ProRea — Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation

Oliver Schiller Nazario Cipriani Bernhard Mitschang

Applications of Parallel and Distributed Systems, IPVS,
Universität Stuttgart, Universitätsstr. 38, 70569 Stuttgart

{schiller, cipriani, mitschang}@ipvs.uni-stuttgart.de

ABSTRACT

The consolidation of multiple tenants onto a single RDBMS instance turned out to be benefical with respect to resource utilization and scalability. The consolidation implies that multiple tenants share the physical resources available for the RDBMS instance. If the available resources tend to get insufficient to meet the SLAs agreed with the tenants, migration of a tenant's database from one RDBMS instance to another is compelling. Highly available services demand for live migration techniques that come with minimal service interruption and low performance impact.

This paper meets the demand for live migration techniques by contributing *ProRea*. ProRea is a live database migration approach designed for multi-tenant RDBMS that run OLTP workloads under snapshot isolation. ProRea extends concepts of existing live database migration approaches to accomplish minimal service interruption, high efficiency and very low migration overhead. Measurements of a prototypical ProRea implementation underpin its good performance.

Categories and Subject Descriptors

H.2.4 [**Database Management Systems**]: [Relational databases]; H.3.4 [**Information Storage and Retrieval**]: [Distributed systems]

General Terms

Design, Management, Performance

Keywords

Database Live Migration, Snapshot Isolation, Multi-tenancy, Cloud Computing, RDBMS

1. INTRODUCTION

Nowadays, relational database management system (RDBMS) instances are available for everyone through pushing a button on a web-based management interface, e. g. Amazon RDS [2]. Some seconds after pushing the button, a new instance runs somewhere in the Cloud without any efforts, except for the money that the service provider debits from the credit card. The amount of money paid depends on the individual demands but also on the fixed overheads which each tenant induces for the service provider. Naturally, lower fixed overheads per tenant enlarge the profit margin of the service provider. However, lower fixed overheads also allow lowering service fees which eventually may attract more tenants. Therefore, minimization of fixed overheads per tenant represents a central challenge of Cloud Computing.

A central opportunity to reduce fixed overheads and essential characteristic of Cloud Computing [15] constitutes *multi-tenancy*. Multi-tenancy describes consolidating multiple tenants onto a single physical resource, which lowers unused capacity and decreases the numbers of resources to manage and maintain. Despite of the consolidation onto a single physical resource, each tenant seems to have individual resources. For this purpose, tenants obtain virtual resources which are mapped to the underlying physical realities.

To stay efficient and to keep in view the demands of tenants, the service provider has to adapt the mapping of virtual resources to physical resources regularly. In fact, an autonomic manager, as drafted in [9], decides about the mapping to guarantee good resource utilization and to meet SLAs upon which the service provider and the tenants agreed, e. g. transaction throughput or transaction response times. At this, a primitive operation for the autonomic manager represents the migration of a tenant (or its virtual resource) from one physical resource to another physical resource. As services nowadays run 24/7, the migration must not interrupt normal processing, which means the migration has to run *live*. The high proliferation of relational database technology to manage data thus calls for *live migration of relational databases*.

In principle, live migration of virtual machines, which is common nowadays, enables live migration of relational databases as well; a virtual machine may simply wrap a corresponding RDBMS instance per tenant. However, virtual machines come with huge footprints which entail large fixed overheads per tenant. This is particularly unfavorable for smaller tenants. A promising model to overcome this drawback represents *Shared Process*. In Shared Process, each tenant has a dedicated database, but they share a single RDBMS instance and, thus, share resources of the RDBMS instance, e. g. logging. To support such models, new live migration approaches for relational databases emerged recently [26, 32, 8, 10]. A central difference of these approaches represents the type of database transfer: *proactive* or *reactive* [1].

Proactive migration approaches [26, 32, 8] push the whole database to the destination of migration, before it starts new transactions at it. These approaches propagate modifications of the database that happen during migration to the destination. This yields redundant work at the source and the destination and causes blocking updates while applying the modifications that occurred during migration. As

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

opposed to proactive migration approaches, reactive approaches [10] directly start new transactions at the destination and pull required database parts on demand. This leads to many page faults which entail pulling the corresponding pages from the source.

To overcome the drawbacks of purely proactive and purely reactive approaches, we contribute *ProRea* which represents a new live database migration approach that combines *pro*active and *rea*ctive measures. ProRea starts with proactively pushing hot pages, i. e. pages that have been recently accessed and, thus, are in the buffer pool of the RDBMS instance. Thereafter, newly arriving transactions start at the destination and pull pages on demand. ProRea reduces page faults and improves buffer pool handling significantly compared to purely reactive approaches. For this purpose, it requires little redundant work, but considerably less than a purely proactive approach. Our detailed contributions are as follows:

- Section 2 embarks upon reviewing common multi-tenancy models: Shared Machine, Shared Process and Shared Table. In this context, it motivates the applicability of Shared Process and reviews database migration approaches which are suitable for Shared Process.
- Section 3 sums up the environment and system conditions for which ProRea is designed. Thereafter, the basic concept of ProRea follows. Referring to the basic concept of ProRea, a semi-formal proof shows that ProRea ensures snapshot isolation during migration.
- Section 4 analyzes costs and trade-offs of ProRea. Moreover, it provides an optimized buffer space handling strategy which supports releasing load from the source of migration.
- Section 5 presents measurements of a prototypical implementation of ProRea that underpin the good performance of ProRea compared to a purely reactive approach.
- Section 6 outlines how to deal with system failures during database migration assuming a log-based recovery approach.
- Finally, the paper finishes with our conclusions and directions for future work.

2. STATE OF THE ART

Virtualization constitutes the fundamental technical concept that drives Cloud Computing. Therefore, a key decision when creating cloud services is concerned with placing the cut between physical and virtual resources. At this, one size does not fit all — where to place the cut between physical to virtual resources depends on many issues: desired abstraction, customization and isolation requirements of tenants, workloads of tenants, size of tenants, applications and so forth. Therefore, different models to enable multi-tenancy with RDBMSs emerged. Subsequently, we give a brief overview of these models, clarifying the position of Shared Process. Thereafter, we review related database migration techniques.

2.1 Multi-tenancy Models: Shared Process

Many different models and implementations have been proposed to enable multi-tenancy with RDBMSs [6, 11, 13, 22]. Jacobs et al. [13] proposed three model categories having different degrees of abstraction and consolidation. For each category industrial implementations or at least academical evaluations exist (we only name some examples):

1. **Shared Machine:** The tenants share a single machine, but each tenant has a private RDBMS instance (Soror et al [28]).
2. **Shared Process:** The tenants share a single RDBMS instance, but each tenant obtains a private database or at least a private set of tables (SQLAzure [16], RelationalCloud [7]).
3. **Shared Table:** The tenants share a single RDBMS instance, a single database instance and a single set of tables (Salesforce.com [23, 33], SuccessFactors [30]).

These categories represent a quite rough classification, but it suffices for the following discussion. For a sub-division of these categories, we refer to [22].

Shared Machine allows consolidating only few tenants onto one machine due to the large main memory footprint of RDBMS instances [13]. Obviously, this gets worse if a virtual machine wraps the RDBMS instance [7]. Therefore, this approach comes with low consolidation efficiency for smaller tenants. Note that we refer to tenants which require little resource capacity as small tenants.

Compared to Shared Machine, Shared Process and Shared Table decrease the overheads per tenant. Shared Table offers the highest degree of sharing between tenants and thus the lowest overheads per tenant. The high degree of sharing in Shared table is mainly useful if many tenants require similar data structures, show similar data access patterns and omit to have high customization or isolation needs. This is often the case in software as a service (SaaS) scenarios for very small tenants. Hence, Shared Table is attractive in this case due to its good performance and very low overheads per tenant [3, 25, 24].

Database as a service (DaaS) scenarios are different. In DaaS, tenants provide different workloads and do not share any data or schemas. In this case, Shared Process is promising due to its good trade-off between isolation, customization and scalability [9]. It is particularly interesting to consolidate many smaller tenants with maximum database sizes of few GBs.

Note that the described models may also occur nested. A software provider may use DaaS offerings, which adopt Shared Process, to provide its SaaS offering by means of Shared Table. Moreover, techniques developed for Shared Process, e. g. migration techniques, may be beneficial for a SaaS provider as well if the SaaS provider manages and maintains multiple Shared Table databases using Shared Process. To conclude, Shared Process and related techniques target a broad range of use cases.

2.2 Database Migration for Shared Process Multi-tenancy

Database migration represents a primitive operation in order to keep in view the changed demands of tenants and to run the service efficiently, e. g. to avoid overload situations and to reduce energy consumption. Subsequently, we review existing techniques that allow database migration within a Shared Process multi-tenancy model. For this purpose, we start with *Stop and Copy* as a straightforward approach and going on with live database migration approaches.

2.2.1 Stop and Copy

Stop and Copy shuts down the database, flushes it to disk, copies the database from the source to the destination, and, finally, restarts it at the destination. Obviously, this approach is straightforward and very efficient but also has some intrinsic drawbacks.

First of all, the database is offline during migration. For example, to copy a database having 2.5 GB over a 1 Gbit/s ethernet requires at least 20 seconds, assumed full network transfer speed. Yet, co-located tenants on the same machine or other machines may observe severe degradation when copying full speed ahead. Limiting transfer speed actually solves this issue, but lengthens service interruption.

In addition to that, the database runs with cold buffer pool after migration which naturally impacts performance. For the points mentioned, other approaches, live migration approaches, that minimize service interruption and performance impact are desired.

2.2.2 Live Migration

Several researchers and developers showed interest in live migration of virtual machines or live migration of processes. Compared to these types of live migration, live database migration for Shared Process multi-tenancy models requires another granularity; it requires transferring databases instead of whole virtual machines or processes. As a result, the algorithms for database live migration are tailored to the concepts relevant in RDBMS and keep in view related dependencies, e. g. transactions, concurrency control and recovery.

Several works into live migration of relational databases exist. A common practice is what we refer to as *fuzzy migration*, as it resembles fuzzy backup and fuzzy reorganization [27]. Fuzzy migration creates and transfers a fuzzy dump of the database from the source to the destination. Thereafter, it applies recorded modifications of the database which occurred while dumping and transferring on the database image at the destination. The previous step may run repeatedly in hope that the volume of modifications that newly comes up during each step decreases. Finally, it shuts down the database at the source, transfers the last set of recorded modifications, applies them on the database image at the destination, and restarts the database at the destination. Now, the database at the destination is able to process transactions. This approach is proactive as it copies the data to the destination before switching transaction processing.

Recent research applied this technique on different system architectures and evaluated different aspects [26, 32, 8]. This approach is quite simple and applicable by adopting standard database tools. However, it also has some considerable drawbacks: redundant work at the source and the destination, higher network traffic compared to stop and copy, blocked updates during replay of changes recorded in the last step, and transaction load stays at the source until whole data is transferred, which is undesired for out-scaling.

To overcome these drawbacks, Elmore et al. [10] proposed a reactive approach, called *Zephyr*. Reactive approaches switch transaction processing from the source to the destination before they copy the data. If a transaction accesses a page, which is not transferred already, it causes a page fault. That is, the page is pulled synchronously from the source. Reactive approaches keep redundant work at a minimum and come without a serious service interruption. Moreover, they are able to disburden the source quickly. However, the page faults increase average transaction latency.

ProRea is a hybrid approach which combines *pro*active and *reac*tive measures. By this, ProRea accomplishes minimal service interruption and minimizes the number of page faults. With respect to the handover of transaction load from the source to the destination, ProRea is similar to Zephyr because both allow running transactions concurrently at the source and the destination. Concurrently running transactions at the source and the destination require synchronization measures which are integrated into the migration protocol. Therefore, the used concurrency control mechanism considerably influences the design of the migration protocol. The used concurreny control mechanism differs in ProRea and Zephyr. ProRea bases upon multi-version concurrency control in order to provide snapshot isolation, whereas Zephyr bases upon strict 2-phase locking. Snapshot isolation only avoids most concurrency anomalies, but it provides higher concurrency than strict 2-phase locking. Thus, it surely is a popular concurrency control mechanism; major RDBMS implementations including Microsoft SQL Server, Oracle, MySQL, PostgreSQL, Firebird, H2, Interbase, Sybase IQ, and SQL Anywhere support snapshot isolation.

3. PROREA

Live database migration techniques, especially techniques which are tightly integrated into the RDBMS, depend on the system and environment conditions, e. g. adopted concurrency control, workload type and system architecture. For this reason, we first describe the conditions for which ProRea is designed. Subsequently, we explain the algorithmic building blocks of ProRea. Finally, a semi formal proof that the presented algorithmic building blocks fulfill snapshot isolation follows.

3.1 Basic System and Environment Conditions

ProRea targets a shared nothing system architecture of the RDBMS cluster, as depicted in Fig. 1. The RDBMS cluster comprises multiple machines, each running a RDBMS instance. A RDBMS instance serves several databases and each database relates to one tenant. Thus, tenants provide a natural partitioning which aligns database boundaries to server boundaries. This assumption implies that a tenant fits on a single machine with respect to its SLAs and requirements, which represents a common scenario [34]. If a tenant's data is horizontally or vertically partitioned and thus distributed over several nodes, one partition could be considered as one database from the perspective of migration. Moreover, we assume an OLTP workload that consists of mainly short-running transactions.

To establish transparency of the database location, we imagine that the tenant connects to a cluster management instance that knows the location of the tenant's database. As shown in Fig. 1, the client library queries the cluster management instance about the machine that serves the tenant's database. Thereafter, it connects to the returned machine and issues queries over this connection. Hence, the cluster management instance stays out of the line regarding query processing, which is necessary for high scalability.

For a short period of time, ProRea allows running transactions concurrently at the source and at the destination. As already mentioned, the measures required to ensure isolation during this period depend on the used concurrency control mechanism. ProRea assumes snapshot isolation [4], which enforces the following two rules [18]:

1. A transaction T reads the latest committed tuple version that exists at the time T started, i. e. it never sees changes of concurrently running transactions.
2. A transaction T aborts if a concurrent transaction of T has already committed a version of a tuple which T wants to write (*First Committer Wins*).

In practice, snapshot isolation is implemented by multi-version concurrency control (MVCC) [21]. Typical MVCC implementations

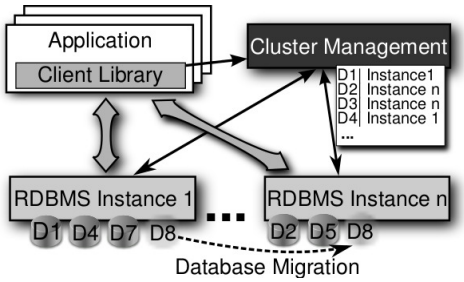


Figure 1: System Architecture

maintain several versions of one tuple in a chronologically arranged *version chain*. This version chain is traversable and may span multiple pages. A page provides information where to find the previous or later version of each tuple it contains. Thus, the access of a specific version of a tuple may require following its version chain, which may cause multiple page accesses.

For instance, PostgreSQL uses a non-overwriting storage manager and a forward chaining approach that allows traversing all tuple versions starting from the oldest version. For this purpose, a tuple stores a reference to its successor with respect to time. By contrast, Oracle's database software updates tuples in place and stores older versions in so-called undo segments. At this, it uses a backward chaining approach that allows traversing tuple versions starting from the newest version. The InnoDB storage manager of MySQL and MS SQL Server use similar approaches.

Instead of the First Committer Wins rule, typical implementations enforce a slightly different rule, referred to as *First Updater Wins* rule. This rule prevents a transaction from modifying a tuple if a concurrent transaction has already modified it, usually by means of locking, preferably tuple-level locking. If the first updater commits and releases locks, the waiter aborts. If the first updater aborts and releases locks, the waiter may modify the tuple. Hence, the First Updater Wins rule eventually leads to the same effect as the First Committer Wins rule.

To sum up, the boundary conditions on which the design of ProRea bases are:

- a RDBMS cluster with shared nothing system architecture and database location transparency,
- snapshot isolation based on MVCC and the First Updater Wins rule as concurrency control mechanism,
- mainly small tenants, such that one machine usually serves multiple tenants (> 10),
- OLTP workloads that mainly consist of short-running transactions,
- and reliable, order-preserving, message-based communication channels having exactly-once semantics.

3.2 Basic Concept

ProRea runs in five successive phases: 1. *Preparation*, 2. *Hot Page Push*, 3. *Parallel Page Access*, 4. *Cold Page Pull* and 5. *Cleanup*. Figure 2 outlines the interaction of the source and the destination in each phase. First, Preparation initializes migration at the source and the destination. Thereafter, Hot Page Push proactively pushes all hot pages, i. e. pages in the buffer pool, from the source to the destination. During Hot Page Push, each transaction that runs modification operations on a already transferred page sends suitable modification records to the destination. The destination applies the modification records to be consistent with the source. Next, the source hands over transaction processing to the destination. From now, the destination processes newly arriving transactions. During Parallel Page Access, transactions run concurrently at the source and at the destination. If a transaction at the destination requires a page which has not been transferred yet, it synchronously pulls the page from the source. After the last transaction at the source completes, Parallel Page Access transitions to Cold Page Push. Cold Page Push pushes the data which has not been transferred yet, the potentially cold data, to the destination. Finally, migration finishes with cleaning up used resources.

During migration, we use the concept of page ownership to synchronize page access between the source and the destination. If a node owns a page, it has the current version. Moreover, only the node that owns the page may modify it. This implies that if the ownership of a page is passed from one node to another node, the new version of the page or at least appropriate delta records have to be passed as well, such that the new owner of the page also has the newest version of it. Note that the ownership concept is a low-level concept that ensures consistent, synchronized page access across the source and the destination. That is, the ownership concept serializes modifications of a page in order to preserve its physical integrity. On the contrary, transaction-level synchronization is accomplished by snapshot isolation which builds upon multi-version concurrency control and tuple-level locks.

$\mathcal{D}_m$	The database to be migrated.
$\mathcal{T}_s/\mathcal{T}_d$	Transactions at the source/destination.
$\mathcal{B}_s/\mathcal{B}_d$	Buffer pool of the source/destination.
$o(P)$	Ownership of page P.
$S(T)$	Start timestamp of transaction T.
$C(T)$	Commit timestamp of transaction T.

Table 1: Notations.

Subsequently, we dive into the details of each phase using the notations listed in Table 1.

3.2.1 Preparation and Hot Page Push

To prepare migration, the source sets up local data structures and migration infrastructure, e. g. processes and communication channels. Thereafter, it sends an initial message to the destination. This message includes the meta data of the database, e. g. table definitions, index definitions, user and role definitions and so forth. From this information, the destination creates an empty database and sets up its local migration infrastruture. Finally, it acknowledges preparation which triggers transitioning to Hot Page Push.

Hot Page Push scans through all pages in $\mathcal{B}_s$. At this, it transfers each page $P \in \mathcal{D}_m \cap \mathcal{B}_s \wedge o(P) = source$, i. e. the hot pages which are not transferred, to the destination and changes $o(P)$ to *destination*, i. e. the destination obtains the ownership of P. The destination inserts the retrieved page in the buffer pool using standard buffer allocation methods.

If a transaction accesses an already transferred page P for modification, it requests *temporal ownership* for P. As the destination does not run transactions of $\mathcal{D}_m$ during Hot Page Push, the request simplifies to changing $o(P)$ to *temporal ownership* at the source. However, to maintain a consistent image of P at the destination, each modification operation of P creates a modification record which it sends to the destination. The modification record contains all information required to synchronize the image of P at the destination. In practice, page-level logical operations, i. e. physiological operations, lend themselves. If the RDBMS already creates such records for logging or replication, it simply has to capture records for P and ship them to the destination. The destination applies the retrieved record on P which ensures that the image of P at the destination is consistent to the image at the source.

The set of contained pages in $\mathcal{B}_s$ may change while the migration process scans $\mathcal{B}_s$ for hot pages; some pages in $\mathcal{B}_s$ give way to new pages. Therefore, $\mathcal{B}_d$ does not necessarily contain the actual set of hot pages of $\mathcal{D}_m$ after the described pass over $\mathcal{B}_s$. To obtain a higher similarity, further passes over $\mathcal{B}_s$ that transfer hot pages of $\mathcal{D}_m$ which are not transferred yet may be useful.

Our current implementation simply scans the buffer pool two times. We have not considered more sophisticated termination conditions so far. Nevertheless, we assume that the change of the

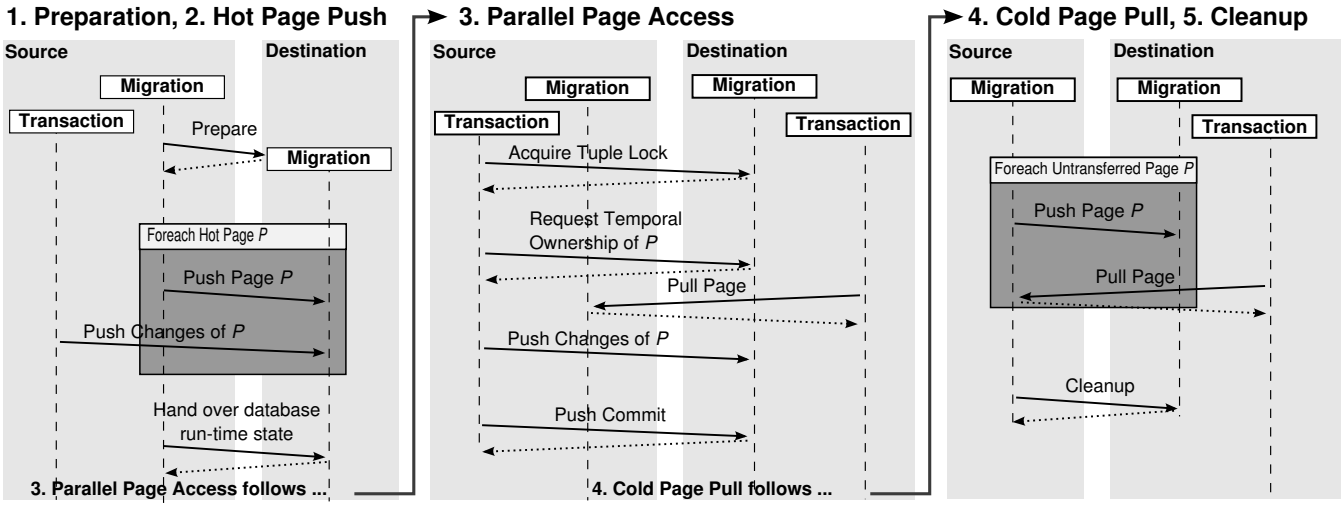


Figure 2: Sequential diagram for the algorithmic phases of ProRea.

difference of transferred pages in two consecutive passes represents a good start. If the change converges to zero, there is no noteworthy progress anymore with respect to obtaining a higher similarity.

After the last pass, Hot Page Push hands over transaction processing to the destination. To ensure valid transaction processing, the destination requires the same *database run-time state* as the source. The database run-time state includes the state information that is globally visible such as currently running transactions, the identifier of the next transaction, and the lock table. The source adds the database run-time state to a handover message. To obtain a consistent state at the source and the destination, the source freezes the database run-time state while preparing the handover message. This prevents starting new transactions, modifying $\mathcal{D}_m$ and completing running transactions. The destination delays starting new transactions until it has received the handover message and has taken over the contained database run-time state. Directly after handover, the source and the destination run transactions on $\mathcal{D}_m$ concurrently, with which Parallel Page Access deals.

Existing and newly arriving clients have to be informed about the new location of the database. For this purpose, the migration task updates the cluster management instance with the new location of $\mathcal{D}_m$. In addition to that, the source notifies clients which want to access $\mathcal{D}_m$ about its new location. Thus, the client library is able to connect to the destination directly, without querying the cluster management. This approach enables updating the cluster management asynchronously.

3.2.2 Parallel Page Access

As transactions on $\mathcal{D}_m$ run concurrently at the source and the destination, Parallel Page Access requires synchronization between both of them in order to comply to the concurrency protocol. Subsequently, we briefly describe the algorithm to achieve snapshot isolation, but defer discussing its correctness to a separate section (see Sec. 3.3).

To complete successfully, transactions which run at the source may require accessing pages that have been already transferred to the destination. The destination could have modified the page already, by what the source does not have the current image of the page. To take into account this situation, the source accesses a tuple using the steps which algorithm 1 shows in pseudo code notation. The listing just shows the case of reading and updating a tuple.

Algorithm 1 Access of a tuple in page P at the source.

```
1: if access = read then
2:     read(P, tuple)
3: else
4:     if acquireTupleLock(tuple) = failed then     ▷ Request tuple lock(includes destination).
5:         abort
6:     end if
7:     if o(P) = source then
8:         (P, Rec_mod) ← update(P, tuple)
9:     else                                          ▷ Destination already owns page P.
10:        res ← requestTemporalOwnership(P)
11:        if includesPage(res) then  ▷ Response may include current version of P.
12:            P ← getPage(res)
13:        end if
14:        (P, Rec_mod, status) ← update(P, tuple)
15:        returnOwnership(P, Rec_mod)
16:        if status = failed then
17:            abort
18:        end if
19:    end if
20: end if
```

The source primarily acquires an exclusive lock for the tuple which it wants to modify. If the source obtains the tuple lock locally, it also acquires the lock at the destination. Thus, the destination has the global view with respect to acquired locks. Acquiring the lock may fail if the destination already locked the tuple under concern. Note that it is essential to acquire the lock at the destination even if the source still owns the ownership. This is because the ownership of the page may change although the source holds the tuple lock. In this case, the destination has to know about the tuple lock to discover potential update conflicts of concurrently running transactions.

If the source has already transferred the page, it requests *temporal ownership* of the page from the destination. The destination tracks temporal ownership of the source and sends an appropriate response. The response includes the current image of the page if a transaction at the destination already modified the page. After updating the page, the source returns ownership to the destination and piggybacks the corresponding modification record. The source proactively returns ownership since the probability that several transactions of the source access the same page is low, in the case of

short-running transactions (more details follow in Sec. 4.1).

Note that the methods which actually read or update the tuple (line 2, 8 and 14) may require following the version chain of the tuple. This yields accesses of tuple version in other pages, which run similar to the previous description.

Furthermore, if a transaction completes at the source, it sends a message that includes its completion status to the destination. This ensures that the destination knows about completion of all transactions, which it requires to release locks related to the transaction and ensure valid snapshots for new transactions.

Tuple access at the destination is simpler (see algorithm 2). Transactions at the destination primarily check the ownership of a page when accessing it. If it already owns the page, no additional efforts are necessary. If the source still owns the page, the destination pulls the page from the source. If the source temporarily obtained ownership of the page, the destination waits until the source returns the ownership.

Algorithm 2 Access of a tuple in Page P at the destination.

```
1: if o(P) = source then
2:     pullFromSource(P)                    ▷ Page not transferred yet.
3: end if
4: if o(P) = temporalsource then
5:     waitForOwnership(P)                  ▷ Wait till source returns ownership.
6: end if
7: if access = read then
8:     read(P, tuple)
9: else
10:    acquireTupleLock(tuple)
11:    (P, Rec_mod, status) ← update(P, tuple)
12:    if status = failed then
13:        abort
14:    end if
15: end if
```

3.2.3 Cold Page Pull and Cleanup

Cold Page Pull starts as soon as the last transaction at the source completes. Analogous to Parallel Page Access, the destination pulls pages which it requires but that have not been transferred yet from the source. To finish migration, the source additionally pushes pages which have not been transferred during Hot Page Push or as response of a pull request from the destination.

Finally, after the destination owns all pages, migration cleans up used resources and completes by a handshake between the source and the destination.

3.3 Snapshot Isolation during Parallel Page Access

During Parallel Page Access, concurrency control has to span the source and the destination since both of them run transactions concurrently. For this reason, the concurrency control measures are different to the standard measures done without migration. Note that the standard measures are totally sufficient during Hot Page Push and Cold Page Pull.

In a RDBMS that promotes the ACID principle, concurrency control must work properly even during migration. To foster the trust in the correctness of ProRea, we subsequently present a semi-formal proof that Parallel Page Access ensures snapshot isolation. For this, we assume the following prerequisites: Snapshot isolation runs correctly without migration (Prereq. 1). The source and the destination access the same pages as they would without migration (Prereq. 2). The ownership concept works correctly (Prereq. 3). Recall that the ownership concept synchronizes page access. The ownership concept ensures that the node which owns the page has its current version and this node is the only one which is allowed to modify the page.

LEMMA 3.1. *During Parallel Page Access, a transaction $T_s \in \mathcal{T}_s$ sees the latest committed tuple version that existed when T_s started.*

PROOF. As the source does not start new transactions anymore, $C(T_d) > S(T_s), \forall T_d \in \mathcal{T}_d$ holds. Therefore, T_s never has to see tuple versions created by T_d. Hence, the latest committed tuple version which T_s has to see must exist at the source and is therefore visible to T_s. □

LEMMA 3.2. *During Parallel Page Access, a transaction $T_d \in \mathcal{T}_d$ sees the latest committed version of a tuple t that existed when T_d started.*

PROOF. We prove this by contradiction. Let us assume that there exists the latest committed tuple version v_{T_s} of transaction T_s, $C(T_s) < S(T_d)$, and v_{T_s} is not visible to T_d. Prereq. 1 and prereq. 3 directly yield that T_s had to run at the source, thus $T_s \in \mathcal{T}_s$.

If T_s modified pages P_1 and P_2 with $o(P_1) = source$ and $o(P_2) = source$ to write v_{T_s} into P_1, T_d pulls page P_1 (or P_2 in case of a forward version chain). Thus, contrary to the assumption, it has to see v_{T_s} (either directly or, in case of a forward chain, by following the version chain to P_1 which T_d also pulls in this case).

If T_s modified a page P_1 and P_2 with $o(P_1) = temporalsource$ and $o(P_2) = temporalsource$ to write v_{T_s} into P_1, T_s returns ownership to the destination with modification records that ensure that the images of P_1 and P_2 are consistent at the source and the destination. As the modification records are applied before the ownership transitions to the destination, T_d waits until the image of P_1 is consistent to the source. Thus, contrary to the assumption, it has to see v_{T_s} (either directly or by following the version to P_1 for whose consistency T_D again waits).

The remaining cases are analogous to the previous cases.

Thus, in all cases, T_d sees v_{T_s}. □

LEMMA 3.3. *During Parallel Page Access, a transactions $T_s \in \mathcal{T}_s$ aborts if the version v_{T_s} of a tuple t which it wants to create conflicts with an already existing version v_{T_d} of t created by a concurrent transaction $T_d \in \mathcal{T}_d$.*

PROOF. T_d acquires an exclusive lock for t in order to write v_{T_d} into page P. If T_s is unable to obtain the lock at the destination since T_d still holds the lock, T_s aborts. If T_s acquires the lock (which implies that T_d has committed or aborted), it eventually gets the current image of P that contains v_{T_d} in the response to the ownership request (analogous to Lemma 2). Hence, T_s sees v_{T_d} and, thus, aborts. □

LEMMA 3.4. *During Parallel Page Access, a transaction $T_d \in \mathcal{T}_d$ aborts if the version v_{T_d} of a tuple t which it wants to create conflicts with an already existing version v_{T_s} of t created by $T_s \in \mathcal{T}_s$ with $C(T_s) > S(T_d)$.*

PROOF. If T_s still runs at the source, it still holds an exclusive lock for t at the destination in order to write v_{T_s}. The exclusive lock semantics ensure that T_d is unable to get the lock for t. Thus T_d has to recognize the conflict and wait for completion of T_s. As T_s notifies the destination about its completion, strict 2-phase locking ensures that T_d gets the lock for t not until it is able to inspect the state of T_s. If T_s has committed, T_d aborts. Note that this also applies if T_s has already completed at the time T_d acquires the tuple lock. □

THEOREM 3.5. *ProRea ensures snapshot isolation during Parallel Page Access.*

PROOF. From Lemma 3.1 and Lemma 3.2 together with Prereq. 1 follows that a transaction $T \in \mathcal{T}_s \cup \mathcal{T}_d$ sees the latest committed tuple versions that existed when T started. Thus, rule 1 of snapshot isolation holds during Parallel Page Access.

Prereq. 1 implies that local conflicts, i. e. conflicts between transactions at the same node, are recognized. Lemma 3.3 ensures that a transaction $T_s \in \mathcal{T}_s$ aborts, if there exists a potential conflict with a transaction $T_d \in \mathcal{T}_d$. Lemma 3.4 ensures that a transaction $T_d \in \mathcal{T}_d$ aborts if it conflicts with a transaction $T_s \in \mathcal{T}_s$ and $C(T_s) > S(T_d)$. Hence, a transaction will successfully commit only if its updates do not conflict with any updates performed by transactions that committed since the transaction under concern started. As consequence, rule 2 of snapshot isolation holds during Parallel Page Access.

Thus, Parallel Page Access meets rule 1 and rule 2 of snapshot isolation. □

Note that Lemma 3.3 is more restrictive than required to accomplish snapshot isolation. If transaction T_s observes a conflict with a still running transaction T_d, it aborts even if T_d eventually fails (which T_s does not know). That is, transactions at the source do not wait for tuple locks or check completion status of other transactions at the destination during Parallel Page Access. From the perspective of T_s, this constitutes a pessimistic decision, as it aborts although it has the chance to commit successfully (in most cases a chance with quite low probability). We argue that this design decision simplifies ProRea considerably. This prevents synchronizing lock release from the destination back to the source. In addition to that, it prevents synchronizing completion of transactions from the destination to the source. Finally, it prevents more complicated deadlock detection and management, as a deadlock cycle can never span the source and the destination.

4. ANALYSIS AND IMPROVEMENTS

This section embarks upon analyzing run-time costs and providing design rationales for ProRea. Thereafter, it presents an improved buffer pool handling approach.

4.1 Analysis and Design Rationales of ProRea

4.1.1 Hot Page Push

Hot Page Push requires a shared latch on a page while transferring it, which blocks writers of the page and thus ensure a physically consistent copy of the page. Yet, the shared latch is very short if pages are transferred asynchronously. The same holds for creating and shipping the modification records. Note that modification records are often created anyway for logging or replication. Therefore, we regard the overhead at the source as negligible.

This is different at the destination, as it inserts retrieved pages and applies retrieved modification records. If the pages under concern are still in $\mathcal{B}_d$, the load caused by applying the modification records tends to be lower than transaction load at the source, with which the destination should be able to deal anyway. In addition to that, the destination has to insert pages into $\mathcal{B}_d$ which is assumed uncritical if the destination has enough available buffer space. If the destination has to free buffer space or has to write out retrieved pages, Hot Page Push may cause severe load at the destination. We regard this an important issue in down-scaling scenarios that requires further evaluations.

4.1.2 Handover

Handover transaction processing from the source to the destination represents a critical part, as it blocks most operations on $\mathcal{D}_m$. The length of the blocking period mainly depends on the transfer time of the handover message which in turn depends on its size and the network latency. Recall that the handover message transfers the database run-time state. During our tests, the main part of the database run-time state consisted of the lock table and the ownership map. Thus, we limit ourselves considering the size of the lock table and the ownership map: A lock table with 500 lock identifiers each having 64 bytes requires 32 Kb. This is a reasonable (perhaps slightly overestimated) size, as we assume simple, short-running transactions, which only update few tuples; a lock table with 500 entries approximates to 50 concurrent transactions each holding 10 locks. The ownership map requires two bits for the state of each page. Thus, for a page size of 8 Kb and a database size of 5 GB, the total size of the ownership map approximately amounts to 160 Kb. Under these prerequisities, the total size of the handover message is about 200 Kb which requires less than 3 ms transfer time in a latency-free 1 Gbit/s ethernet network at half speed. In practice, network latency adds to the transfer time. Yet, even if latency adds additional 10 ms, the transfer of the handover message is short. Thus, the blocking period during handover is short and, thus, its impact is low.

4.1.3 Parallel Page Access

Parallel Page Access entails synchronization between the source and the destination. The synchronization principles from the source to the destination and the reverse direction differ; pages are synchronized eagerly from the source to the destination, whereas the reverse direction is done lazily. The lazy synchronization is justified as eager synchronization of *all* modifications from the destination to the source is obviously unnecessary. This is different the inverse way. As the destination requires the current image of the page for future transactions, eager synchronization of pages from the source to the destination is useful.

A side effect of the lazy synchronization is the fact that the destination returns the whole image of a already modified page only if the source asks for its ownership. At the time the source requires the page, the destination need not have the previous page image anymore. Therefore, it is unable to create the corresponding modification records. It may indeed search the log (if existent) for the corresponding records, but this is more complicated than simply returning the page and may even require additional disk I/O.

The source directly returns the ownership of the page after its modification. It might be objected that the source may keep the ownership such that other transactions at the source may also modify the page without requesting the ownership from the destination again. However, Parallel Page Access is short and transactions only update few tuples. The probability that multiple transactions at the source require writing the same page is in general low and will further decrease in the future. Hence, page ownership would eventually return to the destination without additional accesses at the source in most cases. For this reason, it is useful to take the chance to create modification records, send them directly to the destination and piggyback return of ownership.

The total synchronization overhead during migration highly depends on the write share, page contention and network latency. A high write share and high page contention increase the probability that a transaction at the source and a transaction at the destination desire to write the same page. For example, if all transactions insert tuples in ascending order, many transactions want to write to the same page. This requires transferring the page back and forth be-

tween the source and the destination. However, Parallel Page Access is short, as transactions are short. Thus, in practice the amount of pages that is transferred back and forth should be small.

4.1.4 Cold Page Pull

During Cold Page Pull, two operations affect overall performance: page push from the source to the destination and page pull by the destination.

The former operation requires reading a huge part of the database at the source and writing it at the destination. Although the operations cause mainly sequential disk I/O, their impact is usually too high to run them full speed ahead. Therefore, limiting the throughput of these operations is essential to limit its impact on overall performance. [26] shows how the overall performance depends on the throughput of reading and writing the database. Based on these results, they propose a control-theoretic approach to limit the performance impact due to migration. Their results also hold for pushing the pages during Cold Page Pull.

If a transaction at the destination pulls a page from the source, the access time of the page includes network transfer time and network latency. Let us assume that the network overhead adds 1 ms. Compared to a disk access which may easily require 10 ms on a moderately utilized machine, the network overhead is low. However, compared to a buffer pool access which may take about 0.2-0.5 ms, the network overhead is high. As ProRea transfers the hot pages during Hot Page Push, a page fault typically entails a disk access at the source why we consider the relative degradation by the network overhead as tolerable.

4.2 Index Migration

Note that our previous description of ProRea is not limited to a certain storage structure. For this reason, ProRea works for an index like the primary storage (the storage which stores the actual tuples of the table) if the index uses MVCC, as described in 3.1, in order to serialize access to its tuples.

Alternatively, an index may index all visible versions of a tuple in the primary storage. An index access then implies an access of the looked up tuple in the primary storage to determine its visibility. In this case, locks for index tuples are unnecessary and only latches are required to ensure the physical integrity of index pages. PostgreSQL being an example of such an implementation. As the ownership concept used in ProRea ensures the physical integrity of pages, ProRea is usable for this kind of index implementation as well.

4.3 Improved Buffer Handling

Common buffer pool replacement strategies require to know the access history of a page, e. g. usage counters, to take the right decisions which buffer pages to replace. To save this information, the migration task also transfers the related meta data of a page. Naturally, all phases of ProRea maintain the meta data suitably. For example, if the source requests temporal ownership for a page, the destination also increments the usage counter of the page when granting ownership.

In a Shared Process multi-tenancy model, multiple databases, i. e. multiple tenants, share available buffer pool space. Either all databases share the whole buffer pool space equally or each database obtains a reserved amount of buffer pool space dedicated to it. In both cases, if a database frees buffer pool space, other databases will be able to allocate more space. For example, if ten databases equally share 4 GB of buffer pool space and one is shutdown, the remaining nine database may allocate about 11 % more buffer pool space. For this reason, ProRea frees $\mathcal{B}_s \cap \mathcal{D}_m$ directly after Parallel Page Access, i. e. after all transactions at the source have completed.

To free the buffer pool space, dirty pages require writing out, which causes considerable random disk I/O. However, if a page has been already transferred to the destination and the source does not change the page anymore, it is safe to skip the page immediately from the buffer pool without writing it out. From a conceptual view, the transfer of the page replaces writing out the page to disk. Note that this does not violate durability; the recovery protocol ensures durability.

5. EXPERIMENTAL EVALUATION

We built a prototypical implementation of ProRea integrated into PostgreSQL 8.4 [20], which provides snapshot isolation. Furthermore, we implemented a simple testbed to evaluate the run-time characteristics of our prototypical implementation. For comparison with a purely reactive approach, we used an implementation without Hot Page Push and optimized buffer pool handling, which we refer to as *Pure Rea*.

5.1 Test Environment

5.1.1 Testbed Implementation

The schema of our testbed consists of one table and one index: the *Customer* table of the TPC-C benchmark and an index for the primary key. Based on this schema, we generated three databases, each having a total size of about 5.1 GB.

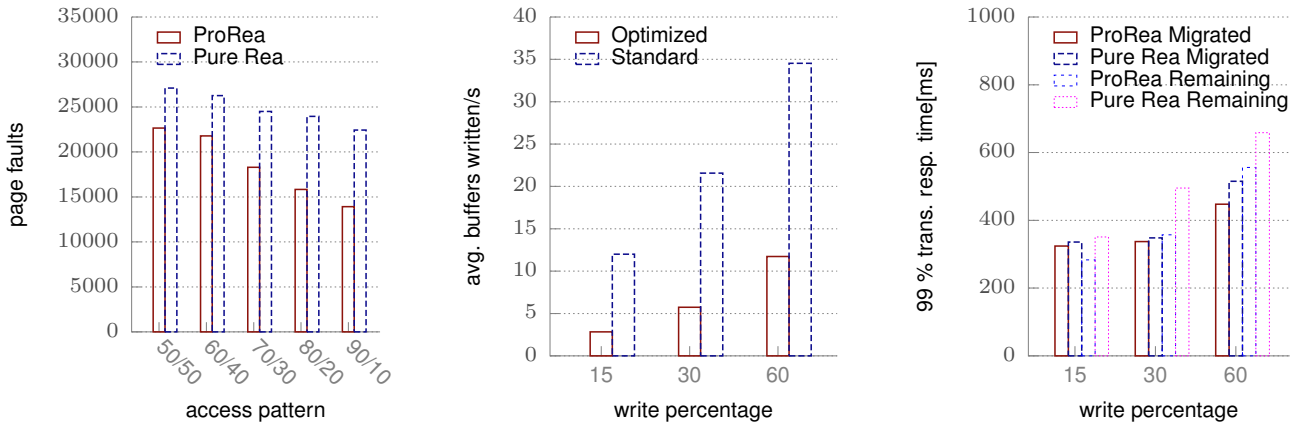
The load simulator of the testbed runs a mix of simple select and update transactions, each accessing a single tuple by its primary key. The load simulator takes parameters for the access pattern, the read/write ratio, and the number of transactions to be issued per second (tps). The access pattern parameter specifies the percentage of transactions that access a certain percentage of the data, e. g. 80 % of transactions access 20 % of the data and 20 % of transactions access 80 % of the data. Within the resulting data ranges, transactions access data uniformly random. We refer to the respective access pattern by *transaction percentage/data percentage*, e. g. 80/20.

The load simulator begins each run with a ramp up phase that starts a new worker thread every 30 seconds till it reaches 25 threads. Each thread connects to all databases. The total warm up period before migration starts is 70 minutes. After this period latencies and throughput turned out to be quite stable for our tests. The load simulator distributes the transactions uniformly random across the configured databases and available worker threads. Hence, all databases have to serve the same share of load. At the beginning, the source serves all three databases. After 70 minutes, one database is migrated from the source the destination. Thus, after migration, the source serves two databases and the destination serves one database.

To limit the impact of pushing the pages during Cold Page Pull, we throttled transfer throughput to 4 Mb/s. Moreover, we ran three passes for each test, whereas each pass started on freshly booted machines. As the results of each pass were similar and each pass would lead to the same conclusions, we only report the results of the last pass.

5.1.2 Test Systems

For our tests, we used two Dell Optiplex 755, one for the source and the other for the destination. The machines were equipped with an Intel Core2 Quad Q9300 CPU running at 2.50 GHz and 4 GB of main memory. We stored the database and its log on two striped 250 GB SATA 3.0 GB/s hard drives spinning at 7.200 RPM. The test machine ran a 64 bit 2.6.32 Linux kernel (Ubuntu release 10.04 Server), configured without swap space. The client machine on which we ran our test tools was equipped with four Dual Core AMD Opteron 875 CPUs running at 2.2 GHz and 32 GB of main memory.



(a) Total page faults. (b) Avg. buffers written per second. (c) 99 % transaction response times.

Figure 3: Figure 3a compares the page faults of ProRea with a purely reactive approach, called Pure Rea. The different data access patterns entail increasing buffer pool hit ratio from left to right (80/20 means that 80 % of transactions access 20 % of the data). Figure 3b compares the standard buffer handling strategy of PostgreSQL with our optimized strategy (Sec. 4.2) during migration. Figure 3c shows the 99 % quantile of transaction response times during migration. Figure 3b and 3c shows the results for different write percentages and the access pattern 80/20.

The operating system was a 64 bit 2.6.18 Linux Kernel (CentOS release 5.8). All machines were connected over a 1 GBit/s ethernet network.

PostgreSQL's buffer pool was configured to 1.5 GB at the source and the destination. Autovacuuming, autoanalyze and checkpoints have been disabled.

5.2 Measurements

To estimate the page fault reduction of ProRea with respect to different buffer pool hit ratios, we started with tests that count the page faults for different access patterns during migration. The results presented in Fig. 3a show that the number of page faults decreases with higher buffer pool hit ratio (from left to right) for ProRea and Pure Rea. This is because higher buffer pool hit ratios yield lower numbers of different pages that are accessed during the period of migration. Obviously, the number of page faults caused by ProRea reduces more relative to Pure Rea. ProRea produces about 16 % less page faults for 50/50 and about 38 % less page faults for 90/10. Thus, regarding page fault reduction, ProRea has it strengths for workloads with good buffer pool hit ratio.

ProRea sends modification records during Hot Page Push to synchronize already transferred Page. During our test, the number of modification records scaled almost linearly with the write percentage. Compared to the number of reduced page faults, the number of modification records was considerably lower (about 8200 less page faults relative to Pure Rea and about 1300 additional modification records for a write percentage of 30 % and access pattern 80/20). Hence, ProRea reduces the total number of messages sent relative to Pure Rea. Despite of the message reduction, ProRea often increases the total amount of data sent across the network. This is because modification records are typically considerably larger than page pull requests. Our current implementation sends the whole modified tuple why the additional amount of data depends on the average tuple size, which was about 320 bytes. For example, the test for 30 % write percentage transferred approximately 420 Kb additionally, which we regard negligible compared to the total database size (5.1 GB). Moreover, the transfer of modification records is less time critical than processing page faults, as it typically does not lead to wait periods for transactions.

To estimate the effectiveness of the optimized buffer pool handling, we measured the average number of buffers written per second at the source during migration. Fig. 3b shows the results for the standard buffer handling strategy of PostgreSQL and for our optimized buffer handling strategy. The results evidence that the optimized buffer handling strategy reduces the number of buffers written at the source. Naturally, the optimized buffer handling strategy mainly takes effect for higher write percentages. Moreover, its effect is limited to the allocated space of the database that is migrated. Anyway, it is a simple but effective optimization. For example, in our tests, it reduces the average number of buffers written per second from 22 to 6 for a write percentage of 30 %. This reduced the average number of disk I/O operations at the source by more than 10 %.

The reduction of disk I/O operations is in line with the 99 % quantiles of transaction response times during migration, as Fig. 3c depicts for ProRea and Pure Rea. The transaction response time is the end-to-end execution time as difference from the time of issuing the query to the time of retrieving the results. The difference between the 99 % transaction response times for the remaining databases of ProRea and Pure Rea is obvious and increases with the write percentage. Interestingly, the 99 % response time of the migrated database is nearly identical between ProRea and Pure Rea, although Pure Rea has to cope with more page faults. This is because the higher number of page faults did not cause a severe penalty in our tests since the network connection between the nodes in our test environment was very fast (< 0.1 ms latency).

Finally, we ran a test which mimics a typical outscaling scenario. In this scenario, the transaction throughput increases from 100 to 150 transactions per second within a time window of 10 minutes. Note that the load is still equally distributed across the three databases. That is, the number of issued transactions per second with which the source has to deal remains the same, before and after migration. The load was generated according to the access pattern 80/20. The write percentage was configured to 30 %. Three minutes after starting with increasing load, migration starts. Fig. 4 depicts the average transaction response time using a moving average over one minute. About 2 minutes after start of migration, the migration task transitions to Cold Page Pull. As a result, the response

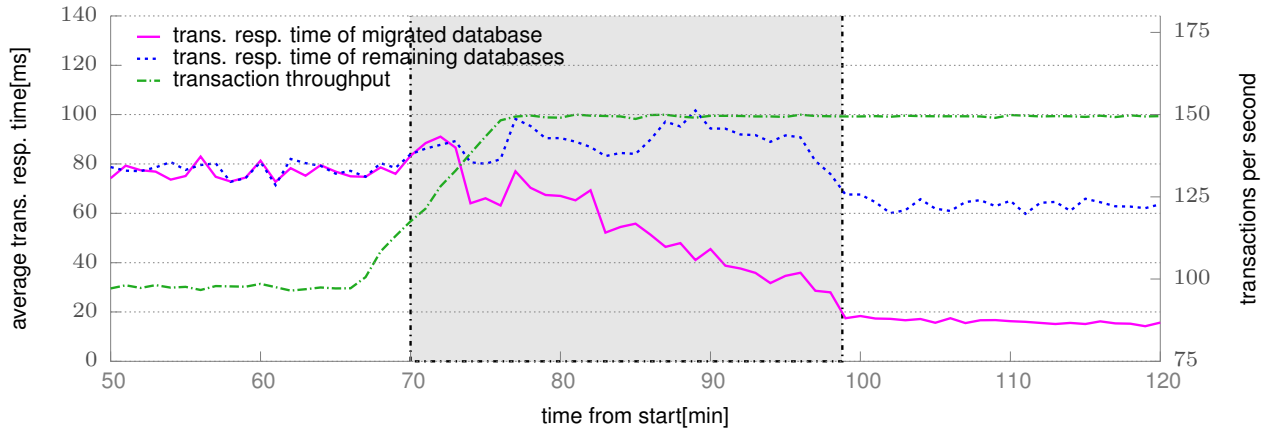


Figure 4: Outscaling scenario that increases transaction throughput from 100 transactions per second to 150 transactions per second. Each data point is calculated as the moving average over the past minute. The grey-shaded box represents the migration window.

times reduce significantly. This is because of the buffer handling that releases load at the source. As the load further increase the response times increase as well, until the load reaches its maximum. Thereafter, the graph shows that the response times of the migrated database decrease with ongoing migration. This is because more and more data is at the destination by what the number of page faults reduces and, thus, the destination gets more and more independent from the source. The response times of the remaining databases degrade between 5 and 25 % compared to the response times before migration, although load at the source does not increase. This results from reading pages to push them to the destination. After migration, the response times improve due to the lower total data volume with which the source has to deal. In another test, we ran the same scenario but limited transfer throughput during Cold Page Pull to 2 Mb/s. In this case, the average response times do not deteriorate more than 10 %, but the migration time almost doubles.

6. SYSTEM FAILURES

Recovery from system failures represents a core functionality in a RDBMS implementation to guarantee the ACID principle. Database migration using ProRea faces recovery with a new challenge: the consistent state of the database can be distributed over the source and the destination in the event of a system failure. This section provides a conceptual discussion about dealing with system failures that occur during database migration with ProRea. This discussion focuses on showing that recovery from system failures is feasible without degrading migration performance significantly.

For this discussion, we assume a log-based recovery approach that uses typical log replay techniques to restore the latest transaction consistent database state before the system failure. Log-based recovery is undoubtedly the most common recovery approach in current RDBMS implementations.

6.1 General Reflections

To ensure common consenus about the current migration phase even in the event of a system failure, the two-way handshake protocol presented in [10] is reasonable for ProRea as well. It asserts permanence of the current phase by logging the sending and receiving of messages related to phase transition.

The handover of the database run-time state does not require to be synchronized with the update of the database location at the cluster management (different to [10]). This is because the source notifies clients about the new location of the database. Thus, the source may update the cluster management instance asynchronously using the same two-way handshake protocol as used for phase transition. Migration certainly can only finish after this update.

In the event of a system failure during Prepare or Hot Page Push, the aborting of the migration represents the most obvious choice. The work done so far is low and aborting of the migration only requires cleaning up few data structures, e. g. meta data entries. A significant advantage of aborting the migration is flexibility. This is due to the fact that the source still owns solely the consistent database image. Moreover, the source and the destination can decide to abort the migration alone due to the common consensus about the migration phase. Hence, if the destination fails, the system can decide to start another migration from the source to an alternative destination without waiting for the failed destination. In Parallel Page Access and later phases, the consistent database image spans the source and the destination, by what both of them have to be alive for successful recovery. This case offers the continuation of the migration which requires restoring its state. The subsequent sections discuss how to enable restoring the state of migration.

6.2 Logging

Just as the database state, the database log gets distributed over the source and the destination after handover. The source logs only updates of page P if $o(P) = source$, i. e. if it actually owns P. The destination logs updates of P if $o(P) = destination$ or $o(P) = temporal source$. The latter case is accomplished because the destination logs applying modification records retrieved from the source, but not until handover. The source and the destination independently maintain the latest log sequence number (LSN) (see Sec. 6.3) which a transaction $t_s \in \mathcal{T}_s$ caused at the source and the destination respectively. Before t_s is allowed to commit, the source and the destination flush their respective local log up to the respective latest LSN of t_s.

The descibed logging approach ensures that all log entries belonging to transaction t_s are flushed, partially at the souce and partially at the destination. This is sufficient for a system which runs a selective redo approach; PostgreSQL being an example of such a system.

An Aries-style [17] recovery approach however mandates a complete redo that repeats all changes of a page in chronological order

(referred to as repeating history), independent of the completion status of the related transaction. Therefore, the previously sketched logging approach additionally requires flushing log entries related to a page before its transfer to the destination. During Hot Page Push, this is actually not required because all transaction still run at the source and hand over of database run-time state flushes the log due to the phase transition anyway. Hence, only Parallel Page Access is affected. As Parallel Page Access is short, we consider the resulting additional overhead as negligible.

6.3 Log Sequence Numbers

The LSNs increase independently at the source and the destination, why they only provide local chronological ordering of log entries. Yet, ProRea implies that all log entries related to a page P at the source are chronologically before all log entries related to P at the destination. This is because, after hand over of the database run-time state, the first transfer of a page P to the destination irreversibly transfers the responsibility for logging modifications of P to the destination. Hence, with respect to P, arranging the log entries from the source before the log entries at the destination creates a partial order that relates entries of P in chronological order, which suffices for page-oriented redo and page-oriented undo. A special marker LSN allows indicating a reference to the latest log entry at the source. After the receipt of a page, the destination resets the LSN recorded in the page to this special marker LSN. During recovery, a special marker LSN in a page indicates that the destination has to apply all existing log records for the page.

Recall that a transaction $t_s \in \mathcal{T}_s$ creates log entries at the source and at the destination during Parallel Page Access. For transaction-oriented logical undo, the chronological order of log entries caused by t_s has to be preserved. The previously induced partial order does not preserve this order. For this reason, additional measures are necessary to enable transaction-oriented logical undo. For instance, overlaying a logical order over the log entries caused by t_s is one approach. The identifiers which establish the order are attached to all log entries caused by t_s (at the source and the destination). Standard messages of the ProRea protocol piggy back the identifiers for related log entries at the destination. Alternatively, logical LSNs (at least partially) allow relating a LSN to the node which produced it, e. g. as done in [29] for similar purposes. This allows building a chronologically ordered chain per transaction, but it requires synchronously shipping and applying of modification records from the source to the destination.

6.4 Durability of Migration State

The goal is to preserve most work of the migration in the event of a system failure, but without severe impact on its run-time performance. The two obvious, extreme approaches are: (I) discard all work done and start the migration from scratch or (II) flush each retrieved page to disk and log its receipt. The first approach has no run-time efforts, but it loses most work done. The second causes a considerable amount of disk I/O requests, which yields a performance loss during run-time. These approaches are naturally far from the desired goal.

Therefore, we propose an approach in which the buffer pool manager at the destination creates a log entry for the successful completion of writing out a page. The corresponding log entry is not forced to stable storage immediately, but it will be flushed to disk prior to or with the subsequent commit entry. Logging the successful write of a page is sometimes anyway done to minimize redo efforts. In our case, it allows determining which pages have been transferred and have been successfully written to stable storage.

After all pages have been transferred to the destination, the destination runs a fuzzy checkpoint [12], which eventually creates a page consistent database image at the destination. Thereafter, the source is allowed to purge the database.

6.5 Recovery

During analyze of the log, the destination creates an ownership map from the log entries the buffer pool manager has written. The destination transfers this ownership map to the source. Thereafter, the source starts redo for the pages it still owns. The destination does the same for the page it owns. If the destination finds a log entry related to a page it does not own, it has to fetch the page from the source. Naturally, prior to this, the source has to apply all redo log entries for this page. Thereafter, the undo pass starts. Each node undoes transactions it has started. If the failure occured during Parallel Page Access, this may require to ship undo logs or pages from one node to the other and vice versa.

If only one node fails, only the failed node has to recover. However, in any case, the ownership map has to be transferred from the destination to the source to ensure commen consensus about the ownership of a page.

6.6 Summary

Although the previous discussion only briefly considers recovery in the event of system failure, it shows that recovery is feasible without significant run-time overheads. At this, Parallel Page Access represents the most critical phase. It may require additional measures to ensure appropriate ordering of log entries. However, these measures are limited to Parallel Page Access, which is assumed to be short anyway.

Log-based recovery is a disscussable point by itself in highly available system environments. In such environments, replicas of a database lend themselves for recovery. [14] has shown that failover and rebuild is efficiently feasible. Furthermore, highly available system environments offer an interesting opportunity. If the source fails, the migration may continue from a replica. The destination simply has to ship the ownership map to the replica.

Even if recovery does not run log-based, the database log might be useful for other features, at least partially. For example, [31] adopts the undo log to implement point in time recovery without the need of restoring a whole database image. Such functionality is quite appealing in system environments that allow rare database backups as highly available systems usually do. Hence, the previously discussed points hold for a broader range than only recovery.

7. CONCLUSIONS AND FUTURE WORK

Live database migration allows satisfying increasing capacity needs of a tenant by moving it to a node with higher capacity. Furthermore, it allows minimizing costs by moving tenants physically closer together during periods of low load. This makes live database migration a compelling functionality to run a multi-tenant RDBMS efficiently. Yet, live database migration constitutes a challenging task, as it must not interrupt service processing or degrade service performance significantly.

ProRea meets this challenge well. By combining proactive and reactive measures to transfer the database state from one node to another node, it provides efficient live migration for multi-tenant RDBMS which fulfill snapshot isolation. The proactive measures in conjunction with the improved buffer pool handling entail less page faults and less disk I/O and, ultimately, less migration overhead compared to a purely reactive approach. ProRea unburdens the source very fast, as it switches transaction load from the source to the destination early. This behavior is particularly useful to migrate a tenant from an (almost) over-utilized node to a lowly utilized node.

As mentioned in the introduction, an autonomous controller ideally decides when to migrate, which database to migrate and where to migrate. In this case, ProRea is automatically initiated and filled with required arguments. Alternatively, an administrator may initiate ProRea manually if a tenant's SLAs are violated or if a machine requires maintenance. The decision on when, which and where to migrate is difficult. Such decisions require to take into account typical capacity planning and tenant placement criteria: load patterns of a tenant, the growth of a tenant, the associated penalty if a tenant's SLAs are violated, diurnal cycles of a tenant's load and so forth. Against this background, we consider an evaluation of existing capacity planning and tenant placement methods with regard to their customization for migration decisions an interesting direction for future work.

Although snapshot isolation became quite popular, it does not avoid all concurrency anomalis. Recent research into serializable snapshot isolation shows that guaranteeing true serializabiliy retains most of the performance benefits of snapshot isolation [5]. A first implementation of serializable snapshot isolation in a production RDBMS release, namely PostgreSQL, underpins these results [19]. These results reveal a new direction for future work: the redesign and clean reimplementation of ProRea to guarantee serializable snapshot isolation during migration.

8. REFERENCES

[1] D. Agrawal, A. E. Abbadi, S. Das, and A. J. Elmore. Keynote: Database scalability, elasticity, and autonomy in the cloud. In *DASFAA*, 2011.

[2] Amazon. Amazon Relational Database Service. http://aws.amazon.com/rds/, 2012.

[3] Aulbach et al. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *Proc. of SIGMOD Conf.*, pages 1195–1206, 2008.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of SIGMOD Conf.*, pages 1–10, 1995.

[5] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proc. of SIGMOD Conf.*, pages 729–738, 2008.

[6] F. Chong and G. Carraro. Architecture Strategies for Catching the Long Tail. Microsoft Corp. Website, 2006.

[7] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *Proc. of CIDR*, 2011.

[8] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[9] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Towards an Elastic and Autonomic Multitenant Database. In *Proc. of NetDB Workshop*, 2011.

[10] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*, pages 301–312, 2011.

[11] G. C. Frederick Chong and R. Wolter. Multi-Tenant Data Architecture. Microsoft Corp. Website, 2006.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] D. Jacobs and S. Aulbach. Ruminations on Multi-Tenant Databases. In *Proc. of BTW Conf.*, pages 514–521, 2007.

[14] E. Lau and S. Madden. An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In *In Proc. VLDB*, pages 703–714, 2006.

[15] P. Mell and T. Grance. The NIST Definition of Cloud Computing. *NIST*, 53(6):50, 2009.

[16] Microsoft. Sql azure. http://msdn.microsoft.com/en-us/library/ee336230.aspx.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[18] R. Normann and L. T. Ostby. A theoretical study of 'snapshot isolation'. In *Proc. of the 13th ICDT*, 2010.

[19] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. In *Proc. VLDB Endow.*, pages 1850–1861, 2012.

[20] PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org`, 2012.

[21] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978.

[22] B. Reinwald. Database support for multi-tenant applications. In *In IEEE Workshop on Information and Software as Services*, 2010.

[23] Salesforce.com. Salesforce. http://www.salesforce.com, June 2012.

[24] O. Schiller, A. Brodt, and B. Mitschang. Partitioned or Non-Partitioned Table Storage? Concepts and Performance for Multi-tenancy in RDBMS. In *Proc. of SEDE Conf.*, 2011.

[25] O. Schiller, B. Schiller, A. Brodt, and B. Mitschang. Native support of multi-tenancy in RDBMS for software as a service. In *Proc. of EDBT Conf.*, pages 117–128, 2011.

[26] Sean Barker et al. "cut me some slack": Latency-aware live migration for databases. In *EDBT*, 2012.

[27] G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41:14:1–14:136, July 2009.

[28] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD Conference*, 2008.

[29] J. Speer and M. Kirchberg. D-aries: A distributed version of the aries recovery algorithm. In *ADBIS Research Communications*, 2005.

[30] SuccessFactors. Distinctive cloud technology platform. http://www.successfactors.com/cloud/architecture/, March 2012.

[31] T. Talius, R. Dhamankar, A. Dumitrache, and H. Kodavalla. Transaction log based application error recovery and point in-time query. *PVLDB*, 5(12):1781–1789, 2012.

[32] Umar Farooq Minhas et al. Elastic scale-out for partition-based database systems. In *International Workshop on Self-Managing Database Systems (SMDB '12), ICDE Workshops*, 2012.

[33] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *Proc. of SIGMOD Conf.*, pages 889–896, 2009.

[34] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.