High Level Design for Logical Replication in Postgres

Andres Freund, 2ndQuadrant Ltd. <andres@2ndQuadrant.com>

Table of Contents

1. Introduction 1
  1.1. Previous discussions 1
  1.2. Changes from v1 2
2. Existing approaches to replication in Postgres 2
  2.1. Trigger based Replication 2
  2.2. Recovery based Replication 3
3. Goals 4
4. New Architecture 5
  4.1. Overview 5
  4.2. Schematics 5
  4.3. WAL enrichement 7
  4.4. WAL parsing & decoding 7
  4.5. TX reassembly 9
  4.6. Snapshot building 10
  4.7. Output Plugin 11
  4.8. Setup of replication nodes 12
  4.9. Disadvantages of the approach 13
  4.10. Unfinished/Undecided issues 13

1. Introduction

This document aims to first explain why we think postgres needs another replication solution and what that solution needs to offer in our opinion. Then it sketches out our proposed implementation.

In contrast to an earlier version of the design document which talked about the implementation of four parts of replication solutions:

1. Source data generation
2. Transportation of that data
3. Applying the changes
4. Conflict resolution

this version only plans to talk about the first part in detail as it is an independent and complex part usable for a wide range of use cases which we want to get included into postgres in a first step.

1.1. Previous discussions

There are two rather large threads discussing several parts of the initial prototype and proposed architecture:

- Logical Replication/BDR prototype and architecture [http://archives.postgresql.org/message-id/201206131327.24092.andres@2ndquadrant.com]
- Catalog/Metadata consistency during changeset extraction from WAL [http://archives.postgresql.org/message-id/201206211341.25322.andres@2ndquadrant.com]

Those discussions lead to some fundamental design changes which are presented in this document.

1.2. Changes from v1

- At least a partial decoding step required/possible on the source system
- No intermediate ("schema only") instances required
- DDL handling, without event triggers
- A very simple text conversion is provided for debugging/demo purposes
- Smaller scope

2. Existing approaches to replication in Postgres

If any currently used approach to replication can be made to support every use-case/feature we need, it likely is not a good idea to implement something different. Currently three basic approaches are in use in/around postgres today:

1. Trigger based
2. Recovery based/Physical [1]
3. Statement based

Statement based replication has obvious and known problems with consistency and correctness making it hard to use in the general case so we will not further discuss it here.

Lets have a look at the advantages/disadvantages of the other approaches:

2.1. Trigger based Replication

This variant has a multitude of significant advantages:

- implementable in userspace
- easy to customize

[1]Often referred to by terms like Hot Standby, Streaming Replication, Point In Time Recovery

- just about everything can be made configurable
- cross version support
- cross architecture support
- can feed into systems other than postgres
- no overhead from writes to non-replicated tables
- writable standbys
- mature solutions
- multimaster implementations possible & existing

But also a number of disadvantages, some of them very hard to solve:

- essentially duplicates the amount of writes (or even more!)
- synchronous replication hard or impossible to implement
- noticeable CPU overhead
  - trigger functions
  - text conversion of data
- complex parts implemented in several solutions
- not in core

Especially the higher amount of writes might seem easy to solve at a first glance but a solution not using a normal transactional table for its log/queue has to solve a lot of problems. The major ones are:

- crash safety, restartability & spilling to disk
- consistency with the commit status of transactions
- only a minimal amount of synchronous work should be done inside individual transactions

In our opinion those problems are restricting progress/wider distribution of these class of solutions. It is our aim though that existing solutions in this space - most prominently slony and londiste - can benefit from the work we are doing & planning to do by incorporating at least parts of the changeset generation infrastructure.

2.2. Recovery based Replication

This type of solution, being built into postgres and of increasing popularity, has and will have its use cases and we do not aim to replace but to complement it. We plan to reuse some of the infrastructure and to make it possible to mix both modes of replication

Advantages:

- builtin
- built on existing infrastructure from crash recovery
- efficient
    - minimal CPU, memory overhead on primary
    - low amount of additional writes
- synchronous operation mode
- low maintenance once setup
- handles DDL

Disadvantages:

- standbys are read only
- no cross version support
- no cross architecture support
- no replication into foreign systems
- hard to customize
- not configurable on the level of database, tables, …

3. Goals

As seen in the previous short survey of the two major interesting classes of replication solution there is a significant gap between those. Our aim is to make it smaller.

We aim for:

- in core
- low CPU overhead
- low storage overhead
- asynchronous, optionally synchronous operation modes
- robust
- modular

- basis for other technologies (sharding, replication into other DBMS’s, …)
- basis for a multi-master solution
- make the implementation as unintrusive as possible, but not more

4. New Architecture

4.1. Overview

Our proposal is to reuse the basic principle of WAL based replication, namely reusing data that already needs to be written for another purpose, and extend it to allow most, but not all, the flexibility of trigger based solutions. We want to do that by decoding the WAL back into a non-physical form.

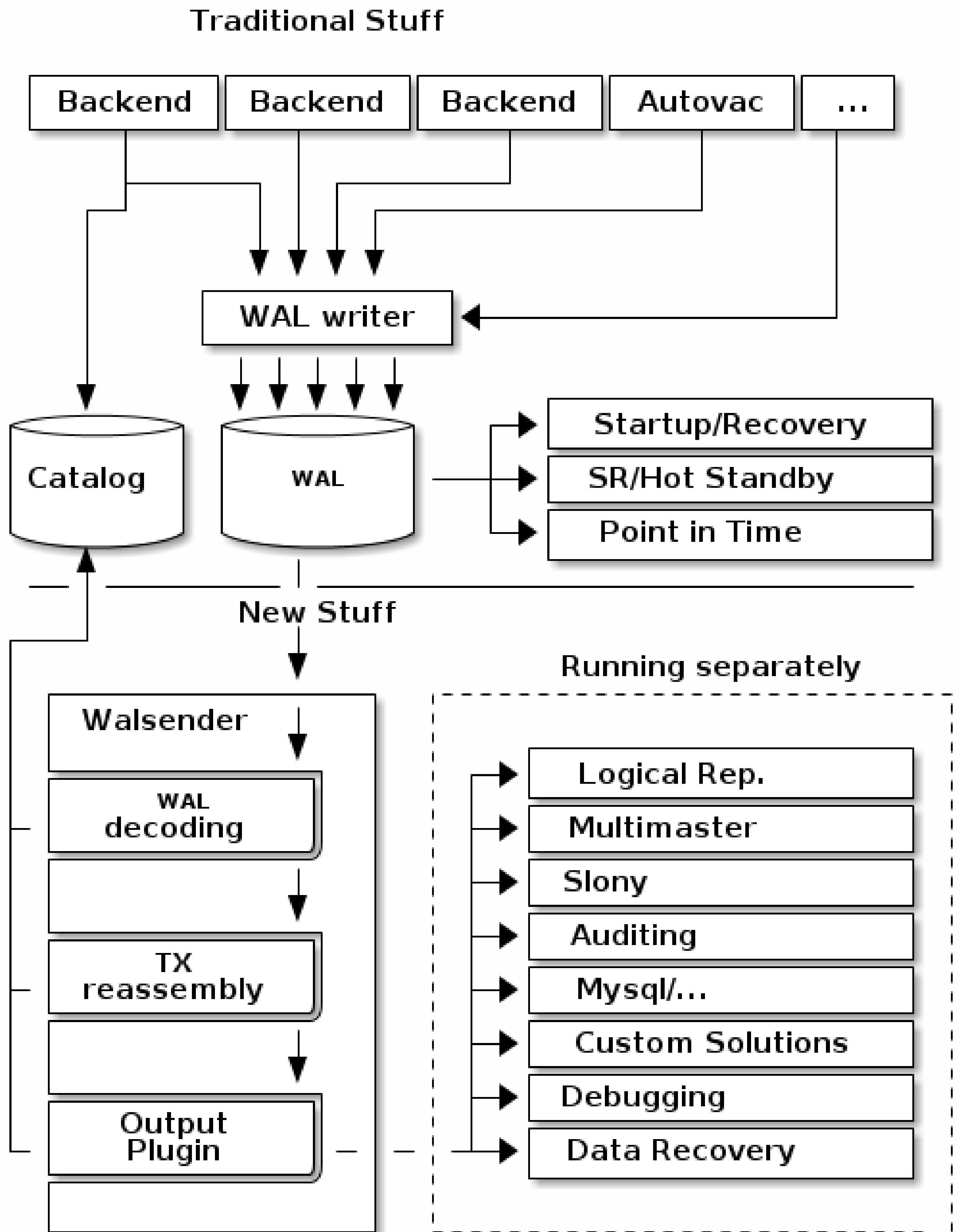
To get the flexibility we and others want we propose that the last step of changeset generation, transforming it into a format that can be used by the replication consumer, is done in an extensible manner. In the schema the part that does that is described as *Output Plugin*. To keep the amount of duplication between different plugins as low as possible the plugin should only do a a very limited amount of work.

The following paragraphs contain reasoning for the individual design decisions made and their highlevel design.

4.2. Schematics

The basic proposed architecture for changeset extraction is presented in the following diagram. The first part should look familiar to anyone knowing postgres' architecture. The second is where most of the new magic happens.

Figure 1. Architecture Schema



4.3. WAL enrichement

To be able to decode individual WAL records at the very minimal they need to contain enough information to reconstruct what has happened to which row. The action is already encoded in the WAL records header in most of the cases.

As an example of missing data, the WAL record emitted when a row gets deleted, only contains its physical location. At the very least we need a way to identify the deleted row: in a relational database the minimal amount of data that does that should be the primary key [2].

We propose that for now it is enough to extend the relevant WAL record with additional data when the newly introduced *WAL_level = logical* is set.

Previously it has been argued on the hackers mailing list that a generic *WAL record annotation* mechanism might be a good thing. That mechanism would allow to attach arbitrary data to individual wal records making it easier to extend postgres to support something like what we propose.. While we don't oppose that idea we think it is largely orthogonal issue to this proposal as a whole because the format of a WAL records is version dependent by nature and the necessary changes for our easy way are small, so not much effort is lost.

A full annotation capability is a complex endeavour on its own as the parts of the code generating the relevant WAL records has somewhat complex requirements and cannot easily be configured from the outside.

Currently this is contained in the Log enough data into the wal to reconstruct logical changes from it [http://archives.postgresql.org/message-id/1347669575-14371-6-git-send-email-andres@2ndquadrant.com] patch.

4.4. WAL parsing & decoding

The main complexity when reading the WAL as stored on disk is that the format is somewhat complex and the existing parser is too deeply integrated in the recovery system to be directly reusable. Once a reusable parser exists decoding the binary data into individual WAL records is a small problem.

Currently two competing proposals for this module exist, each having its own merits. In the grand scheme of this proposal it is irrelevant which one gets picked as long as the functionality gets integrated.

The mailing list post Add support for a generic wal reading facility dubbed XLogReader [http:http://archives.postgresql.org/message-id/1347669575-14371-3-git-send-email-andres@2ndquadrant.com] contains both competing patches and discussion around which one is preferable.

Once the WAL has been decoded into individual records two major issues exist:

1. records from different transactions and even individual user level actions are intermingled
2. the data attached to records cannot be interpreted on its own, it is only meaningful with a lot of required information (including table, columns, types and more)

[2] Yes, there are use cases where the whole row is needed, or where no primary key can be found

The solution to the first issue is described in the next section: Section 4.5, “TX reassembly”

The second problem is probably the reason why no mature solution to reuse the WAL for logical changeset generation exists today. See the Section 4.6, “Snapshot building” paragraph for some details.

As decoding, Transaction reassembly and Snapshot building are interdependent they currently are implemented in the same patch: Introduce wal decoding via catalog timetravel [http://archives.postgresql.org/message-id/1347669575-14371-8-git-send-email-andres@2ndquadrant.com]

That patch also includes a small demonstration that the approach works in the presence of DDL:

Decoding example.

```
/* just so we keep a sensible xmin horizon */
ROLLBACK PREPARED 'f';
BEGIN;
CREATE TABLE keepalive();
PREPARE TRANSACTION 'f';

DROP TABLE IF EXISTS replication_example;

SELECT pg_current_xlog_insert_location();
CHECKPOINT;
CREATE TABLE replication_example(id SERIAL PRIMARY KEY, somedata int, text
varchar(120));
begin;
INSERT INTO replication_example(somedata, text) VALUES (1, 1);
INSERT INTO replication_example(somedata, text) VALUES (1, 2);
commit;


ALTER TABLE replication_example ADD COLUMN bar int;

INSERT INTO replication_example(somedata, text, bar) VALUES (2, 1, 4);

BEGIN;
INSERT INTO replication_example(somedata, text, bar) VALUES (2, 2, 4);
INSERT INTO replication_example(somedata, text, bar) VALUES (2, 3, 4);
INSERT INTO replication_example(somedata, text, bar) VALUES (2, 4, NULL);
COMMIT;

/* slightly more complex schema change, still no table rewrite */
ALTER TABLE replication_example DROP COLUMN bar;
INSERT INTO replication_example(somedata, text) VALUES (3, 1);

BEGIN;
INSERT INTO replication_example(somedata, text) VALUES (3, 2);
INSERT INTO replication_example(somedata, text) VALUES (3, 3);
commit;

ALTER TABLE replication_example RENAME COLUMN text TO somenum;

INSERT INTO replication_example(somedata, somenum) VALUES (4, 1);

/* complex schema change, changing types of existing column, rewriting the table */
ALTER TABLE replication_example ALTER COLUMN somenum TYPE int4 USING
(somenum::int4);
```

```
INSERT INTO replication_example(somedata, somenum) VALUES (5, 1);

SELECT pg_current_xlog_insert_location();

/* now decode what has been written to the WAL during that time */

SELECT decode_xlog('0/1893D78', '0/18BE398');

WARNING:  BEGIN
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  tuple is: id[int4]:1 somedata[int4]:1 text[varchar]:1
WARNING:  tuple is: id[int4]:2 somedata[int4]:1 text[varchar]:2
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  tuple is: id[int4]:3 somedata[int4]:2 text[varchar]:1 bar[int4]:4
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  tuple is: id[int4]:4 somedata[int4]:2 text[varchar]:2 bar[int4]:4
WARNING:  tuple is: id[int4]:5 somedata[int4]:2 text[varchar]:3 bar[int4]:4
WARNING:  tuple is: id[int4]:6 somedata[int4]:2 text[varchar]:4 bar[int4]:
(null)
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  tuple is: id[int4]:7 somedata[int4]:3 text[varchar]:1
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  tuple is: id[int4]:8 somedata[int4]:3 text[varchar]:2
WARNING:  tuple is: id[int4]:9 somedata[int4]:3 text[varchar]:3
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  tuple is: id[int4]:10 somedata[int4]:4 somenum[varchar]:1
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  COMMIT
WARNING:  BEGIN
WARNING:  tuple is: id[int4]:11 somedata[int4]:5 somenum[int4]:1
WARNING:  COMMIT
```

4.5. TX reassembly

In order to make usage of the decoded stream easy we want to present the user level code with a correctly ordered image of individual transactions at once because otherwise every user will have to reassemble transactions themselves.

Transaction reassembly needs to solve several problems:

1. changes inside a transaction can be interspersed with other transactions

2. a top level transaction only knows which subtransactions belong to it when it reads the commit record

3. individual user level actions can be smeared over multiple records (TOAST)

Our proposed module solves 1) and 2) by building individual streams of records split by xid. While not fully implemented yet we plan to spill those individual xid streams to disk after a certain amount of memory is used. This can be implemented without any change in the external interface.

As all the individual streams are already sorted by LSN by definition - we build them from the wal in a FIFO manner, and the position in the WAL is the definition of the LSN [3] - the individual changes can be merged efficiently by a k-way merge (without sorting!) by keeping the individual streams in a binary heap.

To manipulate the binary heap a generic implementation is proposed. Several independent implementations of binary heaps already exist in the postgres code, but none of them is generic. The patch is available at Add minimal binary heap implementation [http://archives.postgresql.org/message-id/1347669575-14371-2-git-send-email-andres@2ndquadrant.com].

Note

The reassembly component was previously coined ApplyCache because it was proposed to run on replication consumers just before applying changes. This is not the case anymore.

It is still called that way in the source of the patch recently submitted.

4.6. Snapshot building

To decode the contents of wal records describing data changes we need to decode and transform their contents. A single tuple is stored in a data structure called HeapTuple. As stored on disk that structure doesn’t contain any information about the format of its contents.

The basic problem is twofold:

1. The wal records only contain the relfilenode not the relation oid of a table

2. The relfilenode changes when an action performing a full table rewrite is performed

3. To interpret a HeapTuple correctly the exact schema definition from back when the wal record was inserted into the wal stream needs to be available

We chose to implement timetraveling access to the system catalog using postgres' MVCC nature & implementation because of the following advantages:

- low amount of additional data in wal

- genericity

- similarity of implementation to Hot Standby, quite a bit of the infrastructure is reusable

- all kinds of DDL can be handled in reliable manner

[3] the LSN is just the byte position int the WAL stream

- extensibility to user defined catalog like tables

Timetravel access to the catalog means that we are able to look at the catalog just as it looked when changes were generated. That allows us to get the correct information about the contents of the aforementioned HeapTuple's so we can decode them reliably.

Other solutions we thought that fell through: * catalog only proxy instances that apply schema changes exactly to the point were decoding using "old fashioned" wal replay * do the decoding on a 2nd machine, replicating all DDL exactly, rely on the catalog there * do not allow DDL at all * always add enough data into the WAL to allow decoding * build a fully versioned catalog

The email thread available under Catalog/Metadata consistency during changeset extraction from WAL [http://archives.postgresql.org/message-id/201206211341.25322.andres@2ndquadrant.com] contains some details, advantages and disadvantages about the different possible implementations.

How we build snapshots is somewhat intricate and complicated and seems to be out of scope for this document. We will provide a second document discussing the implementation in detail. Let's just assume it is possible from here on.

Note

Some details are already available in comments inside *src/backend/replication/logical/snapbuild.{c,h}*.

4.7. Output Plugin

As already mentioned previously our aim is to make the implementation of output plugins as simple and non-redundant as possible as we expect several different ones with different use cases to emerge quickly. See Figure 1, "Architecture Schema" for a list of possible output plugins that we think might emerge.

Although we for now only plan to tackle logical replication and based on that a multi-master implementation in the near future we definitely aim to provide all use-cases with something easily useable!

To decode and translate local transaction an output plugin needs to be able to transform transactions as a whole so it can apply them as a meaningful transaction at the other side.

What we do to provide that is, that very time we find a transaction commit and thus have completed reassembling the transaction we start to provide the individual changes to the output plugin. It currently only has to fill out 3 callbacks:

Callback	Passed Parameters	Called per TX	Use
begin	xid	once	Begin of a reassembled transaction
change	xid, subxid, change, mvcc snapshot	every change	Gets passed every change so it can transform it to the target format

Callback	Passed Parameters	Called per TX	Use
commit	xid	once	End of a reassembled transaction

During each of those callback an appropriate timetraveling SnapshotNow snapshot is setup so the callbacks can perform all read-only catalog accesses they need, including using the sys/rel/catcache. For obvious reasons only read access is allowed.

The snapshot guarantees that the result of lookups are be the same as they were/would have been when the change was originally created.

Additionally they get passed a MVCC snapshot, to e.g. run sql queries on catalogs or similar.

Important

At the moment none of these snapshots can be used to access normal user tables. Adding additional tables to the allowed set is easy implementation wise, but every transaction changing such tables incurs a noticeably higher overhead.

For now transactions won't be decoded/output in parallel. There are ideas to improve on this, but we don't think the complexity is appropriate for the first release of this feature.

This is an adoption barrier for databases where large amounts of data get loaded/written in one transaction.

4.8. Setup of replication nodes

When setting up a new standby/consumer of a primary some problem exist independent of the implementation of the consumer. The gist of the problem is that when making a base backup and starting to stream all changes since that point transactions that were running during all this cannot be included:

- Transaction that have not committed before starting to dump a database are invisible to the dumping process
- Transactions that began before the point from which on the WAL is being decoded are incomplete and cannot be replayed

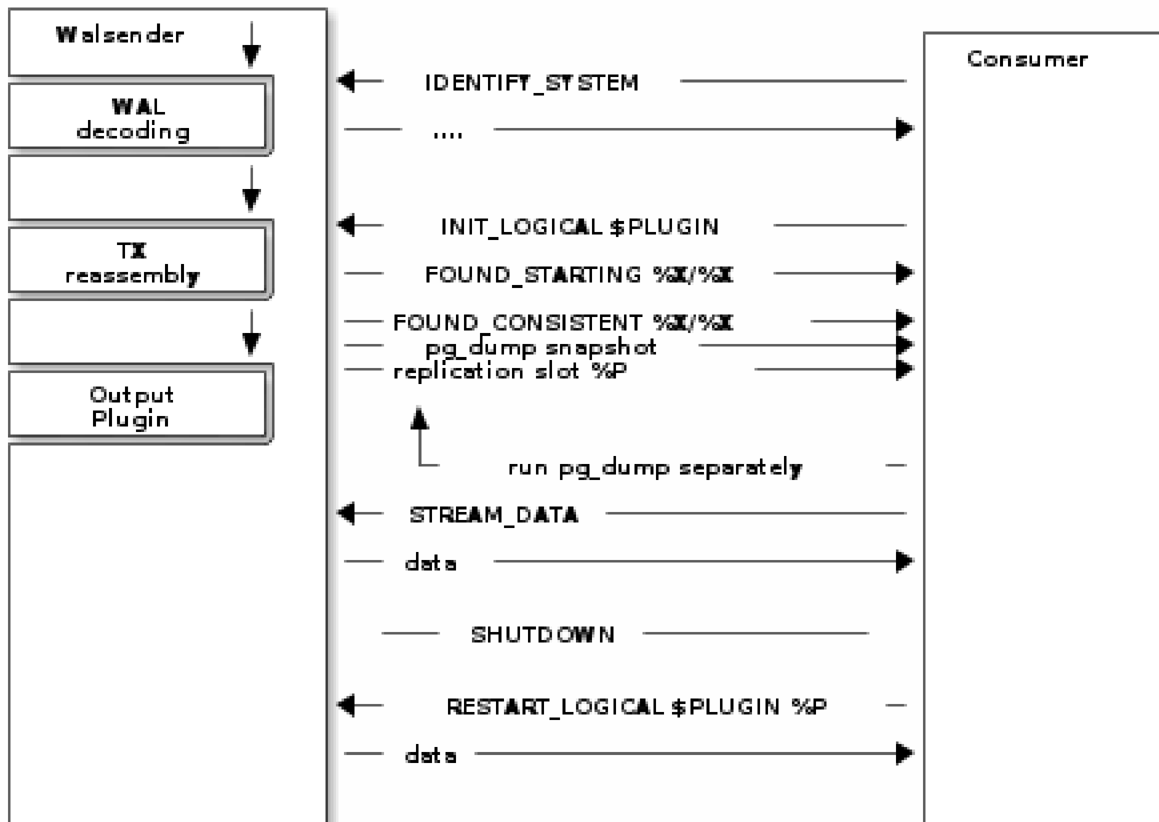
Our proposal for a solution to this is to detect points in the WAL stream where we can provide:

1. A snapshot exported similarly to pg_export_snapshot() [4] that can be imported with `SET TRANSACTION SNAPSHOT` [5]
2. A stream of changes that will include the complete data of all transactions seen as running by the snapshot generated in 1)

See the diagram.

[4] http://www.postgresql.org/docs/devel/static/functions-admin.html#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION
[5] http://www.postgresql.org/docs/devel/static/sql-set-transaction.html

Figure 2. Control flow during setup of a new node



4.9. Disadvantages of the approach

- somewhat intricate code for snapshot timetravel
- output plugins/walsenders need to work per database as they access the catalog
- when sending to multiple standbys some work is done multiple times
- decoding/applying multiple transactions in parallel is hard

4.10. Unfinished/Undecided issues

- declaration of user “catalog” tables (e.g. userspace enums)
- finishing different parts of the implementation
- spill to disk during transaction reassembly
- mixed catalog/data transactions
- snapshot refcounting
- snapshot exporting

- snapshot serialization