

Major Features: Postgres 17

BRUCE MOMJIAN



POSTGRESQL is an open-source, full-featured relational database. This presentation gives an overview of the Postgres 17 release.

<https://momjian.us/presentations>



Creative Commons Attribution License

Last updated: November 2024

Postgres 17 Feature Outline

1. Incremental backup
2. Improved data manipulation
3. Improved optimizer handling
4. Improved logical replicas

Full item list at <https://www.postgresql.org/docs/17/release-17.html>.

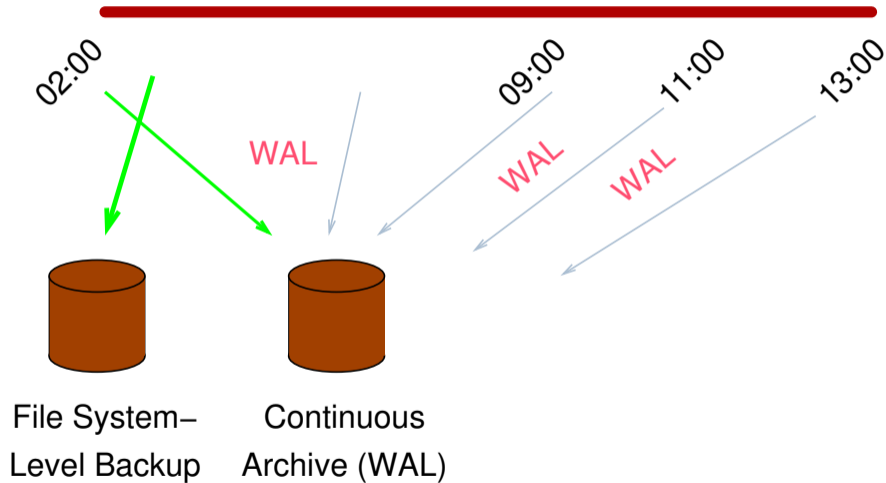
1. Incremental Backup: Backup Methods

Postgres supports three main backup methods:

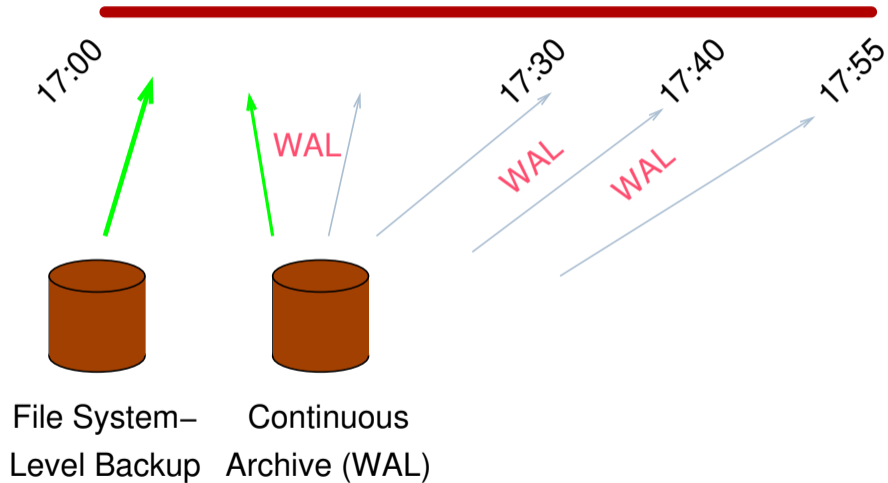
- File system backup/snapshot
- Logical backup, e.g., *pg_dump*
- Continuous archiving

This last method is preferred because it allows recovery to arbitrary times, including up to the most recent transactions. Replicas and delayed-replay replicas are more for failover than backup.

Continuous Archiving



Point-in-Time Recovery



Continuous Archiving Challenges

Continuous archiving requires a file system backup, even one taken while the database is active, plus write-ahead log (WAL) generated from the time of the backup to the restore time. This has some challenges:

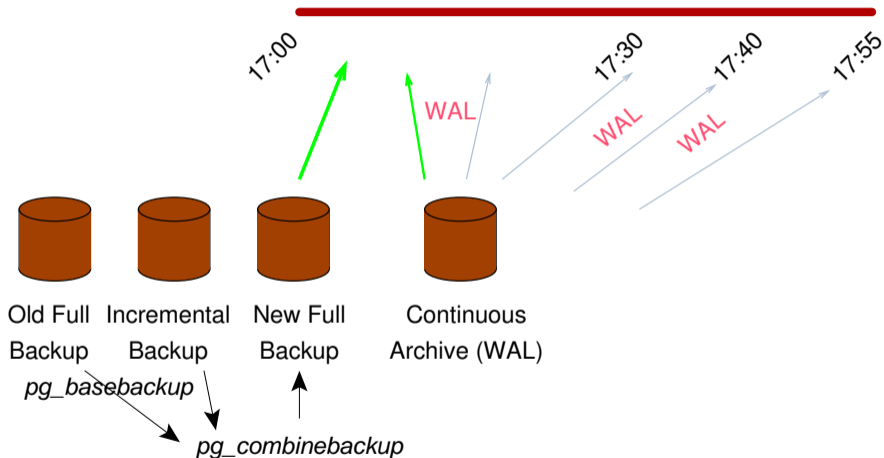
- Replay of the WAL from the time of the backup to the recovery time can be slow
- Replay time can be reduced by requiring FEWER WAL files from being processed
- This can be accomplished with more frequent file system backups
- Unfortunately file system backups are large and require a lot of I/O

Incremental Backups

Postgres's incremental backup feature solves these problems

- Can create an incremental file system backup via *pg_basebackup*
 - only records data blocks modified since the last full or incremental backup
- Combine a full backup with incremental backups to create a newer full backup, via *pg_combinebackup*
 - requires fewer WAL files to restore
 - faster restores
- Combining does not need access to the data directory so it can be done on a separate server
- Existing backup tools could already produce incremental backups, but with poorer granularity or more overhead, e.g. *pg_backrest*, *Barman*

Point-in-Time Recovery with Incremental Backup



pg_combinebackup

-- apply one incremental to create a new full backup

-- full_backup2 is now current as of the end of incremental1's backup

```
$ pg_combinebackup full_backup1 incremental1 -o full_backup2
```

-- apply three incrementals to create a new full backup

```
$ pg_combinebackup full_backup2 incremental2 incremental3 incremental4 -o full_backup3
```

New Deployment Options

- Keep the base backup unchanged
 - create many incremental backups
 - incremental recovery is faster than WAL replay
 - discard WAL for time spans that don't need granular recovery
- Keep the base backup current
 - use *pg_combinebackup* to keep the base backup current by applying frequent incremental backups
 - this is only possible when desired recovery point is short
 - at a rapid frequency, it starts to have options like replicas, but with heavy I/O overhead
- Make a copy of the base backup
 - keep the copy current by applying incremental backups frequently
 - this can greatly reduce recovery time because there is minimal WAL to replay
 - this does not reduce the recovery time window because the old base backup, and necessary WAL, are kept

pg_combinebackup Removal

-- apply one incremental to create a new full backup

```
$ pg_combinebackup full_backup3 incremental5 -o full_backup4
```

-- If we don't need to recover to any time earlier than the end of incremental5,

-- we can delete the previous full backup and incremental5, and all needed WAL

```
$ rm -r full_backup3 incremental5
```

-- do it again

```
$ pg_combinebackup full_backup4 incremental6 -o full_backup5
```

```
$ rm -r full_backup4 incremental6
```

2. Improved Data Manipulation

1. JSON_TABLE()
2. COPY
3. MERGE

2.1 JSON_TABLE()

```
SELECT *
FROM JSON_TABLE('{ "key1": "val1" }',
                '$.key1' COLUMNS (col1 text PATH '$'));
```

col1

val1

```
SELECT *
FROM JSON_TABLE('{ "key1": "val1", "key2": "val2" }',
                '$[*]' COLUMNS (key1 text PATH '$.key1', key2 text PATH '$.key2'));
```

key1 | key2

-----+-----

val1 | val2

Load JSONB Data

```
-- download sample data from https://www.mockaroo.com/  
-- remove 'id' column, output as JSON, uncheck 'array'  
CREATE TABLE friend (id SERIAL, data JSONB);
```

```
COPY friend (data) FROM '/tmp/MOCK_DATA.json';
```

```
SELECT *  
FROM friend  
ORDER BY 1  
LIMIT 2;
```

id	data
1	{"email": "sbouzan0@wikispaces.com", "gender": "Female", ...
2	{"email": "ebruffell1@independent.co.uk", "gender": "Male", ...

Pretty Print JSON

```
SELECT id, jsonb_pretty(data)
FROM friend
ORDER BY 1
LIMIT 1;
```

id	jsonb_pretty
1	{ "email": "sbouzan0@wikispaces.com", "gender": "Female", "last_name": "Bouzan", "first_name": "Sher", "ip_address": "89.153.16.253" }

JSON_TABLE()

```
SELECT json.first_name, json.last_name, json.email
FROM friend, JSON_TABLE(data,
                        '$[*]' COLUMNS (first_name TEXT PATH '$.first_name',
                                         last_name TEXT PATH '$.last_name',
                                         email TEXT PATH '$.email')) AS json
ORDER BY random()
LIMIT 5;
```

first_name	last_name	email
Abbey	Terbeck	aterbeckf7@latimes.com
Giustino	Weeke	gweekerm@fastcompany.com
Bailey	Romi	bromibc@desdev.cn
Guillemette	Hastwell	ghastwell161@microsoft.com
Cherise	Biermatowicz	cbiermatowicz3c@google.com.hk

2.2 COPY Error Handling

```
CREATE TABLE copy_test (int_col INTEGER, date_col DATE, jsonb_col JSONB);
```

```
COPY copy_test (int_col) FROM STDIN;
```

```
test> 1
```

```
test> 2
```

```
test> x
```

```
test> \.
```

```
ERROR:  invalid input syntax for type integer: "x"
```

```
CONTEXT:  COPY copy_test, line 3, column int_col: "x"
```

```
SELECT *
```

```
FROM copy_test;
```

```
int_col | date_col | jsonb_col
```

```
-----+-----+-----
```

COPY Ignore Errors

```
COPY copy_test (int_col) FROM STDIN WITH (ON_ERROR ignore);
test> 3
test> 4
test> a
test> \.
NOTICE: 1 row was skipped due to data type incompatibility
```

```
SELECT * FROM copy_test;
 int_col | date_col | jsonb_col
-----+-----+-----
      3 | (null)  | (null)
      4 | (null)  | (null)
```

COPY Report Error Rows

```
COPY copy_test (int_col) FROM STDIN WITH (ON_ERROR ignore, LOG_VERBOSITY verbose);
```

```
test> 5
```

```
test> 6
```

```
test> b
```

```
test> \.
```

```
NOTICE: skipping row due to data type incompatibility at line 3 for column "int_col": "b"
```

```
NOTICE: 1 row was skipped due to data type incompatibility
```

```
SELECT * FROM copy_test;
```

```
int_col | date_col | jsonb_col
```

```
-----+-----+-----  
3 | (null) | (null)  
4 | (null) | (null)  
5 | (null) | (null)  
6 | (null) | (null)
```

2.3 MERGE Improvements

- Allow MERGE to modify updatable views
- Add WHEN NOT MATCHED BY SOURCE
- Allow MERGE to use the RETURNING clause

3. Improved Optimizer Handling

1. CTE passdown
2. NULL optimizations
3. Correlated subquery optimization
4. Other

3.1 CTE Passdown

```
WITH RECURSIVE dep (classid, obj) AS (  
    SELECT (SELECT oid FROM pg_class WHERE relname = 'pg_class'),  
           oid  
    FROM pg_class  
    WHERE relname = 'deptest'  
    UNION ALL  
    SELECT pg_depend.classid, objid  
    FROM pg_depend JOIN dep ON (refobjid = dep.obj)  
)  
-- statistics and sort order are passed down here  
SELECT (SELECT relname FROM pg_class WHERE oid = classid) AS class,  
       (SELECT typename FROM pg_type WHERE oid = obj) AS type,  
       (SELECT relname FROM pg_class WHERE oid = obj) AS class,  
       (SELECT relkind FROM pg_class where oid = obj::regclass) AS kind,  
       (SELECT pg_get_expr(adbin, classid) FROM pg_attrdef WHERE oid = obj) AS attrdef,  
       (SELECT conname FROM pg_constraint WHERE oid = obj) AS constraint  
FROM dep  
ORDER BY obj;
```

3.2 NULL Optimizations

```
CREATE TABLE null_test (not_null_col INTEGER NOT NULL,  
                        opt_null_col INTEGER);
```

```
-- This disables EXPLAIN cost output  
\set EXPLAIN 'EXPLAIN (COSTS OFF)'
```

```
:EXPLAIN SELECT *  
FROM null_test;  
      QUERY PLAN
```

```
-----  
Seq Scan on null_test
```

```
-- no sequential scan  
:EXPLAIN SELECT *  
FROM null_test  
WHERE not_null_col IS NULL;  
      QUERY PLAN
```

```
-----  
Result  
One-Time Filter: false
```

NULL Optimizations

```
:EXPLAIN SELECT *  
FROM null_test  
WHERE opt_null_col IS NOT NULL;  
      QUERY PLAN
```

```
-----  
Seq Scan on null_test  
  Filter: (opt_null_col IS NOT NULL)
```

```
-- no 'Filter' clause  
:EXPLAIN SELECT *  
FROM null_test  
WHERE not_null_col IS NOT NULL;  
      QUERY PLAN
```

```
-----  
Seq Scan on null_test
```


3.3 Correlated Subqueries in Postgres 16

```
:EXPLAIN SELECT COUNT(*)  
FROM pg_class  
WHERE oid IN (  
    SELECT attrelid  
    FROM pg_attribute  
    WHERE attrelid = pg_class.oid  
);
```

QUERY PLAN

Aggregate

-> Seq Scan on pg_class

Filter: (SubPlan 1)

SubPlan 1

-> Index Only Scan using pg_attribute_relid_attnum_index on pg_attribute

Index Cond: (attrelid = pg_class.oid)

Correlated Subqueries Now as Joins

```
:EXPLAIN SELECT COUNT(*)  
FROM pg_class  
WHERE oid IN (  
    SELECT attrelid  
    FROM pg_attribute  
    WHERE attrelid = pg_class.oid  
);
```

QUERY PLAN

Aggregate

-> Hash Join

Hash Cond: (pg_class.oid = pg_attribute.attrelid)

-> Seq Scan on pg_class

-> Hash

-> HashAggregate

Group Key: pg_attribute.attrelid, pg_attribute.attrelid

-> Seq Scan on pg_attribute

3.4 Other Optimizer Improvements

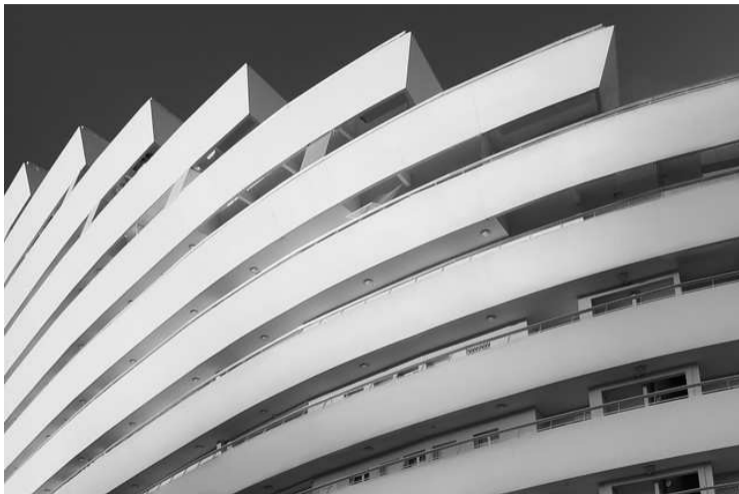
- Partition pruning for boolean columns
- Range value containment
- Optimize LIMIT on partitions
- Allow GROUP BY to be reordered to match ORDER BY
- Improve Merge Append *
- More parallelism

* <https://momjian.us/main/presentations/performance.html#beyond>

4. Improved Logical Replication

- Add tool *pg_createsubscriber* to allow creation of logical replicas from physical ones
- Enable failover of logical replication slots
- Enable *pg_upgrade* to preserve logical replication slots in future major upgrades

Conclusion



<https://momjian.us/presentations>

<https://www.flickr.com/photos/thomasletholzen/>