y_0, z_0, ... (we imagine that x, y, z, ... have all been installed by a progenitor transaction T_0).

2.3 Schedule and History

When a set of transactions are executed concurrently, their operations may be interleaved. We define a *schedule* as the order in which a scheduler of a database server intends to execute operations of the transactions. A *history* indicates the order in which the operations of the transactions on the schedule were actually executed. Note that a schedule is a plan for the future and therefore not all operations can be actually executed. Consider, for instance, two transactions T_i and T_j:

$$T_i : \quad b_i\, s_i\, w_i(x_i)\, c_i$$
$$T_j : \quad b_j\, s_j\, w_j(x_j)\, c_j$$

and suppose a scheduler makes a schedule S as follows:

$$S = b_i\, b_j\, s_i\, s_j\, w_i(x_i)\, w_j(x_j)\, c_i\, c_j$$

In addition, suppose $w_j(x_j)$ has not been executed successfully for some reason. Then T_j cannot be committed but aborted. So the order that all operations were actually executed, namely a history H, is as follows:

$$H = b_i\, b_j\, s_i\, s_j\, w_i(x_i)\, w_j(x_j)\, c_i\, a_j$$

In T_j, a_j were executed in the place of c_j.

2.4 Snapshot Isolation

We designate the time when a transaction T_i starts as $start(T_i)$ which is equal to the time when a snapshot operation s_i has been executed. Also, we address the time when T_i ends as $end(T_i)$ which is equal to the time when

a commit operation c_i has been executed. A transaction T_i executing under SI reads data from the committed state of the database as of start(T_i), namely the *snapshot*, and writes its own snapshot, so it can read the data from the snapshot that it has previously written.

The interval in time from start(T_i) to end(T_i), represented as [start(T_i), end(T_i)], is called *transactional lifetime*. We say that two transactions T_i and T_j are *concurrent* if their transactional lifetime overlap, i.e.:

$$[\,start\,(T_i),\ end\,(T_i)\,] \cap [\,start\,(T_j),\ end\,(T_j)\,] \neq \emptyset$$

Reading from a snapshot means that a transaction never sees the partial results of other transactions: T_i sees all the changes made by transactions that commit before start(T_i), and it sees no changes made by transactions that commit after start(T_i).

In SI, a read operation never blocks write operations, and vice versa. This feature increases concurrent execution and thereby brings higher throughput. A write operation includes a SELECT FOR UPDATE query in addition to UPDATE, DELELE and INSERT queries because the SELECT FOR UPDATE query needs a tuple-level lock as well.

2.5 First Updater Wins Rule

Originally, two concurrent transactions T_i and T_j causing write-write conflict obey the *first committer wins* rule under SI [5]. However, practical database servers such as Oracle and PostgreSQL adopt the *first updater wins* rule [11] as follows.

If T_i updates the data item x, then it will take a write lock on x; if T_j subsequently attempts to update x while T_i is still active, T_j will be prevented by the lock on x from making further progress. If T_i then commits, T_j will abort; T_j will be able to continue only if T_i drops its lock on x by aborting. If, on the other hand, T_i updates x but then commits before T_j attempts to update x, there will be no delay due to locking, but T_j will abort immediately when it attempts to update x (the abort does not wait until T_j attempts to commit). In short, a transaction will successfully commit only if no other concurrent transaction causing conflict has already been committed.

Although the ultimate effect of both rules is the same, the first committer wins rule causes unnecessary execution of operations which will be aborted and thereby wastes time and resources in vain before aborting. We assume that each replica guarantees SI with the first updater wins rule.

This apparatus motivates the following definition of SI-schedule.

Definition 1. (SI-schedule) Let $\mathcal{T}$ be a set of transactions, where each transaction T_i has operations $o_i \in \{b_i, s_i, r_i, w_i, c_i, a_i\}$. An SI-schedule S over $\mathcal{T}$ has the following properties.

(i) $r_i(x)$ is mapped to $w_j(x)$ such that $w_j(x) < c_j < s_i < r_i(x)$ and there are no other operations $w_k(x)$ and $c_k (k \neq j)$ with $w_k(x) < s_i$ and $c_j < c_k < s_i$

(ii) If $w_i(x)$ and $w_j(x)$ causes write-write conflict and $w_i(x) < w_j(x)$, then $(c_i < a_j)$ or $(a_i < c_j)$.

(i) means that the version function maps each read operation $r_i(x)$ to the most recent committed write operation $w_j(x)$ as of the time of execution of s_i. (ii) means the first updater wins rule, i.e., the transaction which includes the first updater operation can commit successfully and the other transaction has to abort.

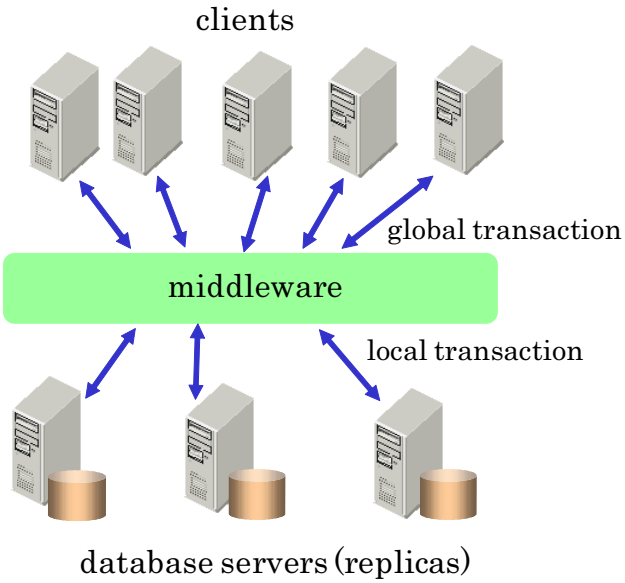

Figure 1: Model of An Eager Replication System

3. SNAPSHOT ISOLATION IN REPLICATED SYSTEMS

In this section, we develop a formal characterization of SI in eager replicated systems with a middleware and then propose a new correctness criterion called *global snapshot isolation*.

3.1 Model of An Eager Replication System

Figure 1 shows the model of an eager replication system. If the system has n replicas, R^m represents the m-th replica ($1 \leq m \leq n$), where the superscript is the index of the replica. A middleware lies between replicas and clients. Transactions are never sent by clients in a single block, i.e., they are processed by the middleware statement by statement.

Basically, in eager replication, when a request is received from a client, the middleware sends this request to replicas. Then, receiving responses from all the replicas, the middleware sends back one response to the client. If we optimize the protocol, the final effect must be the same as that of the basic protocol. Note that clients must not access replicas directly.

3.2 Mapping Function

We assume that clients do not use stored procedures, i.e., business logic is usually implemented in application servers. We also assume that we can distinguish a read-only or an update transaction at the beginning of each transaction[2].

The middleware receives a request including one operation from a client as an operation $o_i \in \{b_i, s_i, r_i, w_i, c_i, a_i\}$ of a *global transaction* T_i, projects o_i onto replica R^m as an operation $o_i^m \in \{b_i^m, s_i^m, r_i^m, w_i^m, c_i^m, a_i^m\}$ of a *local transaction* T_i^m, and then sends the request to R^m. R^m receives the request and executes o_i^m with SI-schedule. We define

[2]In Java clients, for example, this can be done by executing `Connection.setReadOnly()` method.

the mapping function that takes a set of global transactions $\mathcal{T}$ and a set of replicas $\mathcal{R}$ as input and transforms $\mathcal{T}$ into a set of local transactions $\mathcal{T}^m$ $\{T_i^m | R^m \in \mathcal{R}\}$.

Definition 2. (mapping function) Let $\mathcal{R}$ be a set of replicas in a replicated database system. Let $\mathcal{T}$ be a set of global transactions, where each global transaction T_i has operations $o_i \in \{b_i, s_i, r_i, w_i, c_i, a_i\}$. Each replica R^m executes a set of local transactions $\mathcal{T}^m$, where each local transaction T_i^m has operations $o_i^m \in \{b_i^m, s_i^m, r_i^m, w_i^m, c_i^m, a_i^m\}$. The mapping function is as follows:

(i) If a global transaction T_i is a read-only transaction, the mapping function picks up one replica R^p and all operations o_i of T_i are mapped to o_i^p of T_i^p. That is, $T_i^p = T_i$ and the other local transactions $T_i^m (1 \leq m \leq n, m \neq p) = \emptyset$.

(ii) If a global transaction T_i is an update transaction, the mapping function picks up one replica R^q and r_i of T_i are mapped to r_i^q of T_i^q, and b_i, s_i, w_i, c_i, a_i of T_i are mapped to $b_i^m, s_i^m, w_i^m, c_i^m, a_i^m$ of $T_i^m (1 \leq m \leq n)$, respectively. That is, $T_i^q = T_i$ and the other local transactions $T_i^m (1 \leq m \leq n, m \neq q)$ excludes read operations from T_i.

Consider, for example, a read-only global transaction T_i:

$$T_i : b_i\, s_i\, r_i(x_j)\, r_i(y_k)\, r_i(z_l)\, c_i$$

The mapping function transforms T_i into a set of local transactions T_i^m:

$$\begin{aligned} T_i^p &: \quad b_i^p\, s_i^p\, r_i^p(x_j)\, r_i^p(y_k)\, r_i^p(z_l)\, c_i^p \\ T_i^m &: \quad \emptyset (1 \leq m \leq n, m \neq q) \end{aligned}$$

Consider, for example, an update global transaction T_j:

$$T_j : b_j\, s_j\, r_j(x_k)\, w_j(y_j)\, c_j$$

The mapping function transforms T_j into a set of local transactions T_j^m:

$$\begin{aligned} T_j^q &: \quad b_j^q\, s_j^q\, r_j^q(x_k)\, w_j^q(y_j)\, c_j^q \\ T_j^m &: \quad b_j^m\, s_j^m\, w_j^m(y_j)\, c_j^m (1 \leq m \leq n, m \neq q) \end{aligned}$$

In summary, it is necessary that read operations are executed on only one replica for high performance and scalability.

3.3 SI-Equivalence

When we reason about consistency between replicas, the equivalence of two schedules is an inevitable concept like classical serializable theory [6]. Although a similar definition has been made by Lin et al. [14], they assume SI with the first committer wins rule, i.e., the replica checks write-write conflicts only at commit time. Unlike [14], we assume the first updater wins rule. We say two SI-schedule S^m and S^n are *SI-equivalent* as follows:

Definition 3. (SI-equivalence) Let $\mathcal{T}$ be a set of transactions, where each transaction T_i has operations $o_i \in \{b_i, s_i, r_i, w_i, c_i, a_i\}$. Let $\mathcal{R}$ be a set of replicas in a replicated database system, where each replica R^m has a SI-schedule S^m. Let S^m and S^n be two SI-schedules over the same set of transactions $\mathcal{T}$ and have the same operations. S^m and S^n are *SI-equivalent* if for any $T_i, T_j \in \mathcal{T}$ the following holds.

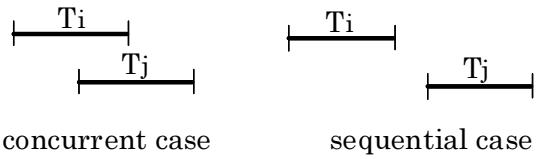

Figure 2: The relative relation between T_i and T_j

(i) $(s_j < c_i) \in S^m \Leftrightarrow (s_j < c_i) \in S^n$; or
$(c_i < s_j) \in S^m \Leftrightarrow (c_i < s_j) \in S^n$.

(ii) w_i and w_j cause conflict, $(w_i < w_j) \in S^m$
$\Leftrightarrow$ w_i and w_j cause conflict, $(w_i < w_j) \in S^n$.

Whether the snapshot of T_j includes the change of T_i or not depends on the relative relation between T_i and T_j as shown in Fig. 2. In the concurrent case, namely $s_j < c_i$, the snapshot of T_j does not include the change of T_i. In the sequential case, namely $c_i < s_j$, the snapshot of T_j includes the change of T_i. Thus, (i) means that the relative order of snapshot and commit operations in S^m must be the same as that in S^n.

(ii) utilizes the fact that the outcome of a concurrent execution of transactions depends on the relative ordering of conflicting write operations. Therefore, the relative order of conflicting write operations in S^m must be the same as that in S^n.

3.4 Global Snapshot Isolation

We propose a new correctness criterion called *global snapshot isolation (GSI)*. GSI is not only to provide SI to clients but also to maintain consistency between replicas. To guarantee GSI with a minimal overhead, SI-equivalence is too strong, i.e., it is not necessary that SI-equivalence holds over the same set of transactions which have the same set of operations. It is necessary that SI-equivalence holds over the set of local transactions mapped from global transactions. We define global snapshot isolation as follows:

Definition 4. (global snapshot isolation) Let $\mathcal{R}$ be a set of replicas in a replicated database system. Let $\mathcal{T}$ be a set of global transactions, where each global transaction T_i has operations $o_i \in \{b_i, s_i, r_i, w_i, c_i, a_i\}$. Let S^m be the SI-schedule over the set of local transactions $\mathcal{T}^m$ at replica $R^m \in \mathcal{R}$. We say that $\mathcal{R}$ guarantees GSI if the following properties hold.

(i) There is a mapping function such that $\cup_m \mathcal{T}^m = f(\mathcal{T}, \mathcal{R})$

(ii) There is an SI-schedule S over $\mathcal{T}$ such that for each S^m and $T_i^m, T_j^m \in \mathcal{T}^m$ being transformations of T_i and T_j:

  (a) If T_i is a read-only transaction,
  there exists $m \in T_i^m = T_i$

  (b) If T_i is an update transaction,
  $(s_j < c_i) \in S \Rightarrow (s_j^m < c_i^m) \in S^m$; or
  $(c_i < s_j) \in S \Rightarrow (c_i^m < s_j^m) \in S^m$.

  (c) If T_i is an update transaction,
  w_i and w_j cause conflict and $(w_i < w_j) \in S \Rightarrow$
  w_i^m and w_j^m cause conflict and $(w_i^m < w_j^m) \in S^m$.

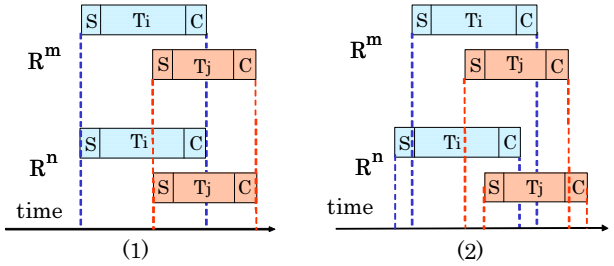

Figure 3: Snapshot Acquisition

4. PANGEA MIDDLEWARE

In this section, we propose Pangea, an eager database replication middleware guaranteeing SI without modification of database servers.

4.1 Concept

Pangea has the following two functionalities to guarantee SI.

(1) snapshot acquisition control: Pangea makes all replicas create the same snapshots on different replicas.

(2) write operation control: Pangea makes every replica execute write-write conflicting operations in the same order.

Furthermore, Pangea has the following functionality to enhance throughput by load balancing.

(3) Pangea makes one replica create the snapshot of a transaction and execute read operations of the transaction.

4.2 Snapshot Acquisition Control

To achieve the first functionality, we propose a snapshot acquisition control which makes a pair of transactions on different replicas acquire the same snapshot.

Schenkel et al. [22] proposed a method to make all database servers get the same snapshots in a federated database system with the atomic commit protocol. Although we can apply it to a replicated database system, it is too strong to construct a practical replicated database system, i.e., not only a local transaction T_i^m at R^m and a local transaction T_i^n at R^n are forced to start at the same time but also commit operations are executed at the same time exactly as shown in Fig. 3.(1).

Rather than their approach, our proposal is to make the relative order of snapshot and commit operations at R^m and those at R^n be the same as shown in Fig. 3.(2). It can realize the snapshot acquisition control with a minimal overhead since our proposal is weaker.

Pangea has several threads and assigns one thread to each client. In order to regulate the relative order of snapshot and commit operations, Pangea uses mutual exclusion. When a thread receives a snapshot operation from a client, it tries to enter the critical region. When it enters the critical region, there are no replicas executing a commit operation. So, by sending the snapshot operation to all replicas, all snapshots must be the same. The reverse is also true. When the thread receiving a commit operation enters the critical region, there are no replicas executing a snapshot operation. In this way, Pangea preserves the relative order of snapshot and commit operations. Rather than the method of [22], multiple snapshot operations can be executed concurrently. Also, commit operations can be executed concurrently. This property leads to better performance.

4.3 Write Operation Control

To achieve the second functionality, we propose the write operation control which makes write-write conflicting operations be executed in the same order. Unlike the conventional table level locking approaches [4], our approach brings tuple level concurrency control. Also, rather than [9, 12] that causes a hidden deadlock due to the fact that the order of write-write conflicting operations changes, our approach prevents the order from changing.

The key point is how to distinguish operations which will conflict on a particular data item. So far, it has been considered very difficult to do this by middleware based approaches, since a middleware cannot know whether operations will conflict or not by parsing SQL statements. This is one of the reasons why middleware based approaches achieve only the table level concurrency control, and therefore cannot enhance performance and scalability.

4.3.1 *Leader and Follower Replicas*

The key idea of our control is to delegate one replica called *leader* to detect conflicts on tuple level and to decide the execution order of write operations, namely the first updater wins rule, and to make the others called *followers* execute the write operations in the same order as the leader has decided. It does not matter which replica Pangea selects as a leader but once Pangea completes the selection, the roles are not changed until the leader fails.

Our proposed write operation control is the next simple protocol that Pangea manages sending write operations and receiving responses as follows:

(1) send all write operations only to the leader,

(2) receive responses from the leader,

(3) send the write operations corresponding to the responses to all the followers,

(4) receive responses from all the followers, and

(5) send back a response to each client.

Consider, for instance, that Pangea sends the three write operations W_a, W_b and W_c shown in Fig.4. We assume that W_a and W_b will conflict. After receiving write operations W_a, W_b and W_c from clients, Pangea sends all the write operations only to the leader (step (1)). Then Pangea receives responses R_a and R_c corresponding to W_a and W_c, respectively (step (2)). In this case, Pangea cannot receive the response R_b corresponding to W_b since W_a and W_b conflict and R_a is returned to Pangea (this means W_a is the first updater). Recall the first updater wins rule: only the transaction that can get a write lock can progress but the others have to wait for the lock. Thus, by checking which of the responses Pangea has received, Pangea can know which of the write operations will not conflict in the followers. Next, Pangea sends only W_a and W_c to all the followers, which are guaranteed not to cause conflict (step (3)). Then Pangea receives responses from all the followers (step (4)) and sends back a response to each client(step (5)).

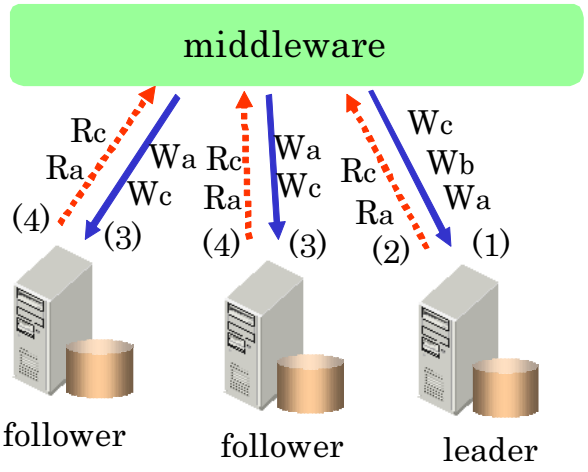

Figure 4: Write Operations Control

Surprisingly, our idea is very simple but very effective, i.e., it can not only avoid the hidden deadlock but can also achieve the tuple level locking. This fine-grained locking leads to higher throughput.

4.3.2 *Correctness*

Let $w_{i,m}$ be a m-th write operation of a transaction T_i. $L(w_{i,m})$ and $F(w_{j,n})$ show the execution of $w_{i,m}$ by the leader and the execution of $w_{j,n}$ by the follower, respectively. e_i means end operation of T_i, commit or abort operation. $L(w_{i,m}) < F(w_{j,n})$ denotes that the execution of $w_{i,m}$ by the leader occurs before the execution of $w_{j,n}$ by the follower.

THEOREM 1. *(write operation control) For any transactions T_i and T_j and corresponding write operations $w_{i,m}$ and $w_{j,n}$, respectively, if $w_{i,m}$ and $w_{j,n}$ try to modify the same data item and $L(w_{i,m}) < L(w_{j,n})$ then $F(w_{i,m}) < F(w_{j,n})$*

PROOF.
case $i \neq j$
$i \neq j$ means T_i and T_j are different transactions and $w_{i,m}$ and $w_{j,n}$ cause write-write conflict. As described in Sect.2.4, the first updater wins rule gives a write lock to the first write operation, i.e., from $L(w_{i,m}) < L(w_{j,n})$, $w_{i,m}$ is executed immediately but $w_{j,n}$ is forced to wait until T_i executes an end operation e_i and releases the write lock. Therefore, for any transactions T_i and T_j,

$$L(w_{i,m}) < e_i < L(w_{j,n}) \quad (1)$$

In the write operation control, Pangea sends the write operation to the leader prior to the follower and Pangea does not send it to the follower until it has received the response from the leader. So, for any transactions T_i,

$$L(w_{i,m}) < F(w_{i,m}) < e_i \quad (2)$$

Like T_i, for any transactions T_j,

$$L(w_{j,n}) < F(w_{j,n}) < e_j \quad (3)$$

Thus, from the equation (1), (2) and (3), for any transactions T_i, T_j, $F(w_{i,m}) < F(w_{j,n})$.
case $i = j$
If $i = j$, $w_{i,m}$ and $w_{j,n}$ are included in the same transaction T_i. In this case, $w_{i,m}$ and $w_{j,n}$ does not cause write-write conflict. Since one write operation overwrites the result of another, of course, $w_{i,m}$ and $w_{j,n}$ must be executed in the same order on the different replicas. In the write operation control, Pangea sends back the response to the client after it has received the responses from the leader and the follower. This means Pangea does not send back the response to the client until all $w_{i,m}$ have been executed. In addition, the client does not send a new operation until it has received the response corresponding to the former operation. Thus, the latter operation can not get ahead of the former operation. So, for any transactions T_i,

$$L(w_{i,m}) < L(w_{i,m+1}) \quad (4)$$

$$F(w_{i,m}) < F(w_{i,m+1}) \quad (5)$$

From $L(w_{i,m}) < L(w_{j,n})$, $i = j$ and the equation(4), $n = m+1$. Thus, the equation (5), $i = j$ and $n = m+1$, for any transactions T_i, T_j, $F(w_{i,m}) < F(w_{j,n})$. □

4.4 Read Operation Control

To achieve the third functionality, we propose the read operation control. It is not necessary that read operations are executed on all replicas because read operations do not change the state of a database. However, to send back a response to a client, one read operation must be executed on one replica. Note that a read operation must be executed on the replica that holds the snapshot of the transaction to which the read operation belongs.

4.5 Algorithm

In this section, we present the algorithm to attain the three functionalities described above. Our algorithm consists of initialization and two states, the NO_SNAPSHOT state and the SNAPSHOT_PREPARED state. Pangea assigns one thread for each client and makes the thread execute the following Algorithm I, II and III.

Algorithm I. Initialization

```
1  state = NO_SNAPSHOT
2  commit_counter = 0
3  snapshot_counter = 0
4  select one replica R^m(1 ≤ m ≤ n, m ≠ leader)
```

An initial state is set to the NO_SNAPSHOT state (I.1). Two counters, one that counts the number of threads executing commit operations and the other that counts the number of threads executing snapshot operations, are initialized to zero (I.2-3). One replica $R^m(1 \leq m \leq n, m \neq leader)$ is selected for load balancing in advance (I.4). To avoid the leader becoming a bottleneck, R^m should be selected among followers.

Algorithm II. read-only transaction

```
1.  send the request to R^m
2.  recv a response from R^m
3.  send the response to the client
```

The behavior of a thread with a read-only transaction is very simple. The thread sends a request to R^m which

has been selected in the initialization, receives a response from R^m and then sends the response to the client (II.1-3). The thread must not send the request to the other replicas because only R^m holds the snapshot of the transaction.

Algorithm III. update transaction

```
1.  if (state == NO_SNAPSHOT)
2.      enter critical region
3.      while (commit_counter > 0)
4.          sleep and wait for signal
5.      snapshot_counter ++
6.      leave critical region
7.      send the request to all replicas
8.      recv responses from all replicas
9.      send a response to client
10.     enter critical region
11.     snapshot_counter --
12.     if (snapshot_counter == 0)
13.         signal to wake up
14.     leave critical region
15.     state = SNAPSHOT_PREPARED
16. else /* SNAPSHOT_PREPARED */
17.     if (request == write operation)
18.         send the request to the leader replica
19.         recv a response from the leader replica
20.         send the request to all follower replicas
21.         recv responses from all follower replicas
22.         send a response to client
23.     else if (request == read operation)
24.         send the request to R^m
25.         recv a response from R^m
26.         send the response to the client
27.     else if (request == abort operation)
28.         send the request to all replicas
29.         recv responses from all replicas
30.         send a response to the client
31.     else /* commit operation */
32.         enter critical region
33.         while (snapshot_counter > 0)
34.             sleep and wait for signal
35.         commit_counter ++
36.         leave critical region
37.         send the request to all replicas
38.         recv responses from all replicas
39.         send a response to client
40.         enter critical region
41.         commit_counter --
42.         if (commit_counter == 0)
43.             signal to wake up
44.         leave critical region
45.         state = NO_SNAPSHOT
```

The behavior of a thread with an update transaction depends on the state. In NO_SNAPSHOT state, a snapshot for the transaction must be created by executing a snapshot operation. The thread enters the critical region and increases the `snapshot_counter` only if any commit operations are not executed (III.1-5). The counter informs the other threads that there is a thread creating snapshot and is used to guarantee that both snapshot and commit operations are not executed simultaneously. Leaving the critical region, the thread makes all replicas execute the request which contains a snapshot operation and acquire the same snapshots (III.6-7). If the thread receives responses from all replicas (III.8), the thread sends back a response to the client (III.9). Again, the thread enters the critical region and decrements the `snapshot_counter` (III.10-11). If the counter equals zero, there are not any threads executing snapshot operations (III.12). Therefore, the thread wakes up all the threads which are about to execute commit operations (III.13). After the thread leaves the critical region, the state of the thread is changed to SNAPSHOT_PREPARED (III.14-15).

In the SNAPSHOT_PREPARED state, the type of a request is examined. Receiving a write operation, the thread controls the operation transmission with our tuple level concurrency control as described in Sect.4.3 (III.17-22). That is, the thread sends the request only to the leader, receives a response and then sends the request to all the followers. Receiving a read operation, the thread sends it to R^m, receives a response from the replica and sends back the response to the client (III.23-26). Receiving an abort operation, the thread sends it to all replicas, receives responses from them and sends back one to the client (III.27-30). If the thread receives a commit operation, it enters the critical region and increases the `commit_counter` only if any snapshot operations are not executed (III.31-35). The counter informs the other threads that there is a thread executing a commit operation and is used to guarantee that both snapshot and commit operations are not executed simultaneously. Leaving the critical region, the thread makes all replicas execute the request (III.36-37). If the thread receives responses from all replicas (III.38), the thread sends back a response to the client (III.39). Again, the thread enters the critical region and decrements the `commit_counter` (III.40-41). If the counter equals zero, there are not any threads executing commit operations (III.42). Therefore, the thread wakes up all the threads which do not execute snapshot operations (III.43). After the thread leaves the critical region, the state of the thread is changed to NO_SNAPSHOT again (III.44-45).

Unlike lazy replication [20, 17, 16, 3, 18, 19, 10], not only read operations of read-only transactions but also those of update transactions can be executed on any replica because Pangea can make any replica create the latest snapshot whereas only the master holds the snapshot in lazy replication. This prevents the leader from becoming a bottleneck.

There is a useful algorithm that we do not write in Algorithm III to make it easier to understand. If Pangea receives a nack from the leader (III.19), Pangea immediately sends back a nack to the client without sending the request (III.20) because the follower would send a nack. This enables Pangea not only to eliminate unnecessary execution but also to use database servers such as SQL Server which guarantees SI without the first updater wins rule, i.e., which returns a nack immediately without blocking if a write operation encounters the write-write conflict.

4.6 Correctness

THEOREM 2. *(guaranteeing GSI) Pangea algorithm guarantees GSI.*

PROOF. Read-only transactions are executed on one follower replica R^m (Algorithm II). Write operations of update transactions are executed on all replicas (III.17-21). Furthermore, snapshot operations of the update transactions

are executed on all replicas (III.7). Moreover, commit operations of the update transactions are executed on all replicas (III.37). Read operations of the update transactions are executed on one follower replica R^m (III.24). Therefore, R^m executes $T_i^m = T_i$ and the other replicas execute a transaction which excludes read operations from T_i. This means Pangea has the mapping function.

If snapshot operations are executed, a commit operation can not be submitted (III.33-34) until the snapshot operations have been executed (III.12-13). This means $(s_j < c_i) \in S \Rightarrow (s_j^m < c_i^m) \in S^m$. Else if commit operations are executed, a snapshot operation can not be submitted (III.3-4) until the commit operations have been executed (III.42-43). This means $(c_i < s_j) \in S \Rightarrow (c_i^m < s_j^m) \in S^m$.

Let $R^l, R^f, \{l, f\} \in m$ be the leader and the follower, respectively. Pangea delegates R^l to decide the execution order of w_i and w_j. That is, if Pangea sends w_i^l and w_j^l to R^l (III.18) and receives a response corresponding to w_i^l (III.19), Pangea knows that R^l has decided that the order of them is $w_i^l < w_j^l$. Therefore, Pangea adopts the order as a global schedule, $w_i < w_j$. Next, Pangea sends only w_i^f to R^f (III.20) and then receives a response from R^f (III.21). After that, Pangea sends c_i^l and c_i^f to R^l and R^f (III.37), respectively. When R^l has executed c_i^l, R^l executes w_j^l and Pangea receives a response corresponding to w_j^l (III.19). Then Pangea sends w_j^f to R^f (III.20) and receives a response from R^f (III.21). Therefore, $w_i^f < c_i^f < w_j^f$.
$w_i < w_j \Rightarrow w_i^m < w_j^m$. □

4.7 Hidden Deadlock Avoidance

Because we assume the first updater wins rule, if the execution order of conflicting write operations on one replica differs from that on another replica, hidden deadlock occurs, which lies across replicas. To avoid the deadlock, the execution orders on different replicas must be the same. As the write operation control, as described in Sect. 4.3, makes all replicas execute conflicting write operations in the same order, Pangea does not suffer from the deadlock.

However, Pangea probably has another hidden deadlock. If a snapshot operation is a write operation, Pangea may cause another hidden deadlock. Consider a write operation w_i is a snapshot operation of a transaction T_i and waits for a lock which has been given by w_j of a transaction T_j. With the first updater wins rule, w_i can not get the lock until T_j commits or aborts. However, T_j can not commit until the snapshot operation w_i has ended.

In practice, this may not be a serious problem because write operations are usually preceded by read operations. To avoid the problem, if Pangea receives a write operation as a snapshot operation, after submitting a dummy read operation as a snapshot operation to create a snapshot to all replicas and receiving responses, Pangea submits the write operation.

4.8 High Availability

We assume that the fault model is fail-stop and multiple failures do not occur. If the leader falls into malfunction, Pangea re-selects a new leader from followers and then clients must re-submit the transactions that have not been finished successfully. If a follower stops due to a fault, Pangea only detaches the follower from the system, i.e., Pangea does not send any requests to the follower and makes progress with the remaining followers. To circumvent Pangea becoming a single point of failure, we must prepare a standby Pangea node in addition to an active Pangea node.

If you assume that the fault model is not fail-stop, e.g., Byzantine fault, or need higher availability, you have to combine Pangea with a reliable protocol, e.g., 2PC. However, note that CAP theorem [7] states it is impossible to achieve all performance, consistency and availability at the same time. Although you can combine them easily, you have to compromise performance or consistency.

4.9 Elimination of Non-Determinism

In eager replication, we must prevent any replica from executing non-deterministic operations. Pangea parses an SQL statement and finds out non-deterministic functions such as `random` and `current_timestamp`, executes them and re-writes them to static values.

If a client submits the insertion of autogeneration value which does not cause the write-write conflict, Pangea must cause the conflict with a SELECT FOR UPDATE query because the order of the autogeneration without the write-write conflict is non-deterministic and therefore it may produce a different value on different replicas. Consider, for example in PostgreSQL, a table "tab(id SERIAL, name TEXT)" and an "INSERT(nextval('tab_id_seq'), 'TOM')" query. Receiving the request including the insertion, Pangea sends a "SELECT nextval('tab_id_seq') FROM tab_id_seq FOR UPDATE" query only to the leader without sending the original request. Then Pangea receives the response of the selection request from the leader and picks up the value of tab_id_seq. For example, tab_id_seq equals 10. Pangea re-writes the original request to an "INSERT(10, 'TOM')" query and sends it to all replicas.

It is easy to implement the mechanism for Pangea because Pangea parses an SQL statement to distinguish a read-only operation from an update operation. Due to the same reason, this mechanism produces small overhead.

4.10 Implementation

In the current implementation, we adopt `libpq` and the type 4 JDBC protocols of PostgreSQL. Surprisingly, Pangea uses less than 2000 lines of C code. This is because the idea of Pangea is very simple.

5. PERFORMANCE EVALUATION

In order to clarify the effectiveness of Pangea, we implemented a prototype and conducted a performance evaluation with the TPC-W benchmark by comparing throughput, response time and CPU utilities of Pangea with those of one of lazy replication middlewares.

5.1 Reference Middleware

So far, several middlewares have been proposed but most of them were not compared with the others. However, in order to verify the validity of Pangea, it is inevitable to compare the performance of Pangea with that of existing middlewares. There is not any existing eager replication middlewares guaranteeing SI without modification of database servers. In lazy replication middlewares, there are two approaches in terms of creating writesets: one called the *trigger approach* uses triggers [17, 10, 19] and the other called the *statement-copy approach* copies SQL statements in a middleware layer [19].

Salas et al. [21] came to the conclusion that the trigger-based replication caused an unacceptable overhead. Furthermore, Plattner et al. [19] implemented both approaches and observed that triggers produced non-trivial overhead and the statement-copy approach, which they called the *generic approach*, outperformed the trigger approach. Therefore, we implemented a prototype called LRM (Lazy Replication Middleware) with the statement-copy approach (the generic approach). Consequently, it is sufficient that we compare the performance of Pangea with that of LRM.

LRM selects one replica as a master and the others as slaves in advance. Rather than Pangea which can send a read operation to followers (slaves), receiving a request of an update transaction from a client, LRM must send it to the master whether it is a read operation or not because only the master holds snapshots. At the same time, LRM holds a copy of the request except a read operation as a writeset. To keep consistency between the master and the slaves, the order of executing commit operations on the master must be the same as that of applying writesets to the slaves. Because typical database servers offer no mechanism to specify a commit order externally, LRM must submit each commit operation serially, waiting for each commit to complete. After the update transaction commits on the master, LRM sends the writeset of the transaction to all the slaves.

Receiving a request of a read-only transaction from a client, LRM sends it to one of the slaves. The slaves must execute not only read-only transactions but also writeset transactions[3] with guaranteeing SI. Unfortunately, there are also no mechanisms to execute them concurrently with changing databases of the slaves to the same state of the master's database. Therefore, the slaves must execute writeset transactions in the commit order serially.

Moreover, in the worst case, read-only transactions may be delayed until suitable writesets have been applied on the slaves not to read stale data.

5.2 Analysis of Overhead

Before we show the measured results, we analyze which action of each middleware possibly causes overhead.

In Pangea, the first overhead may be caused due to the fact that both commit and snapshot operations of update transactions can not be executed simultaneously. Because read-only transactions do not produce this overhead at all, the higher the ratio of update transactions become, the bigger overhead Pangea suffers from. Even if there is only one replica, i.e., the leader also does the follower's work, Pangea yields this overhead. The second overhead is the round trip time of write operations, which may be bigger because Pangea delegates the leader to resolve the write-write conflict. The higher the ratio of update operations become, the bigger overhead Pangea suffers from. However, if there is only one replica, Pangea does not suffer from the overhead because Pangea does not have to send write operations to any followers.

In LRM, the first overhead may be caused by the fact that all commit operations of update transactions must be executed serially. Because read-only transactions do not produce this overhead at all, the higher the ratio of update transactions become, the bigger overhead LRM suffers from. Even if there is only one replica, i.e., the master also does slave's work, LRM yields this overhead. The second overhead may be caused due to the fact that writeset transactions must be executed in the serial fashion and no other transaction, even if it is a read-only transaction, can be executed during this time on the slaves. Because read-only transactions do not produce this overhead at all, the higher the ratio of update transactions become, the bigger overhead LRM suffers from. However, if there is only one replica, LRM does not suffer from the overhead because LRM does not have to apply writeset transactions to the slaves. The third overhead may be caused due to the fact that read-only transactions may wait to be executed until needed writesets have been applied on the slaves. Because read-only transactions do not produce this overhead at all, the higher the ratio of update transactions become, the bigger overhead LRM suffers from. Like the second overhead, if there is only one replica, LRM does not suffer from the overhead due to the same reason.

[3]A writeset transaction is the transaction that excludes read operations from an update transaction executed by the master.

5.3 TPC-W Benchmark

The TPC-W benchmark [2] models customers that access an online book store. The database manages eight tables: item, country, author, customer, orders, order_line, cc_xacts and address. There are 14 interactions, six are read-only and eight have update transactions. The benchmark specifies three different workloads: browsing, shopping and ordering mixes. A workload simply specifies the relative frequency of the different interactions. As table 1 shows, all three workloads consist of the same basic read-only or update interactions.

Table 1: three workloads in TPC-W benchmark

	read-only	update
browsing mix	95%	5%
shopping mix	80%	20%
ordering mix	50%	50%

From a database point of view, the browsing workload is read-dominated while the ordering workload is update-intensive. The shopping workload is between the two. The throughput metric used is web interactions per second (WIPS).

5.4 Experimental Setup

We use up to 7 replicas, one is the leader (master) and the others are followers (slaves) for database node. In addition, there are single middleware node, and two application server and client simulator nodes. All nodes have the same specification (two 2.4 GHz Xeon, 2 GBytes RAM and one 70 GB SCSI HDD) and connected by 1Gb/s Ethernet. We used PostgreSQL, version 8.3.7, as database replicas providing SI. We set the `default_transaction_isolation` to SERIALIZABLE[4], and did not change the other parameters. We used Apache Tomcat, version 6.0.18, as web and application servers. All nodes run the 2.6.18 Linux kernel. To measure the performance of the prototype, we used a TPC-W benchmark implementation [1] that had to be modified to support a PostgreSQL database. The TPC-W scaling parameters chosen were 1 million items and 2.88 million customers. This results in a database of around 6 GBytes.

[4]SI is called SERIALIZABLE in PostgreSQL.

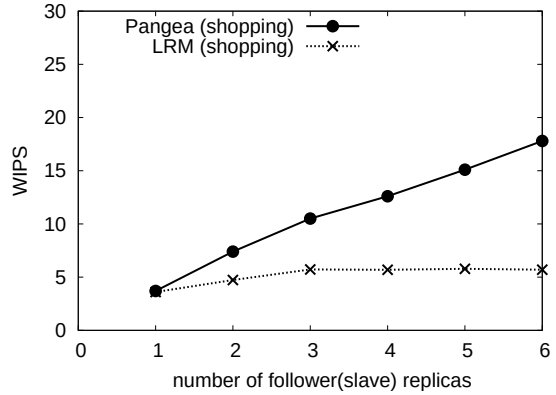

Figure 5: shopping mix

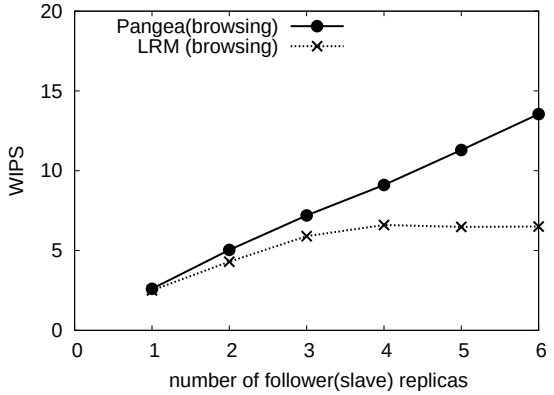

Figure 6: browsing mix

5.5 Results and Discussion

5.5.1 Overhead

In the first experiment, we wanted to analyze the overheads of Pangea and LRM with actual measurements.

We measured maximum throughputs of a single PostgreSQL instance with no middleware called PGSQL, Pangea with one replica and LRM with one replica. Table 2 shows measured maximum throughputs.

Table 2: maximum throughput with one replica

	PGSQL	Pangea	LRM
browsing (WIPS)	2.65	2.63	2.46
shopping (WIPS)	3.58	3.58	3.32
ordering (WIPS)	8.43	8.39	7.55

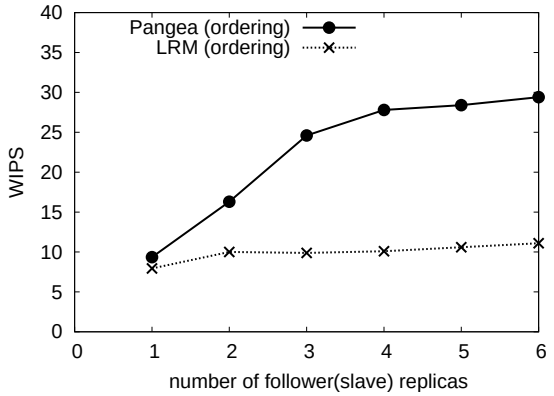

Figure 7: ordering mix

With only one replica, the middleware simply forwards requests to the single replica, Pangea may suffer from only the first overhead of Pangea, and LRM may also suffer from only the first overhead of LRM as described in Sect. 5.2. The ratios of throughput reduction in browsing, shopping, ordering mixes are 0.75%, 0% and 0.47% in Pangea, respectively, and 7.20%,7.26% and 10.4% in LRM, respectively. The overheads of LRM are bigger than those of Pangea. This is because commit operations of update transactions can be executed concurrently in Pangea whereas those must be executed serially in LRM. Unlike LRM, Pangea has the disadvantage that both commit and snapshot transactions of update transactions can not be executed concurrently. However, this experiment shows that the overhead is smaller than that of the serial execution of LRM. Surprisingly, even if the workload is read-intensive, namely the browsing mix, the overhead of LRM is not negligible.

5.5.2 Scalability

The second part of the evaluation analyzes scalability. We measured the throughputs of Pangea and LRM in different configurations, from 1 up to 6 followers (slaves). Unlike in the first experiment, the measured throughputs include the influence of all the overheads.

Figure 5 shows the maximum throughputs for the shopping mix, which is the most representative workload for the TPC-W benchmark. The throughput of LRM increases with the number of replicas moderately and reaches a peak of 5.72 WIPS at 3 replicas and then flattens. On the other hand, the throughput of Pangea increases linearly and reaches 17.8 WIPS at 6 replicas, which means Pangea outperformed LRM by approximately 210% in throughput. We can expect that by adding more follower replicas we can increase the achievable throughput since even read operations of update transactions can be executed on any follower.

Figure 6 shows the maximum throughputs for the browsing mix. The curves are lower than those in Fig. 5, since this workload contains more read operations, which are generally more complex than update operations. The fact helps LRM distribute the read load and thereby the throughput of LRM increases at 4 replicas. Although the difference between throughput of Pangea and that of LRM becomes smaller, the maximum throughputs of Pangea and LRM are 13.6 WIPS and 6.48 WIPS, respectively, which means Pangea outperformed LRM by approximately 110% with regard to throughput.

Figure 7 shows the maximum throughputs for the ordering mix. Because simple update transactions account for 50% of the load, Pangea and LRM achieve higher throughputs. However, the throughput of LRM increases slightly even if LRM has 6 replicas. This is because there are few opportunities to distribute read operations and the overhead of the serial commit execution and the serial writeset application is very big. Similarly, Pangea saturates at 4 replicas. How-

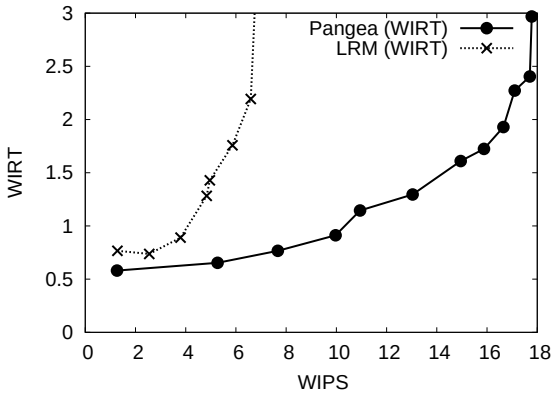

Figure 8: WIRT (Web Interaction Response Time)

ever, the throughput of Pangea is much higher than that of LRM, i.e., the maximum throughputs of Pangea and LRM are 29.4 WIPS and 11.1 WIPS, respectively. This means Pangea outperformed LRM by approximately 165% in terms of throughput.

5.5.3 *Response Time*

The second overhead of Pangea increases the round trip time of write operations and thereby the response time of Pangea is possibly bigger than that of LRM. To answer this question, we measured WIRTs (Web Interaction Response Time) of Pangea and LRM.

Figure 8 shows WIRTs for the shopping mix. Because the tendencies of the browsing and the ordering mixes are the same as the shopping mix with 6 follower replicas, we show only the graph of the shopping mix. WIRT of LRM is bigger than that of Pangea. This means that the overhead of sequential transmission of write operations is smaller than the overhead of LRM.

5.5.4 *Bottleneck Avoidance*

Pangea does not allow more than one leader replica. So, the leader is possibly a bottleneck. To answer the question, we examined usage of CPU time of the experiments with the `vmstat` utility. Table 3 shows usage of CPU time for the ordering workload with 6 follower replicas and 240 EBs (Emulated Browsers), which is a heavy workload in our environment.

Table 3: usage of CPU time

	leader	follower	Pangea	tomcat
utilities	0%	50 ∼ 70%	0%	3%

In this setting, the usages of CPU time of the leader and Pangea are less than 1%. This means that neither the leader node nor the Pangea node is a bottleneck. Even if we added follower replicas to the setting, neither the leader node nor the Pangea node would be a bottleneck because the usages of CPU time of the leader and Pangea are much smaller than those of followers.

In Pangea, the possibility that the leader becomes a bottleneck is smaller than existing middlewares. Because every replica has the same snapshot, even read operations of update transactions can be executed on any follower. Also, the possibility that Pangea becomes a bottleneck is smaller because the algorithm is very simple and therefore the code size is very small.

6. RELATED WORK

6.1 Lazy Replication Middleware

Replication middlewares guaranteeing SI without modification of database servers can be categorized into two approaches, namely the trigger approach [17, 10, 19] and the statement-copy approach [19] as noted in Sect. 5.1. In addition to the two approaches, there is another approach, the log-shipping approach [16]. However, no existing middleware with the approach guarantees SI.

Plattner et al. [17, 19] introduce Ganymed, the original middleware of LRM against which we compared Pangea in Sect. 5. In order to keep consistency without modification of database servers, this approach must submit commit operations serially on the master. Furthermore, writeset transactions must be executed in a serial fashion. Moreover, read-only transactions may be delayed not to read stale data until needed writesets have been applied on slaves. We showed these overheads decreased throughput and disabled scale-out. Although they also implemented prototypes with the trigger approach and evaluated it, they presented that the overhead of the trigger approach was bigger than that of the statement-copy approach.

Elnikety et al. [10] introduce *Tashkent-MW*, a middleware which collects writesets by triggers. Their strong point is to reduce the overhead of write action by integrating several write accesses into a single disk write. However, there still exists the problems of the statement-copy approach. Moreover, if write operations are not executed frequently, there are few chances of the integration. Generally read operations are often executed and some of them are complex. Few applications may benefit from Tashkent-MW.

6.2 Eager Replication Middlewares

Unfortunately, no existing eager replication middleware guarantees SI without modification of database servers.

All middlewares by Cecchet et al. [9], Fujiyama et al. [12] and Amza et al. [4] guarantee one-copy-serializability[5] [6]. Unfortunately, the middlewares by Cecchet et al. [9] and Fujiyama et al. [12] suffer from hidden deadlock. The middleware by Amza et al. [4] avoids the deadlock. However, unlike Pangea, the middleware can not achieve the tuple level concurrency control but the table level.

6.3 Guaranteeing SI in Federated Database Systems

Schenkel et al. [22] propose two approaches, a pessimistic approach and an optimistic approach, which guarantee SI in a federated database system. These approaches can also be used in replicated database systems.

In the pessimistic approach, all replicas are forced to start a transaction at the same time with the assumption of using an atomic commit protocol. The assumption implies that the end of the transaction of all replicas is also the same. Unlike Pangea, they assume the first committer wins rule.

[5]The resulting schedules are equivalent to a serial schedule on a single database.

The synchronized start of a transaction guarantees that all replicas get the same snapshot. However, the synchronization is too strong. In fact, it is sufficient that the relative order of snapshot and commit operations in all replicas are the same. Pangea adopts the weaker synchronization and therefore the overhead is smaller. Unlike Pangea, the pessimistic approach can not control the execution of conflicting write operations. Ensuring consistency between replicas relies on the atomic commit protocol. When the execution order in one replica differs from that in another, which produces inconsistency between replicas, one executes write operations successfully and another fails or vice versa. Hence, all replicas abort the transaction because all replicas can not commit successfully. Unfortunately, the unnecessary abort causes deterioration of throughput.

The optimistic approach checks whether T_i and T_j are executed serially in one replica and T_i and T_j are executed concurrently in another replica. If this situation is found before T_i is committed, T_i is aborted. Unfortunately, this approach also causes unnecessary aborts. Consequently, Pangea is superior to both approaches because Pangea can avoid re-execution of unnecessary aborted transactions.

6.4 Guaranteeing SI in Replicated Database Systems

Lin et al. [14] propose *1-Copy-SI*, a new criterion for replicated database systems guaranteeing SI. Unlike Pangea, 1-Copy-SI is based on the first committer wins rule, which is not used by practical database servers such as Oracle, SQL Server and PostgreSQL.

They also propose *SI-Rep*, a middleware which guarantees 1-Copy-SI. However, SI-Rep needs group communication and writeset handling, which needs modification of existing database servers.

7. CONCLUSION

In this paper, we defined a new correctness criterion for eager replicated database systems called global snapshot isolation (GSI). We proposed a new concurrency control which exploits the first updater wins rule effectively and avoids a hidden deadlock. To the best of our knowledge, this paper is the first to show the effective exploitation of the first updater wins rule. We proposed Pangea, a novel eager database replication middleware guaranteeing GSI using the control.

The experimental results with the TPC-W benchmark showed that the overhead of Pangea was very small. Moreover, Pangea outperformed one of existing middlewares for all workloads because Pangea offers not only higher concurrency but also smaller overhead. Furthermore, Pangea achieved linear scale-out for read-intensive workload.

Note that Pangea requires no changes to database servers but is implemented only in a middleware layer. Furthermore, the code size of Pangea is very small. Consequently, it is concluded that Pangea is very practical and effective.

8. REFERENCES

[1] Java TPC-W implementation distribution, http://www.ece.wisc.edu/ pharm/tpcw.shtml.

[2] Transaction processing performance council, tpc-w.

[3] F. Akal, C. Türker, H.-J. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *VLDB*, pages 565–576, 2005.

[4] C. Amza, A. L.Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, pages 282–304, 2003.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD*, pages 1–10, 1995.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, Massachusetts, 1987.

[7] E. Brewer. Towards robust distributed systems (invited talk). In *PODC*, 2000.

[8] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *ACM SIGMOD*, 2008.

[9] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX*, 2004.

[10] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, 2006.

[11] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005.

[12] K. Fujiyama, N. Nakamura, and R. Hiraike. Database transaction management for high-availability cluster system. In *PRDC*, 2006.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD*, pages 173–182, 1996.

[14] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *ACM SIGMOD*, 2005.

[15] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, November 2005.

[16] J. G. Per-Ake Laarson and J. Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *ICDE*, 2004.

[17] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[18] C. Plattner, G. Alonso, and Özsu. Dbfarm: A scalable cluster for multiple databases. In *Middleware*, 2006.

[19] C. Plattner, G. Alonso, and Özsu. Extending dbmss with satellite databases. *VLDB Journal*, 2006.

[20] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Fas – a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB*, 2002.

[21] J. Salas, R. Jiménez-Peris, M. Patiño-Martínez, and B.Kemme. Lightweight reflection for middleware-based database replication. In *SRDS*, 2006.

[22] R. Schenkel, G. Weikum, N. Weibenberg, and X. Wu. Federeated transaction management with snapshot isolation. *Lecture Notes in Computer Science*, January 2000.