credativ

Fast & Prepared

Prepared Statements in PostgreSQL

Josef Machytka <josef.machytka@credativ.de>

2025-11-14 - credativ Tech Talk

credativ GmbH

- Founded 1999 in Jülich, Germany
- Close ties to Open-Source Community
- More than 40 Open-Source experts
- Consulting, development, training, support (3rd-level / 24x7)
- Open-Source infrastructure with Linux, Kubernetes, Proxmox
- Open-Source databases with PostgreSQL
- DevSecOps with Ansible, Puppet, Terraform and others
- Since 2025 independent owner-managed company again

Josef Machytka

credativ

- Professional Service Consultant - PostgreSQL specialist at credativ GmbH
- 33+ years of experience with different databases
- PostgreSQL (13y), BigQuery (7y), Oracle (15y), MySQL (12y), Elasticsearch (5y), MS SQL (5y)
- 10+ years of experience with Data Ingestion pipelines, Data Analysis, Data Lake and Data Warehouse
- 3+ years of practical experience with different LLMs / AI / ML including architecture and principles
- From Czechia, living now 12 years in Berlin

- LinkedIn: linkedin.com/in/josef-machytka
- Medium: medium.com/@josef.machytka
- YouTube: youtube.com/@JosefMachytka

- GitHub: github.com/josmac69/conferences_slides
- ResearchGate: researchgate.net/profile/Josef-Machytka
- Academia.edu: academia.edu/JosefMachytka
- sessionize: sessionize.com/josefmachytka

All My Slides:

Recoded talks:

Topics Covered

- Why, what, when to use, when to avoid
- Protocol-level vs SQL commands
- EXPLAIN, plan caching, generic vs custom plans
- Prepared statements & partitioned tables
- Internal implementation overview
- Plan invalidation & replanning
- Hidden usage in FDWs, PL/pgSQL
- Common use cases

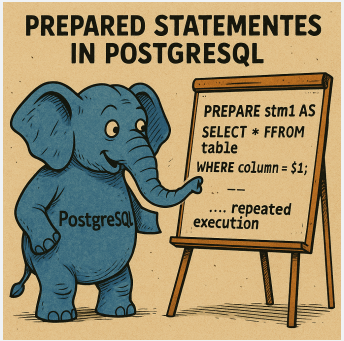

Image credit ChatGPT

Why Prepared Statements?

- What does postgres do with a query?
  - Parses the SQL statement
  - Analyzes it
  - Rewrites it
  - Plans it
  - Executes the plan & Returns the result
- Every step increases total query runtime
- Prepared statements can optimize this process
  - Parse, Analyze, Rewrite, Plan only once
  - Execute many times

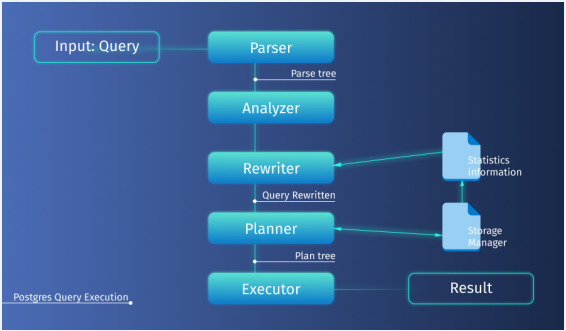

Image from the article
PostgreSQL EXPLAIN for Analyzing Query Execution Plans

What Are These Prepared Statements Anyway?

credativ

- Prepared statements are stored in a server-side cache
- They are kept in memory, session-scoped
- Survive across transactions & multiple queries

- **Server-side** prepared statements:
  - - managed by PostgreSQL server -
  - - created using SQL commands PREPARE, EXECUTE, DEALLOCATE
- **Client-side** prepared statements:
  - - managed by client libraries
  - - emulated inside library, not created on server

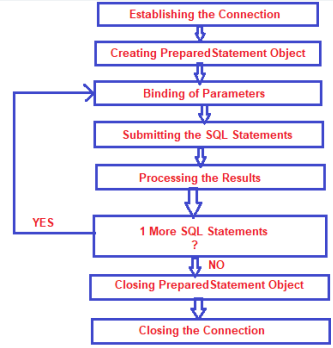

Image from the article
Prepared Statement in JDBC

Prepared Statements vs Prepared Transactions?

credativ

- **Prepared Statements** are cached query plans
- They are performance optimization feature

- **Prepared Transactions** are 2-phase commit transactions
- Prepared Transactions are managed with:
  - PREPARE TRANSACTION;
  - COMMIT PREPARED;
  - ROLLBACK PREPARED;
- They are internal PostgreSQL feature for distributed transactions
- Intended for ensuring atomicity across multiple databases
- Not related to Prepared Statements

When Shall We Use Prepared Statements?

- When a single session executes the same query multiple times
- Just with different parameters
- When performance is critical
- And query planning overhead needs to be minimized
- Advantages of using prepared statements:
  - Prevent repeated parsing & planning overhead
  - Prevent SQL injection attacks
  - Reduce network traffic
  - Improve code readability

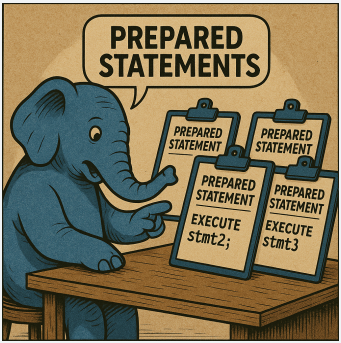

Image credit: ChatGPT

When Shall We NOT Use Prepared Statements?

- When queries are executed only once
- When queries are highly dynamic
- When query planning time is negligible
- When using connection pooling
- When prepared statement cache size is limited
- When connection lifetime is short
- Gotchas:
  - Can switch to bad plans over time
  - After 5th execution, re-planning happens (default)
  - Generic plans can be much less efficient
  - Increased memory usage with many prepared statements

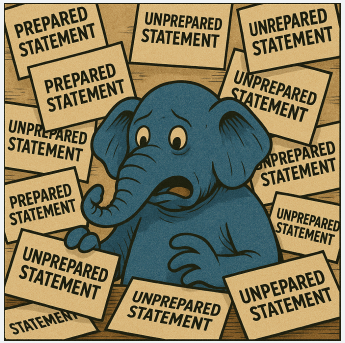

Image credit: ChatGPT

SQL commands vs Protocol-level

credativ

- **SQL-level commands** - standard SQL commands:
  - PREPARE, EXECUTE, DEALLOCATE
  - Must be parsed as text to be understood
- **Protocol-level commands** - ”wire protocol” between client & server
  - Parse - create a prepared statement on the server
  - Bind - attach parameters to a prepared statement
  - Execute - execute the prepared statement with parameters
  - Close - deallocate the prepared statement (added in PostgreSQL 17)
- See in PostgreSQL documentation Chapter 54. Frontend/Backend Protocol
- Part 54.2.3. Extended Query describes prepared statements
- Most client libraries use protocol-level commands
- PgBouncer can intercept protocol-level commands & handle them in transaction mode

Prepared Statements and Connection Pooling

- Prepared statements are session-scoped
- Some poolers do not support them in transaction mode
- PgBouncer supports protocol-level prepared statements
- But not SQL-level prepared statements
- Feature implemented in versions 1.21 and 1.22
- Functionality requires PostgreSQL 17 or higher

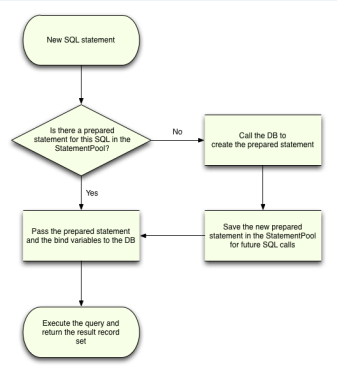

Image from the article
Show some love for
prepared statements in Rails 3.1

View: pg_prepared_statements

credativ

- System view showing prepared statements in current session
- Columns:
  - name - name of the prepared statement
  - statement - original query string
  - prepare_time - timestamp of when it was prepared
  - parameter_types - array of parameter data types
  - result_types - array of result column data types
  - from_sql - boolean indicating if prepared via SQL commands (true) or protocol (false)
  - generic_plans - number of times generic plan was used
  - custom_plans - number of times custom plan was used
- Useful for monitoring prepared statements usage

EXPLAIN command & Prepared Statements

credativ

- EXPLAIN shows the execution plan of a query for specific parameters
- We can test prepared statements with new GENERIC_PLAN option

```
CREATE TABLE test (id int PRIMARY KEY, data text);
-- Insert data, run ANALYZE

EXPLAIN GENERIC_PLAN SELECT * FROM test WHERE id = \$1;

PREPARE my_stmt (int) AS SELECT * FROM test WHERE id = \$1;

SET plan_cache_mode = 'force_generic_plan';
EXPLAIN EXECUTE my_stmt (42);

SET plan_cache_mode = 'force_custom_plan';
EXPLAIN EXECUTE my_stmt (42);
```

Plan Caching & Heuristics

credativ

- plan_cache_mode (default: auto)
  - Controls the behavior of the plan cache for prepared statements
  - Options: auto, force_custom_plan, force_generic_plan
  - auto = plan is reevaluated after 5 executions
- First 5 executions use custom plans (plan_cache_mode = auto)
- On 6th execution, PostgreSQL compares generic vs average custom plan costs

Plan Caching & Heuristics

credativ

- **Generic plan**
  - Parameter agnostic - "one size fits all"
  - Uses average distribution of parameter values
  - After creation does not require replanning (no planning costs)
  - Generic plan cannot prune partitions - validates/locks all partitions
- **Custom plan**
  - Optimized for specific parameter values based on statistics
  - Code includes costs of planning into custom plan costs
- Index scan vs sequential scan flip is common difference
- PostgreSQL switches to generic plan when:
  - Generic plan costs are cheaper than execution + replanning costs of custom plans
- For table with many partitions force custom plan

Prepared Statements and Partitioned Tables

credativ

- Partitioned tables have significant difference between generic & custom plans
- Custom plans can prune partitions based on parameter values
- During PREPARE/Parse backend will:
  - Parse-analyze the query
  - Create parse tree in a cached plan structure
  - But no parameter values are known yet
- During EXECUTE/Bind:
  - It will create the actual plan
  - With custom plan partitions can be pruned now

- Generic plans must validate/lock all partitions
- This can lead to significant overhead with many partitions
- Recommendation: force custom plans for partitioned tables

- Postgres: on performance of prepared statements with partitioning

What PostgreSQL Does Internally

credativ

- When PREPARE or protocol Parse command is received:

- Name is validated (not empty, not duplicate)
- Internal memory structure are prepared - raw parse tree etc.
- Parameters types are determined and converted into OIDs
- Query is parsed, analyzed, and rewritten
- Plan is completed and stored in plansource structure
- All metadata is stored in PreparedStatement hash table

What PostgreSQL Does Internally

credativ

- When EXECUTE or protocol Execute command is received:

- PreparedStatement hash table is looked up by name
- Parameters are evaluated and converted to internal format
- In case of partitioned table, pruned plan is created now
- Process creates ”portal” for execution - query instance with parameters
- Cached plan is obtained from plansource structure
- Query ”portal” is associated with the plan
- Portal is executed, results returned, portal deallocated

Internal Implementation

credativ

- `src/backend/commands/prepare.c`
- Implements SQL commands PREPARE, EXECUTE, DEALLOCATE
- As well as memory structures for prepared statements
- Defines per-backend hash table to store prepared statements
- Structure `PreparedStatement` holds:
  - Name of the prepared statement
  - Cached query plan (plansource structure - all details of the plan, quite big)
  - Boolean - prepared via SQL (true) or protocol (false)
  - Timestamp of creation

Internal Implementation

credativ

- Plan cache management is in `src/backend/utils/cache/plancache.c`
  - Decides when to use generic vs custom plans
  - Tracks schema changes invalidating cached plans
  - Defines plansource structure as well
- Prepared statements memory structures - use several memory contexts:
  - Source context - holds CachedPlanSource and parse tree
  - Query context - holds analyzed and rewritten query tree & dependency data
  - Plan context - for each CachedPlan, created by BuildCachedPlan
- Memory contexts are cleaned up when prepared statement is deallocated

Implementation of protocol-level Prepared Statements

credativ

- Implemented in the libpq client library
- Functions:
  - `PQprepare()` - prepares a statement - sends Parse command to server
  - `PQexecPrepared()` - executes a prepared statement - sends Bind & Execute
  - `PQdescribePrepared()` - sends Describe command to get metadata about prepared statement
  - `PQclear()` - deallocates a prepared statement - sends Close command to server (PostgreSQL 17+)
- Connection libraries use these functions to manage prepared statements
- Described in PostgreSQL documentation 32.3. Command Execution Functions

Plan invalidation & Replanning

credativ

- Cached plans can become invalid due to schema changes
- Code in `plancache.c` registers ”invalidation callbacks” on system catalog
- Processed by `src/backend/storage/ipc/sinval.c` code -> ”sinval” events
- When an invalidation event matches a cached plan’s dependencies, it is marked as invalid
- During next execution, invalid plan is re-analyzed or re-planned
- Usage of generic vs custom plan is re-evaluated based on current parameters
- New statistics are stored in the plansource structure
- New numbers are reported in pg_prepared_statements view

Hidden Usage of Prepared Statements

credativ

- Some PostgreSQL features use prepared statements internally
- Example is postgres_fdw foreign data wrapper
- See in `contrib/postgres_fdw/postgres_fdw.c`
- It creates prepared statements for execution of INSERT/UPDATE/DELETE
- This optimizes performance when accessing foreign tables

- PL/pgSQL code automatically turns static queries into prepared statements
- This improves performance of repeated executions of functions
- Dynamic statements must be parsed each time, cannot be prepared

Use Cases of Prepared Statements

credativ

- Web applications with high frequency of SELECT queries
  - Same SELECTs with different parameters
  - Reduces parsing/planning overhead significantly (up to 36% in some benchmarks)
- Data ingestion pipelines / ETL with repeated INSERT/UPDATE operations
  - Prepared statements speed up data loading by reusing plans
  - Protect against SQL injection attacks
- Repeated complex analytical queries in reporting applications
  - Complicated queries often have high planning overhead
  - Planning often takes longer than execution

Additional Resources

- Articles:
  - PostgreSQL Performance Insights: Extended Query Protocol vs. Simple Query Protocol
  - EXPLAIN (GENERIC_PLAN): New in PostgreSQL 16
  - Puzzling Postgres: solving an unreproducible performance issue
  - Supporting Postgres Named Prepared Statements in Hyperdrive
  - When are PostgreSQL prepared statements fully prepared for execution?
  - More Memory, More Problems (generic plans incident)

- Talks & Videos:
  - Comparing Postgres connection pooler support for prepared statements | POSETTE 2024

Thank you for your attention!

All my slides

Recorded talks