2ndQuadrant®
PostgreSQL

PostgreSQL Extensions

PGConf India 2019

Pavan Deolasee, 2ndQuadrant
PostgreSQL Developer

PostgreSQL is extensible

- Hyperconverged Database!
- PostgreSQL is extensible because its operation is catalog-driven and it stores information about data types, functions, access methods etc in the catalogs.
- The catalogs can be modified by the users and that makes PostgreSQL highly extensible.
- PostgreSQL also allows users to dynamically load arbitrary code in the engine
- And remember, it’s Open Source!

What is an Extension?

- A package of functions, operators, data types, index types, that can be installed and removed as a unit.
- First appeared in PostgreSQL 9.1, though similar capability existed even before in form of *modules.*
- Special SQL commands such as CREATE EXTENSION, DROP EXTENSION.
- A few important extensions are bundled with the core, several others are written and managed by third party developers.

What can be extended?

- PostgreSQL provides a bunch of hooks to incept and override default behaviour of planner, executor, transaction manager, DDLs, start/stop background worker processes, request system resources such as shared memory, low-level locks etc
- Define your own data types, aggregate functions, operators, operator classes
- Indexes and storage systems.

What can be extended?

- Write Foreign Data Wrappers to talk to a completely different datasource (including another database engine too).
- Define your own user-defined-functions in the language of your choice.
- Doesn’t stop there - implement your own language as well.

Why Extensions are important?

- Nearly impossible to handle all workloads in core PostgreSQL
  - Development process is often slow and conservative
  - Extensions allow rapid development and experimentation
- PostgreSQL can be modified without forking it (though our liberal licensing allows that)

Syntax

```
CREATE EXTENSION [ IF NOT EXISTS ]
extension_name
    [ WITH ] [ SCHEMA schema_name ]
             [ VERSION version ]
             [ FROM old_version ]
```

```
DROP EXTENSION [ IF EXISTS ] extension_name [,
...] [ CASCADE | RESTRICT ]
```

Syntax

ALTER EXTENSION ***extension_name*** UPDATE [TO ***new_version***]

ALTER EXTENSION ***extension_name*** SET SCHEMA ***new_schema***

ALTER EXTENSION ***extension_name*** ADD ***member_object***

ALTER EXTENSION ***extension_name*** DROP ***member_object***

Built-In Extensions

Built-In Extensions

- Bundled with PostgreSQL source code, standard packages
- Full supported and maintained by PostgreSQL development team
- Examples -
  - hstore
  - pg_stat_statements
  - auto_explain
- Some are moved to the server core
  - pg_rewind, pg_waldump, full text search

auto_explain

- The `auto_explain` module provides a means for logging execution plans of slow statements automatically, without having to run EXPLAIN by hand.
- The module provides no SQL-accessible functions. To use it, simply load it into the server.
  - `shared_preload_libraries`
  - `session_preload_libraries`
  - `LOAD ‘auto_explain’;`

auto_explain

- `auto_explain.log_min_duration (integer)`
  - minimum statement execution time, in milliseconds, that will cause the statement's plan to be logged
- `auto_explain.log_analyze (boolean)`
  - log `EXPLAIN ANALYZE` output, rather than just `EXPLAIN` output
- `auto_explain.log_nested_statements (boolean)`
  - log nested statements, rather than just the top level statement
- `auto_explain.sample_rate (real)`
  - Control logging rate

pg_stat_statements

- provides a means for tracking execution statistics of all SQL statements executed by a server.
- The module must be loaded by adding `pg_stat_statements` to `shared_preload_libraries` in `postgresql.conf`, because it requires additional shared memory. This means that a server restart is needed to add or remove the module.
- Provides functions and views to access/manipulate stats. Requires `CREATE EXTENSION`

pg_stat_statements

Name	Type	Description
`userid`	`oid`	OID of user who executed the statement
`dbid`	`oid`	OID of database in which the statement was executed
`queryid`	`bigint`	Internal hash code, computed from the statement's parse tree
`query`	`text`	Text of a representative statement
`calls`	`bigint`	Number of times executed

pg_stat_statements

Name	Type	Description
`total_time`	`double precision`	Total time spent in the statement, in milliseconds
`min_time`	`double precision`	Minimum time spent in the statement, in milliseconds
`max_time`	`double precision`	Maximum time spent in the statement, in milliseconds
`mean_time`	`double precision`	Mean time spent in the statement, in milliseconds
`stddev_time`	`double precision`	Population standard deviation of time spent in the statement, in milliseconds

postgres_fdw

- A data wrapper to speak to remote PostgreSQL databases (replaces dblink extension)
- Push WHERE clauses, JOINS, ORDER BY, aggregates
- Use transaction hooks to control remote transactions
- ANALYZE remote tables
- **You have a distributed database!**

HStore: A Key-Value Store

- Implements the `hstore` data type for storing sets of key/value pairs within a single PostgreSQL value
- A set of operators to operate on the `hstore` data type
- A set of functions
- New index types
- Integration with JSON and JSONB types

And More..

- pg_buffercache
- pg_prewarm
- pg_visibility
- pg_trgm
- pg_crypto
- pgstattuple

Third Party Extensions

Geospatial Database - PostGIS

- Several geometrical datatypes
- Point, Line, Rectangle, Polygon
- Associated operators, functions
- Associated index access methods
- Maintained and developed by PostGIS community
- **You have a fully OpenGIS compatible geospecial database!**

Logical Replication: pglogical

- PostgreSQL core now has logical replication
- Publisher-subscriber model
- Pglogical extends the in-core features
  - Connects to different data sources
  - Row and column filtering
  - Seamlessly replicate DDLs on the subscriber nodes
- **You have a complete logical replication in PostgreSQL!**

Bi-Directional Replication

- Uses built-in logical replication, the pglogical extension to create a multi-master, bi-directional replication solution
- Always-on architecture
- Rolling upgrades
- Geographically distributed database
- **You have a multi-master clustering solution!**

And many more..

- Miss planner hints?
  - pg_hint_plan
- Columnar store?
  - Cstore_fdw
- Timeseries data?
  - TimescaleDB
- Distributed data?
  - Citus

Write Your Own Extension?

- Faced with PostgreSQL’s limitation?
  - ○ Check if a work-around is available
  - ○ Check someone else has already solved the problem for you (and if the solution is publicly available)
  - ○ Talk to your PostgreSQL support provider.
  - ○ Roll out your own?

Knowing PGXS

- Build infrastructure provided by PostgreSQL for building/distributing extensions
- Mainly used for extensions which include C code (as most extensions would do), but can be used otherwise too
- Automates simple build rules
- For very complex extensions, you may need to write your own

Sample Makefile

```
# contrib/pg_prewarm/Makefile
MODULE_big = pg_prewarm
OBJS = pg_prewarm.o autoprewarm.o

EXTENSION = pg_prewarm
DATA = pg_prewarm--1.1--1.2.sql pg_prewarm--1.1.sql

PGFILEDESC = "pg_prewarm - preload relation data into system buffer cache"

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Server-side Hooks

- Parser Hooks
- Planner Hooks
- Executor Hooks
- Transaction Control Hooks
- Utility command hooks

An Example

- Our customer reported TOAST corruption
- Queries started failing with ERRORs; no easy way to find the extent of corruption and the problematic rows
  - Sequential scan of the table ends at the first error
  - Index scan on each PK is very costly

Simple Way

```
DO $$
DECLARE
  baddata TEXT;
  badid INT;
BEGIN
FOR badid IN SELECT id FROM badtable LOOP
  BEGIN
    SELECT badcolumn
    INTO columndata
    FROM badtable where id = badid;
  EXCEPTION
    WHEN OTHERS THEN
      RAISE NOTICE 'Data for ID % is corrupt', badid;
      CONTINUE;
  END;
END LOOP;
END;
$$
```

More hackish (superfast) way

A `toastcheck` extension

```
/*
 * toast_check(relid regclass)
 *
 * Verify integrity of toast table.
 */
Datum
toast_check(PG_FUNCTION_ARGS)
{
    Oid         relid = PG_GETARG_OID(0);
```

Toastcheck: Scan the Heap

```
    /*
     * Scan all tuples in the base relation. Uses a global
heapTuple pointer to
     * track the heap tuple so that toast routine can
quickly know the current
     * TID.
     */
    while ((state.heapTuple = heap_getnext(state.scan,
ForwardScanDirection)) != NULL)
    {
        ...
        heap_deform_tuple(state.heapTuple, tupdesc, values,
nulls);
```

Toastcheck: ERROR -> NOTICE

```
        if (residx != nextidx)
        {
            elog(NOTICE, "unexpected chunk number %d
(expected %d) for toast value %u in %s ctid (%u,%u),
column %s",
                 residx, nextidx,
                 toast_pointer.va_valueid,
                 RelationGetRelationName(state->heaprel),

ItemPointerGetBlockNumber(&state->heapTuple->t_self),

ItemPointerGetOffsetNumber(&state->heapTuple->t_self),
                 state->colName);
```

Toastcheck: Makefile

```
# contrib/toastcheck/Makefile

MODULE_big  = toastcheck
OBJS        = verify_toast.o

EXTENSION = toastcheck
DATA = toastcheck--1.0.sql
PGFILEDESC = "toastcheck - function for verifying toast
relation integrity"

REGRESS = toastcheck

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Summary

- Significantly extend PostgreSQL capabilities, outside the core
- A huge open source community which is contributing significantly to PostgreSQL and its adoption
- New businesses are being built purely on PostgreSQL extensions

2ndQuadrant PostgreSQL Solutions

Website	https://www.2ndquadrant.com/
Blog	https://blog.2ndquadrant.com/
Email	info@2ndQuadrant.com

Thank you!