Most popular DBs https://db-engines.com/en/ranking

PostgreSQL

What is PostgreSQL and why do enterprise developers and start-ups love it?

Kris Sharma

PostgreSQL is a powerful, open source object-relational database system that is known for reliability, feature robustness, and performance. PostgreSQL is becoming the preferred database for more and more enterprises. It is currently ranked #4 in popularity amongst hundreds of databases worldwide according to the DB-Engines Ranking.

The basics first – What is PostgreSQL?

PostgreSQL is a relational database. It stores data points in rows, with columns as different data attributes. A table stores multiple related rows. The relational database is the most common type of database in use. It differentiates itself with a focus on integrations and extensibility. It works with a lot of other technologies and conforms to various database standards, that ensures it is extensible.

In recent years, many companies have officially supported the development of the PostgreSQL project. Let's dig deeper into why it is gaining popularity.

Why use PostgreSQL?

An enterprise class database, PostgreSQL boasts sophisticated features such as Multi-Version Concurrency Control (MVCC), point in time recovery, tablespaces, asynchronous replication, nested transactions, online/hot backups, a sophisticated query planner/optimiser, and write ahead logging for fault tolerance.
PostgreSQL works on most popular operating systems – almost all Linux and Unix distributions, Windows, Mac OS X. Its open source nature makes it easy to upgrade or extend. In PostgreSQL, you can define your own data types, build custom functions, and even write code in another programming language (e.g. Python) without recompiling the database. And, of course, PostgreSQL is free!

Reliable database

PostgreSQL isn't just relational, it's object-relational and supports complex structures and a breadth of built-in and user-defined data types. It provides extensive data capacity and is trusted for its data integrity. This gives it some advantages over other open source SQL databases like MySQL, MariaDB and Firebird. PostgreSQL comes with many features aimed to help developers build applications, administrators to protect data integrity and build fault-tolerant environments.

It offers its users a huge (and growing) number of functions. These help programmers to create new applications, admins better protect data integrity, and developers build resilient and secure environments. PostgreSQL gives its users the ability to manage data, regardless of how large the database is.

Extensible database

In addition to being free and open source, PostgreSQL is highly extensible. There are two areas that PostgreSQL shines when users need to configure and control their database. First, it is compliant to a high degree with SQL standards. This increases its interoperability with other applications.

Second, PostgreSQL gives users, control over the metadata. PostgreSQL is extensible because its operation is catalog-driven. One key difference between PostgreSQL and standard relational database systems is that PostgreSQL stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. These tables can be modified by the user, and since PostgreSQL bases its operation on these tables, this means that PostgreSQL can be extended by users. By comparison, conventional database systems can only be extended by changing hard-coded procedures in the source code or by loading modules specially written by the DBMS vendor.

How is PostgreSQL used?

PostgreSQL has a rich history for support of advanced data types, and supports a level of performance optimisation that is usually associated with commercial database counterparts, like Oracle and SQL Server. PostgreSQL is used as the primary data store or data warehouse for many web, mobile, geospatial, and analytics applications.
PostgreSQL can store structured and unstructured data in a single product. Unstructured data, found in audio, video, emails and social media postings, can be used to improve customer service, discover new product requirements, and find ways to prevent a customer from churning among countless other uses.

PostgreSQL also has superior online transaction processing capabilities (OLTP) and can be configured for automatic fail-over and full redundancy, making it suitable for financial institutions and manufacturers. As a highly capable analytical database it can be integrated effectively with mathematical software, such as Matlab and R. Due to PostgreSQL's replication capabilities, websites can easily be scaled out to as many database servers as you need.

PostgreSQL when used with the PostGIS extension, supports geographic objects, and can be used as a geospatial data store for location based services and geographic information systems (GIS).

Day-N operational challenges of using PostgreSQL

Despite the benefits, there are challenges that enterprises shall be faced with when it comes to PostgreSQL adoption. PostgreSQL has one of the fastest-growing communities but unlike traditional database vendors, the PostgreSQL community does not have the luxury of a mature database ecosystem. In addition, PostgreSQL is often used in tandem with several different databases, such as Oracle or MongoDB and each database requires specialised expertise and hiring technical staff with relevant PostgreSQL skill set can be a challenge for enterprises. In addition to management tools for PostgreSQL, DevOps teams and database professionals need to be able to manage multiple databases from multiple vendors without having to change existing processes.
Second, as PostgreSQL is open-source, different IT development teams within an organisation may start using it organically. This can give rise to another challenge – no single point of knowledge for all instances of PostgreSQL in enterprise IT landscape. Further, there is redundancy and duplication of work, as different teams may be solving the same problem with it, independently.

How can start-ups and larger enterprises deal with these challenges? Let's look at a Canonical offering that aims to solve this.

Optimised PostgreSQL, managed for you

Managed PostgreSQL from Canonical is a trusted, secure, and scalable database service deployable on the cloud of your choice or on-prem. Never worry about maintenance operations or updates – we handle that for you. With Canonical's managing your PostgreSQL, you get the following benefits.

- Canonical's PostgreSQL experts manage your database servers. You do not have to go through the delay and difficulties of hiring DevOps engineers who know how to stand up a high availability cluster.
- Canonical's open source app management team does the heavy operational lifting so you can focus on building your applications.
- Canonical will manage PostgreSQL on any conformant Kubernetes on the cloud of your choice or on-premise. This means, you get to bring your cloud, and we handle the rest.

Summary

PostgreSQL, an advanced enterprise class open source relational database backed by over 30 years of community development is the backbone to many key technologies and apps we use each day. Canonical supports PostgreSQL through a fully managed database service that automates the mundane task of application operations so enterprises and developers can focus on building their core apps with PostgreSQL.

MySQL vs PostgreSQL -- Choose the Right Database for Your Project

Krasimir Hristozov

The choice of a database management system is usually an afterthought when starting a new project, especially on the Web. Most frameworks come with some object-relational mapping tool (ORM) which more or less hides the differences between the different platforms and makes them all equally slow. Using the default option (MySQL in most cases) is rarely wrong, but it's worth considering. Don't fall into the trap of familiarity and comfort – a good developer must always make informed decisions among the different options, their benefits and drawbacks.

Database Performance

Historically, MySQL has had a reputation as an extremely fast database for read-heavy workloads, sometimes at the cost of concurrency when mixed with write operations.

PostgreSQL, also known as Postgres, advertises itself as "the most advanced open-source relational database in the world". It was built to be feature-rich, extendable and standards-compliant. In the past, Postgres performance was more balanced - reads were generally slower than MySQL, but it was capable of writing large amounts of data more efficiently, and it handled concurrency better.

The performance differences between MySQL and Postgres have been largely erased in recent versions. MySQL is still very fast at reading data, but only if using the old MyISAM engine. If using InnoDB (which allows transactions, key constraints, and other important features), differences are negligible (if they even exist). These features are absolutely critical to enterprise or consumer-scale applications, so using the old engine is not an option. On the other hand, MySQL has also been optimized to reduce the gap when it comes to heavy data writes.

When choosing between MySQL and PostgreSQL, performance should not be a factor for most run-of-the-mill applications – it will be good enough in either case, even if you consider expected future growth. Both platforms are perfectly capable of replication, and many cloud providers offer managed scalable versions of either database. Therefore, it's worth it to consider the other advantages of Postgres over MySQL before you start your next project with the default database setting.

Postgres Advantages over MySQL

Postgres is an object-relational database, while MySQL is a purely relational database. This means that Postgres includes features like table inheritance and function overloading, which can be important to certain applications. Postgres also adheres more closely to SQL standards.

Postgres handles concurrency better than MySQL for multiple reasons:

Postgres implements Multiversion Concurrency Control (MVCC) without read locks Postgres supports parallel query plans that can use multiple CPUs/cores Postgres can create indexes in a non-blocking way (through the CREATE INDEX CONCURRENTLY syntax), and it can create partial indexes (for example, if you have a model with soft deletes, you can create an index that ignores records marked as deleted) Postgres is known for protecting data integrity at the transaction level. This makes it less vulnerable to data corruption.

Default Installation and Extensibility of Postgres and MySQL

The default installation of Postgres generally works better than the default of MySQL (but you can tweak MySQL to compensate). MySQL has some outright weird default settings (for example, for character encoding and collation).
Postgres is highly extensible. It supports a number of advanced data types not available in MySQL (geometric/GIS, network address types, JSONB which can be indexed, native UUID, timezone-aware timestamps). If this is not enough, you can also add your own datatypes, operators, and index types.
Postgres is truly open-source and community-driven, while MySQL has had some licensing issues. It was started as a company product (with a free and a paid version) and Oracle's acquisition of MySQL AB in 2010 has led to some concerns among developers about its future open source status. However, there are several open source forks of the original MySQL (MariaDB, Percona, etc.), so this is not considered a huge risk at the moment.

When to Use MySQL

Despite all of these advantages, there are still some small drawbacks to using Postgres that you should consider.
Postgres is still less popular than MySQL (despite catching up in recent years), so there's a smaller number of 3rd party tools, or developers/database administrators available.

Postgres forks a new process for each new client connection which allocates a non-trivial amount of memory (about 10 MB).

Postgres is built with extensibility, standards compliance, scalability, and data integrity in mind - sometimes at the expense of speed. Therefore, for simple, read-heavy workflows, Postgres might be a worse choice than MySQL.

How does Postgres work?

Divya Nagar

If you are reading this article, I am assuming that you have used PostgreSQL and you want more details about it. So what is PostgreSQL? How it works and how does it manage to understand our simple English queries? Let's first understand what is PostgreSQL and then we will see how it works.

In simple words, PostgreSQL is a system which helps you in managing your huge databases by hiding the underlying complicated details. Let's understand it with an analogy of a car. When you drive a car, if you press the accelerator and change the gear, the speed changes. You need not bother what is happening to the engine and other systems of the car. When it comes to databases Postgres does same for us. You just write a simple English query and it gives you the desired results while hiding all the tortuous processes. It does the optimisation for you which makes your work fast.

So how does Postgres do it? Let's take a simple example and understand it in more details. This is a simple INSERT query.

INSERT INTO table_name (col_1, col_2) VALUES(val_1, val_2)

So these are the things which happen when above query goes to Postgres.

1. Parse
2. Analyse & Rewrite
3. Plan
4. Execute

Let's understand them one by one. To understand the entire flow, from parsing to execution, you can go through the source code links given with functions. First function which PostgreSQL executes is **exec_simple_query**. Inside this you can find all the functions which are being explained below.

Parse (pg_parse_query)

Initially this query goes to parser, which converts string query to a list of parse tree (RawStmt nodes). Lexical scanner breaks the query into tokens. Parser uses the grammar from gram.y and tokens generated by scanner to get the query type. It finally loads the proper structure for the query and generates the raw tree. The output for the above query will be something like this.

[{ InsertStmt: { relation: [Object], cols: [Array], selectStmt: [Object] } }]

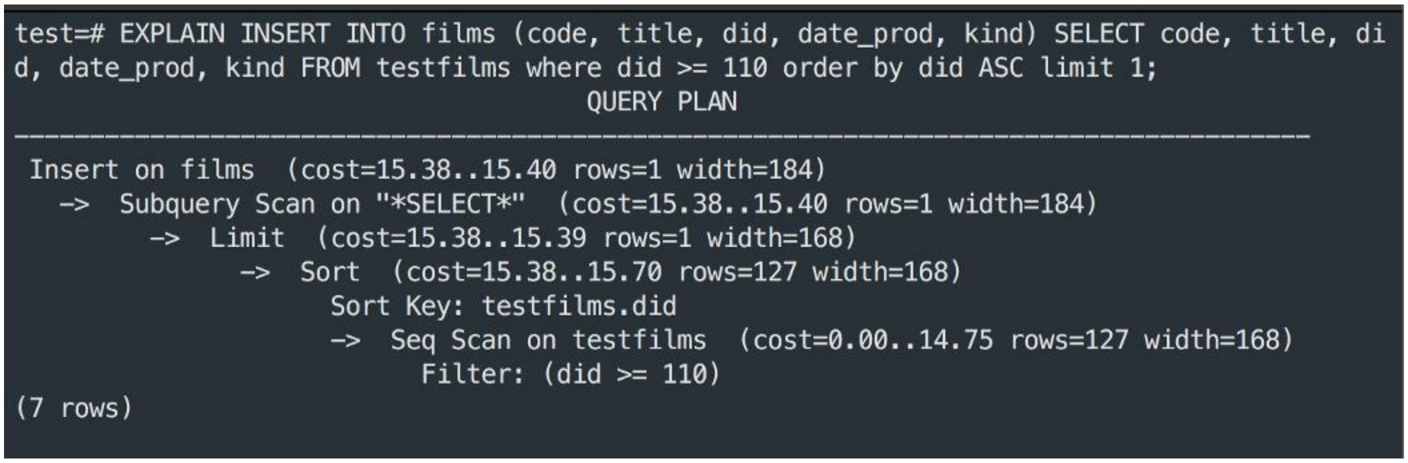
Analyse & Rewrite (pg_analyze_and_rewrite)

Once the raw tree is generated, if we see in exec_simple_query the next function which we encounter is pg_analyze_and_rewrite. It does some analysis on raw tree and tries to optimise it. To analyse the query it applies some transform operations on different clauses present in our query. For example if we have a WHERE clause, it calls TransformWhereClause and checks if the output value is boolean. It avails different transformations for different clauses like transformLimitClause, transformGroupClauseExpr etc.

INSERT query modifies some columns of the database. In this step Postgres creates a target list in which it stores what columns will get modified when this query executes. After the analysis and rewrite, the next thing is optimisation. Postgres has one optimiser module to optimise the queries. It uses query structure to determine the best table join order i.e. in what order should the join operations happen to make the execution faster.

Plan (pg_plan_queries)

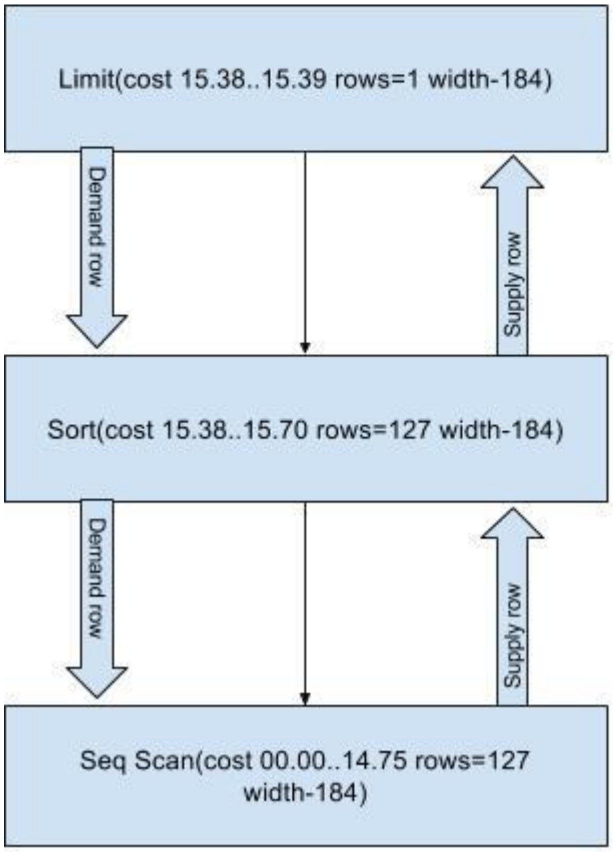
This module generates a plan tree headed by ***PlannedStmt*** node***.*** PlannedStmt contains the information which execution module needs to execute the query. Plan tree contains plan nodes. Plan nodes have the information about their tasks. Each node can be considered as a task which has some cost and other factors like plan_rows, plan_node_id. These factors are necessary to execute that node. Each node in plan tree has one branch. It takes input from it's child and passes the output to it's parent. Based on the cost in plan nodes, Postgres estimates the total query execution time. To get more insights on this, try EXPLAIN in Postgres. As you can see in the output, there are some tasks with their costs. This is the output of plan module which goes to execution module. It can also help you in profiling and understanding which part of the query is taking more time and hence optimising it.


```
test=# EXPLAIN INSERT INTO films (code, title, did, date_prod, kind) SELECT code, title, di
d, date_prod, kind FROM testfilms where did >= 110 order by did ASC limit 1;
                                        QUERY PLAN
---------------------------------------------------------------------------------------
 Insert on films  (cost=15.38..15.40 rows=1 width=184)
   ->  Subquery Scan on "*SELECT*"  (cost=15.38..15.40 rows=1 width=184)
         ->  Limit  (cost=15.38..15.39 rows=1 width=168)
               ->  Sort  (cost=15.38..15.70 rows=127 width=168)
                     Sort Key: testfilms.did
                     ->  Seq Scan on testfilms  (cost=0.00..14.75 rows=127 width=168)
                           Filter: (did >= 110)
(7 rows)
```

EXPLAIN on insert-select query

Execution

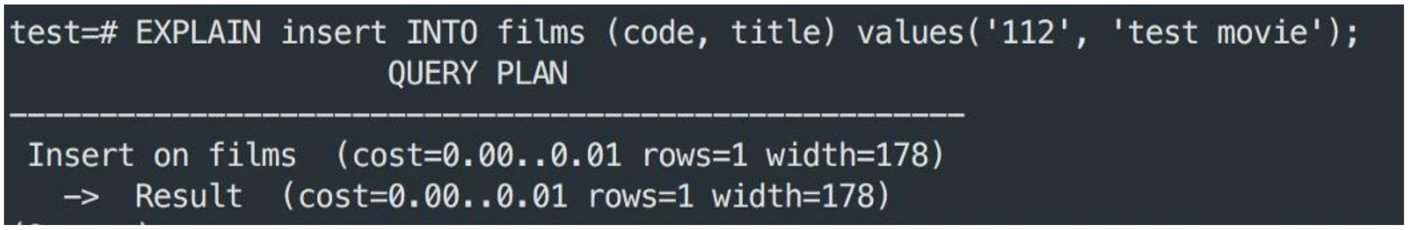
After so much of work, Postgres finally has a plan to execute. Plan tree, generated by planner and optimiser goes to Postgres executor. Executor has a demand-pull mechanism. Every plan node will demand rows from its child node. Executer will keep doing this recursively until it gets the desired results.


Limit(cost 15.38..15.39 rows=1 width-184)
Demand row
Supply row
Sort(cost 15.38..15.70 rows=127 width-184)
Demand row
Supply row
Seq Scan(cost 00.00..14.75 rows=127 width-184)

Demand-pull Plan tree

This image denotes the plan tree which we generated in plan section. Limit plan node requires at least one row to execute. It demands that row from Sort plan node which recursively demands row from Seq Scan. While executing, Sort plan node will repetitively call Seq Scan plan node to get the rows to be sorted. When the input is exhausted, Sort plan node will keep the sorted rows so that it can deliver to Limit plan node when demanded.

For our previous insertion query, each returned row is inserted in the specified table. EXPLAIN on that insert query will give trivial results with single Result plan node.



```
test=# EXPLAIN insert INTO films (code, title) values('112', 'test movie');
                     QUERY PLAN
-----------------------------------------------------
 Insert on films  (cost=0.00..0.01 rows=1 width=178)
   ->  Result  (cost=0.00..0.01 rows=1 width=178)
```

EXPLAIN on simple insert query

So this is how Postgres executer works. For more details on update and delete query execution you can look at Postgres official manual. Queries might get complicated, parse and plan trees might get larger but process stays the same.

Sources
https://ubuntu.com/blog/what-is-postgresql
https://medium.com/@divya.n/how-postgres-works-733bc5cf61a
https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres