



# Anarchy in the Database: A Survey and Evaluation of Database Management System Extensibility

Abigale Kim  
UW–Madison  
abigale@cs.wisc.edu

Marco Slot  
Crunchy Data  
marco.slot@crunchydata.com

David G. Andersen  
Carnegie Mellon University  
dga@cs.cmu.edu

Andrew Pavlo  
Carnegie Mellon University  
pavlo@cs.cmu.edu

## ABSTRACT

Extensions allow applications to expand the capabilities of database management systems (DBMSs) with custom logic. However, the extensibility environment for some DBMSs is fraught with perils, causing developers to resort to unorthodox methods to achieve their goals. This paper studies and evaluates the design of DBMS extensibility. First, we provide a comprehensive taxonomy of the types of DBMS extensibility. We then examine the extensibility of six DBMSs: PostgreSQL, MySQL, MariaDB, SQLite, Redis, and DuckDB. We present an automated extension analysis toolkit that collects static and dynamic information on how an extension integrates into the DBMS. Our evaluation of over 400 PostgreSQL extensions shows that 16.8% of them are incompatible with at least one other extension and can cause system failures. These results also show the correlation between these failures and factors related to extension complexity and implementation.

## PVLDB Reference Format:

Abigale Kim, Marco Slot, David G. Andersen, and Andrew Pavlo. Anarchy in the Database: A Survey and Evaluation of Database Management System Extensibility. PVLDB, 18(6): 1962 - 1976, 2025.  
doi:10.14778/3725688.3725719

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/cmu-db/ext-analyzer>.

## 1 INTRODUCTION

There are two choices when building a DBMS to support a specific use case. The first is to create a new system (or fork an existing one) and customize it for the target workload and environment [152]. The other approach is to expand an existing DBMS through an extension. An *extension* (aka plug-in) is custom code that adds new features to a DBMS while maintaining its core functionality and infrastructure. Examples of extensions include user-defined types, password authentication protocols, or a storage manager.

Extensions allow a DBMS to support more use cases with less code than writing a new system. As the DBMS improves with new features, such extensions ideally benefit from these updates without refactoring. They also reduce feature bloat in the DBMS, improving engineering cadence by separating the DBMS’s development cycle from the extensions. In some cases, extensions are so beneficial that

DBMSs convert them into core features. For example, PostgreSQL’s auto-vacuum started as a “contrib” extension in 2003 (v7.4 [1]) and became a built-in component in 2005 (v8.1 [2]).

The database industry has recognized the importance of extensions for decades. Many leading systems support extensibility, including PostgreSQL [119], DuckDB [117], Oracle [55], MySQL [45], SQLite [104], SQL Server [43], Redis [93], and Elasticsearch [30]. There are even companies offering DBMS products based on extensions, such as Citus [24], TimescaleDB [112], and ParadeDB [56].

Despite the benefits of extensions and wide adoption, there has been almost no research into this aspect of DBMSs. Furthermore, some design decisions in DBMSs to make a system more extensible are fraught with problems. Unfettered extensions lead to unexpected errors or compatibility issues with other extensions, causing the DBMS to produce incorrect results, corrupt data, or crash. In other cases, insufficient APIs and support infrastructure cause extension developers to resort to bad practices, such as copying source code or exploiting existing APIs for uses other than their intention.

This paper provides a thorough analysis of DBMS extensions from the perspective of system internals. We propose a taxonomy of extensibility supported by DBMSs and their design considerations, and then discuss the mechanisms that DBMSs can offer to support extensions. We use this taxonomy to survey six DBMSs: PostgreSQL, MySQL, MariaDB, SQLite, Redis, and DuckDB. We discuss the good and not-so-good aspects of each DBMS’s extension support. We then introduce **ExtAnalyzer!** [116], our automated analysis toolkit for DBMS extensions. We evaluated 441 PostgreSQL extensions and find that 16.8% are incompatible with at least one other extension, causing unexpected behavior and errors. Based on these results, we provide guidance on how DBMSs can better support extensibility.

## 2 BACKGROUND

To our knowledge, the first extensible DBMS was INGRES in the 1970s, with its support for UDFs written in C [153]. The academic version of INGRES also added UDTs in the early 1980s [143]. The INGRES developers then transferred this prototype’s design and code into the initial version of PostgreSQL [151, 154].

In the 1990s and early 2000s, other DBMSs added support for UDFs and UDTs, including Oracle [3, 5], IBM DB2 [135], and Microsoft SQL Server [8, 19]. MySQL v3.22 added its storage engine interface in 2001 [145]. The SQL standard also included UDFs in SQL:1996 [136] and UDTs in SQL:1999 [137]. Since then the trend towards more DBMS extensibility has continued. MySQL began supporting system plugins in 2004, while PostgreSQL added extension hooks in 2006. These extensibility mechanisms allowed applications to override DBMS code, increasing extension development.

Some DBMSs provide extensibility as a first-class feature with explicit methods of extending the capabilities of the DBMS. We

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.  
doi:10.14778/3725688.3725719

exclude methods that rely on unintended behaviors or security breaches to add extensions (e.g., using code injection to add new features instead of the core DBMS APIs).

### 3 DATABASE EXTENSIBILITY TAXONOMY

This section presents our taxonomy of implementation design decisions for DBMS extensibility. We begin in Section 3.1 with the types of extensibility that a DBMS can support. We then discuss the five areas of a DBMS’s architecture and environment for extensions: (1) integration interfaces, (2) state modification methods, (3) protection mechanisms, (4) supporting components, and (5) developer APIs. We discuss problems with real-world implementations of these five areas in Section 4. Although extending a DBMS in more ways than described here is possible, our taxonomy targets the most common extensibility types in the most popular DBMSs.

#### 3.1 Extensibility Types

An extension is user-provided logic that augments a DBMS’s behavior. This broad definition encompasses high-level extensions (i.e., UDFs) and low-level extensions that rely on a DBMS’s internal APIs. These types are not mutually exclusive, and many extensions we examine in Section 5 comprise multiple types.

**User-Defined Types (UDTs):** A UDT provides a data type that does not already exist in the DBMS’s built-in type system. UDTs can be logical aliases of an existing type or a combination of existing types (i.e., a struct). Some DBMSs support new custom physical types with their own binary encoding scheme and support functions (e.g., comparators and access methods). Examples of the latter include vector embeddings [58] and LiDaR points [76].

**User-Defined Functions (UDFs):** Most DBMSs support UDFs written in procedural languages like PL/SQL or Python. The most common UDF takes in scalar input arguments and produces either scalar or table results. Another is a *user-defined aggregate* (UDA) that takes in multiple rows of data as input and returns a single scalar result. *User-defined operators* (UDOs) implement expression operators such as “=” and “+” for UDTs via a UDF. Lastly, some systems support executing UDFs as standalone *stored procedures*.

**Client Authentication:** These extensions add or modify the DBMS’s client authentication mechanisms and protocols. For example, an extension could enable user authentication through an external Kerberos server [34]. Some DBMSs allow extensions to change how they handle passwords internally, such as MySQL’s `validate_password` [50] and PostgreSQL’s `passwordcheck` [71].

**Utility Commands:** The next group overrides or introduces administrative commands that read or modify the internal state of the DBMS. These commands target a variety of database objects and system artifacts, including schema, permissions/ACLs, log files [36], and configuration settings. We also include package managers in this category that install other extensions (see Section 3.6).

**Parser Modifications:** A DBMS parser validates the syntax of a query and converts it into an internal representation for subsequent optimization and execution. A parser extension could change the query dialect the DBMS supports to add new syntax. Such extensions differ from UDFs because they enable new syntactic features in a DBMS’s query language that are not expressed as functions. Alternatively, they may not change the syntax but instead change

validation rules (e.g., reject queries modifying specific tables). Developers extend a DBMS’s parser by modifying its code or injecting custom rewrite rules (MySQL’s Query Rewriter [52]).

**Query Processing:** The next type is a broad category of extensions that modify the DBMS’s query processing stack, including its planner and execution engine. The extensions that provide additional execution support either (1) modify only a small part of the execution path and, therefore, are smaller in scope or (2) rewrite the entire query processing layer to support distributed execution or to handle new physical data layouts (e.g., Citus [24], Timescale [112]). Some query processing extensions collect runtime telemetry about queries for performance analysis and debugging.

**Storage Managers:** A storage manager is responsible for organizing, retrieving, and updating data on storage devices. Some DBMSs allow extensions to replace their storage manager either at the (1) file access layer or (2) table access layer. The former is when an extension replaces the DBMS’s components interacting with physical files (e.g., SQLite’s Unix VFS [107]). For example, an extension could retrieve object store data (e.g., Amazon S3) using a non-POSIX API. Only the extension interacts with the object store to retrieve data; the rest of the DBMS is unaware of where that data came from, and the data’s interpretation remains the same. Other extensions override a DBMS’s API for retrieving data from logical tables. The most common are “connectors” that interact with tables backed by external data sources. More expansive extensions involve concurrency control and buffer management components [48].

**Index Access Methods:** Some DBMSs allow extensions to introduce new indexes on top of its existing storage. Custom index implementations sometimes rely on UDTs. For instance, PostgreSQL’s `pgvector` [58] and DuckDB’s `VSS` [113] provide vector UDT indexes for similarity search. Other index extensions leverage custom UDT operators, such as PostgreSQL’s `GIST` [67].

#### 3.2 Interfaces

A DBMS’s programming interface (API) determines how developers integrate their extensions into the system. The DBMS exposes this API through a SQL interface or a procedural programming language (PL). The API’s expressiveness determines whether an extension (1) adds new components to the DBMS or (2) overrides existing components’ capabilities with custom logic.

**Adding Components:** A DBMS can provide APIs to add new types, functions, and other components via handler functions without changing the system’s core functionality. An application then loads the extension into the DBMS and invokes the new functionality explicitly (e.g., by calling the UDF). The APIs for adding components to the DBMS are limited and cannot support all extensibility types. Additionally, performance can suffer depending on the language and runtime context. For example, a Python UDF is slower to execute than a C/C++ UDF due to runtime overhead.

**Overriding Components:** Another approach is where extensions override parts of the DBMS’s code to augment it with new capabilities. The DBMS invokes the extension to perform some task instead of using its original, built-in code. Consider an extension that overrides the DBMS’s query execution: when a query arrives, the DBMS bypasses its original code and invokes the extension to

process it. One method of supporting overriding is via function pointer *hooks* at various places in a DBMS's source code.

### 3.3 State Modification

Most extensibility types in Section 3.1 access or modify state in the DBMS. There are four state types with different implications for how the DBMS supports them: (1) database state, (2) system state, (3) extension state, and (4) ephemeral state.

**Database State:** This consists of a database's logical contents and its physical structures. Database state consists of user data, such as tables, indexes, materialized views, and the metadata about these objects. Extensions modify this state by either (1) submitting SQL commands, (2) accessing table data via the host DBMS's internal API, or (3) modifying the contents of the database's physical files.

**System State:** This state is how the DBMS tracks its runtime operations independent of extensions. It includes the DBMS's catalog, internal data structures that persist from one query to the next (e.g., buffer pools), and temporary data structures that the DBMS creates per query (e.g., plans). Extensions modify system state by (1) registering new components in the catalog (e.g., UDFs, UDTs), (2) writing into data structures passed to extension handler functions, and (3) updating data in shared memory.

**Extension State:** Each extension can also maintain data for its internal operations, including configuration settings and metrics. Extensions store their state as (1) a database table, (2) an entry in the DBMS's catalog, or (3) in-memory data structures if their contents persist across multiple queries. The first two rely on the DBMS's built-in recovery methods to ensure durability.

**Ephemeral State:** We deem any intermediate state that an extension modifies during its execution but does not persist in the database as ephemeral. Such state is usually related to query execution and only persists for the lifetime to a query (e.g., tuples projected from an external table). Extensions create or modify ephemeral state within their runtime contexts. They can also pass this state to the host DBMS for additional processing.

### 3.4 Protection Mechanisms

We now discuss how to ensure that extensions do not interfere with one another and limit the DBMS's exposure to security attacks.

**Isolation:** To ensure extensions do not interfere with each other, a DBMS should isolate their effects. As we discuss in our analysis in Section 5.4, we found examples where two extensions cause problems when installed together, even if individually they do not. One type of static isolation is to restrict the DBMS's extension API. For example, SQLite only allows one extension to control a given extensibility pathway. Other methods include limiting allocations to a unique namespace with the appropriate ACLs. The level of isolation provided by processes does not prevent resource management exploits using extensions. To mitigate this, the system could run an extension in a sandbox (e.g., container, jail) to restrict an extension's resource consumption to prevent out-of-memory or priority inversion issues.

**Security:** Unsafe extensibility opens the DBMS up to attacks that access sensitive data and cause the system to malfunction. Thus, it is imperative to consider security from the onset of adding

extension support. The DBMS can diminish such threats by (1) supporting extensions in safer languages (e.g., not C/C++), (2) disallowing extensions from accessing sensitive data via internal APIs, and (3) placing restrictions on what extensions users can install. These are preventative measures for malicious extension developers, but they are not sufficient for security.

### 3.5 System Components

We describe the internal components exposed to extensions.

**Background Workers:** Most extensibility types modify the critical path of query execution in the DBMS. Others implement asynchronous maintenance tasks that do not operate during query execution. Some DBMSs provide background workers for these tasks by spawning processes or threads to execute custom code. Examples of these maintenance tasks include garbage collection [4], periodic job scheduling [73], and LSM compaction [70]. Extensions can invoke background workers by utilizing (1) custom APIs provided by the DBMS or (2) native threads (e.g., POSIX threads).

**Memory Allocation:** When extensions store state in memory, they should utilize the DBMS's built-in allocator instead of their PL's native allocators (e.g. `malloc`). Using the DBMS's memory allocator allows the system to track memory via pools that are coupled with the lifecycle of a query, transaction or session, and cleared in case of error to avoid memory leaks. We found most extensions use the DBMS's built-in memory mechanisms.







**Configuration Options:** Extensions can expose configuration options to allow users to customize their behavior without needing to recompile the source code. Common options include (1) resource limits, (2) debug logging levels, and (3) other extension-specific settings. Most DBMSs expose registration functions to add new options to the existing configuration infrastructure. After new options are added, the users can set these options by (1) modifying configuration files, (2) passing in command line arguments when they start up a database instance, or (3) writing SQL commands.

**Concurrency Control:** To support parallelism, some DBMSs allow extensions to declare custom locks and use existing latches, which are managed by the system. Extensions use locks to implement concurrency control protocols or modify table data, and use latches to synchronize access to extension managed memory or internal data structures. Extensions declare and register custom locks by calling DBMS API functions in their source code, and leverage the host system's existing concurrency control scheme.

### 3.6 Developer Ecosystem

Lastly, we discuss the ways developers create extensions and how users manage them. These include the supported languages and a DBMS's package manager, building tools, or testing infrastructure.

**Programming Languages:** A developer's first decision when writing an extension is which PL to use since DBMSs often support multiple languages. These include the language of the DBMS source code or SQL for UDFs and UDTs. Additionally, language binding frameworks allow developers to create extensions in previously unsupported languages. DBMSs implement these as foreign function interfaces (FFIs) bundled with their original API. Implementing an extension overriding a DBMS component in the same language as its source code is easier, as the extension can call the original

	 PostgreSQL	 MySQL	 MariaDB	 SQLite	 Redis	 DuckDB
User-Defined Functions	Yes (408)	Yes (2)	Yes (1)	Yes (79)	Yes (57)	Yes (41)
User-defined Types	Yes (139)	No	Yes (13)	No	No	Yes (4)
Utility Commands	Yes (43)	No	No	No	No	No
Parser Modifications	No	Yes (2)	Yes (1)	No	No	Yes (4)
Query Processing	Yes (46)	Yes (7)	Yes (5)	No	No	Yes (4)
Storage Managers	Yes (44)	Yes (13)	Yes (18)	Yes (43)	No	Yes (9)
Index Access Methods	Yes (67)	No	No	No	No	Yes (3)
Client Authentication	Yes (17)	Yes (3)	Yes (10)	No	No	No
Version Examined	v16	v8	v11	v3	v7	v1
Number of Extensions	441	29	68	98	57	44+

**Table 1: DBMS Extensibility** – An overview of extensibility types supported by the DBMSs considered in Section 4.

implementation as needed. However, it requires knowledge of the target DBMS’s internals. Writing extensions in a high-level PL (e.g., Python, JavaScript) results in less faulty code because extensions address the DBMS’s state through abstractions and not directly. This increases the security and isolation of the extension.

**Installation:** Developers must also work with a DBMS’s APIs for installing and removing extensions. Examples of installation interfaces include (1) SQL commands, (2) separate extension manager, and (3) configuration files. Similar to PLs, extension developers can build on these installation interfaces to support a new API.






**Package Managers:** Configuring and installing many extensions on the same DBMS is difficult. As a result, the DBMS or ecosystem sometimes provide package managers to download, install, and run tests on extensions. Package managers are implemented as (1) command line tools, (2) websites, or (3) built-in utilities.


**Build & Test Tooling:** Developers rely on a DBMS’s infrastructure to decrease the boilerplate code required to create a new extension. Examples include build scripts and additional PL support. Likewise, test tooling is also tedious to implement and an DBMS should ideally provide these to expedite extension development.

## 4 SYSTEM SURVEY

Given this taxonomy, we qualitatively analyze how DBMSs support extensions. For each element in Section 3, we read the DBMSs’ documentation and then confirmed our findings by examining their source code. This survey aims to understand the prevalence of trade-offs that DBMSs make when designing extensibility. We will provide a quantitative analysis of DBMS extensibility in Section 5.

We chose six DBMSs to examine based on (1) source code availability, (2) support for at least two extensibility types other than UDTs/UDFs, and (3) usage popularity. The first criterion is critical, but it unfortunately precludes the most popular enterprise DBMSs.


-  **PostgreSQL (1986)** [18]: This relational DBMS is written in C and was designed at its inception to be extensible [154]. As such, PostgreSQL has the most diverse extensibility network.
-  **MySQL (1994)** [45]: This relational DBMS is written in C++ and is most known for its storage manager plug-in architecture.
-  **MariaDB (2009)** [37]: A MySQL fork written in C++ that supports more extensions than the original MySQL codebase.
-  **SQLite (2000)** [104]: An embedded DBMS written in C that works on the most varied hardware and operating environments.
-  **Redis (2009)** [93]: An in-memory key-value store written in C++. Its extensibility is unique because it only supports extensions that operate above the DBMS’s key-value storage.


-  **DuckDB (2018)** [117]: An embedded DBMS for analytics written in C++ with a burgeoning extensibility ecosystem.


We first discuss which extensibility types each DBMS supports (see Table 1). We then provide our analysis of each DBMS’s internals based on our taxonomy. Table 2 shows a summary of this survey.


### 4.1 Extensibility Types

For each DBMS, we surveyed extensions found from three sources: (1) GitHub, (2) officially supported internal extensions, and (3) extensions supported by cloud providers (e.g., Amazon, Microsoft).

 **PostgreSQL:** This DBMS supports the most extensibility types (seven of eight) and supports relatively advanced features for each type. We found over 441 extensions, whereas the other DBMSs have less than 100 per system. PostgreSQL’s source code includes a contrib directory with ~50 extensions. The DBMS supports UDFs written in C, SQL, or PL/pgSQL by default, but extensions can also add new PLs [62, 63]. Combining UDTs, UDOs, and index access extensions enables specialized domains, such as geospatial [64] and vector search [58]. PostgreSQL extensions use query processing hooks to provide alternative scan and join algorithms, or augment the planner and executor with additional functionality [128]. It also provides two forms of storage manager extensibility: (1) foreign data wrappers [84] for external tables (e.g., file-based, other DBMSs) and (2) table access methods for local storage (e.g., columnar [25]). It is the only DBMS that supports utility command extensibility.

 **MySQL:** The DBMS’s installation package includes 44 extensions, but there are a small number of third-party extensions. There are ~10 storage managers for MySQL [158], including InnoDB [48], in-memory [49], and federated [47] engines. MySQL’s extension ecosystem is less active than PostgreSQL despite historical trends of the former being more widely deployed than the latter.

 **MariaDB:** Since it is a MySQL fork, MariaDB has the same extensibility types, but it added support for UDTs in 2019 [6]. MariaDB’s installation package contains 65 extensions, including 22 storage managers that are more specialized than the ones available for MySQL [158]. Examples include text search [39], S3 backup [42], and parallel query execution [38].

 **SQLite:** This DBMS supports overriding its filesystem access layer. Extensions use this extensibility to optimize storage for different workloads, support additional operating systems, and implement security features, such as encryption [105]. SQLite restricts overriding more than one filesystem simultaneously. Apart from this, extensibility support in SQLite is minimal.



🔴 **Redis:** This DBMS’s extensibility support differs from the others because it does not allow extensions to access its internal APIs. It only supports UDFs, which operate above the DBMS’s key-value storage. Despite this limited support, Redis has an actively maintained catalog of 57 extensions. One of its most popular extensions provides a new query and indexing engine [99]. Redis supports extensions written in either C or Lua.

🟡 **DuckDB:** Similar to PostgreSQL, DuckDB supports many extensibility types. For example, it supports planner extensions to add custom optimizer passes. It also provides vectorized UDFs, which means that each invocation of the UDF processes a batch of values to avoid the overhead of context switching. It enables developers to override its catalog to support accessing external DBMSs, including PostgreSQL [28] and SQLite [29]. DuckDB has modified its parser to be extensible at runtime to allow extensions to add their own syntax without the need for UDFs [142].

## 4.2 Interfaces

We now examine the DBMSs’ interfaces for extensions. Extensions that replace DBMS components have a wider variety of capabilities; some extensions override so much of a DBMS that they transform it into a different system. For example, Citus [24] turns PostgreSQL into a distributed DBMS. Maintaining such complex extensions is difficult because developers must ensure compatibility as a DBMS’s API evolves with newer versions. When a DBMS’s interface does not provide developers with what they need, they resort to using brittle methods, like duplicating the DBMS source code.

🔵 **PostgreSQL:** The DBMS provides mechanisms to add functionality via handler functions for UDTs, external tables, storage engines, and index access methods. One problem with PostgreSQL’s UDT API is that some handler functions are optional, which causes problems when using them with index extensions if not implemented (see Section 5.4). PostgreSQL provides the other extensibility types by allowing extensions to override DBMS functionality via hooks in parts of its code. The DBMS declares hooks as function pointers in global variables that it will call instead of its code if the hook is set. Extensions replace a part of PostgreSQL’s functionality or perform additional steps before or after calling the original code. As of 2024, PostgreSQL supports 34 hooks.

🟡 **MySQL / 🦋 MariaDB:** Their interfaces for adding components is similar to PostgreSQL. However, instead of setting hook pointers, their extensions set handler functions in structs. The DBMS stores extension metadata, including the addresses of the handlers, in its catalog. Then, it searches and invokes the extension at various points in the code. Each extension has a single type association and thus cannot contain multiple types of extensibility.

🔵 **SQLite:** The UDF and storage manager interfaces support adding components via handler functions. Similar to MySQL and MariaDB, extensions set handler functions in structs, and the DBMS invokes the extension when the component is used. SQLite allows extensions to override filesystem functionality by replacing system call wrapper functions (e.g., open, read, write).

🔴 **Redis:** This DBMS has the most restrictive API that only supports UDFs (called “commands” [95]) that process query commands and read/write data via its storage layer. Redis UDFs cannot override core DBMS functionality. Despite this limited API, as we

describe in Section 4.1 Redis has some of the most diverse extensions that transform the system into a different type of DBMS (e.g., graph [100], full-text search [99], relational [115]). This is akin to building a full-featured DBMS (e.g., TiDB [134]) on top of an embedded key-value DBMS (e.g., RocksDB [131]).

🟡 **DuckDB:** Similar to MySQL, this DBMS’s extensions add components via handler functions. Since DuckDB is a C++ codebase, extensions define these functions in a class instead of structs. But DuckDB is switching to a C API for extensions to make it easier to interoperate with other PLs [120].

## 4.3 State Modification

Next, we consider the state types extensions can modify. Extensions that modify system state wield more control over the DBMS’s behavior to support expansive features. But if an extension is allowed to modify internal data structures, it can corrupt them, causing the DBMS to crash or produce incorrect results.

🔵 **PostgreSQL:** Extensions access database and system state via global functions and variables, and the DBMS passes ephemeral state into the extension via function arguments. Extensions can create tables and other database objects to store extension-specific state, but they must also handle upgrades and dump-and-restore scenarios. PostgreSQL also allows extensions to register functions for copying and (de)serializing data structures. Extensions often use this functionality to inject data into query plans.

🟡 **MySQL / 🦋 MariaDB:** They allow extensions to create database objects (e.g., tables), access system state via C-style pointers, use the built-in memory allocation mechanisms for extension state, and define ephemeral state.







🔵 **SQLite:** Its extensions can use the built-in memory allocator to access ephemeral state. Both SQLite’s virtual table (i.e., connectors) and filesystem extensions access system state via arguments to their handlers. Although extensions cannot create new databases, they can write tuple data in database files to modify state.

🔴 **Redis:** Its extensions can modify all state types except system. Extensions can store metadata or database contents inside the DBMS’s built-in key-value store, use built-in memory mechanisms, and define ephemeral state in their source code.

🟡 **DuckDB:** The DBMS exposes its system state via top-level database session instances and extensibility interfaces. Extensions can also augment the DBMS’s C++ classes with ephemeral state. They can create tables and store state in the database, though such facilities are limited.


## 4.4 Protection Mechanisms

We next analyze the protection mechanisms in the DBMSs. There is contention between how much freedom a DBMS gives extensions and how vulnerable it makes the system. Query-invoked extensions (e.g., UDFs) can circumvent a DBMS’s ACLs since they execute as the calling user and not the original user that installed it [14]. If users can install extensions written in an unsafe language, the only protection is whatever the OS provides. If extensions run in the DBMS’s address space, the OS cannot prevent them from accessing certain resources. Such problems are why DBaaS vendors restrict what extensions a user can enable. Static analysis methods and

	 PostgreSQL	 MySQL	 MariaDB	 SQLite	 Redis	 DuckDB
Adding Components	Yes	Yes	Yes	Yes	Yes	Yes
Overriding Components	Yes	Yes	Yes	Yes	No	Yes
State Modification	All state	All state	All state	All state	Etxn. + Ephmrl.	All state
Isolation/Security	None	Low	Low	Medium	High	Low
Background Workers	Yes	Yes	Yes	No	No	No
Memory Allocation	Yes	Yes	Yes	Yes	Yes	Yes
Configuration Options	Yes	Yes	Yes	No	Yes	Yes
Source Code	Yes	Yes	Yes	Yes	No	Yes
Programming Languages	C, C++, Rust	C++	C++	C, Rust	C, Lua	C++
Installation Interface	SQL, configs	SQL	SQL	SQL	SQL, configs	SQL
Build & Test Tooling	Both	Testing	Testing	Both	None	Both
Package Manager	Yes (community)	No	Yes (OS)	Yes (community)	No	Yes



**Table 2: Extensibility Support** – An overview of extensibility support for the DBMSs evaluated for the survey in Section 4.


runtime sandboxing could prevent some problems with unsafe extensions, but none of the DBMSs we examined do these. Enterprise DBMSs (e.g., Oracle [118]) execute extensions as a separate process.

 **PostgreSQL:** Users’ calls to UDFs in higher-level languages are subject to regular database ACLs. However, it is possible for an extension to define UDFs with administrative privileges. An administrator normally loads extensions, though installation interface issues create opportunities for less privileged users to gain higher privileges [11, 13, 17]. An administrator can mark extensions as “trusted” to allow less privileged users to load them [66].


Extensions written in C have no restrictions once called via a hook. Low-level functions for reading and writing to the database are not subject to access controls since the DBMS only enforces those at the planning and execution layer. It is up to the extension developer to enforce ACLs. PostgreSQL provides a keyword `SECURITY LABEL` [87] which provides access control for extensions.


Multiple extensions can interfere with each other when they use the same hook. Thus, the end-to-end behavior of combining multiple extensions may be unpredictable. Some extensions require the DBMS to invoke them last to avoid interference, which causes problems if two extensions have this requirement. Suppose a hook is already set by extension A when extension B is loaded, and B sets it to its function. In that case, the convention is for B’s function to call the original hook (A’s function) to form a chain of function calls, but the DBMS does not enforce this, and the extension may make changes to the data structures passed as arguments. We study this issue further in Section 5.4.

 **MySQL /  MariaDB:** Since the DBMS handles extension invocation, one extension cannot modify another’s execution. The DBMS attempts to ensure extensions do not overwrite each other’s state, but this behavior is not guaranteed. For auditing plugins, the DBMS passes the same event state to each plugin, which allows other extensions to modify it. Because most of MySQL’s extensions are in C/C++, it is prone to security vulnerabilities. For example, users with low privilege access can execute denial of service attacks [12, 46]. Like PostgreSQL, MySQL’s allows developers to specify whether an extension requires higher privileges to install.

 **SQLite:** Most extensibility types cannot interact, and the DBMS only allows one filesystem extension at a time. Like other DBMSs, a malicious user can cause denial of service attacks through


the extension API [7, 10]. SQLite supports disabling extension loading to prevent such attacks and SQL injections. Since SQLite is in C, it is prone to language-supported security attacks.



 **Redis:** This DBMS is the most isolated in our survey, as all its extensibility types do not interact with one another. Since the DBMS is written in C, it is also prone to language-related security attacks. For example, `pam_auth` [98] overrides the client authentication component of the DBMS without an established interface. It supports ACLs to disallow users from installing extensions.

 **DuckDB:** The APIs in DuckDB prevent extensions from overriding each other, though they can sometimes alter each other’s state. For instance, optimizer extensions have access to the same query plan data structure. To prevent exploits, DuckDB cryptographically signs its core extensions and community repository extensions. The DBMS is also prone to some language-level attacks, but it mitigates them using modern C++ memory constructs.

## 4.5 System Components

We now describe the internal mechanisms that the DBMSs provide to developers to help them create extensions. The DBMS’s system components impact extension design by making development more convenient. Implementing workarounds for components that do not exist or have unsuitable APIs is also possible. For example, extensions can create database tables to store metadata, even without memory allocation APIs.

 **PostgreSQL:** The DBMS has a large C codebase where some functions and variables defined in header files are accessible to extensions. Extensions allocate short-term memory using the DBMS’s built-in region-based allocator; the DBMS automatically releases this memory at the end of the current operation or transaction. Extensions can also request long-term memory to share state across its process-based sessions. PostgreSQL provides specialized background workers that work with its process-per-worker model. Extensions can also define new configuration options. Users set these options via SQL or regular configuration files. Allowing users to change options via configuration files is unique to PostgreSQL.

 **MySQL /  MariaDB:** They both provide generous extensibility mechanisms to developers. For example, the DBMS’s daemon extensibility allows extensions to spawn additional threads. This is similar to background workers, although it is supported differently within the DBMS. Extensions using daemons typically run one process in the background to perform various tasks. On the

other hand, extensions using background workers use them to supplement existing functionality (e.g., garbage collection for an index).

🔧 **SQLite:** The DBMS provides a C header [111] that gives extensions access to all its API routines. These include concurrency control, memory allocation, and string processing functionality.

🔴 **Redis:** It provides a dynamic memory allocation API that allows extensions to allocate and free memory. Redis also provides an extensive utility API, allowing extension developers to call its internal commands from their extension code. It does not offer its core source code for extension developers. Instead, extension developers build extensions using the key-value store as needed instead of directly modifying Redis.

🐧 **DuckDB:** It uses C++’s memory management features (e.g., smart pointers) to mostly obviate the need for custom allocation primitives, but a custom allocator is available for allocating data buffers. Extensions can add data structures to maps tied to the session or database instance. Extensions can also extend classes in DuckDB’s source code to add new functionality. DuckDB relies on conventional C++ mutexes for concurrency control, and extensions can do the same. Since DuckDB is an embedded DBMS, it does not support background workers. Extensions add configuration options by calling internal functions upon initialization.

## 4.6 Developer Ecosystem

We discuss methods for creating and installing extensions. A common problem is ensuring version compatibility when a DBMS’s API changes. If a DBMS does not require an extension to provide a supported versions list, developers resort to two methods. First, they release separate variants of their extension for each DBMS version, which imposes a maintenance burden on developers and users. Alternatively, developers write explicit version handling in their extensions, increasing the code’s complexity and brittleness.

🐘 **PostgreSQL:** SQL scripts define the DBMS extensions to create database objects and an optional shared library with internal functions and hooks. Developers can write a shared library in C, C++, or Rust (pgrx [86]). `pg_tle` [89] also enables developers to write extensions in procedural languages. Extensions can be installed via package repositories, compiling the extension from source, or a specialized extension manager (e.g., PGXN [59]). No official extension manager exists, but PGDC [68] distributes commonly used extensions. Some extensions require users to modify configuration files to load them on startup.

PostgreSQL provides two extension development tools: (1) PGXS [60] (build system for extension development) and (2) `pg_regress` [57] (opaque-box testing for extensions).

🐳 **MySQL:** Extensions are SQL scripts with UDFs or shared library objects written in C++. The DBMS does not have build tooling, so extensions create their own. It provides example modules and testing infrastructure, although extensions sparingly use the latter. Users install extensions with SQL commands or by passing in a configuration option at server startup. Our survey did not find any extension package manager for MySQL. This lack of support is notable, given that the MySQL extensibility has many features.

🐬 **MariaDB:** Its ecosystem is similar to MySQL but has better documentation and a tool for managing extensions [41]. MariaDB also relies on OS package managers (e.g., Debian `apt`, Redhat `yum`) for distributing and upgrading extensions [40].

🔧 **SQLite:** There is no DBMS-provided build infrastructure for extensions written in C, but there are community-driven efforts for other PL frameworks [106, 109]. Developers compile extensions into shared library objects and load them through SQL commands. They can also write tests using SQLite’s built-in Tcl testing framework. SQLite has one community package manager (`sqlean`) [108].

🔴 **Redis:** The DBMS supports extensions written in either C or Lua (UDFs). `RedisModulesSDK` [97] is a collection of utility functions, test tooling, and documentation to help developers write extensions. The `redismodule-rs` is a framework for writing extensions in Rust. Redis does not have a package manager, but its official documentation used to provide a list of popular, vetted extensions [96]; this list was removed as part of Redis’ re-branding and license change in 2024 [94].

🐧 **DuckDB:** The DBMS provides a C++ extension template that includes build and test infrastructure. Extensions compile to shared libraries. When the DBMS loads an extension, it installs objects like functions and types to its catalog. DuckDB also provides extensibility that targets local applications. For instance, developers can use the C API to define custom scans and the Python API to define custom UDFs in Python, though we do not consider such programs in this paper. DuckDB is the only DBMS that has a built-in extension manager. When users install an extension with SQL commands, the DBMS downloads it from either a DuckDB-managed repository or third-party repository [27]. Developers can also manually link extensions into the DuckDB binary.

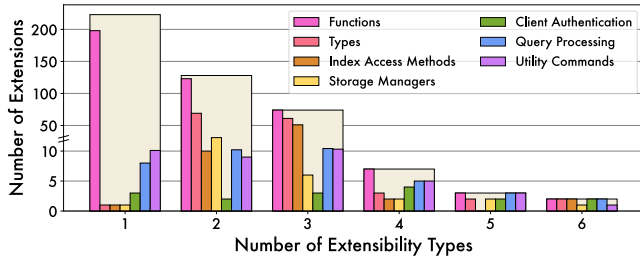
## 5 EXTENSION ANALYSIS

Our survey in Section 4 provides a qualitative overview of DBMS extensibility in widely used systems. To better understand how extensions use these features in the real world, we next analyze the implementations of PostgreSQL extensions. We examined the other DBMSs but found no incompatible extensions, and the amount of copied code was minimal. We discovered that 15.5% of DuckDB extensions contained code from the DBMS, which was all less than 3%. Most of the code copied by SQLite and Redis extensions were from their extension header files. Lastly, we found that MySQL and MariaDB have no extensions with copied source code.

In this section, we evaluate PostgreSQL’s extensibility in three ways. The first two rely on static analysis of the extensions’ source code (Sections 5.1 and 5.2). Our third evaluation step runs the extensions in the DBMS and observes their behavior (Section 5.4).

We created the *Extension Analyzer Toolkit* (ExtAnalyzer! [116]) for this investigation. The toolkit automatically downloads and installs PostgreSQL v16 [18] extensions from the following sources: (1) PostgreSQL’s `contrib` directory [16], (2) supported extensions from AWS RDS [20], Google Cloud SQL [33], and Azure PostgreSQL [22], (3) PostgreSQL Extension Network (PGXN) [59], and (4) other popular extensions (e.g., Citus [24], TimescaleDB [112], PostGIS [64]). Since we do not know which extensions are the most used, the supported extension list from the cloud vendors is a suitable approximation and 89% of them are open-source.





**Figure 1: Distribution of Extensibility Types** – The number of extensions using each extensibility type, grouped by number of types used.

Extensibility Type	# of Extensions
User-Defined Functions	408 (92.5%)
User-Defined Types	139 (31.5%)
Index Access Methods	67 (15.2%)
Storage Managers	44 (10.0%)
Client Authentication	17 (3.9%)
Query Processing	45 (10.2%)
Utility Commands	43 (9.8%)

**Table 3: Extensibility Types** – The number of PostgreSQL extensions per extensibility type evaluated in our experiments.

### 5.1 Source Analysis: Extension Characteristics

We first measure the prevalence of extensibility types (Section 3.1) and system components (Section 3.5) to understand how real-world extensions use these elements. For each extension, our analysis toolkit downloads its source code, extracts relevant code, and checks for keywords that indicate usage of an extensibility feature. Since our toolkit can produce false positives due to keyword misuse, we manually inspected a random sample of the extensions to verify that the analysis works as intended. Extensions declare which hooks they want to override in their initialization code. The toolkit also extracts SQL keywords that install different components of the extension (e.g., `CREATE FUNCTION`). We count the number of extensibility types per extension. We then identify combination groupings of extensibility types in the more complex extensions.

Figure 1 groups extensions by the number of extensibility types used and shows the number of extensions that use a specific type within each group. In addition, Table 3 presents the number of extensions using each type. UDFs are the most common type; over 92% of extensions use them. This pervasiveness is expected because PostgreSQL extensions use UDFs for user-facing features and to define other extensibility types (e.g., UDTs).

The second most common extensibility type is UDTs, with ~32% of extensions using them. Extensions are more likely to employ UDTs when combined with other types. All but one extension creates UDTs using C/C++ code handlers (represented as UDFs) to read and write the type representation into memory.

Most extensions are simple and only use one or two extensibility types. 81 of 441 (~20%) extensions use three or more types, and we classify these as complex extensions. Of these extensions, 31 of 81 (35.6%) use (1) client authentication, (2) query processing, or (3) utility command types, which means they use hooks and override functionality in the host DBMS. We also observed that 36 of 81 (44.4%) extensions with three or more extensibility types leverage the DBMS’s system components described in Section 4.5.

Rank	Categorization Info	# of Extensions
#1	<b>NEW INDEXES</b> UDFs, UDTs, index access methods	55 (67.9%)
#2	<b>QUERY PROCESSING FEATURES</b> UDFs, query processing, utility cmds	17 (21.0%)
#3	<b>UDT-OPTIMIZED QUERY ENGINE</b> UDFs, UDTs, storage mgr., query proc.	4 (4.9%)
#4	<b>NEW STORAGE MANAGER</b> UDFs, storage manager, utility cmds	4 (4.9%)
#5	<b>FULL-FEATURED EXTENSIONS</b> All extensibility types	1 (1.2%)

**Table 4: Complex Extension Categorization** – An overview of the extension categories found by the K-modes clustering algorithm.

We convert each extension’s extensibility types into a one-hot encoding and then cluster them using K-modes [124]. We found that five clusters provided the most meaningful results through the elbow method. Then, we manually each group of extensions (shown in Table 4). All categories use UDFs, which are required for other extensibility types. The most common category is new data types with type-specific indexes. One example is `pgvector`, an extension that implements a custom embedding type and an HNSW index to retrieve them efficiently. PostgreSQL has supported custom types and index access methods since 1995 [133], whereas it added query processing extensibility in the mid-2000s [26, 91, 103].

The second complex extension category targets query processing. These extensions combine query processing and utility hooks to provide additional functionality for each command (e.g., statistics collection). Storage manager extensions add foreign tables or access methods with table-level utility commands (e.g., `COPY` or `TRUNCATE`). New query engines with UDTs add storage and execution features for custom types. The smallest category in our cluster is the full-featured extensions that transform PostgreSQL into a new DBMS by adding layered features on top of it; the only extension in this category is `Citus` [24]. It includes a custom planner for distributed queries, columnar table storage, and UDFs/UDTs.

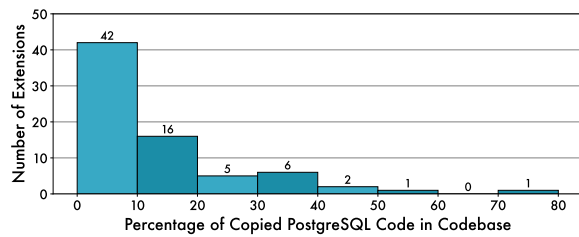
### 5.2 Source Analysis: Duplicate Code

We next evaluate the source code quality of PostgreSQL extensions to understand whether the DBMS’s API sufficiently supports the extensions’ use cases or whether developers contort the system to implement their features. We look for two problems that we observed while surveying these extensions. The first is when developers copy large amounts of the DBMS’s code into their extensions. The second problem is when extensions have custom logic for API changes in DBMS versions (see Section 5.3).

To identify duplicate code, the toolkit extracts two metrics from each extension: (1) total number of code lines and (2) number of lines copied from PostgreSQL’s source code. We use PMD Copy/Paste Detector (CPD) [31] to find duplicate code blocks. We provide CPD with each extension’s source code and PostgreSQL source code. The toolkit then examines code blocks with a minimum length of 100 tokens (i.e., the smallest unit of a PL with meaning). CPD records false positives when it determines that an extension duplicates its own code and the toolkit omits these cases in our results.

We found 73 of 441 extensions (16.6%) include at least one line of copied PostgreSQL code. The histogram in Figure 2 shows the distribution of these extensions based on the percentage of copied





**Figure 2: Duplicate Code** – Distribution of extensions based on the percentage of copied PostgreSQL code in their codebases.

Extensibility Type	# of Extensions
User-Defined Functions	29 (93.5%)
User-Defined Types	18 (58.1%)
Index Access Methods	7 (22.6%)
Storage Managers	0 (0%)
Client Authentication	3 (9.7%)
Query Processing	5 (16.1%)
Utility Commands	6 (19.4%)

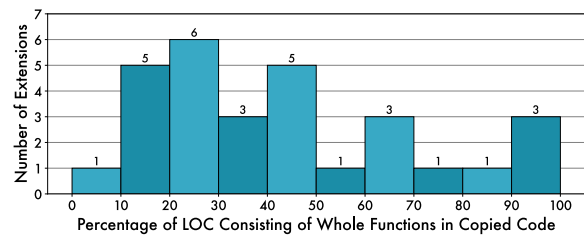
**Table 5: Duplicate Code per Extensibility Type** – Number of extensions with > 10% copied code (31) using each extensibility type.

code relative to their entire codebase. The biggest “offender” is `pg_ivm` [75] (incremental view maintenance) where over 75% of its code is copied from PostgreSQL!

We ran an additional cross-analysis on the extensions with more than 10% copied code to understand what differentiates them. Extensions copying source code use more UDTs, index access methods, client authentication, query processing, and utility commands than others. All extensibility types that require hooks are on this list. When setting these hooks, an extension should preserve the DBMS’ original functionality and may need to call existing functions. Extensions that use storage manager APIs have little duplicated code. A possible explanation is that PostgreSQL’s storage manager APIs were created in tandem with related extensions (e.g., `postgres_fdw` [84]), so any gaps in those interfaces would have been apparent during development.

Through a manual inspection of the copied code and discussions with developers, we found that extensions copy PostgreSQL’s code for three reasons. Foremost, extensions often need to use built-in functions within the PostgreSQL source code. However, those functions are declared `static` in the code, meaning any other code that wants to call the function must reside in the same file as the function’s definition (this is a C restriction). Therefore, developers resort to copying these functions into their repository so their extension can call them. This is the reason why `pg_ivm` contains so much copied code. We found that 29 of 73 (39.7%) extensions that duplicate code contain copied entire functions. In other cases, extensions copy functions but change small parts of their functionality to suit their needs. Lastly, we identified cases where extensions copy a portion of logic from PostgreSQL code but not entire functions.

To differentiate between these three scenarios, we measured what percentage of each extension’s lines of copied code resides in an entire function from PostgreSQL. Figure 3 shows the percentage of copied code relative to an extension’s total number of lines of code. These results show that an average of 42.7% of copied code consisted of entire functions. One common anti-pattern is when an extension contains a copy of the same PostgreSQL function for



**Figure 3: Percentage of Function Code in Extensions** – Distribution of extensions based on the percentage of lines of copied code consisting of whole functions.

each major DBMS version. An example of an extension that does this is `rum` [101] (inverted index for full-text search), which has over 10k lines of copied code related to sorting tuples that only changes slightly from one version of PostgreSQL to the next. Maintaining a codebase with copied functions is laborious and results in many bugs. For example, Citus’ GitHub repository contains 44 issues related to problems with PostgreSQL copied functions [24]. We discuss the implications of this issue further in Section 6.5.

### 5.3 Source Analysis: Versioning Logic

Given the above example with the profuse copied code based on versions, we next measure how often extensions include such custom logic. This provides insight into how often extension developers must handle changes in the DBMS’s internal API. For each extension, our toolkit analyzes its code to determine whether (1) it uses versioning logic, (2) the number of versions supported, and (3) how many lines of code are version-specific. The toolkit identifies pre-processor directives (`#ifdef`) in the code using the PostgreSQL’s `PG_VERSION_NUM` macro, and then calculates the number of lines of code encapsulated in these versioned conditionals.

Figure 4 shows the distribution of the extensions’ code wrapped in versioning logic, with 153 of 441 (34.7%) extensions using it at least once; 38 of these 153 (24.8%) extensions are supported by the cloud vendors. Although versioning exists in a third of the extensions, on average only 5.8% of their code is version-specific.

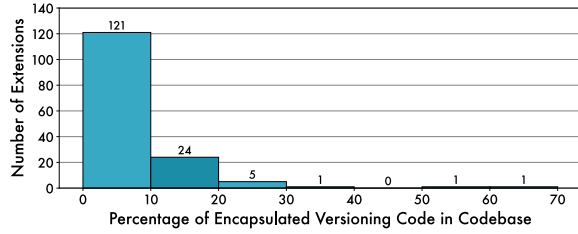
We also ran a cross-analysis on extensibility types to explain why some extensions contain more versioning logic than others. Table 6 shows query processing (41.9%) and utility command (48.4%) extensions have more versioning logic. These extensions rely on internal representations of query plans and execution engine logic that often changes between versions. For example, the function signature for PostgreSQL’s utility command hook changed four times in the last decade. But not all extensions using these extensibility types contain versioning logic: 13 of 45 (28.9%) query processing and 12 of 43 (27.9%) utility command extensions do not handle versioning. The developers for these extensions instead release separate binaries for specific PostgreSQL’s versions. This analysis highlights that PostgreSQL API inconsistency between versions causes developers to write more convoluted code.

### 5.4 Runtime Analysis: Compatibility

As discussed in Section 4.2, PostgreSQL has the most permissive extension API out of the other DBMSs. Our last experiment measures how PostgreSQL’s flexible API impacts extension isolation.

Extensibility Type	# of Extensions
User-Defined Functions	22 (71.0%)
User-Defined Types	3 (9.7%)
Index Access Methods	1 (3.2%)
Storage Managers	6 (19.4%)
Client Authentication	4 (12.4%)
Query Processing	13 (41.9%)
Utility Commands	15 (48.4%)

**Table 6: Versioning Logic per Extensibility Type** – Number of extensions with >10% encapsulated versioning code (31) per extensibility type.



**Figure 4: Versioning Logic** – Distribution based on the percentage of encapsulated versioning code in extension codebases.

We test all unique pairs of extensions to evaluate their compatibility. Two extensions are deemed *compatible* if our toolkit completes three steps: (1) installs them into the same DBMS instance, (2) runs the extension-provided unit tests, and (3) runs pgbench [65]. Running pgbench in the last step provides a smoke test to determine whether the rest of the DBMS still functions correctly with the extensions installed. We also separately test each UDT and index extension combination. We use Claude v3.5 Sonnet [127] to automatically generate an SQL script that populates a small table with a UDT column and then attempts to build an index on that column.

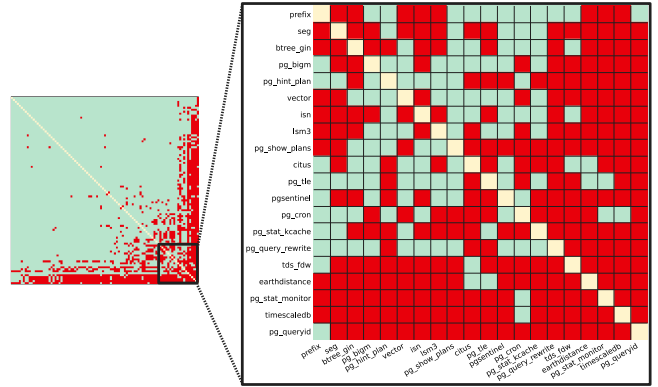
Some failures only occur when the toolkit installs one extension first because it determines the order in which the DBMS invokes them. Hence, for each extension pair, our toolkit installs them in both permutations (i.e.,  $A \rightarrow B$ ,  $B \rightarrow A$ ). We ran these tests in our toolkit for the 96 extensions with the necessary installation scripts.

Our tests found that 16.8% of extension pairs failed to work together. The matrix in Figure 5 shows the compatibility testing results. Each green square in the graph indicates a successful, compatible pair of extensions, while each red square indicates that the pair of extensions failed to operate together correctly. The extensions in the graph are sorted from lowest to highest compatibility failure rate. This figure reveals that while most extensions are compatible with one another, some extensions have higher failure rates.

The zoomed view in Figure 5 highlights the 20 extensions with the highest failure rates. Of these tests, 255 of 380 (67.1%) fail and about a quarter (29.2%) of those fail due to brittle test cases. Most PostgreSQL extensions use the built-in testing harness (pg\_regress). The problem with this harness is that it uses string comparisons on the output of SQL commands to determine whether a test case succeeds. When our toolkit loads multiple extensions together, it can change the output of some commands (e.g., catalog queries) and cause spurious test failures.

Through manually examining our output, we found bugs in extensions and parts of PostgreSQL. We discuss these below:

- **Extension Bugs:** `lsm3` introduces a latch deadlock that causes the DBMS to hang with 16 of 95 (16.8%) extensions due to a



**Figure 5: Compatibility Tests** – Left matrix shows the results of all compatibility testing. Right matrix shows the results of the 20 extensions with the highest failure rate.

background worker synchronization bug. `plprofiler` [61] conflicts with 12 of 95 (12.6%) extensions due to a shared memory initialization error.

- **Brittle Configuration Settings:** Since `test_decoding` [88] and `wal2json` [114] are both logical decoding extensions, they use replication slots. Their tests produced an error due to too few allocated replication slots.
- **Memory Errors:** The toolkit caused the DBMS to crash due to a segmentation fault with `pg_show_plans` [80] and `pg_repack` [78] when the former runs out of memory. The same crash occurs when `logerrors` [35] and `plprofiler` [61] run out of memory.
- **UDO Incompatibilities:** PostgreSQL misinterprets UDOs for several extensions [69, 70, 72, 85] as their own. When their function signatures do not match the DBMS’s API, it errors out.
- **Query Processing Bugs:** If the DBMS uses `timescaledb` [112] with `pg_hint_plan` [74], then the latter is unable to override the query’s join order, causing unexpected behavior. Installing `pg_stat_monitor` [82] together with `pg_stat_statements` [83] causes the former to record incorrect data in its auxiliary tables.
- **Installation Order:** The DBMS fails to start when it does not load (1) `citus` [24] first or (2) `pg_queryid` [77] last.

These failures indicate there are rampant compatibility issues between extensions. We next identified what aspects of their implementations cause problems. We ran paired T-tests with a p-value of 0.1 on the compatibility failure rates. Although there are risks with using paired T-tests on multiple hypotheses, we decided to run tests on many extensibility factors to determine possible correlations. We split them into two groups based on whether an extension:

- Uses a certain hook in their source code.
- Uses more than  $n$  hooks in their source code ( $n = [0, 7]$ ).
- Utilizes a certain extensibility type.
- Contains source code copied from PostgreSQL.
- Contains versioning logic.
- Contains more than  $n$  lines of source code ( $n = \{500, 750, 1000\}$ ).
- Uses more than  $n$  extensibility types ( $n = [1, 4]$ ).
- Uses certain system components.

Of the 55 factors we considered in our T-paired analysis, we found that the following have the strongest correlation to failures.

	Feature	p-value
ARCH.	Function Usage	1.13e-6
	Storage Manager Component	0.0375
CODE	Duplicate PostgreSQL Code	0.0172
	More than 1000 LOC	0.0448
	More than 750 LOC	0.0915
IMPL.	More than 3 Components	0.0920
	More than 3 Hooks	0.0715
	More than 4 Hooks	0.0023
	More than 4 Components	0.0015

**Table 7: Failure Correlations by General Features** – Lists the p-values associated with each of our paired T-tests grouped by the extensions’ architecture (ARCH.), source code (CODE), and implementation (IMPL.) features.

The foremost reason is that higher complexity causes lower compatibility: if an extension uses many system components, extensibility types, or hooks, it is more likely to break other extensions.

Second, using larger-scope query planner and execution hooks (e.g., `planner_hook`, `ProcessUtility_hook`) and smaller scope hooks (e.g., `set_join_pathlist_hook`) decreases compatibility. Using such hooks in an extension requires a deep understanding of the DBMS’s query engine to ensure that it does not cause problems; if two extensions modify the same query engine state, they break unpredictably.

Other codebase complexity measures, such as duplicate code usage or having a larger codebase, also decreased compatibility. This aligns with our observations since extensions with these features require more maintenance and effort to ensure they work correctly.

Using UDFs correlates to failures since many complex extensions use UDFs. Notably, extensions only using UDFs had an average compatibility failure rate of 10.7%, while extensions that used UDFs and other extensibility types had an average failure rate of 22.2%.

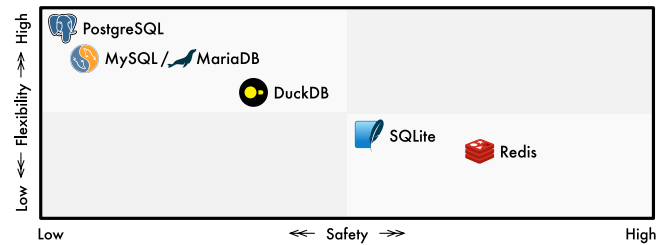
## 6 DISCUSSION

We now summarize our findings on DBMS extensibility and provide recommendations for developers who want to add or improve extension support in a DBMS. PostgreSQL has the most comprehensive extensibility APIs and the largest extension ecosystem. Many of its design decisions are worth replicating in other DBMSs, but our survey and analysis also showed that it is not without its flaws.

### 6.1 Safety vs. Flexibility

Designing extensibility with the proper safety mechanisms means ensuring the DBMS will not produce incorrect results or crash. This goal can conflict with the desire to support expansive extensions that can modify the DBMS’s runtime behavior. An example of opting for safety is SQLite’s limitation that only allows one filesystem extension at a time to avoid compatibility issues. PostgreSQL is the other extreme, where extensions are unrestricted in manipulating the core system and each other.

Figure 6 shows the six DBMSs surveyed on the safety-flexibility spectrum. We consider PostgreSQL’s framework the most flexible and Redis’ the safest. We consider Redis the least flexible because it does not allow users to access system state and it only supports one interface for writing extensions. This trade-off is important to consider because it determines the API that a DBMS exposes to extension developers and the possible extensions. For example, most of SQLite’s extensions are UDFs, but the extensions for PostgreSQL,



**Figure 6: Safety vs. Flexibility** – An approximation of how the DBMSs consider safety vs. flexibility trade-offs from Section 6.1.

MySQL, MariaDB, and DuckDB are more diverse. On the other hand, we showed in Section 5.4 that a too-permissive extensibility API can have stability issues.

### 6.2 Extension Composability

Our survey showed that DBMS extensibility is not inherently composable. There is no built-in way for an extension to reference another extension’s code, components, or state. Instead, developers rely on their understanding of an extension and then implement their extension to match this behavior. This does not stop developers from using extensions as dependencies in their implementations. For example, many PostgreSQL extensions [74, 77, 79, 81] rely on `pg_stat_statements` [83] to obtain planning and execution statistics. Another example is the InnoDB-specific catalog extensions in MySQL and MariaDB. These examples provide a compelling argument for DBMSs to support extension composability formally.

PostgreSQL’s hook “chaining” is the root of many problems we identified, as it creates a new call path that an extension is unlikely to test beforehand. PostgreSQL requires that any extension that connects via a hook must also call the previously set hook to ensure both extensions are called, but it has no way of checking whether an extension complies with this. Furthermore, extensions might call internal query execution functions that then call other hooks. Such chaining means the DBMS might invoke an extension in an unfamiliar state. Given this, using a SQL interface over low-level hooks (e.g., event triggers for DDL changes) is preferable.

### 6.3 Extension Redundancies

Our survey reveals that many extensions re-implemented the same functionality for different DBMSs. For example, both MySQL and PostgreSQL have job scheduling extensions: `mysql_query_queue` [51] and `pg_cron` [73]. Almost all the DBMSs also have vector index extensions: PostgreSQL’s `pgvector` [58], SQLite’s `sqlite-vss` [110], DuckDB’s `duckdb_vss` [113], and MySQL’s `mysql_vss` [53]. To reduce this repetitive effort, we argue that the database community should develop a standardized POSIX-like API [129, 146] for extensions so that DBMSs can share extensions.

### 6.4 DBMS Extensibility in the Cloud

Managed DBMS services want to allow users to create extensions, but they do not want to compromise the integrity and security of their systems. This is problematic, especially for PostgreSQL, since extensions can expose low-level access to the host machine (e.g., file system, remote shell).

There are different strategies to deal with this problem. The first is to modify the DBMS to introduce a new admin role with reduced permissions for installing extensions. For example, when a user installs an extension in AWS RDS PostgreSQL [20], the DBMS switches to an administrator role with unique permissions (e.g., no file system access) to add an extension. The second option is to modify the source code of allowed extensions to avoid these problems. Google Cloud SQL [33] allows a regular user to create an extension, which prevents escalation issues but is risky because it means regular users own extension catalog tables. For instance, a user install a trigger on a `pg_cron` [73] catalog table and then wait for the background worker with full administrative permissions to write to that table and then execute the trigger. However, Google’s developers modified `pg_cron` to prevent this from happening. Both approaches are laborious and error-prone; a better approach is to use a sandbox to reduce the exposure of the DBMS to such attacks.

## 6.5 API Lessons

Our correlation analysis shows that extensions relying on coarse-grained hooks have higher compatibility failure rates. However, the extensions using these hooks often add relatively simple functionality, such as statistics collection. The DBMS should expose more fine-grained hooks for common scenarios that do not carry the compatibility risks identified in Section 5.4.

PostgreSQL extensions copy code when they need to process internal data structures that the extension API permits them to access. The DBMS gives extensions a lot of responsibility to manage the system’s internal state (e.g., overriding core DBMS functions) but without the necessary capability to do so (e.g., internal helper/API functions to modify these internal data structures). Therefore, developers copy code to use and modify these data structures. Other problems occur when extensions rely on the representations of a DBMS’s internal objects, but those objects change in a new DBMS version (e.g., adding a new field to a struct) [149]. PostgreSQL is designed to be extensible, and this is commendable. However, it is a beautiful monstrosity because extensions can do essentially anything in PostgreSQL.

Lastly, PostgreSQL makes building and testing extensions relatively easy via its tooling. This partly explains why it has a more extensive ecosystem than MySQL and MariaDB, even though they have similar extensibility features. However, PostgreSQL lacks an authoritative platform for publishing extensions. In contrast, DuckDB maintainers provide a platform for community extensions. Given the success of package managers for programming languages (e.g., Rust [102], Python [92], JavaScript [54]), every DBMS should strive to have an effective management tool for extensions. Such a platform can also address quality concerns by recompiling extensions for new patch releases and running cross-extension tests.

## 7 RELATED WORK

We now discuss prior work on extensibility across system fields.

**DBMS Extensibility:** One of the first studies on DBMS extensibility from the early 1990s examined emerging frameworks for building DBMSs [125]. This study considers extensible DBMSs [122, 126, 144, 148, 154] and identifies three extensibility types: (1) user interface, (2) query processing, and (3) storage extensions.

Since then, several works have proposed new types of DBMS extensibility. For example, Umbra proposed user-defined operators (UDOs) to support custom operations on UDTs [90]. Second, a few recent works [129, 146, 147] have argued that the databases community should prioritize composability, which refers to building new systems from a collection of its naturally well-defined components. Both extensibility and composability argue for reusable, customizable components within DBMSs. Unlike extensibility, composability argues that the DBMS should consist of these components.

**Operating System Extensibility:** Understanding OS extensibility is helpful because we can apply existing terminology, techniques, and support mechanisms to categorize and analyze database system extensibility. Linux’s extensions [9] are shared objects written in C, while MacOS [21] provides a user-space API. Safer versions of extensibility that use sandboxing, verification, type-safe languages, and isolation techniques have been implemented successfully in OSs [15, 123, 132, 140, 155]. These efforts introduce many valuable ideas for safe, reliable extensibility frameworks. In the future, it may be helpful to apply the ideas of user-space extensions, sandboxing, formal extensibility frameworks, crash recovery, and user-controlled research management to DBMS extensibility.

**Browser Extensibility:** There are ~130,000 Google Chrome extensions [130] and ~44,000 Mozilla Firefox extensions [32]. As a result, security is a significant focus in browser extensibility research [121, 139, 156, 157]. Both Chrome [23, 121] and Firefox [44] support security features for extensions. Applying the lessons learned to ensure safety in browser extensions is highly relevant to developing security measures for DBMS extensions.

**Feature Interactions:** In software engineering, feature interactions research [138, 141, 150] focuses on understanding how two different features in a software system modify each other’s behavior. Such research is relevant to DBMS extensibility because one can represent extensions as features in a system. This means applying these results to DBMSs and using static and dynamic analysis to make extension development easier is possible.

## 8 CONCLUSION

This paper presented an evaluation of modern DBMS extensibility. We first provided an overview of the major DBMS extensibility types and then expounded on five implementation considerations for extensions: (1) integration interfaces, (2) state modification methods, (3) protection mechanisms, (4) supporting components, and (5) developer APIs. We examined six open-source DBMSs and showed how their extension implementations differ. We then performed a comprehensive analysis of PostgreSQL’s extensibility ecosystem. To help with this analysis, we developed a toolkit that automatically deploys and tests extensions. Our results demonstrate the challenges in trading off DBMS safety for extension flexibility. This work highlights existing problems of DBMS extensibility and motivates future research on improving this critical capability.

## ACKNOWLEDGMENTS

The authors would like to thank the database extensions community for their support and feedback on this work. Particularly, we are very grateful to everyone who interacted with our talk at the PostgreSQL Development Conference 2024!



## REFERENCES

- [1] 2003. *PostgreSQL v7.4.0 Release Notes*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/release/7.4.0/>
- [2] 2005. *PostgreSQL v8.1.23 Documentation: Appendix E. Release Notes*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/8.1/release-8-1.html>
- [3] 2008. *Oracle v8 Release Notes*. Retrieved 2025-04-04 from [https://www.oraFAQ.com/wiki/Oracle\\_8](https://www.oraFAQ.com/wiki/Oracle_8)
- [4] 2013. *PostgreSQL: pg\_autovacuum*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/8.3/catalog-pg-autovacuum.html>
- [5] 2016. *Oracle v7 Release Notes*. Retrieved 2025-04-04 from [https://www.oraFAQ.com/wiki/Oracle\\_7](https://www.oraFAQ.com/wiki/Oracle_7)
- [6] 2019. *MariaDB: UUID Data Type*. Retrieved 2025-04-04 from <https://mariadb.com/kb/en/mariadb-plugin/>
- [7] 2019. *MySQL: CVE-2019-19959*. Retrieved 2025-04-04 from <https://nvd.nist.gov/vuln/detail/CVE-2019-19959>
- [8] 2019. *SQL Server: User-defined Functions*. Retrieved 2025-04-04 from <https://learn.microsoft.com/en-us/archive/msdn-magazine/2003/november/data-points-sql-server-user-defined-functions>
- [9] 2020. *The Linux Documentation Project*. Retrieved 2025-04-04 from <https://tldp.org/>
- [10] 2020. *MySQL: CVE-2020-13630*. Retrieved 2025-04-04 from <https://nvd.nist.gov/vuln/detail/CVE-2020-13630>
- [11] 2020. *PostgreSQL: CVE-2020-14350*. Retrieved 2025-04-04 from <https://www.postgresql.org/support/security/CVE-2020-14350/>
- [12] 2022. *MySQL: CVE-2022-21454*. Retrieved 2025-04-04 from <https://nvd.nist.gov/vuln/detail/CVE-2022-21454>
- [13] 2022. *PostgreSQL: CVE-2022-2625*. Retrieved 2025-04-04 from <https://www.postgresql.org/support/security/CVE-2022-2625/>
- [14] 2022. *Redis: Lua scripts can be manipulated to overcome ACL rules*. Retrieved 2025-04-04 from <https://github.com/redis/redis/security/advisories/GHSA-647m-2wmq-qmvm>
- [15] 2023. *eBPF Documentation*. Retrieved 2025-04-04 from <https://ebpf.io/what-is-ebpf/>
- [16] 2023. *PostgreSQL: Contrib module*. Retrieved 2025-04-04 from <https://pgpedia.info/c/contrib-module.html>
- [17] 2023. *PostgreSQL: CVE-2023-39417*. Retrieved 2025-04-04 from <https://www.postgresql.org/support/security/CVE-2023-39417/>
- [18] 2023. *PostgreSQL v15.3 Release Notes*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/release/15.3/>
- [19] 2023. *SQL Server: CLR User-Defined Types*. Retrieved 2025-04-04 from <https://learn.microsoft.com/en-us/sql/relational-databases/clr-integration/database-objects-user-defined-types/clr-user-defined-types?view=sql-server-ver16>
- [20] 2024. *Amazon RDS for PostgreSQL*. Retrieved 2025-04-04 from <https://aws.amazon.com/rds/postgresql/>
- [21] 2024. *Apple OS X System Extensions*. Retrieved 2025-04-04 from <https://developer.apple.com/documentation/systemextensions>
- [22] 2024. *Azure Database for PostgreSQL*. Retrieved 2025-04-04 from <https://azure.microsoft.com/en-us/products/postgresql>
- [23] 2024. *Chrome Extensions: Manifest - Sandbox*. Retrieved 2025-04-04 from <https://developer.chrome.com/docs/extensions/reference/manifest/sandbox>
- [24] 2024. *Citus*. Retrieved 2025-04-04 from <https://github.com/citusdata/citus>
- [25] 2024. *Citus Columnar*. Retrieved 2025-04-04 from <https://github.com/citusdata/citus/tree/main/src/backend/columnar>
- [26] 2024. *Create hooks to let a loadable plugin monitor (or even replace) the planner*. Retrieved 2025-04-04 from <https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=604ffd280b955100e5fc24649ee4d42a6f3ebf35>
- [27] 2024. *DuckDB: Community Extensions Repository*. Retrieved 2025-04-04 from <https://github.com/duckdb/community-extensions>
- [28] 2024. *DuckDB: Postgres Scanner*. Retrieved 2025-04-04 from [https://github.com/duckdb/postgres\\_scanner](https://github.com/duckdb/postgres_scanner)
- [29] 2024. *DuckDB: SQLite Scanner*. Retrieved 2025-04-04 from [https://github.com/duckdb/sqlite\\_scanner](https://github.com/duckdb/sqlite_scanner)
- [30] 2024. *ElasticSearch*. Retrieved 2025-04-04 from <https://www.elastic.co/elasticsearch>
- [31] 2024. *Finding duplicated code with CPD*. Retrieved 2025-04-04 from [https://pmd.github.io/pmd/pmd\\_userdocs\\_cpd](https://pmd.github.io/pmd/pmd_userdocs_cpd)
- [32] 2024. *Firefox Browser Add-Ons*. Retrieved 2025-04-04 from <https://addons.mozilla.org/en-US/firefox/search/?type=extension>
- [33] 2024. *Google Cloud SQL*. Retrieved 2025-04-04 from <https://cloud.google.com/sql/postgresql>
- [34] 2024. *Kerberos Pluggable Authentication*. Retrieved 2025-04-04 from <https://dev.mysql.com/doc/mysql-security-excerpt/8.0/en/kerberos-pluggable-authentication.html>
- [35] 2024. *logerrors*. Retrieved 2025-04-04 from <https://github.com/munakoiso/logerrors>
- [36] 2024. *Logical Decoding Concepts*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/logicaldecoding-explanation.html>
- [37] 2024. *MariaDB*. Retrieved 2025-04-04 from <https://mariadb.com/docs/server/ref/cs10.4/>
- [38] 2024. *MariaDB: ColumnStore Engine*. Retrieved 2025-04-04 from <https://mariadb.com/docs/columnstore/>
- [39] 2024. *MariaDB: Mroonga*. Retrieved 2025-04-04 from <https://mariadb.com/kb/en/mroonga/>
- [40] 2024. *MariaDB: Package Repository Setup and Usage*. Retrieved 2025-04-04 from <https://mariadb.com/kb/en/mariadb-package-repository-setup-and-usage/>
- [41] 2024. *mariadb-plugin*. Retrieved 2025-04-04 from <https://mariadb.com/kb/en/mariadb-plugin/>
- [42] 2024. *MariaDB: S3 Storage Engine*. Retrieved 2025-04-04 from <https://mariadb.com/kb/en/s3-storage-engine/>
- [43] 2024. *Microsoft SQL Server*. Retrieved 2025-04-04 from <https://www.microsoft.com/en-us/sql-server>
- [44] 2024. *Mozilla Wiki: Security/Sandbox*. Retrieved 2025-04-04 from <https://wiki.mozilla.org/Security/Sandbox>
- [45] 2024. *MySQL*. Retrieved 2025-04-04 from <http://www.mysql.com>
- [46] 2024. *MySQL: CVE-2024-20985*. Retrieved 2025-04-04 from <https://nvd.nist.gov/vuln/detail/CVE-2024-20985>
- [47] 2024. *MySQL: FEDERATED Storage Engine*. Retrieved 2025-04-04 from <https://dev.mysql.com/doc/refman/8.0/en/federated-storage-engine.html>
- [48] 2024. *MySQL: Introduction to InnoDB*. Retrieved 2025-04-04 from <https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html>
- [49] 2024. *MySQL: MEMORY Storage Engine*. Retrieved 2025-04-04 from <https://dev.mysql.com/doc/refman/8.0/en/memory-storage-engine.html>
- [50] 2024. *MySQL: Password Validation Component*. Retrieved 2025-04-04 from <https://dev.mysql.com/doc/refman/8.3/en/validate-password.html>
- [51] 2024. *MySQL: Query Job Queue*. Retrieved 2025-04-04 from [https://github.com/adrrpar/mysql\\_query\\_queue](https://github.com/adrrpar/mysql_query_queue)
- [52] 2024. *MySQL: Rewriter Query Rewrite Plugin Reference*. Retrieved 2025-04-04 from <https://dev.mysql.com/doc/refman/8.0/en/rewriter-query-rewrite-plugin-reference.html>
- [53] 2024. *mysql\_vss*. Retrieved 2025-04-04 from [https://github.com/stephenc22/mysql\\_vss](https://github.com/stephenc22/mysql_vss)
- [54] 2024. *npm*. Retrieved 2025-04-04 from <https://www.npmjs.com/>
- [55] 2024. *Oracle*. Retrieved 2025-04-04 from <https://www.oracle.com>
- [56] 2024. *ParadeDB*. Retrieved 2025-04-04 from <https://www.paradedb.com/>
- [57] 2024. *pg\_regress*. Retrieved 2025-04-04 from [https://github.com/postgres/postgres/blob/master/src/test/regress/pg\\_regress.c](https://github.com/postgres/postgres/blob/master/src/test/regress/pg_regress.c)
- [58] 2024. *pgvector*. Retrieved 2025-04-04 from <https://github.com/pgvector/pgvector>
- [59] 2024. *PGXN: PostgreSQL Extension Network*. Retrieved 2025-04-04 from <https://pgxn.org/>
- [60] 2024. *pgxs*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/extend-pgxs.html>
- [61] 2024. *plProfiler*. Retrieved 2025-04-04 from <https://github.com/bigsql/plprofiler>
- [62] 2024. *PL/Rust*. Retrieved 2025-04-04 from <https://github.com/tcdi/plrust>
- [63] 2024. *PLV8*. Retrieved 2025-04-04 from <https://github.com/plv8/plv8>
- [64] 2024. *PostGIS*. Retrieved 2025-04-04 from <https://postgis.net/>
- [65] 2024. *PostgreSQL Benchmark (pgbench)*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/pgbench.html>
- [66] 2024. *PostgreSQL: CREATE EXTENSION*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/15/sql-createextension.html>
- [67] 2024. *PostgreSQL: GiST and GIN Index Types*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/9.1/textsearch-indexes.html>
- [68] 2024. *PostgreSQL Global Development Group*. Retrieved 2025-04-04 from <https://www.postgresql.org/developer/core/>
- [69] 2024. *PostgreSQL: isn*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/isn.html>
- [70] 2024. *PostgreSQL: lsm3*. Retrieved 2025-04-04 from <https://github.com/postgrespro/lsm3>
- [71] 2024. *PostgreSQL: passwordcheck*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/passwordcheck.html>
- [72] 2024. *PostgreSQL: pg\_bigm*. Retrieved 2025-04-04 from [https://github.com/pgbigm/pg\\_bigm](https://github.com/pgbigm/pg_bigm)
- [73] 2024. *PostgreSQL: pg\_cron*. Retrieved 2025-04-04 from [https://github.com/citusdata/pg\\_cron](https://github.com/citusdata/pg_cron)
- [74] 2024. *PostgreSQL: pg\_hint\_plan*. Retrieved 2025-04-04 from [https://github.com/oss-c-db/pg\\_hint\\_plan](https://github.com/oss-c-db/pg_hint_plan)
- [75] 2024. *PostgreSQL: pg\_ivm*. Retrieved 2025-04-04 from [https://github.com/sraoss/pg\\_ivm](https://github.com/sraoss/pg_ivm)
- [76] 2024. *PostgreSQL: pgpointcloud*. Retrieved 2025-04-04 from <https://github.com/pgpointcloud/pointcloud>
- [77] 2024. *PostgreSQL: pg\_queryid*. Retrieved 2025-04-04 from [https://github.com/rjuju/pg\\_queryid](https://github.com/rjuju/pg_queryid)

- [78] 2024. *PostgreSQL: pg\_repack*. Retrieved 2025-04-04 from [https://github.com/reorg/pg\\_repack](https://github.com/reorg/pg_repack)
- [79] 2024. *PostgreSQL: pgstatintel*. Retrieved 2025-04-04 from <https://github.com/pgstatintel/pgstatintel>
- [80] 2024. *PostgreSQL: pg\_show\_plan*. Retrieved 2025-04-04 from [https://github.com/cybertec-postgresql/pg\\_show\\_plans](https://github.com/cybertec-postgresql/pg_show_plans)
- [81] 2024. *PostgreSQL: pgstatkcache*. Retrieved 2025-04-04 from [https://github.com/powa-team/pg\\_stat\\_kcache](https://github.com/powa-team/pg_stat_kcache)
- [82] 2024. *PostgreSQL: pg\_stat\_monitor*. Retrieved 2025-04-04 from [https://github.com/percona/pg\\_stat\\_monitor](https://github.com/percona/pg_stat_monitor)
- [83] 2024. *PostgreSQL: pg\_stat\_statements*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/pgstatstatements.html>
- [84] 2024. *PostgreSQL: postgres\_fdw*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/postgres-fdw.html>
- [85] 2024. *PostgreSQL: Prefix Range Module*. Retrieved 2025-04-04 from <https://github.com/dimitri/prefix>
- [86] 2024. *PostgreSQL: Rust Extensions Framework (pgrx)*. Retrieved 2025-04-04 from <https://docs.rs/pgrx/latest/pgrx/>
- [87] 2024. *PostgreSQL: SECURITY LABEL*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/sql-security-label.html>
- [88] 2024. *PostgreSQL: test\_decoding*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/test-decoding.html>
- [89] 2024. *PostgreSQL: Trusted Language Extensions*. Retrieved 2025-04-04 from [https://github.com/aws/pg\\_tle](https://github.com/aws/pg_tle)
- [90] 2024. *PostgreSQL: User-Defined Operators*. Retrieved 2025-04-04 from <https://www.postgresql.org/docs/current/xoper.html>
- [91] 2024. *Provide a function hook to let plug-ins get control around ExecutorRun*. Retrieved 2025-04-04 from <https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=6cc88f0af5b12b22ce1826a26b1a953c434bd165>
- [92] 2024. *Python PIP*. Retrieved 2025-04-04 from <https://pypi.org/project/pip/>
- [93] 2024. *Redis*. Retrieved 2025-04-04 from <https://redis.io/>
- [94] 2024. *Redis Adopts Dual Source-Available Licensing*. Retrieved 2025-04-04 from <https://redis.io/blog/redis-adopts-dual-source-available-licensing/>
- [95] 2024. *Redis: Commands*. Retrieved 2025-04-04 from <https://redis.io/docs/latest/develop/reference/modules/>
- [96] 2024. *Redis Modules (Archived from 2024-04-06)*. Retrieved 2024-04-06 from <https://web.archive.org/web/20240406192236/https://redis.io/resources/modules/>
- [97] 2024. *Redis Modules Software Development Kit*. Retrieved 2025-04-04 from <https://github.com/RedisLabsModules/RedisModulesSDK>
- [98] 2024. *Redis: pam\_auth*. Retrieved 2025-04-04 from [https://github.com/RedisLabsModules/pam\\_auth](https://github.com/RedisLabsModules/pam_auth)
- [99] 2024. *RedisSearch*. Retrieved 2025-04-04 from <https://github.com/RedisSearch/RedisSearch>
- [100] 2024. *RedisGraph*. Retrieved 2025-04-04 from <https://github.com/RedisGraph/RedisGraph>
- [101] 2024. *RUM - RUM access method*. Retrieved 2025-04-04 from <https://github.com/postgrespro/rum>
- [102] 2024. *Rust Cargo*. Retrieved 2025-04-04 from <https://github.com/rust-lang/cargo>
- [103] 2024. *Some infrastructure changes for the upcoming auto-explain contrib module*. Retrieved 2025-04-04 from <https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=cd35e9d7468e8f86dd5a7d928707f4ba8cdae44d>
- [104] 2024. *SQLite*. Retrieved 2025-04-04 from <https://www.sqlite.org/index.html>
- [105] 2024. *SQLite: Encryption Extension*. Retrieved 2025-04-04 from <https://www.sqlite.org/see/doc/trunk/www/readme.wiki>
- [106] 2024. *SQLite Extensions*. Retrieved 2025-04-04 from <https://github.com/riyaz-ali/sqlite>
- [107] 2024. *SQLite: os\_unix*. Retrieved 2025-04-04 from [https://sqlite.org/src/file?name=src/os\\_unix.c&ci=trunk](https://sqlite.org/src/file?name=src/os_unix.c&ci=trunk)
- [108] 2024. *SQLite: sqlean*. Retrieved 2025-04-04 from <https://github.com/nalgeon/sqlean>
- [109] 2024. *SQLite: sqlite-loadable-rs*. Retrieved 2025-04-04 from <https://github.com/asg017/sqlite-loadable-rs>
- [110] 2024. *SQLite: sqlite-vss*. Retrieved 2025-04-04 from <https://github.com/asg017/sqlite-vss>
- [111] 2024. *SQLite: sqLite3ext.h*. Retrieved 2025-04-04 from <https://www2.sqlite.org/src/file?name=src/sqlite3ext.h>
- [112] 2024. *TimescaleDB*. Retrieved 2025-04-04 from <https://github.com/timescale/timescaledb>
- [113] 2024. *Vector Similarity Search*. Retrieved 2025-04-04 from [https://github.com/duckdb/duckdb\\_vss](https://github.com/duckdb/duckdb_vss)
- [114] 2024. *wal2json*. Retrieved 2025-04-04 from <https://github.com/eulerto/wal2json>
- [115] 2024. *zeeSQL*. Retrieved 2025-04-04 from <https://zeesql.com>
- [116] 2025. *Database Extensions Analyzer*. Retrieved 2025-04-04 from <https://github.com/cmu-db/ext-analyzer>
- [117] 2025. *DuckDB*. Retrieved 2025-04-04 from <https://duckdb.org/>
- [118] 2025. *Oracle: Introduction to Oracle Database Extensions for .NET*. Retrieved 2025-04-04 from [https://docs.oracle.com/cd/E11882\\_01/win.112/e17724/intro.htm](https://docs.oracle.com/cd/E11882_01/win.112/e17724/intro.htm)
- [119] 2025. *PostgreSQL*. Retrieved 2025-04-04 from <https://www.postgresql.org>
- [120] Sam Ansmink. 2024. *DuckDB – C API Extensions*. Retrieved 2025-04-04 from <https://github.com/duckdb/duckdb/pull/12682>
- [121] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. 2010. Protecting Browsers from Extension Vulnerabilities. In *Network and Distributed System Security Symposium*.
- [122] D.S. Batoory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise. 1988. GENESIS: an extensible database management system. *IEEE Transactions on Software Engineering* 14, 11 (1988), 1711–1730. <https://doi.org/10.1109/32.9057>
- [123] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility Safety and Performance in the SPIN Operating System. *SIGOPS Oper. Syst. Rev.* 29, 5 (dec 1995), 267–283. <https://doi.org/10.1145/224057.224077>
- [124] Fuyuan Cao, Jiye Liang, and Liang Bai. 2009. A new initialization method for categorical data clustering. *Expert Syst. Appl.* 36 (09 2009), 10223–10228. <https://doi.org/10.1016/j.eswa.2009.01.060>
- [125] Michael Carey and Laura Haas. 1990. Extensible Database Management Systems. *SIGMOD Rec.* 19, 4 (dec 1990), 54–60. <https://doi.org/10.1145/122058.122064>
- [126] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M. Muralikrishna. 1991. *The Architecture of the EXODUS Extensible DBMS*. 231–256. [https://doi.org/10.1007/978-3-642-84374-7\\_15](https://doi.org/10.1007/978-3-642-84374-7_15)
- [127] Claude. 2024. *Generating small UDT datasets*. Retrieved 2025-04-04 from <https://claude.ai/chat>
- [128] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 2490–2502. <https://doi.org/10.1145/3448016.3457551>
- [129] Voltron Data. 2023. *The Composable Codex*. Retrieved 2025-04-04 from <https://voltrondata.com/codex>
- [130] Brian Dean. 2024. *Google Chrome Statistics*. Retrieved 2025-04-04 from <https://backlinko.com/chrome-users>
- [131] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (Oct. 2021), 32 pages. <https://doi.org/10.1145/3483840>
- [132] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. 251–266. <https://doi.org/10.1145/224056.224076>
- [133] Joseph M Hellerstein, Jeffrey F Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*. 562–573.
- [134] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084. <https://doi.org/10.14778/3415478.3415535>
- [135] IBM. 2000. *DB2 Universal Database for OS/390 IBM Release Planning Guide Version 6*. International Business Machines Corporation.
- [136] International Organization for Standardization 1996. *ISO/IEC 9075-4:1996* (1st ed.). International Organization for Standardization.
- [137] International Organization for Standardization 1999. *ISO/IEC 9075-2:1999* (1st ed.). International Organization for Standardization.
- [138] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. 2017. *On the Relation of External and Internal Feature Interactions: A Case Study*. Technical Report 1712.07440. arXiv. <https://arxiv.org/abs/1712.07440>
- [139] David M Martin Jr, Richard M Smith, Michael Brittain, Ivan Fetch, and Hailin Wu. 2001. The privacy practices of web browser extensions. *Commun. ACM* 44, 2 (2001), 45–50.
- [140] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (San Diego, California) (USENIX'93). USENIX Association, USA, 2.
- [141] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems (ASE '16). 483–494. <https://doi.org/10.1145/2970276.2970322>
- [142] Hannes Mühleisen and Mark Raasveldt. 2025. Runtime-Extensible Parsers. In *15th Annual Conference on Innovative Data Systems Research (CIDR '25), Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org. <https://duckdb.org/pdf/CIDR2025-muehleisen-raasveldt-extensible-parsers.pdf>
- [143] James Ong, Dennis Fogg, and Michael Stonebraker. 1983. Implementation of data abstraction in the relational database system INGRES. *SIGMOD Rec.* 14, 1 (sep 1983), 1–14. <https://doi.org/10.1145/984540.984541>
- [144] Sylvia L. Osborn and T. E. Heaven. 1986. The Design of a Relational Database System with Abstract Data Types for Domains. *ACM Trans. Database Syst.* 11, 3 (aug 1986), 357–373. <https://doi.org/10.1145/6314.6461>

- [145] Sasha Pachev. 2007. *Understanding MySQL Internals*. O'Reilly Media, Inc., Chapter 1. MySQL History and Architecture.
- [146] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2679–2685.
- [147] Danica Porobic. 2019. Revisiting RISC-style Data Management System Design.. In *CIDR*.
- [148] P. Schwarz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. 1986. Extensibility in the Starburst Database System. In *Proceedings on the 1986 International Workshop on Object-Oriented Database Systems (OODS '86)*. 85–92.
- [149] Marco Slot. 2024. . Retrieved 2025-04-04 from <https://twitter.com/marcoslot/status/1858132850383421570>
- [150] Larissa Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Almeida. 2018. VarXplorer: Lightweight Process for Dynamic Analysis of Feature Interactions. In *VAMOS 2018: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. 59–66. <https://doi.org/10.1145/3168365.3168376>
- [151] Michael Stonebraker. 2023. Personal Correspondence.
- [152] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 2–11. <https://doi.org/10.1109/ICDE.2005.1>
- [153] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. 1976. The design and implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (sep 1976), 189–222. <https://doi.org/10.1145/320473.320476>
- [154] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. 340–355. <https://doi.org/10.1145/16894.16888>
- [155] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. 2002. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop (EW 10)*. 102–107. <https://doi.org/10.1145/1133373.1133393>
- [156] Mike Ter Louw, Jin Soon Lim, and Venkat N Venkatakrishnan. 2007. Extensible web browser security. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 4th International Conference, DIMVA 2007 Lucerne, Switzerland, July 12-13, 2007 Proceedings* 4. Springer, 1–19.
- [157] Mike Ter Louw, Jin Soon Lim, and Venkat N Venkatakrishnan. 2008. Enhancing web browser security against malware extensions. *Journal in Computer Virology* 4 (2008), 179–195.
- [158] Wikipedia. 2024. *Comparison of MySQL database engines* — Wikipedia, The Free Encyclopedia. Retrieved 2025-04-04 from [https://en.wikipedia.org/wiki/Comparison\\_of\\_MySQL\\_database\\_engines](https://en.wikipedia.org/wiki/Comparison_of_MySQL_database_engines)