Postgres-R (8) Architecture

Markus Wanner <markus@bluegap.ch>

January 2009

Abstract

This document describes the design and architecture of Postgres-R (8), a multi-master replication system for Postgres. It is an extension of the work presented by [KA00] and incorporates enhancements from the subsequent paper Postgres-R (SI) by [WK05]. Further inspiration originates from Slony-II of Neil Conway and Gavin Sherry and from conversation with other fellow hackers of Postgres.

Please note that this paper describes the underlying concept and does not necessarily reflect the status of the prototype implementation, which is available from `http://www.postgres-r.org`.

The reader is supposed to be familiar with Postgres internals, especially with Multi-Version Concurrency Control (MVCC) and transaction isolation issues.

1 Design Goals

Database replication promises to solve a broad range of very different problems. Possible use cases vary in number of nodes, type of transactional load, throughput and latency of the interconnect and (perhaps most importantly) user expectations and requirements. This section outlines the design goal and limitations of the Postgres-R approach, as presented in this paper.

- Multiple nodes with independent memory and storage are connected via a mostly reliable and preferably low latency interconnect. This cluster of nodes forms a distributed database system, which looks like one big database system to the database users.
- Postgres-R is a pure update-everywhere (or multi-master) replication solution, which allows every node of the cluster to process writing transactions at any time (in the absence of failures).
- To ensure high availability, Postgres-R must be tolerant to various failure cases, including single node crashes, network congestion, temporary network outages as well as permanent network failures. Consequences of these failures must be clear and predictable.
- Different applications and transactions require different trade offs between commit latency, visibility and durability. However, we do not currently consider lazy replication, but instead focus on conflict-free replication using fully synchronous or eager replication.
- ACID properties of the database system must be maintained by default to ensure application transparency. For better performance Postgres-R optionally offers compromises, which leads to a behavior slightly different from a single node system and possibly violating ACID properties.
- The nodes of a Postgres-R cluster may run on hardware platforms and operating systems and may use different Postgres major versions underneath.
- After a full cluster outage the distributed system must be able to recover into a consistent state again. The transaction durability guarantees must be respected.
- Initialization of new nodes and recovery of re-joining nodes as well as full cluster restarts must be performed

automatically without requiring human intervention.

- All network communication (reliable multi-casting, consensus protocols, split brain detection, etc.) is delegated to a group communication system.
- Integration into Postgres must be tight to benefit from MVCC.

2 Architecture

2.1 Overview

Postgres-R extends Postgres in various areas. These extensions can be split into modules or building blocks. Some of them are very specific to replication, others can potentially be used for other purposes as well. This first section provides an overview of Postgres-R from various angles. The following sections will each describe such a module or building block of Postgres-R.

2.1.1 Processes on a node

Postgres features a multi-process architecture, where every client connection is connected to exactly one so called backend process. A postmaster process controls these backends and forks new backends as required. But there are also various auxiliary processes which preform different background tasks. As of version 8.3 the following permanent auxiliary processes are running for a Postgres database system: an autovacuum launcher, a background writer, a WAL writer, a statistics collector and possibly also a WAL archiver. The autovacuum launcher may request additional autovacuum worker processes, which do the actual work of vacuuming a database.

To take care of all communication requirements, Postgres-R introduces an additional coordinator process[1]. Similar to the autovacuum launcher, it may request

additional helper processes, which do the actual work[2]. To communicate with other nodes Postgres-R relies on a Group Communication System (GCS), which may be implemented as its own daemon process or dynamically loaded into the coordinator process from a library. Besides passing messages between the helper backends and the communication system, the coordinator also keeps track of the state of local processes and remote nodes. However, it must guarantee short response times and may thus not perform lengthy calculations.

2.1.2 Lifecycle of a replicated transaction

In distributed database system the client connects to one of the nodes and its transaction is processed by a backend process on that node. That's the origin backend and origin node of the transaction. Read-only transactions need not be replicated, so there is no difference to stand alone operation. Changes from writing transaction get collected into change sets[3], which are then replicated to the other nodes in the cluster via the GCS

On the remote node the change set first arrives at the coordinator process, which only forwards it to a helper backend[4]to do the actual processing. To apply the change set the helper backend needs to checks for conflicts with concurrent transactions on the same node. Depending on that check and on the agreed ordering of the transaction with regard to other transactions, it then gets committed or aborted.

---

1. In the original Postgres-R paper [KA00] the coordinator process is referred to as the replication manager. To avoid confusion with the Resource Manager (abbreviated RMGR as well) this process has been renamed to coordinator.

2. It might make sense to merge the coordinator process added for replication with the autovacuum launcher, as both processes are similar in concept: they control worker processes.

3. Again, the original Postgres-R paper [KA00] used another term for the serialized changes: writesets. The term has been altered because its meaning and function has changed as well

4. These helper backends have been referred to as remote backends, because they apply transactions originating from remote nodes. However, they are in no way *remote* but reside on the same node as so called local backends. In this paper the more general term *helper backend* is used.

2.2 Inter-Node Communication

2.2.1 Services of the Group Communication System

Postgres-R delegates all of its inter-node communication to a GCS, which provides the required abstracted services for such a distributed system. For use with Postgres-R a GCS must provide group membership services, failure detection, reliable uni- and multi-cast messaging and a consensus protocol[5]. As part of the group membership service the GCS must detect node and network failures and appropriately inform the remaining nodes.

Upon partitioning of the network the GCS must be able to congruently decide on a majority of nodes to stay active. To prevent split brain situations Postgres-R forces all other nodes to leave the group. If there's no such majority partition, the group becomes inactive which leads to an interruption of the replicated database system. In that case administrator intervention is required to either fix the network problem and thus reactivate the replication group or to decide on a minority group to become the new replication group, in which case the administrator must take care to prevent split brain situations.

Various approaches for implementing consensus and atomic broadcast protocols have been studied. [DSU04] provides classification of atomic broadcast protocols and a theoretical comparison of around 55 different approaches. For Postgres-R non-uniform total ordering within a closed group is sufficient. Support for multiple, independent groups might be beneficial for performance reasons.

2.2.2 Global Object Identifier (GOI)

Most of the names for databases, relations, tables, etc. can be of varying size and are often longer than 4 bytes. To abstract from these names and to keep the size of messages small, Postgres-R does not send full names in every change set. Instead the whole cluster of nodes agrees on a mapping of objects identified by name to 32 bit globally unique object identifiers (GOI). This also helps reducing the number of varying size elements to parse and thus simplifies the structure of messages.

New records are created on demand by requesting an agreement from the GCS on an id for the next object to be inserted. To prevent endless growth of this mapping each node may also request removal of a record after it hasn't been used for some time. These object identifiers are supposed to change very rarely, so a timeout of several days before deleting an unused GOI should be reasonable in most cases. The GOI is very similar to the object identifier (OID) used by Postgres internally, but provides a mapping so that these OIDs can differ between nodes and Postgres versions. Note that not every object must have a GOI at every point in time. Additionally, objects which do not have an OID in Postgres may have a GOI, for example the nodes of a cluster.

2.2.3 Global transaction identifiers (GID)

To refer to a transaction from a node, Postgres-R uses globally unique transaction identifiers. Postgres itself already offers transaction identifier (XID) consisting of 32 bits. Postgres-R prepends a unique node id given from the GCS or mapped via the above GOI mechanism, resulting in a 64bit wide global transaction Identifier (GID). Wrapping around of the per-node XIDs is not a problem, because Postgres-R only needs to refer to recent transactions. Also note that Postgres-R only compares GIDs for equality - ordering information is stored independently.

2.2.4 Transaction State Tracking

On every node a mapping between global and local transaction ids must be maintained, so that backends can translate from GIDs to XIDs and back. To keep the administrative overhead as low as possible Postgres-R only stores GIDs of transactions which are still in progress on at least one node. Transactions which are known to be committed

5. Some GCS offer an atomic broadcast primitive which provides totally ordered delivery of messages instead of a consensus protocol. As shown in [CT96] this can be reduced to the consensus problem and vice versa. For the remainder of this paper we concentrate on the consensus abstraction, because it is more general and allows delivery of payload data to the application before having reached consensus (optimistic delivery).

on all nodes are not of interest to the replication system anymore. To keep track of each node's progress in applying remote transactions the nodes regularly exchange information about committed or aborted transactions.

From this information each node derives the latest transaction which is known to be globally committed (KGC). However, a node may not drop a GID until it is sure it won't be referenced anymore by any other node. Thus another limit must be kept track of: the latest known globally deprecated transaction (KGD). This limit can be derived from change sets received from other nodes, as each change set carries the KGC of the origin node at time of creation. Alternatively each node also sends its local KGC value with the above mentioned transaction state exchange messages, in case it didn't send a change set for a while. As these limits are only required for cleaning up the transaction state information, they are not time critical and can easily lag behind.

2.3 Collection of Transactional Changes

Changes performed by transactions must be transferred between the nodes via a network. This section describes the contents and the serialized format of change set messages and at which point in time during the execution of the original transaction the collection of changes takes place.

2.3.1 Scope of Tuple Collection

To maintain consistency of data between the nodes it is sufficient for Postgres-R to replicate only relational data. There is no need to replicate changes to derived data such as indices, statistics, free space maps, etc. because each replicated node can maintain its own set of derived data based on the relational data. To remain independent of the underlying storage format, of the specific Postgres version and of the CPU architecture used, Postgres-R does not need to transfer any of that data either. Instead Postgres-R uses primary keys to uniquely identify tuples, which in turn means that only relations with such a primary key can be replicated. However, this restriction can easily be circumvented by adding a hidden SERIAL attribute to relations which do not have an application specific primary key.

Not only the common INSERT, UPDATE and DELETE operations are considered to change tuples, but also COPY FROM as well as row level locking operations such as SELECT FOR UPDATE and SELECT FOR SHARE. However, those row level locking operations are only relevant in synchronous replication level and are not part of a change set for eagerly replicated transactions, see 2.6. Changes of sequences are not part of a change set either, because these have different locking requirements, see 2.7.1.

2.3.2 Time of Tuple Collection

On the origin backend changes must be collected just before they are effective for the local transaction within the `ExecInsert`, `ExecUpdate` and `ExecDelete` functions, so as to be able to block before the change[6] is effective. The collection of changes is independent of the cause for the change and thus also captures changes from operations from within stored procedures, triggers and rules.

2.3.3 Representing Tuple Changes

Based on the three basic types of operations Postgres-R stores different information per inserted, updated or deleted tuple. Each change set holds a collection of tuple changes of the same type and from within the same subtransaction. Both of these properties must be stored in the change set header, together with the number of tuples included and the current number of attributes per tuple.

For updates and deletes Postgres-R saves the (former) primary key attributes to identify the tuple to be changed as well as the GID of the transaction which has last touched it (from the tuple's `xmin`). The later is required for conflict detection during application on remote nodes, see 2.5.1. The primary key attributes are serialized without any additional information, because they cannot be NULL.

6. While attribute data changes of the tuple are not visible to other transactions due to MVCC, locking a tuple for updating or deletion is immediately visible to other transactions.

For updates all changed attributes get serialized in addition to the primary key attributes which identify the tuple. Note that upon a change of a primary key attribute, the serialized tuple change information contains both variants of these attributes: the old ones to identify the tuple as well as the new ones within the list of changed attributes. To distinguish between changed and unchanged attributes, a bitmask stores two bits per attribute of a relation, resulting in a maximum of four different ways an attribute can change: unchanged, reset to NULL, set to a new value or set to a new value derived from the old one and a given delta. The fourth variant is an optimization for large attribute values, which can safe network bandwidth in case of small changes to large attributes.

To unify handling of inserts, those are treated as changes against NULL. Unlike updates there is no need to store a primary key to identify some pre-existing tuple to be changed. The new primary key attributes are stored as part of the normal attribute changes.

2.3.4 Representing Attribute Data

The bitmap field indicates how many attribute values need to follow, because attributes which remained unchanged or which have been reset to NULL are skipped. Postgres-R stores attribute data in their binary representation using the `send` and `recv` functions provided by Postgres. For fixed size attributes no length information is required, whereas for variable length attributes a length indicator is prepended before the actual data, similar to VARLENA datums of Postgres. Datums beyond a certain size threshold are transmitted out-of-bound, resembling Postgres' TOAST mechanism.

2.4 Distributing Change Sets

All change sets are sent by reliably multi-casting them via the GCS. This guarantees eventual delivery of the messages or a failure notification in case of delivery failures. Unlike the original Postgres-R approach we do not use atomic broadcast for a totally ordered delivery, but instead decouple the ordering agreement and the change set payload transmission, which is known as optimistic delivery. The change set payload may be delivered before the nodes of the cluster reach an agreement on the ordering of the transaction, so that Postgres-R can start applying its changes before having received the ordering agreement, which benefits parallelism.

2.4.1 Timing and Client Confirmation

Change sets are sent at different times, depending on the chosen replication level. In fully syrchronous mode the change set needs to be sent before acquiring the row-level lock necessary for the operation on a single tuple. Execution of the transaction can only proceed after having received an agreement from the GCS that the current transaction is the next one to acquire the lock, see 2.6 for more details.

The eager replication level requires much fewer messages per transaction, ideally only one per transaction with few changes. However, Postgres-R must respect an upper boundary on the change set size, because some GCS enforce such a limitation. Additionally it might make sense to respect the network's MTU for performance reasons (preventing fragmentation on lower levels). During long running transactions Postgres-R thus sends partial change sets whenever reaching a certain maximum change set size. This also allows early application of changes from long running transactions on remote nodes and thus helps detecting conflicts as early as possible.

In both modes, the last change set of a transaction is sent together with a request for a commit ordering agreement just before committing the transaction. The backend must then await the ordering decision from the GCS so that conflicting transactions are committed in the same order on all nodes, which guarantees congruent commit or abort decisions.

2.4.2 Logging for Persistence

Postgres uses a Write Ahead Log (WAL) which serves two slightly different purposes. One is to make sure changes have made it to permanent storage before confirming a trans-

action to the client, the other is taking counter-measures against partial page writes. The WAL is very much bound to the internal representation of the data so it cannot be used to recover other nodes. As replication already provides additional safety against single node failures it is not necessary to perform logging on every node of the cluster. Instead it is beneficial for performance and manageability to allow separation of transaction logging from transaction processing[7]. Postgres-R thus provides an additional change set logging daemon which replaces ordinary transaction logging[8]. Before committing a transaction the backends must not only await an ordering decision, but also wait for confirmation from one or more of these change set logging services.

2.5 Application of Change Sets

Upon receiving a change set from another node, the coordinator checks if it belongs to a transaction which is already in progress. In that case it gets forwarded to the helper backend in charge of that remote transaction. If the change set is the first one of a new remote transaction, the coordinator must either assign it to an idle backend or cache it until a helper backend gets available.

The helper backends must then deserialize the changes from the set, lookup the tuples, check for conflicts and apply changes. Depending on the results of the conflict checks, the transaction can either be committed at the end of change set application or it needs to be aborted, because the GCS decided for a conflicting transaction to take precedence. This mechanism provides concurrent application of transactions from remote nodes and works directly from binary data, without transferring back to SQL.

2.5.1 Distributed Concurrency Control

Before committing an eagerly replicated remote transaction, the helper backend needs to await the ordering decision from the GCS, even if all conflict checks so far have been fine and the change sets have been applied. The coordinator turns the ordering decision into a list of transactions, which must terminate before the waiting transaction can commit. This ensures that all conflicting transactions get committed or aborted in the same order on all nodes, even if optimistic delivery and early application of change sets disregards this ordering. As long as a transaction is not committed, the changes can easily be rolled back to allow another transaction to apply its change sets and commit first, as required by the decided ordering.

While scanning for tuples to change, a helper backend checks against all tuples with matching primary key attributes - independent of their visibility. The change set carries the GID of the origin transaction for the tuple to be changed. This is compared against the origin transaction's GID of each tuple found, derived from the tuple's `xmin` which can be converted into a GID to compare. If the GIDs do not match the tuple can be skipped, because it belongs to a conflicting transaction[9]. If no tuple matches or if the matching tuple isn't visible, yet, the origin transaction may still be in progress on that node. In such a case the change for that tuple must be deferred and retried later.

Otherwise at most one tuple should match in primary key attributes and GID. In case the tuple may be updated the change can be applied. If the tuple has already been updated, there is a conflict between the current transaction and the one which updated the tuple (identified by the tuple's `xmax`. If the conflicting transaction has already committed it is clear that the GCS has decided to give it precedence, thus the current transaction must be aborted with a serialization error.

If the conflicting transaction is still in progress as well,

7. This has recently been confirmed by the Tashkent project [EDP06], which shows improved scalability by turning fsync off and instead perform logging within the middle-ware, very much like proposed here. Together with memory-aware load balancing as proposed in the subsequent paper [EDZ07], the authors claim to reach super-linear scalability with up to 16 nodes. Note however, that this mainly applies for databases which don't fit into a single node's memory, but which are smaller than the sum of memory available within the cluster.

8. It may still be useful to let WAL protect against torn pages due to partial writes (with `full_page_writes` enabled), because that allows a node to recover into a consistent state after a crash and then perform partial recovery, see 2.8.4.

9. The conflict between that transaction and the current one will be detected as soon as the old version of the same tuple is found. It's `xmax` is expected to be set to the conflicting transaction's id.

possible ordering decisions as well as the replication levels have to be taken into account. A synchronously replicated transaction always takes precedence over an eagerly replicated one, because the former ones may not be aborted with serialization failures. In case at least one of two eagerly replicated transactions has received its ordering decision, the backends can determine which of the two takes precedence. The other one must be deferred until the preceding one either commits or aborts. If neither of the transactions have received an ordering decision, the change has to be deferred as well.

2.5.2 Constraint Checking

Ordinary constraints are checked only once by the origin backend. All other nodes do not need to repeat such checking, because they operate on the same snapshot of data and apply the same changes. However, to ensure that foreign key constraints are respected, all applying nodes check for referenced tuples during change set application. To ensure the referenced tuple is valid when applying a remote transaction, the helper backend acquires the necessary shared lock during change set application. This eliminates the need to include primary key attribute information for referenced tuples in change sets, while still guaranteeing consistency with regard to conflicts on foreign key constraints.

2.5.3 Conflict resolution

In Postgres-R a conflict is normally solved by aborting a transaction with a serialization failure, either for a single command (synchronous) or just before committing (eager). This requires the application to retry the transaction to cope with these failures. For applications using the serializable isolation level such a retry loop should be common practice anyway, other applications might need to be adjusted to handle these situations.

To reduce the number of serialization failures, automatic conflict resolution function (CRF) may be used to automatically solve conflicts by merging changes for columns with meanings. However, these functions must satisfy associativity, because in case of a conflict Postgres-R executes them in different orders on different nodes expecting the result to be the same. The following trivial but useful CRF are provided: texttsum, texttproduct, textttmin and textttmax.

2.5.4 Read-Only Nodes

Another method to reduce or eliminate conflicts is to limit writing transactions to only few nodes or even only one, just like for master-slave replication. The GCS ideally adjusts to the situation of having only few writer nodes, so that the writing performance is independent of the read-only nodes.

Unlike most master-slave solutions, Postgres-R does not block write access to the database on any node, except on failure conditions. We consider the host base access control provided by Postgres to be sufficient for that purpose.

2.5.5 Deadlocks between Backends

Postgres features a deadlock detection mechanism which is capable of resolving deadlocks between multiple transactions by aborting one of them. When using only eagerly replicated transactions, such deadlocks can only occur between local transactions. But using synchronously replicated transactions might lead to deadlocks between local and remote transactions. In such a case the GCS needs to be queried to decide on a transaction to abort, so that all nodes congruently continue with the same set of transactions.

2.5.6 Transaction Rollback and Node Failures

Eagerly replicated transactions which have not sent a change set can be aborted and rolled back locally without any effect on remote nodes. However, all other transactions need to inform the remote nodes upon rollback so they can release the associated resources and locks.

2.6 Distributed Locking

Transactions using the eager replication level do not exchange any locking information. Instead they optimistically assume that conflicts with other transactions are rare and locks can be granted most of the time. The overall

cost for retrying conflicting transactions is expected to be lower than the cost for exchanging locking information between the nodes. Explicit table level locking is not replicated (ACCESS SHARE and ROW SHARE) or not allowed (ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE) for such transactions.

In synchronous replication level Postgres-R transfers locking information between the backends. This approach is known to scale with the number of concurrent transactions, which is not feasible for general purpose usage as shown by [GHOS96]. However, certain transactions or DDL operations are better performed with this kind of pessimistic locking, because conflicts are expected to occur frequently. The following two sections cover synchronous replication exclusively.

2.6.1 Locking agreements

Every locking operation requires a cluster wide agreement on which node[10] is next to acquire a certain lock. In case two or more nodes concurrently claim a lock, the GCS decides on an ordering in which to grant the lock to the requesting nodes. Unlocking only requires a reliable multi-cast message to be sent, to allow all other nodes to continue to grant the lock in the agreed ordering.

In case of a node failure, the remaining nodes release the locks when aborting the failed node's remote transactions. Also note that unlike two phase commit architectures, this locking scheme does not require confirmation from every node, but just an agreement from the GCS. Depending on the GCS and its configuration that means awaiting a message from a majority of nodes or even less.

2.6.2 Representation of Locks

In synchronous replication mode all operations acquiring a row level lock generate a normal change set, which includes the primary key attributes of the tuple to be locked. To avoid duplication of transmitted data, the request for a locking agreement only needs to refer to that change set (and should be sent together with it). As already mentioned in 2.4.1, the transaction processing backend must await the ordering decision and make sure to acquire the requested lock only after all preceding transactions have released the lock.

Table level and advisory locks are not covered by change sets. Instead these are serialized separately. They consist of a lock type and a GOI or advisory lock identifiers.

2.7 Replication of Meta Data

Postgres stores the data definition in so called system catalogs. These contain information about the structure of the data (number and names of databases, relations, attributes, views, etc.), configuration settings as well as statistics for the contained databases. Postgres-R does not replicate object identifiers (OID) because those are considered an implementation detail of the Postgres version and may differ between the nodes of a Postgres-R cluster.

However, all of the data structure information is replicated as well as all custom aggregates, operators, casts, conversions, enums, languages and functions. This includes the vast majority of system catalogs, which need to be replicated. However, note that system objects with OIDs lower than `FirstNormalObjectId` are provided by Postgres for every node indepentently and are thus not replicated. See also 2.8.2.

Entity dependency information (in `pg_depend` and `pg_shdepend`) is maintained separately on each node. So are the storage implementation relations for large objects and toasted data (`pg_largeobject` and `pg_toast_*`). Tablespaces and autovacuum settings are considered per-node configuration and are not replicated either.

2.7.1 Sequences

Unlike relational data, sequences do not underly an MVCC mechanism in Postgres. Instead they are updated immediately and cannot be rolled back. To achieve the same behav-

10. For simplicity, we only consider nodes here, but in fact the transactions on the nodes are requesting the locks, so the agreement is about an ordering between these transactions, which may (or may not) originate from different nodes.

ior on a distributed database system, Postgres-R thus performs a sequence increment via the GCS by requesting an agreement on which node gets the next value from a sequence, similar to locking. Setting a sequence to a specific value is treated similarly and the GCS must decide on an ordering of these operations for all nodes. As with relational data, performing sequence increment and assignment operations in the same order guarantees consistent sequence values.

As this requires a relatively expensive agreement round for every value from a sequence, Postgres-R also supports pre-assigning and caching of values from a sequence, very much like already implemented in Postgres on a per backend basis. This can greatly reduce the messaging overhead, but leads to "holes" or seemingly random sequence numbers. That happens as well during normal operation due to rolled-back transactions and most applications cope with it just fine. So this is a worthwhile optimization for the vast majority of use cases of sequences.

2.7.2 Data Definition Changes

While most changes of the data definition are irrelevant for the replication system, there are some exceptions which affect change set collection and application. When adding columns to a relation, the change set needs to hold one more attribute per tuple of the relation. If such a change is discovered by a concurrently running transaction during collection of changes, the current change set is finished and multi-casted, then a new one with more attributes per tuple is created for the relation in question. As attributes are referenced by their position in the relation, neither renaming nor dropping of a column has an effect on change set collection.

As Postgres-R can only replicate tables with primary keys it's not possible to drop the primary key of a relation without adding a new one within the same transaction. To prevent confusion between change sets addressing tuples based on different primary keys, all concurrently writing transactions which are using that primary key are blocked or aborted. Altering the primary key constraint of a relation is rare enough to not be worth further optimization.

Creation of new relations as well as dropping of existing ones is covered by ordinary Postgres relation locks. Thanks to the GOI abstraction, renaming of relations doesn't affect replication.

2.7.3 Two-Phase Commit

A frequent use case of Two-Phase Commit (2PC) is to implement replication in the application or in a middleware. While this is covered by Postgres-R, there are other use cases for 2PC and it is a standard SQL feature. Postgres-R thus supports preparation of a transaction on one node and accepting commit or abort commands from another node. Independent of the transaction replication level the final commit ordering request of every transaction thus includes a flag to mark a prepared transaction and possibly carries a `transaction_id` string from the PREPRAE statement. All nodes must then store the prepared transaction so that any one of them can later on commit or abort the prepared transaction.

2.7.4 User-Defined Functions (UDFs)

Many user defined functions are written in script languages which are portable to different architectures. However, the interpreter for the language itself must be available on all nodes to support the UDF. These interpreters as well as precompiled UDFs are linked into Postgres from dynamically loadable module. To ensure a replicated UDF can be executed on all nodes Postgres-R checks for the loadable module and only allows creation of a language or UDF if the required module is available on all nodes. Newly joining nodes must be checked for the required modules and can only participate in the replication group if they provide all necessary modules.

2.7.5 Per Node Configuration and Maintenance

Postgres-R does not replicate per node settings and configuration as those may differ between the nodes of a distributed

database for good reasons. This also applies to maintenance tasks such as those performed by VACUUM, REINDEX and CHECKPOINT, which should in most cases not be done in parallel for performance reasons, but performed on one node after another, if all nodes require the same maintenance steps at all. Note that the host based access control is considered to be part of the per node configuration as well, even if that relies on roles of the globally shared database.

2.7.6 Event Notification

Postgres provides a non-standard event notification mechanism via the LISTEN and NOTIFY commands. To support notifications across nodes Postgres-R sends a reliable multi-cast message for every notification, so that remote listeners get notified as well.

2.8 Initialization and Recovery

After adding a node to a Postgres-R cluster, it needs to be initialized with data from the cluster. For simplicity, this process is kept similar to recovering a node after a crash or after an intentional shutdown: first a consistent copy of a single node's system catalogs is transferred, then the schema on the recovering node (the subscriber) adapts its local system catalogs accordingly and initiates the parallelized data transfer afterwards.

2.8.1 Relation Data Transfer

To transfer the contents of a relation via a network, it must be serialized and split into messages. As nodes may fail any time, the recovery process must be able to continue retrieving data from another node. A recovery provider[11] in Postgres-R thus sends chunks of serialized tuple data starting from a given primary key limit, ordered by ascending primary key attributes and up to a given maximum chunk size. This allows nodes to be initialized or recovered by requesting chunks from any other node and simply query another one if the actual recovery provider fails. The serialization of tuple data for recovery messages uses the same format as used for inserts in change sets.

To ensure good parallelism (with regard to network latency as well as storage requests) multiple recovery lanes can run in parallel. This is achieved by limiting each recovery lane to a certain range of tuples by hashing the primary key attributes and limiting to a certain modulo, i.e. `hash(primary key attributes) mod N = X`, where `N` is the number of lanes and `X` the current lane to be limited to.

2.8.2 Schema Adaption

Before transferring relation data, the relations themselves need to be created. Postgres maintains all meta information about relations in so called system catalogs, which resemble ordinary tables. To unify schema adaption, a recovering node uses the relation data transfer method outlined above to retrieve a copy of another node's system catalogs. It can then compare its own system catalogs to the copy and adapt its own meta-data accordingly. This separation into transmission and adaption steps simplifies the adaption, because it doesn't need to care about network issues anymore. Note that supporting varying Postgres major versions requires different adaption methods. Upon changes of the system catalogs or upon failure of the recovery provider, the schema catalog transmission is simply restarted to ensure a consistent copy.

2.8.3 Change set application during recovery

To avoid long running transactions on the recovery provider and to reduce the amount of known deprecated data to be transmitted, Postgres-R uses MVCC during recovery as well. In contrast to transmitting a complete snapshot and later applying all change sets collected in between as proposed for example by [LK08], this algorithm avoids having to send lots of known deprecated data. It also relieves the recovering node from having to cache change sets and in-

11. We talk of a recovery provider and a recovery subscriber only during recovery. Every active node can provide recovery information for initializing or recovering nodes. After the subscriber has completed recovery, it becomes an active node, which may then act as a recovery provider for other nodes as well.

stead allows it to start operating immediately after the data transmission is done. This is achieved by merging these two stages of recovery into one by continuously applying change sets during data transmission.

Each recovery packet is generated based on a current snapshot, but must carry snapshot information with it (the GID of the latest committed transaction). During recovery, the subscriber retrieves and already applies changes to already recovered tuples, but discards changes regarding tuples which are not recovered, yet.

To make sure the recovery process provides these changes in one of the forthcoming recovery packets, application of a change set must be deferred somewhat. Due to the recovery process going through a relation by incrementing primary key attribute values, there are two limits to keep track of: the primary key attributes values which mark the progress of recovery within a relation and the GID of the latest committed transaction covered by the recovery packets. Changes from transactions which committed after that are not included in the recovery packet. During the recovery process, both of these limits possibly advance with every replication packet received. The change set for a certain transaction may well arrive before the first recovery packet which includes changes from that transaction. Postgres-R must ensure that no change falls through because its original tuple happens to be in between both limits (slightly above the primary key limit, but the recovery packet covering it didn't advance up to that transaction's snapshot and thus does not include the change). This can be achieved by deferring application of change sets until the recovery process is known to have passed that transaction and will deliver only newer data from that point on. This guarantees that every change since the start of the recovery process is either included in a recovery packet or applied from a change set.

The backend which does the application of change sets compares against the current primary key limit of the recovery process. If a change targets a tuple above that limit it can be discarded safely, because later recovery packets will deliver the changed tuple. Otherwise it needs to be applied or recovery has already provided the same or newer data in which case the change can also be discarded.

2.8.4 Partial Node Recovery

A node which has been shut down cleanly or which can recover into a consistent state after a crash by other means (see 2.4.2) does not need to perform full data recovery. Instead it can do partial recovery from the last checkpoint on. Much like regular recovery from WAL a Postgres-R node recovers by receiving all the change sets starting from the consistent state's checkpoint on and applying them in order, before continuing with regular change set application and serving client requests. This technique is called partial node recovery and requires knowledge of the latest applied transaction's GID as well as the local to global transaction id mapping[12]. Each node of a Postgres-R cluster thus saves these upon a clean shutdown as well as with every checkpoint. If the node cannot guarantee to recover into a consistent state it must perform a full recovery.

2.8.5 Full Cluster Shutdown and Restart

To perform a full cluster shutdown, Postgres-R needs to ensure that at least a majority of the nodes have terminated and successfully written all its changes and the above mentioned transaction id mapping to disk. Only after that, the connection to the GCS can be closed.

During a full cluster restart the replication group is kept inactive until at least a majority of nodes are available again. As soon as that's the case, the group becomes active and Postgres-R decides on a known consistent state from where to perform partial recovery. This also works for recovering after a full cluster crash, but requires at least one node to be able to recover into a consistent state.

2.9 Testing

Testing of a distributed system poses some difficulties because of the many components involved. Thanks to the del-

12. That mapping of local transaction ids to global ids (GIDs) is required for conflict detection during application of change sets. During partial recovery as well as during normal operation it gets extended by newly applied change sets and transactions.

egation of communication issues to a GCS, Postgres-R is very flexible and can be set up in different ways for different needs, which also benefits testing.

2.9.1 Regression Testing

For regression testing we assume a correctly working GCS and concentrate on Postgres-R itself. A simple emulated group communication system (EGCS) runs on a single host and emulates all the services of a real GCS while not having to providing the same availability guarantees, thus being simple and offering reproducible results.

Postgres-R allows to run multiple instances on the same node, so regression testing does not require a full cluster. To provide quick test results the initialization and startup of the test databases is automated and controlled by a test driver which guards and controls all database processes and the EGCS.

Besides emulating the services of a GCS the emulated group communication system is also capable of simulating the results of various network anomalies and network failures so as to be able to test failure resilience of Postgres-R.

To test the GCS interfaces and Postgres-R in combination with the GCS it turned out to be beneficial to use virtual machines with a simulated network. Automating such a setup is harder due to having to control full systems and a network instead of only multiple processes on a single system.

2.9.2 Performance Testing

Simple performance testing can be done on such a virtual cluster by varying various parameters of the simulated network (like latency, throughput, drop rate, etc.). As all virtual machines normally share the same physical resources, such tests are only meaningful for exclusively network constrained systems, which should ideally not be the case for Postgres-R. For more accurate performance measures the storage subsystem as well as the CPU performance and memory throughput needs to be taken into account as well.

2.10 Future Extensions

While Postgres-R is limited according to the design goals, the following extensions are kept in mind and should be possible on the basis of Postgres-R, due to replicating independently of the storage format and close to the database system's internals.

2.10.1 Partial Replication and Data Partitioning

In its current design, Postgres-R is intended to replicate all relations to all nodes in the system. This increases availability and allows load balancing of read-only queries. However, the overall load for writing transactions increases with every node.

To reduce that load again when going beyond 3 or more nodes it might make sense to partition the data between the nodes. This allows for a compromise between availability (due to having data duplicated on multiple nodes) and performance (due to balancing the write load as well). Postgres already implements partitioning between table spaces by splitting the data of a single relation into multiple relations which can then reside on different table spaces. The data is combined and made available like a single relation through using inheritance or custom rules.

Postgres-R could use a similar approach and spread tables between nodes. However, to be able to combine the data to be queried, the nodes would have to request data from remote nodes (remote querying). For queries which involve multiple relations, spread differently on different nodes this easily results in lots of possible plans for how to satisfy the query.

2.10.2 Lazy Replication

As proposed, Postgres-R provides two timing levels for replicated transactions: synchronous and eager replication. Both increase commit latency compared to a single node system and are thus not appropriate for all situations. Reducing the network latency could be done by deferring the consensus round for change sets and by collecting multiple change sets into a single multi-cast message. However,

this possibly results in transactions conflicting after being committed, which violates consistency. To reach a common consistent state again the conflict needs to be resolved somehow. Often this is only possible by late aborting one of the transactions, which then violates durability.

Besides the conflict detection and resolution, this would also require Postgres-R to maintain an intermediate transaction state between committed and uncommitted. Such intermediate state transactions are already visible on the node they have been processed, but can still be aborted by later reconciliation with other nodes. Also note that this creates a dependency tree of transactions which modify locally committed data, so that late aborting one transaction possibly means having to abort several subsequent transactions as well.

Lazy replication does not induce network dependent commit latency and therefore provides good performance even via high-latency networks. Even disconnected operation could be considered. However, it is even more of a relaxation from ACID properties with rather severe impacts on the application.

3 Conclusion

Developing an update-everywhere database system on a cluster of shared-nothing nodes is a challenge, but can increase availability by an order of magnitude compared to single node database systems. Postgres-R breaks the complexity of multi-master replication into pieces by using an abstraction called group communication. The additional coordinator and worker processes are similar to other Postgres processes and integrate well into the existing architecture. The ability to separate transaction logging enhances flexibility and improves performance by reducing overall disk I/O. Postgres' MVCC is not only used for conflict detection during normal operation, but also to provide flexibility during recovery phase. A fully synchronous replication level allows DDL to be replicated and offers a fall-back for transactions requiring user level or special row locking, while the eager replication level offers optimal performance for update-everywhere replication.

References

[CT96] Chandra and Toueg. Unreliable failure detectors for reliable distributed systems. *JACM: Journal of the ACM*, 43, 1996.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372--421, December 2004.

[EDP06] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *In EuroSys 2006: Proceedings of the 1st European Conference on Computer Systems*, pages 117--130. ACM Press, 2006.

[EDZ07] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *In EuroSys 2007: Proceedings of the 2nd European Conference on Computer Systems*, pages 399--412, 2007.

[GHOS96] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 International Conference on Management of Data*, pages 173--182, Montreal, Canada, 1996. ACM-SIGMOD.

[KA00] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the* 26^{th} *International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, 2000.

[LK08] WeiBin Liang and Bettina Kemme. Online recovery in cluster databases. In Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu, editors, *EDBT*, volume 261 of *ACM International Conference Proceeding Series*, pages 121--132. ACM, 2008.

[WK05] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422--433. IEEE Computer Society, 2005.