Latest developments in PostgreSQL

Amit Kapila

PostgreSQL Committer and Major Contributor

Fujitsu

Agenda

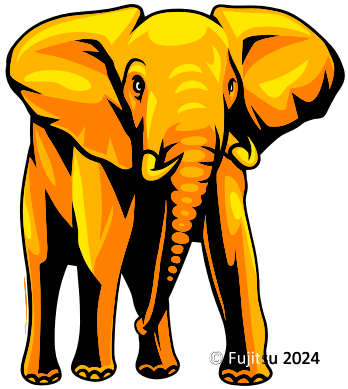
- Key features and performance improvements in PostgreSQL 17
- PostgreSQL 18 and beyond


FUJITSU
Agenda
Key features and performance improvements in PostgreSQL 17
PostgreSQL 18 and beyond

Storage 1/4

FUJITSU

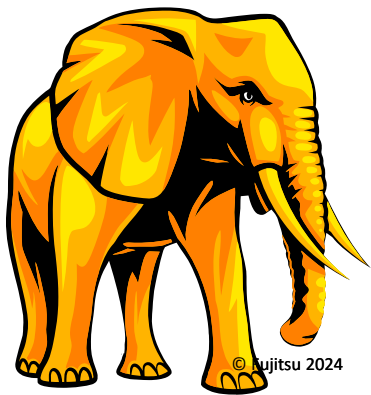
- Incremental Backups
  - Useful for taking backups of large data
  - pg_basebackup can be used to take incremental backups by specifying the `--incremental` option
  - Specify the backup manifest to an earlier backup from the same server
  - In the resulting backup only the changed blocks are copied
  - To figure out which blocks needs to be copied, the server uses WAL summaries stored in the data directory
    - A GUC summarize_wal needs to be enabled to collect these WAL summaries by a background process
  - The tool pg_combinebackup is used to reconstruct a full backup from an incremental backup and earlier backups upon which it depends



Storage 2/4

FUJITSU

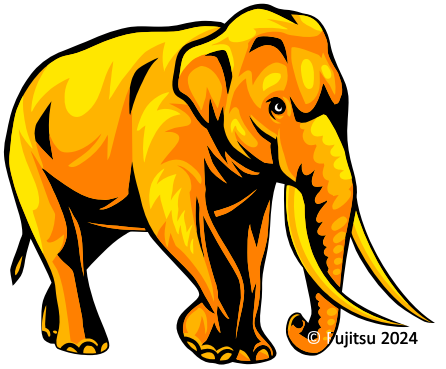
- Improved the mechanism to remove dead tuples during vacuum
  - Replaced the array used to store dead tuples with efficient TIDStore based on adaptive radix tree
  - Since the backing radix tree makes small allocations as needed, the 1GB limit is now gone.
  - Total memory used is now often smaller by an order of magnitude or more
  - This makes multiple rounds of heap scanning and index cleanup an extremely rare event
  - TID lookup during index cleanup is also several times faster
- Reduced the WAL volume for Vacuum by combining freezing and pruning steps such that we now emit a single WAL record containing changes from both steps
  - As a consequence of this, WAL sync and write time is reduced
- Optimize vacuuming of relations with no indexes
  - Items can be marked LP_UNUSED instead of LP_DEAD when pruning
  - This significantly reduces WAL volume



Storage 3/4

FUJITSU

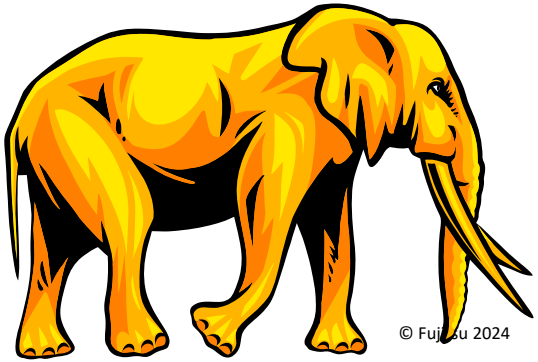
- Faster reads by using streaming APIs
  - This happens by allowing pages to be prefetched and performing vectored reads in chunks up to io_combine_limit
  - The operations improved are sequence scans, analyze, and pg_prewarm
- Improved performance of subsystems on top of SLRU
  - We achieved this by having configurable SLRU cache sizes
  - The cache is divided in "banks" so that eviction buffer search only affects one specific bank
  - Changed the locking regime for the SLRU banks, so that each bank uses a separate LWLock



Storage 4/4

FUJITSU

- Allow Table Am's to skip fetching a block from the heap
  - The block fetch can be skipped if none of the underlying data is needed and the block is marked all visible in the visibility map
  - Previously such an optimization was only used in BitmapHeapScan
- Optimized array matches in BTree-index
  - This significantly improves execution time of queries that use the IN/ANY clause with a B-tree index
- Improved performance of heavily-contended WAL writes, especially at a higher client count (256 and above)



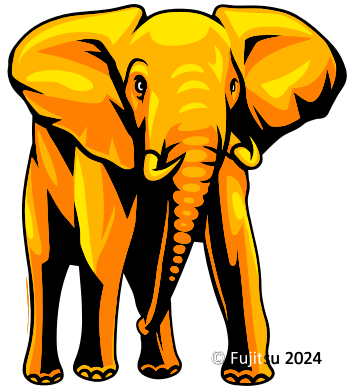
Logical Replication

FUJITSU

- Sync/Failover slots
  - Allow subscribers to follow standbys after primary/publisher goes down
  - The failover slots are copied from primary to hot standby at regular intervals by a slotsync worker process
  - Users can manually sync the slots by using pg_sync_replication_slots()
  - Enabling failover allows us to smoothly transition to the promoted standby, ensuring that we can subscribe to the new primary without losing any data
  - One can enable failover option for a subscription as follows:

```
CREATE SUBSCRIPTION sub CONNECTION '$connstr'
       PUBLICATION  pub WITH (failover = 'true’)
```

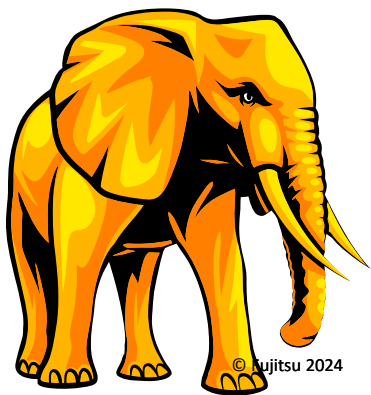
  - Subscribers can continue subscribing to publications now on the new primary server without losing any data that has been flushed to the new primary server
  - For more information, read docs



Logical Replication

FUJITSU

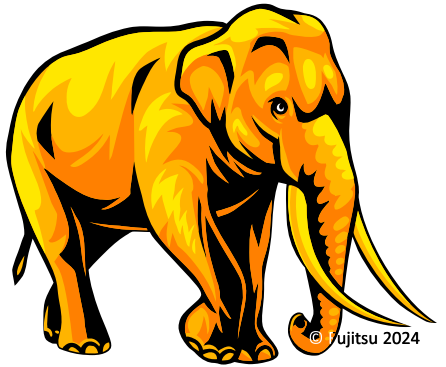
- Allow upgrade of logical replication nodes
  - Prior to this feature, users manually need to re-create the slots on upgraded publisher and the subscription set up on new subscribers also need to be re-defined which sometimes may need to copy the data again.
  - Migrate logical slots to new node during upgrade of publisher node
  - Upgrades preserve the full subscription’s state
  - Migration of logical replication clusters is possible only when all the members of the old logical replication clusters are version 17.0 or later
  - While upgrading a subscriber, write operations can be performed in the publisher. These changes will be replicated to the subscriber once the subscriber upgrade is completed



Logical Replication

FUJITSU

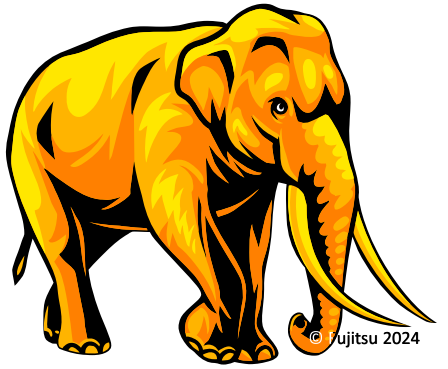
- pg_createsubscriber to create a logical replica from a physical standby server
  - Speed up creation of logical subscriber
  - It can be used for upgrading physical replication nodes. Say there is a physical replication setup between node-A and node-B. Follow below steps to upgrade both nodes in the physical replication setup:
    - Stop the standby server (node-B).
    - Run pg_createsubscriber on node-B.
    - Upgrade node-B and then start node-B.
    - Create a physical replica from node-B, say node-C. So both node-B and node-C are on newer server versions.
    - Transition all writes from node-A to node-B.
    - Decommission node-A.
  - By the end, we have a physical replica setup (node-B → node-C) of the newer version without stopping operations.



Logical Replication

FUJITSU

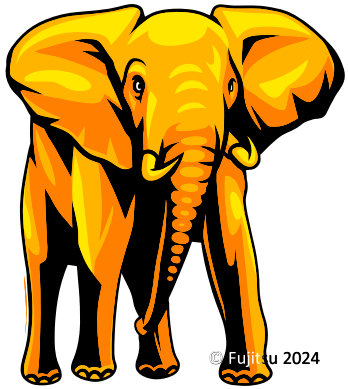
- Speed up logical decoding in cases where there are many subtransactions
  - Previously, we use to check all the (sub)transactions to find the largest transaction to evict
  - The new eviction algorithm uses max-heap with transaction size as the key to efficiently find the largest transaction in O(1)
  - A speed up of 30x has been observed in decoding a transaction with 100k subtransactions
- Allow the use of hash indexes for lookups when PK or REPLICA IDENTITY are not available on the subscriber



FUJITSU

- Use multiple workers to build BRIN indexes
  - Each worker builds BRIN summaries on the subset of table and store those in a sorted form
  - The leader read these sorted stream of ranges and adds the resulting ranges into the index
  - For large tables this often results in significant speedup when the build is CPU-bound
- Queries that generate initPlans can use parallel workers to execute initPlan

```
EXPLAIN (COSTS OFF) SELECT c1 FROM t1 WHERE c1 = (SELECT 1);
               QUERY PLAN
----------------------------------------
   Gather
     Workers Planned: 2
     InitPlan 1
       ->  Result
     ->  Parallel Seq Scan on t1
           Filter: (c1 = (InitPlan 1).col1)
```



FUJITSU

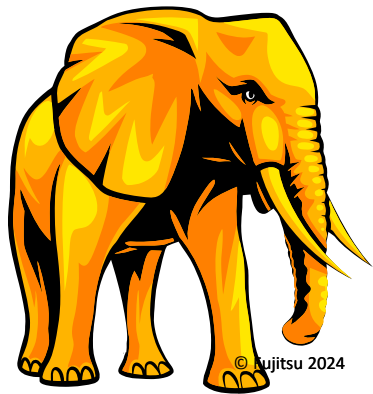
- Eliminated **`IS NOT NULL`** query restrictions on **`NOT NULL`** columns

```
CREATE TABLE pred_tab (a int NOT NULL, b int, c int NOT NULL);
EXPLAIN (COSTS OFF) SELECT * FROM pred_tab t WHERE t.a IS NOT NULL;
       QUERY PLAN
   ------------------------
     Seq Scan on pred_tab t
```

- Eliminated scans on **`NOT NULL`** columns if **`IS NULL`** is specified

```
EXPLAIN (COSTS OFF) SELECT * FROM pred_tab t WHERE t.a IS NULL;
        QUERY PLAN
   --------------------------
    Result
        One-Time Filter: false
```

- COPY adds a new option, ON_ERROR ignore, that allows a copy operation to continue in the event of an error

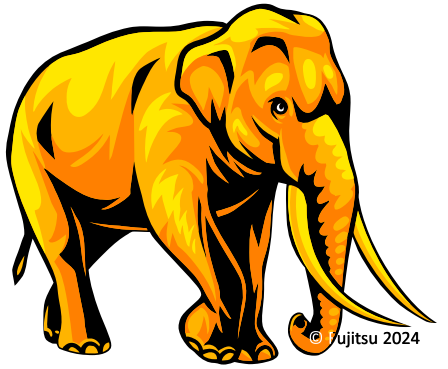


FUJITSU

- Allow correlated **`IN`** subqueries to be transformed into joins

```
EXPLAIN (costs off) SELECT * from tenk1 A WHERE hundred in
 (select hundred from tenk2 B where B.odd = A.odd);
                          QUERY PLAN
   ------------------------------------------------------------
   Hash Join
     Hash Cond: ((a.odd = b.odd) AND (a.hundred = b.hundred))
     ->  Seq Scan on tenk1 a
     ->  Hash
           ->  HashAggregate
                 Group Key: b.odd, b.hundred
                 ->  Seq Scan on tenk2 b
```

- Improved CTE plans by considering the statistics and sort order of columns referenced in earlier row output clauses
  - This improves the execution time of such queries significantly

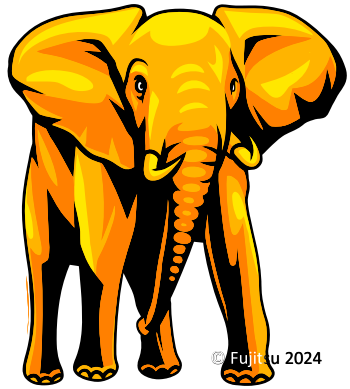


FUJITSU

- Support identity columns in partitioned tables
  - A newly created partition inherits identity property
  - An identity column shares the same underlying sequence across all partitions of a partitioned table
  - In regular inheritance, identity cols in a child table are independent of those in its parent tables
  - A table being attached as a partition inherits the identity property from the partitioned table
  - The identity columns of the partition being detached lose their identity property
- Allow exclusion constraints on partitioned tables
  - As long as exclusion constraints compare partition key columns for equality, other columns can use exclusion constraint-specific comparisons

```
CREATE TABLE idxpart (a int4range, b int4range, c int4range,
            EXCLUDE USING GIST (b with =, c with &&)) PARTITION BY RANGE (a);
ERROR:  unique constraint on partitioned table must include all partitioning columns
DETAIL:  EXCLUDE constraint on table "idxpart" lacks column "a" which is part of the partition key.
```

```
CREATE TABLE idxpart (a int4range, b int4range, c int4range,
            EXCLUDE USING GIST (a with =, b with =, c with &&)) PARTITION BY RANGE (a, b);
```

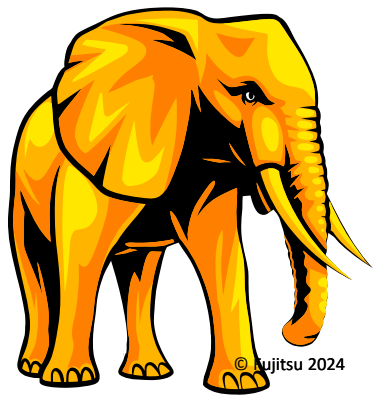


FUJITSU

- Allow pushdown of EXISTS and IN subqueries to the postgres_fdw foreign server

```
EXPLAIN (VERBOSE, COSTS OFF) SELECT t1.c1 FROM ft1 t1 WHERE EXISTS (SELECT 1 FROM ft2 t2 WHERE t1.c1 = t2.c1)
                          ORDER BY t1.c1 OFFSET 100 LIMIT 10;
Foreign Scan
   Output: t1.c1
   Relations: (public.ft1 t1) SEMI JOIN (public.ft2 t2)
   Remote SQL: SELECT r1."C 1" FROM "S 1"."T 1" r1 WHERE EXISTS (SELECT NULL FROM "S 1"."T 1" r2
               WHERE ((r2."C 1" = r1."C 1"))) ORDER BY r1."C 1" ASC NULLS LAST LIMIT 10::bigint OFFSET 100::bigint
```

- Allow joins with non-join qualifications to be pushed down to foreign servers and custom scans

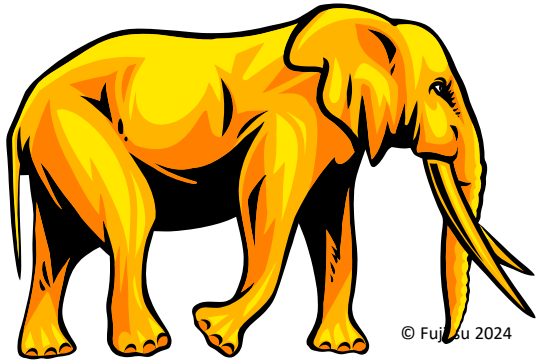


FUJITSU

- **MERGE** command now supports **RETURNING** clause
  - New function merge_action() can be used with **RETURNING** to report the DML that generated the row

```
MERGE INTO products p  USING stock s ON p.product_id = s.product_id
      WHEN MATCHED AND s.quantity > 0 THEN UPDATE SET in_stock = true, quantity = s.quantity
      WHEN NOT MATCHED THEN  INSERT (product_id, in_stock, quantity) VALUES (s.product_id, true, s.quantity)
      RETURNING merge_action(), p.*;
 merge_action | product_id | in_stock | quantity
--------------+------------+----------+----------
 UPDATE       |       1001 | t        |       50
 INSERT       |       1003 | t        |       10
```

- **MERGE** command supports **WHEN NOT MATCHED BY SOURCE**
  - This operates on rows that exist in the target relation, but not in the data source
- **MERGE** command can modify updatable views

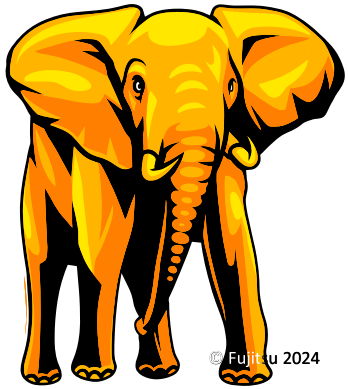


FUJITSU

- Introduced trigger on login event, allowing to fire some actions right on the user connection
  - Useful for logging users login info
  - Can disallow logins for certain duration in a day
  - For verifying the connection and assigning roles according to current circumstances
  - These can be fired on standby servers as well
- Speeded up the serial portion of parallel aggregates and better scales the following in parallel queries:

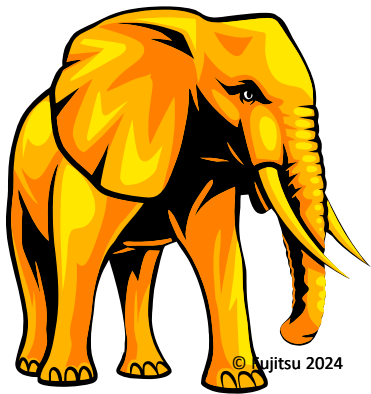
`sum(numeric)`	`avg(numeric)`	`var_pop(numeric)`	`sum(numeric)`
`variance(numeric)`	`stddev_pop(numeric)`	`stddev_samp(numeric)`	`stddev(numeric)`
`array_agg(anyarray)`	`string_agg(text)`	`string_agg(bytea)`	

  - Reduced pallocs and memcpy during deserialization



FUJITSU

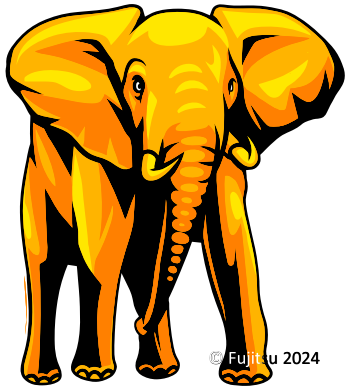
- Introduced 'builtin' collation provider
  - Only the C and C.UTF-8 locales are supported for this provider
  - The C locale behavior is identical to the C locale in the libc provider
  - The C.UTF-8 locale is available only when the database encoding is UTF-8, and the behavior is based on Unicode
    - Faster sorting and case conversion (e.g. LOWER()) as compared to libc variant
  - This new collation ensures that the return values of your sorts won't change, regardless of what system your PostgreSQL installation runs on



Security/SQL 9/9

FUJITSU

- Avoid the need to grant superuser privileges for following
  - pg_maintain role allows executing **`VACUUM`**, **`ANALYZE`**, **`CLUSTER`**, **`REFRESH MATERIALIZED VIEW`**, **`REINDEX`**, and **`LOCK TABLE`** on all relations
  - Alternatively, one can grant **`MAINTAIN`** privilege to users on a table
- Make TLS connections without a network round-trip negotiation
  - Enabled with the client-side option **`sslnegotation=direct`**
  - Requires ALPN
  - Only works on PostgreSQL 17 and later servers
  - PostgreSQL is registered as '**`postgresql`**' in the ALPN directory
- **`ALTER SYSTEM`** improvements
  - Allow **`ALTER SYSTEM`** to set unrecognized custom server variables
  - Add system variable **`allow_alter_system`** to disallow **`ALTER SYSTEM`**
    - Useful in environments where configuration is managed by external tools



SQL/JSON 1/3

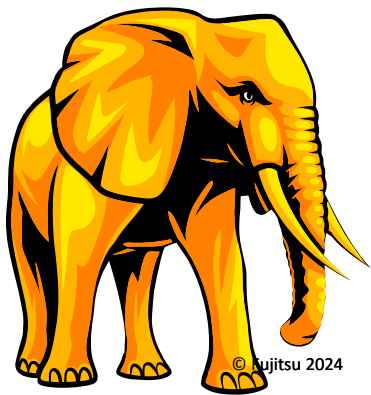
FUJITSU

- Introduced function **`JSON_TABLE()`** to convert JSON data to a table representation

```
CREATE TABLE my_films ( js jsonb );

INSERT INTO my_films VALUES (
'{ "favorites" : [
   { "kind" : "horror", "films" : [
     { "title" : "Psycho",
       "director" : "Alfred Hitchcock" } ] }
 ] }');
SELECT jt.* FROM my_films,
       JSON_TABLE (js, '$.favorites[*]'
         COLUMNS (id FOR ORDINALITY,
                  kind text PATH '$.kind',
                  title text PATH '$.films[*].title’,
                  director text PATH '$.films[*].director')) AS jt;

 id |  kind  | title  |     director
----+--------+--------+------------------
  1 | horror | Psycho | Alfred Hitchcock
```



SQL/JSON 2/3

FUJITSU

- Introduced SQL/JSON constructor functions **`JSON()`**, **`JSON_SCALAR()`**, and **`JSON_SERIALIZE()`**

JSON () ——— Converts a given expression specified as text or bytea string (in UTF8 encoding) into a JSON value

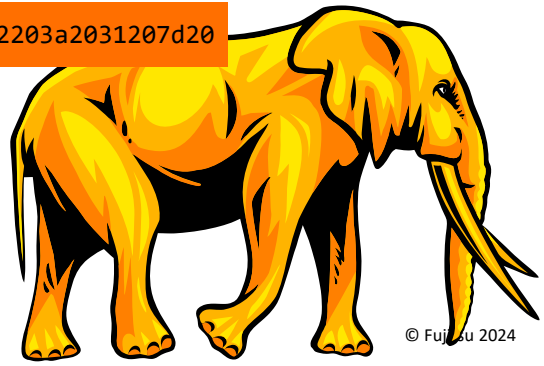
```
JSON('{"a":123, "b":[true,"foo"], "a":"bar"}')  >  {"a":123, "b":[true,"foo"], "a":"bar"}
```

JSON_SCALAR () — Converts a given SQL scalar value into a JSON scalar value

```
JSON_SCALAR(123.45)  >  123.45
```

JSON_SERIALIZE () - Converts an SQL/JSON expression into a character or binary string

```
JSON_SERIALIZE('{ "a" : 1 }' RETURNING bytea)  >  \x7b20226122203a2031207d20
```



SQL/JSON 3/3

FUJITSU

- Introduced SQL/JSON query functions **`JSON_EXISTS()`**, **`JSON_QUERY()`**, and **`JSON_VALUE()`**

JSON_EXISTS () — Returns true if the SQL/JSON path_expression applied to the JSON value yields any items

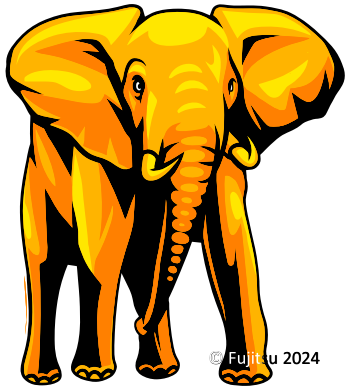
```
SELECT JSON_EXISTS(jsonb '{"key1": [1,2,3]}', '$.key1[2]');
```
> `t`

JSON_QUERY () — Returns the result (JSON, array, or string) of applying the SQL/JSON path_expression to the JSON value

```
SELECT JSON_QUERY(jsonb '{"a": "[1, 2]"}', '$.a');
```
> `[1, 2]`

JSON_VALUE () — Returns the result (SQL/JSON scalar) of applying the SQL/JSON path_expression to the JSON value

```
SELECT JSON_VALUE(jsonb '[1,2]', '$[1]');
```
> `2`



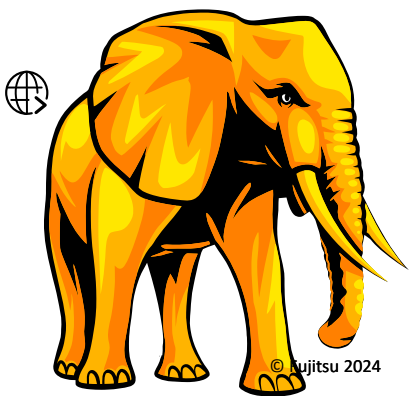
Monitoring 1/2

FUJITSU

- New view pg_wait_events
  - It primarily gives the information on wait event details/description

```
-[ RECORD 1 ]---+-------------------------------------------
pid | 21090
state|
wait_event_type | Activity
wait_event | CheckpointerMain
description | Waiting in main loop of checkpointer process
```

- All checkpointer-related stats could be found in pg_stat_checkpointer
  - Previously, some of this info was stored in pg_stat_bgwriter, which is trimmed now
  - For more information:
    www.postgresql.org/docs/17/monitoring-stats.html#MONITORING-PG-STAT-CHECKPOINTER-VIEW

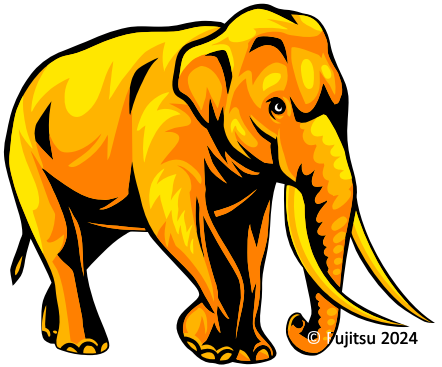


Monitoring

2/2

FUJITSU

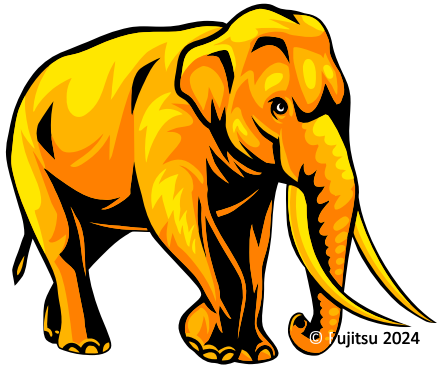
- Index Vacuum progress in pg_stat_progress_vacuum
  - indexes_total: total number of indexes that will be vacuumed or cleaned up
  - indexes_processed: number of indexes for which vacuum has been performed



Backward compatibility 1/1

FUJITSU

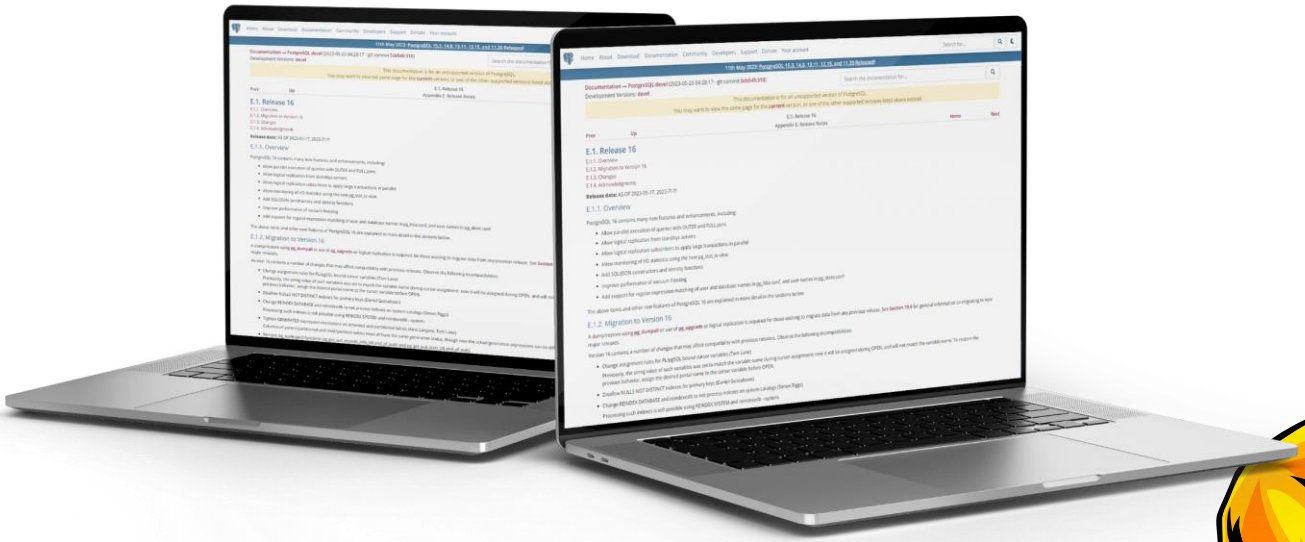
- Removed the parameter old_snapshot_threshold
  - The parameter defines the time threshold for a snapshot during which old row versions will not be deleted
  - When querying the vacuumed rows, PostgreSQL returns “Snapshot too old” error
  - As it turns out, there are issues with the parameter’s implementation, including some performance-related ones
- Change functions to use a safe search_path during maintenance operations
  - While executing maintenance operations (ANALYZE, CLUSTER, REFRESH MATERIALIZED VIEW, REINDEX, or VACUUM), set search_path to 'pg_catalog, pg_temp' to prevent inconsistent behavior
- Remove wal_sync_method value fsync_writethrough on Windows
  - This value was the same as fsync on Windows.
- Remove buffers_backend and buffers_backend_fsync from pg_stat_bgwriter
  - These fields are considered redundant to similar columns in pg_stat_io.

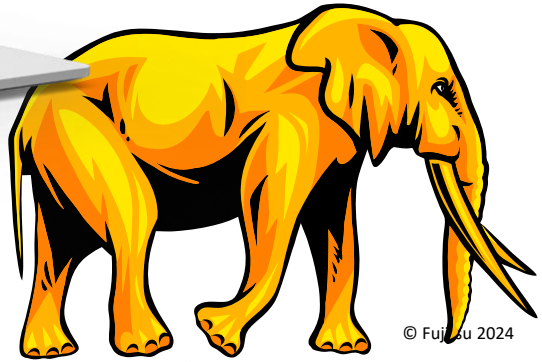


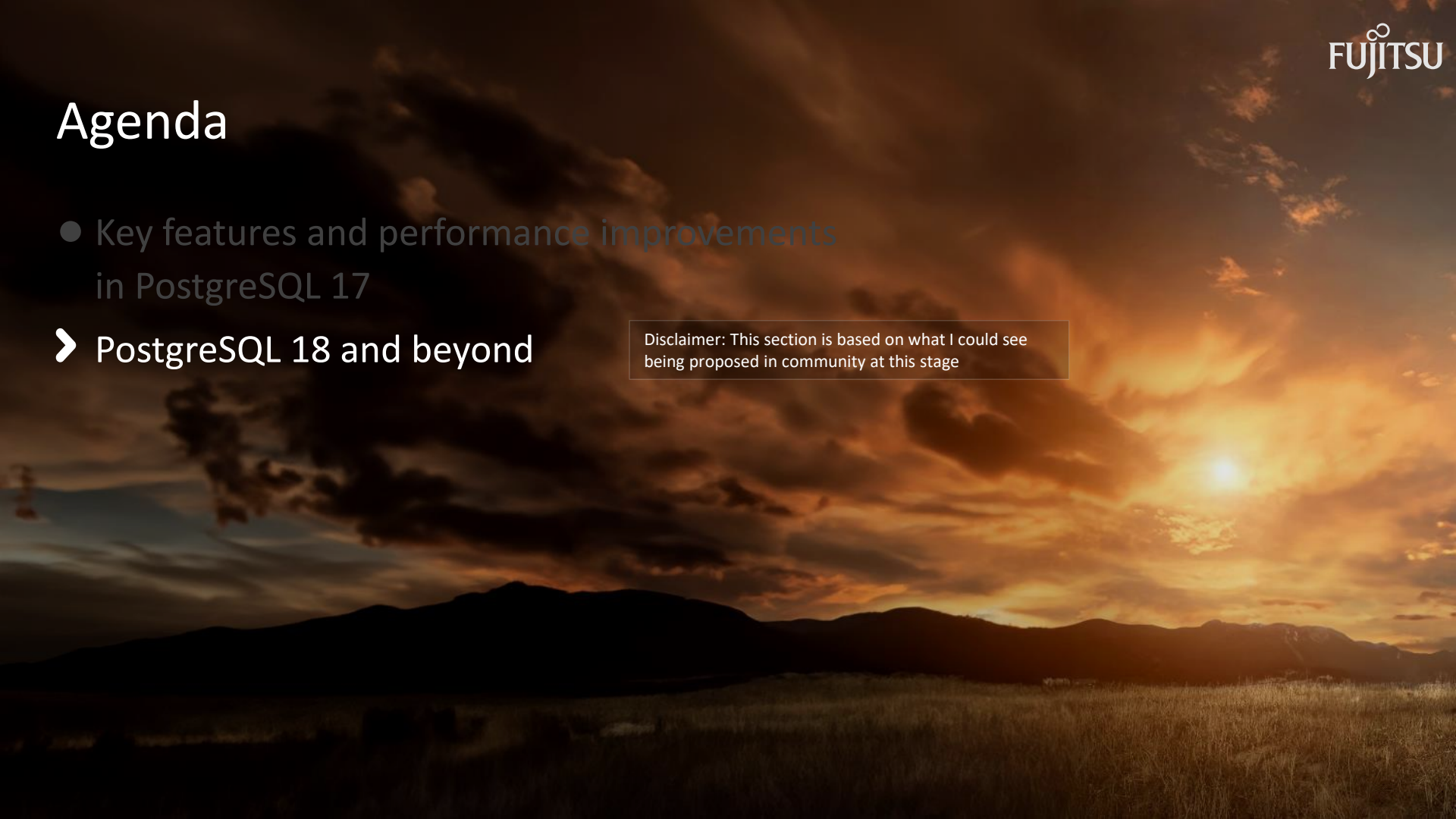
Changes in PostgreSQL 17

FUJITSU

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



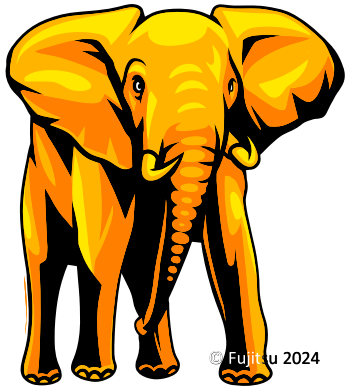



FUJITSU
Agenda
Key features and performance improvements in PostgreSQL 17
PostgreSQL 18 and beyond
Disclaimer: This section is based on what I could see being proposed in community at this stage

PostgreSQL 18 and beyond

FUJITSU

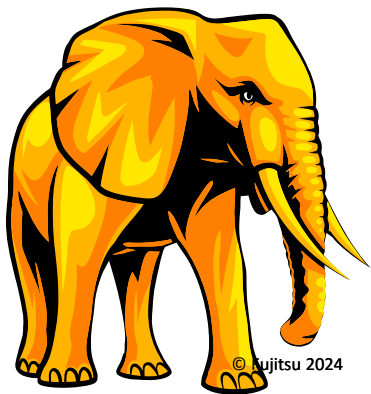
- Asynchronous I/O
  - Index prefetch: This will improve index access performance
  - Will allow prefetching data and will improve system performance
  - Vectored I/O for bulk writes
- Import/Export Statistics
  - This will help to run queries after upgrade without first running Analyze
- Skip Scans in btree
- Allow WITHOUT OVERLAPS clause to PRIMARY KEY and UNIQUE constraints
  - These will be backed by GiST indexes instead of B-tree indexes
- Muti-threaded
  - A very large project but making slow infrastructural improvements



PostgreSQL 18 and beyond

FUJITSU

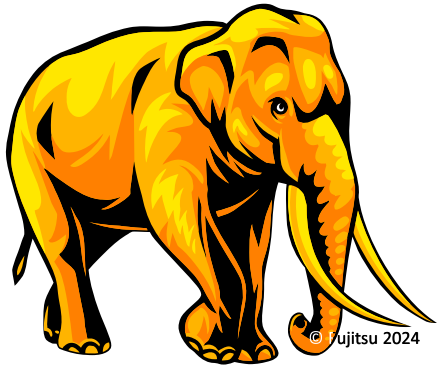
- Various improvements in Logical Replication
  - Replication of sequences
  - Conflict detection and resolution
  - DDL Replication
  - Node management APIs
  - Slot invalidation for unused slots
- Executor improvements
  - Special-case executor expression steps for common combinations (JIT generated code simpler)
  - JIT compilation per plan node
  - SQL standard Row Pattern Recognition (RPR)



PostgreSQL 18 and beyond

FUJITSU

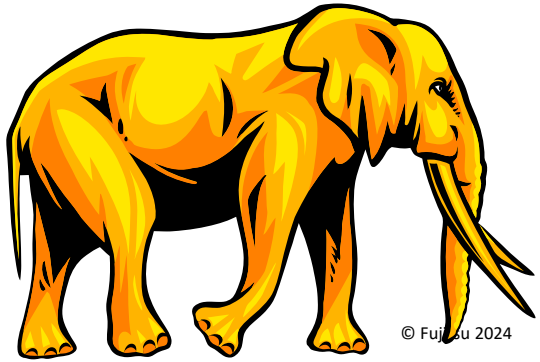
- SQL property graph queries, according to SQL/PGQ standard
- Improvements in partitioning technology, especially in pruning when large number of partitions are present
- Optimizer improvements to make various kind of queries work better
- Read my writes on standby by using pg_wal_replay_wait() stored procedure
- Enhance incremental backups to work for tar format
- Parallelism
  - Parallelize vacuum on tables
  - Parallel Create Index for GIN Indexes
  - Parallelize correlated subqueries
  - TID range scan



PostgreSQL 18 and beyond

FUJITSU

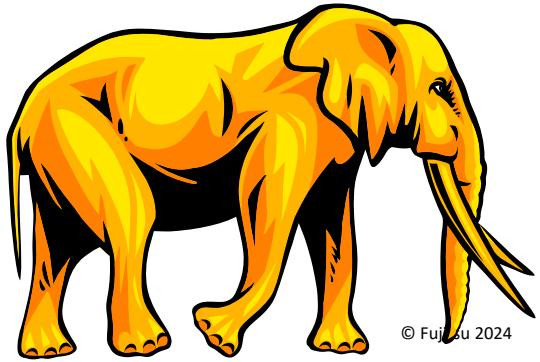
- Transparent column encryption
  - Automatic, transparent encryption and decryption of particular columns in the client
- Introduce compression at wire_protocol_level
- 64bit XIDs
  - Can avoid freezing and reduce the need of autovacuum
- WAL Size reduction
  - Smaller headers in WAL
- TOAST improvements
  - Custom formats
  - Compression dictionaries

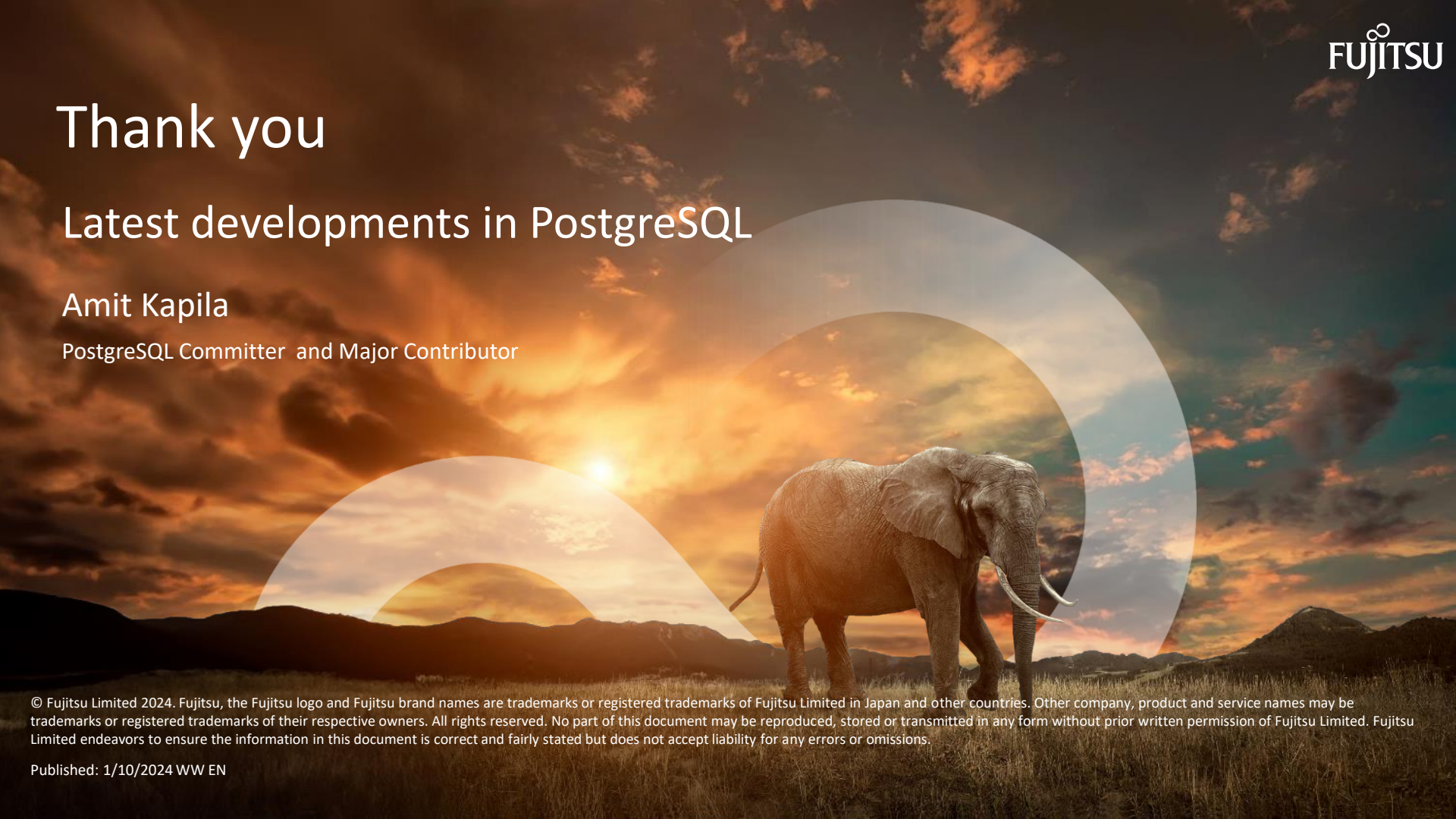


PostgreSQL 18 and beyond

FUJITSU

- Stats
  - Pluggable APIs for Cumulative Statistics. This allows out-of-core extensions to plug their own custom kinds of cumulative statistics.
  - Additional vacuum stats to observe index bloats or other similar unexpected cases
  - Per backend I/O stats
  - More stats
- Enhance Table AM APIs to suite for different storage engines
- CI and build system improvements





FUJITSU

Thank you

Latest developments in PostgreSQL

Amit Kapila

PostgreSQL Committer and Major Contributor

Published: 1/10/2024 WW EN