Fujitsu PostgreSQL blog

sing PostgreSQL: What es in version 16, how we re, and what to expect in releases releases

la | September 15, 2023

Earlier this year, I was in Canada for PGConf 2023, where I talked about the evolution of PostgreSQL from a Berkeley research project to its current status as the most advanced open-source database, and discussed the various changes introduced for PostgreSQL 16, particularly regarding logical replication.

Now that the PostgreSQL community has released version 16, I thought it would be the perfect opportunity to talk to a wider audience, especially if you did not have the

In my talk, I focused on the exciting new features introduced in version 16, but I also touched on the past of PostgreSQL - the timeline of its major features in previous versions -, and its future - what the community has been discussing, possibly for implementation in PostgreSQL 17.

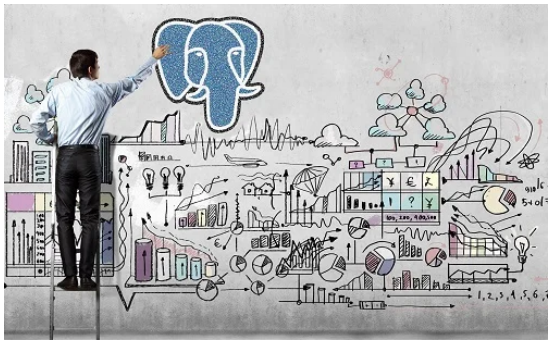
Contents

Search by topic

- PostgreSQL (63)
- PostgreSQL community (50)
- Fujitsu Enterprise Postgres (38)
- PostgreSQL development (21)
- Database security (19)
- PostgreSQL event (16)
- Logical replication (13)
- Data Masking (11)

To better appreciate the long journey that PostgreSQL has taken since its early days, I think it's important to revisit its progress, and the impressive list of features it accumulated along the way, thanks to the work of our committed and dedicated community.

Evolution of PostgreSQL with its major milestones across versions



We have started this project in 1997 from the University of California, at Berkeley Project, which had been running since 1986, and from then on, a new version with major features has been released every year.

It is interesting to note how, from inception, PostgreSQL has been geared towards handling large volumes of data, and has been evolving since then in that direction.

For example, version 9.0 brought streaming replication, which helps with failover, where one node can take over in case another goes down. Version 9.1 introduced foreign tables and unlogged tables to its list features, alongside synchronous replication, helping make it more reliable and further enhance its value.

With 9.2, we have added support for JSON data type, where document and its related data could be stored using this data type. This version also brought index-only scans, which improved the speed of various kinds of queries. The next year, version 9.3 added updates to foreign data wrappers and materialized views, and then 9.4 introduced JSONB data type, an improved version of JSON that allows faster processing and indexes on that data type.

which can be leveraged by OLAP applications. And to further enhance our reliability, we introduced multiple standby servers in synchronous replication.

In version 10, a new landscape has opened for Postgres, with the introduction of logical replication and declarative partitioning. Version 11 introduced partitioning by hash key and SQL stored procedures, widening the appeal for organizations considering migration from other databases. With version 12, we enhanced performance of partitioning and introduced table access methods, where people can write their specialized storage engines, which they can integrate with PostgreSQL. Some of the key features of version 13 are de-duplication in the B-tree index, incremental sorting, and parallel vacuum for indexes.

Version 14 introduced a large number of important features and enhancements. We achieved better read scalability by improving our snapshot mechanism, and allowed logical replication for in-progress transactions, which reduce lag when applying large transactions. We also reduced bloat for B-tree index updates, and allowed parallel foreign table scan using postgres_fdw.

With version 15, we introduced the MERGE command, which was being discussed for a couple of years in the community, and we finally had the chance to implement it. We also introduced shared memory statistics, which improved over the previous statistics mechanism, and improved logical replication further, by introducing row

PostgreSQL 16 enhancements and new features

Version 16 introduces a lot of new features, with several improvements to the logical replication mechanism. One of the most important ones in my view is the ability to perform logical replication from the standby node, which I will discuss below.

Improvements to logical replication

- Data can be filtered based on origin during replication.

```
CREATE PUBLICATION mypub FOR ALL TABLES;
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=postgres'
       PUBLICATION  mypub WITH (origin = none);
```

  Prior to Postgres 16, setting a bi-directional or logical replication among nodes was difficult, because if we set up replication for a table, it would lead to an infinite loop. By adding the ability to filter data based on origin, we can set up n-way logical replication, and that will prevent loops when performing bi-directional replication.

- Logical decoding can be performed from the standby server.

  This requires `wal_level = logical` on both primary and standby.

  This ability can be used for workload distribution, by allowing subscribers to subscribe from the standby when the primary is busy.

```
CREATE SUBSCRIPTION mysub CONNECTION …
       PUBLICATION mypub WITH (run_as_owner = false);
```

- Non-superusers can create subscriptions.

  The non-superusers must have been granted pg_create_subscription role, and are required to specify a password for authentication.

  Superusers can set `password_required = false` for non-superusers that own the subscription.

- Large transactions can be applied in parallel.

```
CREATE SUBSCRIPTION mysub CONNECTION …
       PUBLICATION mypub WITH (streaming = parallel);
```

  Performance improvement in the range of 25-40% has been observed (for further details, check here).

  Each large transaction is assigned to one of the available workers, which improves lag by immediately applying instead of waiting till whole transaction is received by the subscriber. The worker remains assigned until the transaction completes.

  max_parallel_apply_workers_per_subscription sets the maximum number of parallel apply workers per subscription.

- Logical replication can copy tables in binary format.

```
CREATE SUBSCRIPTION mysub CONNECTION …
       PUBLICATION mypub WITH (binary = true);
```

  Copying tables in binary format may reduce the time spent, depending on column types

- Indexes other than PK and REPLICA IDENTITY can be used on the subscriber

The index that can be used must be a btree index, not a partial index, and the leftmost field must be a column (not an expression) that references the remote relation column.

The performance improvement is proportional to the amount of data in the table.

Enhancements to storage

- Performance of relation extension has been improved.

  The newest version provides significant improvement (3x for 16 clients) for concurrent COPY into a single relation.

  The relation extension lock is held just during relation extension, whereas previously the relation extension lock was held while the system:

  - Acquired a victim buffer for the new page (this could require further writing out the old page contents, including possibly needing to flush WAL).
  - Write a zero page during the extension, and then write out the actual page contents (this could nearly double the write rate).

- HOT updates are allowed if only BRIN-indexed columns are updated.

- Direct I/O

  This allows to ask the kernel to minimize caching effects for relation data and WAL files. Currently, this feature reduces system performance, and is not intended for end users, so it is disabled by default. This option can be enabled by GUC debug_io_direct (valid values are: data, wal, wal_init).

  The further plan is to introduce our own I/O mechanisms like read-ahead, etc. to replace the facilities that the kernel disables with this option. Align all I/O buffers at 4096 to have a better performance with direct I/O.

- Allow freezing at page level during vacuum.

  This reduces the cost of freezing by reducing WAL volume.

- pg_stat_io view to show detailed I/O statistics

  The view contains one row for each combination of backend type, target I/O object, and I/O context, showing cluster-wide I/O statistics.

  - Example of backend types: background worker, autovacuum worker, checkpointer, etc.
  - Possible type of target I/O objects: Permanent or Temporary relations.
  - Possible values of I/O context: normal, vacuum, bulkread, bulkwrite.

  The view tracks various I/O operations like reads, writes, extends, hits, evictions, reuses, fsyncs. A high evictions count can indicate that shared buffers

had to be fetched from disk and that which already resided in the kernel page cache.

- Allow vacuum/analyze to specify buffer usage limit

  A new option BUFFER_USAGE_LIMIT has been added, which allows user to control the size of shared buffers. Larger values can make vacuum run faster at the cost of slowing down other concurrent queries.

  vacuum_buffer_usage_limit (GUC) provides another way to control, but BUFFER_USAGE_LIMIT would take precedence. GUC allows even autovacuum to use the specified limit.

  We have also added --buffer-usage-limit option to vacuumdb.

New SQL features

The new version comes with a host of new features that give users more options, among others, for collation and JSON data manipulation.

- Support for text collations (which provide rule for how text is sorted) has been improved.

  ```
  CREATE COLLATION en_custom (provider = icu, locale = 'e
  ```

  The line above places g after a, before b.

  New options are added to CREATE COLLATION, CREATE DATABASE, createdb, and initdb to set the rules.

- ICU to be allowed as the default collation provider.

  Determine the ICU default locale from the environment.

- SQL/JSON standard-conforming constructors for JSON types
  - JSON_ARRAY() - Constructs a JSON array from either a series of value_expression parameters or from the results of query_expression.

```
SELECT json_array(1,true,json '{"a":null}');
     json_array
-----------------------
[1, true, {"a":null}]
```

  - JSON_ARRAYAGG() - Behaves in the same way as json_array but as an aggregate function so it only takes one value_expression parameter.

```
SELECT json_arrayagg(v NULL ON NULL) FROM (VALUES(2
 json_arrayagg
-----------------
[2, 1, 3, null]
```

  - JSON_OBJECT() - Constructs a JSON object of all the key/value pairs given, or an empty object if none are given.

```
SELECT json_object('code' VALUE 'P123', 'title': 'J
            json_object
-----------------------------------
{"code" : "P123", "title" : "Jaws"}
```

  - JSON_OBJECTAGG() - Behaves like json_object, but as an aggregate function, so it only takes one key_expression and one value_expression parameter.

```
SELECT json_objectagg(k:v) FROM (VALUES ('a'::text,
               json_objectagg
```

IS JSON [VALUE], IS JSON ARRAY, IS JSON OBJECT, IS JSON SCALAR

```
SELECT js, js IS JSON "json?", js IS JSON SCALAR "scala
       js IS JSON OBJECT "object?", js IS JSON ARRAY "a
FROM (VALUES ('123'), ('"abc"'), ('{"a": "b"}'), ('[1,2
    js     | json? | scalar? | object? | array?
-----------+-------+---------+---------+--------
123        | t     | t       | f       | f
"abc"      | t     | t       | f       | f
{"a": "b"} | t     | f       | t       | f
[1,2]      | t     | f       | f       | t
```

- Parallel hash full join

```
EXPLAIN (COSTS OFF)
 SELECT COUNT(*)
 FROM simple r FULL OUTER JOIN simple s USING (id);
                        QUERY PLAN
-----------------------------------------------------
Finalize Aggregate
  ->  Gather
        Workers Planned: 2
        ->  Partial Aggregate
              ->  Parallel Hash Full Join
                    Hash Cond: (r.id = s.id)
                    ->  Parallel Seq Scan on simple r
                    ->  Parallel Hash
                          ->  Parallel Seq Scan on simp
```

- Parallel aggregate is allowed on string_agg and array_agg

```
EXPLAIN (COSTS OFF)
 SELECT y, string_agg(x::text, ',') AS t, array_agg(x)
 FROM pagg_TEST GROUP BY y;
                    QUERY PLAN
-------------------------------------------------
Finalize HashAggregate
  Group Key: y
  ->  Gather
        Workers Planned: 2
        ->  Partial HashAggregate
```

.

Previously, we always needed to sort tuples before doing aggregation.

With PostgreSQL 16, an index can provide pre-sorted input, which will be directly used for aggregation

```
EXPLAIN (COSTS OFF)
 SELECT SUM(c1 ORDER BY c1), MAX(c2 ORDER BY c2) FROM p
                        QUERY PLAN
-----------------------------------------------------
Aggregate
  ->  Index Scan using presort_test_c1_idx on presort_t
SET enable_presorted_aggregate=off;
EXPLAIN (COSTS OFF)
 SELECT SUM(c1 ORDER BY c1), MAX(c2 ORDER By c2) FROM p
          QUERY PLAN
-------------------------------
Aggregate
  ->  Seq Scan on presort_test
```

- The last found partition for RANGE and LIST partition lookups is cached.

  This reduces the overhead of bulk-loading into partitioned tables where many consecutive tuples belong to the same partition.

- Left join removals and unique joins on partitioned tables are allowed.

```
CREATE TEMP TABLE a (id int PRIMARY KEY, b_id int);
CREATE TEMP TABLE parted_b (id int PRIMARY KEY) PARTITI
CREATE TEMP TABLE parted_b1 PARTITION OF parted_b FOR V

EXPLAIN (COSTS OFF)
 SELECT a.* FROM a LEFT JOIN parted_b pb ON a.b_id = pb
 QUERY PLAN
---------------
Seq Scan on a
```

Additions to security/privileges

- **reserved_connections** provides a way to reserve connection slots for non-superusers.
- **pg_use_reserved_connections** allows the use of connection slots reserved via reserved_connections

- Support for Kerberos credential delegation has been added
  - This allows the PostgreSQL server to use those delegated credentials to connect to another service, such as with postgres_fdw or dblink or theoretically any other service which is able to be authenticated using Kerberos

- A new libpq connection option require_auth was added to specify a list of acceptable authentication methods.

  The following methods may be specified: `password`, `md5`, `gss`, `sspi`, `scram-sha-256`, or `none`.

  This option can also be used to disallow certain authentication methods, by prefixing the method with `!`.

  If the server does not use the required authentication request, the connection attempt will fail.

- libpq can now use the system certificate pool for certificate verification.

The GRANT ... SET option was added

The SET option, if set to TRUE, allows the member to change to the granted role using the SET ROLE command. To create an object owned by another role or to give ownership of an existing object to another role, you must have the ability to SET ROLE to that role. Otherwise, commands such as ALTER ... OWNER TO or CREATE DATABASE ... OWNER will fail.

Miscellaneous performance improvements

- Performance of pg_strtoint*nn* functions was improved.

  Testing has shown about 8% speedup of COPY into a table containing 2 INT columns.

- Performance of hash index builds was improved.

  In the initial data sort, if the bucket numbers are the same, then sort on hash value next. The build is sped up by skipping needless binary searches. Hash Index build speed up by 5-15%.

- Performance of memory management was improved and overhead was reduced.

  The header size for each allocation was reduced from 16+ to 8 bytes, and performance of the slab memory allocator, which is used to allocate memory during logical decoding, was improved.

- Support for CPU acceleration using SIMD for both x86 and ARM architectures was added.

  This optimizes processing of subtransaction searches and ASCII and JSON strings.

hosts and addresses to be connected to in random order. This parameter can be used in combination with target_session_attrs to load balance over standby servers only.

It is recommended to also configure a reasonable value for connect_timeout to allow other nodes to be tried when the chosen one is not responding.

- LZ4 and Zstandard compression options were added to pg_dump.
- COPY into foreign tables to add rows in batches in now supported.

  This is controlled by the postgres_fdw batch_size option.

Compatibility

- Supports a minimum version of Windows 10 for Windows installations.
- Removes the promote_trigger_file option to enable the promotion of a standby server.

  Users should use the pg_ctl promote command or pg_promote() function to promote a standby.
- The server variable vacuum_defer_cleanup_age has been removed.

  This has been unnecessary since hot_standby_feedback and replication slots were added.
- libpq support for SCM credential authentication was removed.

The full list of new/enhanced features and other changes can be found here.

What 's next: PostgreSQL 17 and beyond

Before wrapping up, I would like to share some of the changes that the PostgreSQL community is currently discussing for future PostgreSQL versions, like PostgreSQL 17 and later. Please keep in mind that this list isn't set in stone, and there's no guarantee that all these features will make it to the final cut - it's based on my own observations from the discussions happening within the PostgreSQL community.

- Various improvements in logical replication:
  - DDL Replication

replication slots to allow failover

- Upgrade of logical replication nodes
- Reuse of tablesync workers

- Reduced number of commands that need superuser privilege.
- SQL/JSON improvements to make it more standard-compliant.
- Incremental backups
- Transparent column encryption/decryption of particular columns on the client.
- Asynchronous I/O will allow prefetching data and will improve system performance.
- Large relation files to reduce open/close for huge numbers of file descriptors.
- Enhance Table AM APIs.
- Amcheck for Gist and Gin indexes.
- Improve locking for better scalability.
- Improvements in vacuum technology by using performance data structure.
- Improvements in partitioning technology.
- Improve statistics/monitoring.

autovacuum

- Parallelism
  - Allow parallel-safe initplans.
  - Parallelize correlated subqueries.
- Smaller headers in WAL to reduce overall WAL size.
- Move SLRU into main buffer pool.
- TOAST improvements: custom formats, and compression dictionaries.
- CI and build system improvements

Topics: PostgreSQL, PostgreSQL community, PostgreSQL development, PostgreSQL event

Senior Director of Fujitsu's PostgreSQL team, PostgreSQL committer

Amit Kapila is the Senior Director of Fujitsu's PostgreSQL team, a PostgreSQL commiter and Major Contributor. Amit specialty is working in Database Internals (SQL Engine, Storage Engine, and Replication), and is known for setting up teams from scratch for R&D work.

View all Amit's blog posts

We also have a series of technical articles for PostgreSQL enthusiasts of all stripes, with tips and how-to's.

Subscribe to be notified of future blog posts

If you would like to be notified of my next blog posts and other PostgreSQL-related articles, fill the form here.

Read our latest blogs

Read our most recent articles regarding all aspects of PostgreSQL and Fujitsu Enterprise Postgres.

Discussing PostgreSQL: What changes in version 16, how we got here, and what to expect

Bi-directional replication using origin filtering in PostgreSQL
by Vigneshwaran C

How to fix transaction wraparound in PostgreSQL?
by Nishchay

How to configure PAM authentication in Fujitsu Enterprise Postgres
by Nishchay

Receive our blog

Fill the form to receive notifications of future posts

Company

Business email address

* Denotes mandatory field

We respect your data and privacy. By clicking below, you consent to the storage and processing of the data submitted for the purpose of providing you the content requested.

Tick to receive additional offers and product updates.

You can unsubscribe or update your communication preferences at any time. See our Privacy Policy for details.

Subscribe

Search by topic

- PostgreSQL (63)
- PostgreSQL community (50)
- Fujitsu Enterprise Postgres (38)
- PostgreSQL development (21)
- Database security (19)
- PostgreSQL event (16)
- Logical replication (13)
- Data Masking (11)

see all >