About PostgreSQL 9.5

PGCon Japan 2015
27th November 2015, Tokyo
Michael Paquier / VMware

Summary

- SQL features
- Management and performance
- WAL, Standbys & HA

UPSERT

- Replace constraint error with other actions
- New INSERT clauses
  - ON CONFLICT .. DO NOTHING
  - ON CONFLICT .. DO UPDATE
- Row-based approach
- Works with VALUES, SELECT statement
- More performant than plpgsql. Etc. equivalents
- Keyword EXCLUDED to reference former data

UPSERT (1) – DO NOTHING

```
CREATE TABLE tab (a int PRIMARY KEY, b text);
INSERT INTO tab VALUES (1, 'old'), (3, 'old');

INSERT INTO tab
    SELECT a, 'inserted'
    FROM generate_series(1, 4) AS a
    ON CONFLICT DO NOTHING;

SELECT * FROM tab;
 a |    b
---+----------
 1 | old
 3 | old
 2 | inserted
 4 | inserted
(4 rows)
```

UPSERT (2) – DO UPDATE

```
INSERT INTO tab
    SELECT a, 'inserted'
    FROM generate_series(1, 4) AS a
    ON CONFLICT (a) DO UPDATE SET b = 'upserted';

SELECT * FROM tab;
 a |     b
---+-------------
 1 | upserted
 2 | inserted
 3 | upserted
 4 | inserted
(4 rows)
```

UPSERT (3) - EXCLUDED

```
INSERT INTO tab VALUES (1, 'excluded');
ERROR:  23505: duplicate key value violates
                unique constraint "tab_pkey"
DETAIL:  Key (a)=(1) already exists.

INSERT INTO tab VALUES (1, 'excluded')
    ON CONFLICT (a) DO UPDATE SET b = EXCLUDED.b;

=# SELECT * FROM tab WHERE a = 1;
 a |     b
---+-----------
 1 | excluded
(1 row)
```

BRIN indexes

- **BRIN** = **B**lock **R**ange **IN**dex
- Minimum/Maximum values for a range of blocks
- Mix of sequential scan and index scan
- Good for clustered (ordered!) data

```
CREATE TABLE tabdata AS
SELECT generate_series (1,10000000) AS a;
CREATE INDEX tabdata_btree ON tabdata(a);
CREATE INDEX tabdata_brin ON tabdata USING brin(a);
SELECT relname, pg_size_pretty(pg_relation_size(oid))
       FROM pg_class WHERE relname LIKE 'tabdata_%';
    relname    | pg_size_pretty
---------------+----------------
 tabdata_brin  | 24 kB
 tabdata_btree | 214 MB
(2 rows)
```

GROUPING SETS, ROLLUP, CUBE

- GROUPING SETS
  - multiple GROUP BY in single query
  - Equivalent with UNION ALL with NULL columns
- CUBE => GROUPING SETS (list of subsets)

```
CUBE (x1,x2,x3) =>  GROUPING SETS ((x1,x2,x3),
                                   (x1,x2   ),
                                   (x1,   x3),
                                   (x1      ),
                                   (   x2,x3), ...
```

- ROLLUP => GROUPING SETS

```
ROLLUP (x1,x2,x3) =>  GROUPING SETS ((x1,x2,x3),
                                     (x1,x2   ),
                                     (x1      ))
```

JSONB additions

- Concatenation: jsonb || jsonb

```
=> SELECT '{"a1":"v1"}'::jsonb ||
          '{"a2":"v2"}'::jsonb AS field;
          field
----------------------------
 {"a1": "v1", "a2": "v2"}
(1 row)
```

- Substraction: jsonb – {text, int, text[]}

```
=# SELECT '{"a1":"v1","a2":"v2"}'::jsonb -
          'a1' AS field;
     field
--------------
 {"a2": "v2"}
(1 row)
```

- jsonb_pretty, jsonb_replace, jsonb_set

Row-level security (RLS)

- Tuple-level control of user visibility
  - RLS = horizontal
  - existing REMOTE/GRANT = vertical
- Parametrization
  - GUC parameter row_security to 'on' (default)
  - ALTER TABLE .. ENABLE ROW LEVEL SECURITY
  - CREATE POLICY on a table
- USING clause for policy granularity (user name, timestamp, function input, etc.)

RLS – Example setup

```
CREATE TABLE salaries (id int, name text, salary int);
ALTER TABLE salaries ENABLE ROW LEVEL SECURITY;
INSERT INTO salaries VALUES (1, 'salaryman1', 1500000);
INSERT INTO salaries VALUES (2, 'salaryman2', 150);
GRANT ALL ON TABLE salaries TO PUBLIC;

CREATE USER salaryman1;
CREATE USER salaryman2;

CREATE POLICY salary_control ON salaries
    FOR ALL TO PUBLIC USING (name = current_user);
```

RLS – let's query it!

```
=# SET SESSION AUTHORIZATION salaryman1;
SET
=> SELECT * FROM salaries ;
 id |    name    | salary
----+------------+---------
  1 | salaryman1 | 1500000
(1 row)
=> SET SESSION AUTHORIZATION salaryman2;
SET
=> SELECT * FROM salaries ;
 id |    name    | salary
----+------------+--------
  2 | salaryman2 |    150
(1 row)
```

FDW improvements

- IMPORT FOREIGN SCHEMA
- Foreign tables as part of inheritance tree
- CHECK constraints for foreign tables as planner hints
- Basics for JOIN pushdown with custom scans

Performance and scalability

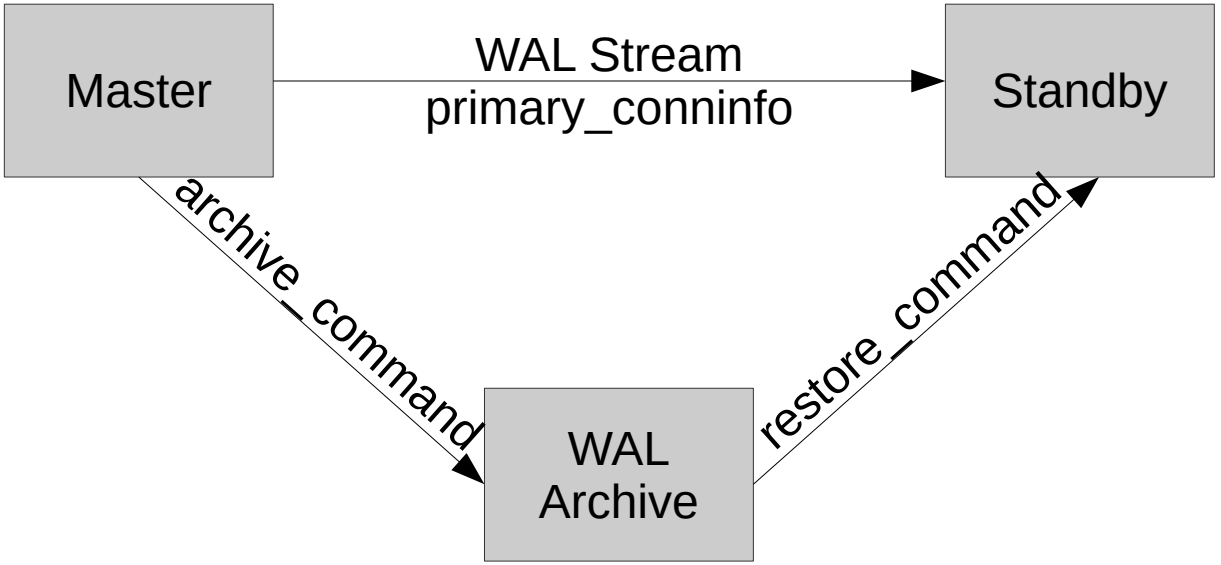
- In-memory sorting for text, varchar, numeric
  - Speed up CREATE INDEX, CLUSTER
  - ORDER BY speed up
- 128-bit accumulators for aggregates
- In-memory hash improvements (number of buckers bumped from 16 to 128)
- Scalability on multi-socket machines
- Speed up CRC calculation (CRC32-C)

Standbys

- Built from base backup of existing node
  - FS-level backup or snapshot
  - pg_basebackup
  - pg_start_backup() and pg_stop_backup() with custom method
- Backup taken from master or other standby
- Can be read-only:
  - hot_standby = on
  - wal_level >= hot_standby

Streaming standby

- Consumes WAL via streaming replication
- Can optionally consume WAL archives
- primary_conninfo in recovery.conf
- Can be synchronous



Archive and promotion <= 9.4

Master	Standby	Archive
000000010000000000000022	000000010000000000000022	**000000010000000000000022**
000000010000000000000023	000000010000000000000023	**000000010000000000000023**
000000010000000000000024	000000010000000000000024	**000000010000000000000024**
	Promotion	
000000010000000000000025	**000000010000000000000025**	**000000010000000000000025**
	000000020000000000000025	**000000020000000000000025**
	000000020000000000000026	**000000020000000000000026**

- Standby archives last, partial WAL segment of old timeline at promotion
  - Conflicts if archived from standby **and** master
  - Size of 16MB, with garbage after fork point

Archive and promotion >= 9.5

Master	Standby	Archive
000000010000000000000022 **000000010000000000000023** **000000010000000000000024** **000000010000000000000025**	000000010000000000000022 000000010000000000000023 000000010000000000000024 Promotion **000000010000000000000025** **000000020000000000000025** **000000020000000000000026**	**000000010000000000000022** **000000010000000000000023** **000000010000000000000024** **000000010000000000000025** **000000010000000000000025.partial** **000000020000000000000025** **000000020000000000000026**

- Standby still archives last, partial segment at promotion of old timeline
  - With suffix .partial
  - No name conflict
  - Format not recognized by backend, should be renamed manually if used at recovery

Lost WAL segments

Master	Standby	Archive
000000010000000000000022 **000000010000000000000023** 000000010000000000000024 000000010000000000000025	000000010000000000000022 000000010000000000000023 000000010000000000000024 Promotion **000000010000000000000025** **000000020000000000000025** **000000020000000000000026**	**000000010000000000000022** **000000010000000000000023** **000000010000000000000025.** **partial** **000000020000000000000025** **000000020000000000000026**

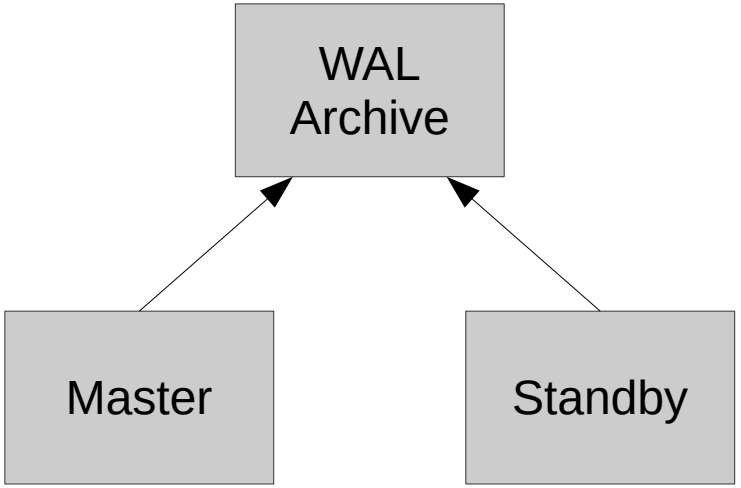
- Master crashed before archiving all segments
  - Game over to recover from older backups on new timeline
  - Series of backups now useless

archive_mode = 'always'

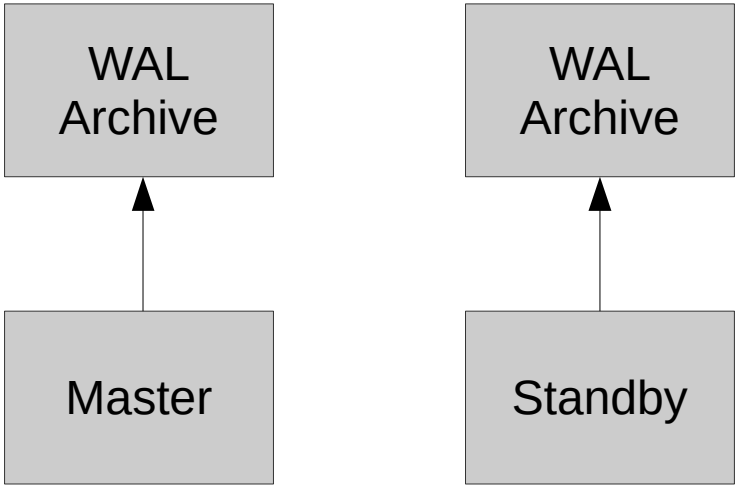
Master	Standby	Archive
000000010000000000000022 **000000010000000000000023** 000000010000000000000024 000000010000000000000025	**000000010000000000000022** **000000010000000000000023** **000000010000000000000024** Promotion **000000010000000000000025** **000000020000000000000025** **000000020000000000000026**	**000000010000**000000000022 **000000010000**000000000023 **000000010000000000000024** **000000010000000000000025.partial** **000000020000000000000025** **000000020000000000000026**

- No changes on master node compared to 'on'
- In recovery mode
  - standby will archive segments whose reception is finished
  - For <= 9.4, forcibly switched to .done
  - May want to use same archive_command. Or not.

archive_mode = 'always' (2)



- Be aware of name conflicts



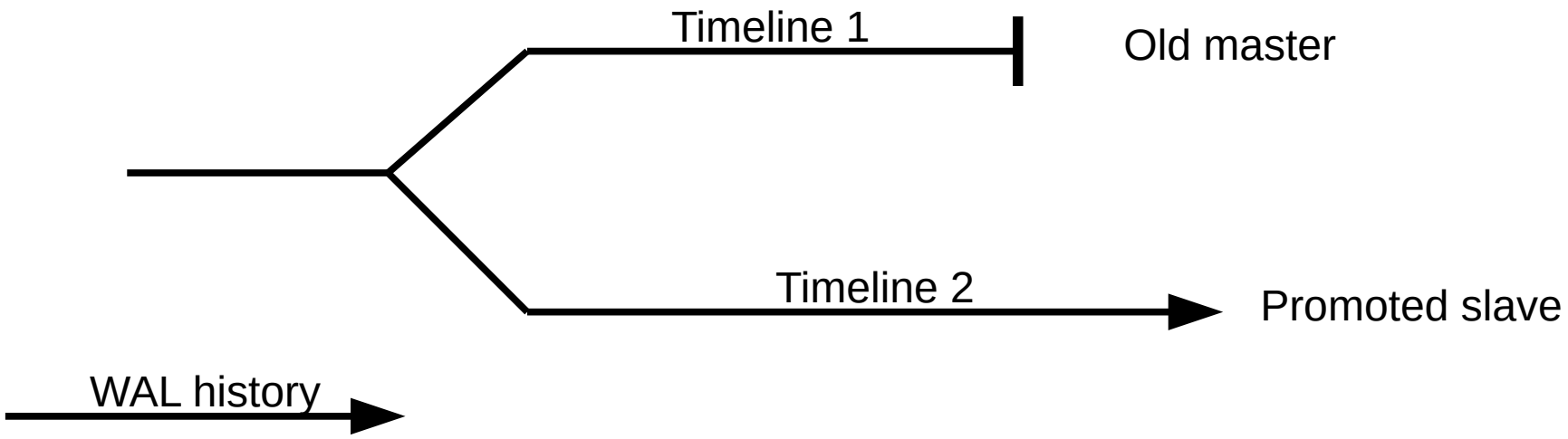
- Be careful with restore_command

pg_receivexlog

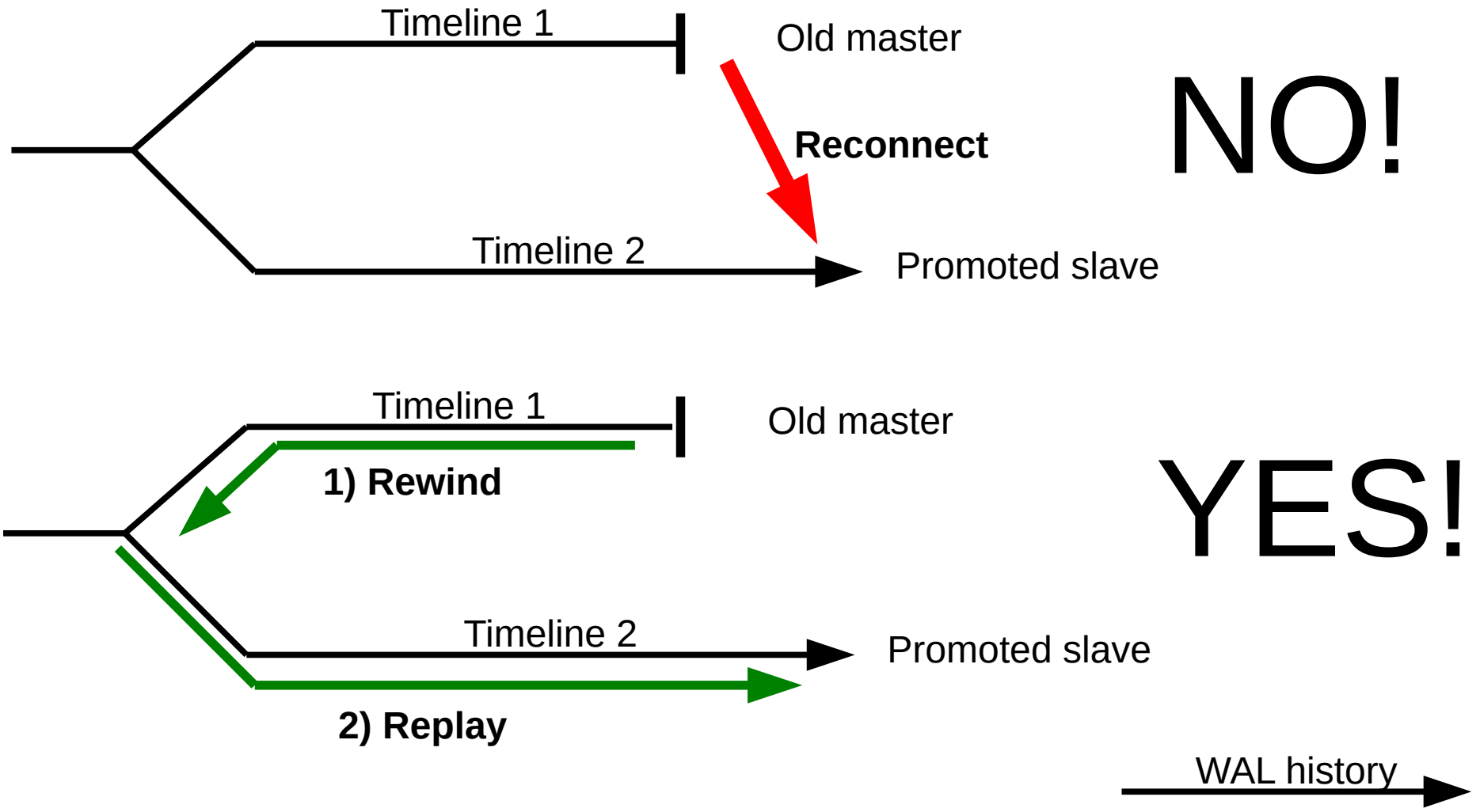
- Useful to leverage archiving
- Segments marked as .partial if not completed
- Should be used on archive host
- New features
  - Support for timeline switches in 9.3
  - Replication slots in 9.4
  - Synchronous mode in 9.5
- Take advantage with cascading replication and standbys (works <= 9.2)

pg_rewind (1)

- Resync an existing data directory without new base backup
  - Replug old master to existing cluster => “rewind it”



pg_rewind (2)


Timeline 1
Old master
Reconnect
Timeline 2
Promoted slave
NO!
Timeline 1
Old master
1) Rewind
YES!
Timeline 2
Promoted slave
2) Replay
WAL history

pg_rewind (3)

- Scan old master data folder
  - Start from WAL fork point
  - Record data blocks touched
- Copy all changed blocks from promoted slave to old master
- Copy clog, conf files...
- Replay WAL from master, starting from last checkpoint before WAL forked
- WAL format refactored for block tracking

pg_rewind (4)

- Largely faster than a new base backup
- Limitations
  - Need wal_log_hints = on or data checksum
  - Might not have all WAL for replay, can copy them though
  - No handling of timeline switches (WIP for 9.6)

min_wal_size and max_wal_size

- checkpoint_segments removed
- max_wal_size
  - Maximum WAL size between checkpoints
  - Soft limit
- min_wal_size
  - Control of WAL segment recycling
  - Useful to handle spikes in WAL usage

wal_compression?

- Not really WAL compression...
- Compression of full-page writes in WAL records
  - Image of block saved in WAL during modification after checkpoint.
- Compression with pglz
  - CPU consumer (may want to switch to lz4 in future)
  - Pushed in src/common, available for extensions
  - https://github.com/michaelpq/pg_plugins/tree/master/compress_test
- Reduction of sequential I/O write induced by WAL at the cost of some CPU.

wal_compression - performance

- Reduction of WAL size, for example:
  - 15% for FPW with UUID datatype
  - 27% for FPW with integers
- Reduction of recovery time, less segments to fetch.
  - Local machine, 500MB of FPWs.
  - UUID: 14.7s/15.3s
  - Integers: 18.5s/21.8s
- Synchronous replication
  - Bottleneck may be WAL record length when master and standby hosts are close
  - Contention with SyncRepLock
  - Can increase overall output

wal_compression - security

- Compression rate of a page gives hints on content
- Only PGC_SUSET
- Hide WAL position to lambda users
  - pg_current_xlog_position()
  - pg_current_xlog_insert_location()
  - pg_last_xlog_receive_location()
  - pg_last_xlog_replay_location()

```
REVOKE ALL ON FUNCTION pg_current_xlog_location()
   FROM PUBLIC;
-- etc.
```

wal_retrieve_retry_interval

- In 9.5, to control interval of time to fetch WAL from source after failure, either WAL archive or WAL receiver.
- Useful for archive recovery
  - Limit requests to WAL archive host
  - Accelerate detection of archived segment

Other things:

http://www.postgresql.org/docs/devel/static/release-9-5.html

Thanks!

Questions?