Survey and Evaluation of Database Management System Extensibility

Abigale Kim

CMU-CS-23-144

January 2024

Computer Science Department
School of Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andrew Pavlo, Chair
David Andersen

Submitted in partial fulfillment of the requirements for the Master's degree in Computer Science.

Keywords: database management systems, static analysis, system extensibility, database management system extensibility

Abstract

Database management system (DBMS) extensibility is a feature that enables users to extend the DBMS with user software. However, the DBMS extensibility environment is fraught with perils, and DBMS developers have to resort to unspecified methods of developing extensions, including copying core DBMS source code and casing between different versions of the DBMS. Extending a DBMS to support new functionality is challenging due to the tight coupling between the system's internal components. This thesis studies and evaluates the design of DBMS extensibility. We first provide a comprehensive taxonomy of the types of extensibility supported by DBMSs and the effects of supporting their functionality within the DBMS. Given that PostgreSQL has the most variegated extensibility ecosystem, we also provide an in-depth analysis of it, where we evaluate how compatible extensions were with one another, extension source code quality, and extension complexity. To assist us with this evaluation, we introduce an automated PostgreSQL extension analysis framework that collects information on how an extension integrates into the DBMS. We present results from static and dynamic analysis for over 100 extensions. We show correlations between the lack of compatibility of extensions and several factors related to their complexity and source code. We conclude by discussing the design decisions and trade-offs with supporting extensions in a DBMS.

Acknowledgments

This thesis would not have been possible without the support and guidance of my advisor, Andy Pavlo. Andy has helped me learn so much about databases, encouraged me when I was down, and made me laugh on many occasions. He is the reason I discovered my love for databases. Additionally, words cannot express my gratitude to Dave Andersen for his generously provided general systems knowledge (e.g., operating systems) and research advice throughout the last seven months. I am also deeply grateful to Marco Slot for his guidance and enthusiasm throughout this research project and for answering all of my questions about complicated PostgreSQL internals.

I would also like to thank the students in the Carnegie Mellon Databases Group and Parallel Data Lab for their feedback on my research, general advice, late night study sessions, and friendship over the past years. This list includes but is not limited to: Rohan Aggrawal, Sam Arch, Mayank Baranwal, Jennifer Brana, Matt Butrovich, Kyle Booker, Arham Chopra, Val Choung, Kevin Gaffney, Chris Laspias, Wan Shen Lim, Yuchen Liang, Lin Ma, Prashanth Menon, Ben Owad, Ritu Pathak, Shubham Shastri, Patrick Wang, Sherry Wang, Chi Zhang, and Will Zhang.

I thank Tracy Farbacher, Karen Lindenfelser, and Joan Digney for efficiently running the 5th Year Master's program and PDL (respectively) behind-the-scenes. Last but not least, I would like to thank the CMU Database Group dogs, Terrier and Scout, for distracting me from my research and providing me with lots of joy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database management systems (DBMSs) are the backbone of modern applications. They must support various use cases, such as bioinformatics, artificial intelligence, geospatial, and web applications. One method in which DBMSs achieve this is by allowing users the ability to extend the capabilities of their system with custom software. We refer to this as *extensibility*, and the software that extends the DBMS as an *extension*. Some examples of extensions include adding a user-defined type, providing an additional password authentication protocol, or overriding a component of a database system, such as its planner, execution engine, or storage engine.

There are several benefits to supporting extensibility within a DBMS. First, supporting extensibility allows the DBMS to support additional use cases in a lightweight manner explicitly supported by the DBMS. Second, extensibility provides the ability to support multiple extra features simultaneously without adding them to the core system. Adding too many features to the core system can result in feature bloat, harming the system's performance. Extensions can help alleviate this problem by allowing developers to package important but niche features as extensions. Lastly, supporting extensibility can improve the database system itself. In some cases, features initially implemented as extensions were eventually supported as core features of the DBMS. One example of this phenomenon is the PostgreSQL garbage collection feature [144], which the committee merged as a feature instead of an extension in 2005. This occurrence shows that extensions are a powerful development mechanism that leads to more DBMS innovation. Overall, these benefits make extensibility helpful to support in a DBMS.

The database system industry recognizes the importance of extensibility. For example, many leading systems support extensibility, including PostgreSQL, DuckDB, Oracle, MySQL, SQLite, Microsoft SQL Server, Redis, and Elasticsearch [32, 38, 77, 80, 84, 142, 154, 171]. These systems have thriving ecosystems containing many kinds of extensions. PostgreSQL has the most vibrant extensibility ecosystem, with over a hundred commonly used extensions. Additionally, the industry has realized the importance of extensions to users. For instance, Google Cloud's and Amazon Web Service's PostgreSQL as-a-service offerings [2, 48] support almost one hundred extensions.

Despite the benefits of DBMS extensibility and industry-wide adoption, there are still problems with database extension ecosystems. First, faults in the general API design of extensibility support cause inefficient extension development. For example, when the extensibility API offers limited support, extension developers may not be capable of implementing the extensions

they want to implement. They may also have to restore to hacks or inefficient design protocols, such as copying many core PostgreSQL code or using a different extensibility mechanism for uses other than its intention to implement the extension they want. Additionally, a complicated or convoluted extensibility API can cause extension developers to not fully understand their code's effects on the DBMS. Second, depending on the system and API provided, extensions are prone to modify each other's executions, which can cause unexpected errors, such as causing the DBMS to crash, error out, or output incorrect results. This occurrence decreases the benefit of using extensions.

1.1 Thesis Contributions

This thesis provides a thorough analysis of the existing DBMS extension ecosystems. After discussing related work, we provide a complete definition of a DBMS that supports extensibility and a formal definition of an extension within that system. We provide an overview of the following:

1. The types of extensibility supported by DBMSs we surveyed
2. The design decisions we observed that each DBMS had to make about extensibility
3. The extensibility support mechanisms offered by the DBMSs we surveyed

Then, we provide a taxonomy of database system extensions, which provides an essential set of distinctions to enhance our understanding of DBMS extensibility. We also profile five widely-used DBMSs that support extensibility (PostgreSQL, DuckDB, SQLite, MySQL, Redis).

After reviewing our taxonomy, we introduce `pgext_cli`, our automated analysis framework for PostgreSQL extensions. This framework includes four main components: the compatibility analysis component, source code analysis component, function analysis component, and API information analysis component. We analyze the results obtained from running our framework on 97 PostgreSQL extensions. Then, we provide a comprehensive discussion on the current state of affairs in the database extension sphere, determining the industry's strengths and weaknesses concerning supporting extensibility. Lastly, we conclude our thesis by presenting our results and suggesting future work.

Chapter 2

Related Work

We now discuss existing research on database extensibility, database composability, operating system extensibility, and feature interactions.

2.1 Software System Extensibility

2.1.1 Database System Extensibility

To our knowledge, there exists only one survey and categorization of database extensibility [17]. The DBMSs mentioned in this paper include [9, 18, 86, 160, 176]. Carey and Haas categorize three different types of database extensibility in this survey. First, they define user-interface extensions, which include abstract data types (UDTs), UDAs, and relation-transformation extensions. Second, they define query processing extensions, which they define as extensions supporting new execution operators. The third type of extension they mention is a data storage extension, which involves creating extensible manners of storing data and index structures.

The ecosystem of database extensibility and extensions has changed a lot since 1990, so this categorization is incomplete. First, our survey contains DBMSs that did not exist in 1990. Additionally, the extension ecosystem of PostgreSQL has changed dramatically. It did not support hooks until 2006 [49]. Hooks are a PostgreSQL extensibility mechanism that allow users to override core PostgreSQL source code. They dramatically changed the types of extensions that developers could create. Lastly, this thesis does not evaluate current systems' versions of extensibility and classify the design decisions that one should consider when creating or modifying an extensible database system.

2.1.2 Operating System Extensibility

Understanding operating system extensibility is helpful because we can apply existing terminology, techniques, and support mechanisms to categorize and analyze database system extensibility.

The current version of MacOS supports extensibility by allowing users to implement user-space system extensions [179]. MacOS supports the implementation of networking extensions,

security extensions, and device drivers in user space via various frameworks, APIs, and libraries. These frameworks give this user code the privilege to run in kernel space. When finished developing these extensions, the developer packages their extension as an application and then use Apple's custom system extension tool to install it. Deleting the extension in user space is equivalent to uninstalling the extension. Since MacOS 11, Apple has disallowed running extensions that interact directly with the kernel unless the operating system is configured to a reduced security mode [72]. Before switching to user space extensions, MacOS kernel extensions were implemented as Mach-O shared object files, similar to many database system extensions, represented as shared object libraries.

The Linux operating system's extensibility is implemented via kernel modules. These small, dynamically loaded shared libraries can extend the kernel's functionality without requiring a complete recompilation or system reboot. Linux kernel modules implement device drivers, filesystem drivers, system calls, network drivers, and executable interpreters [185]. This implementation of extensibility support is similar to database extensibility support.

Another method of extensibility is the Berkeley Packet Filter [74], kernel extensions initially designed to filter network traffic. This project eventually morphed into the extended Berkeley Packet Filter (eBPF) project [36] project, which runs sandboxed, verified programs in the operating system. eBPF has a variety of use cases, including security extensions, network extensions, and performance monitoring. eBPF programs are written in a subset of C, then verified, JIT-compiled, and dynamically loaded into the kernel. They are called after events occur that trigger them. Possible events include network events and syscalls. eBPFs have also been used to implement DBMS features. For example, Tigger [16] is a DBMS proxy implemented as an eBPF extension.

The Spin Operating System [11] provides similar extensibility mechanisms to the current eBPF project. Unlike the eBPF project, which provides an extensibility framework for Linux, the Spin OS extensibility framework is built with a custom operating system. The researchers explicitly designed Spin OS to be extensible. It exports an API, giving applications fine-grained access to processors, memory, and I/O. It enables users to install extensions that happen in response to events, such as a hardware interrupt or context switch. In addition, the language chosen for Spin extensions (Modula-3) is type-safe. This choice ensures that the integrity of the kernel is not compromised when the extension is loaded into the kernel.

Like the Spin OS, the Nooks architecture [177] also provides higher reliability for device drivers, a significant component of operating system extensibility. As motivation for this project, the researchers noticed that most OS bugs were from third-party developer-written device drivers. A lack of deep kernel knowledge caused these bugs, extensively copying code between device drivers and poorly documented kernel features. Nooks ensures the operating system does not crash due to faulty device drivers. It ensures device drivers run in protected environments called "nooks". The Nooks architecture also prevents device drivers from writing to memory outside its protection domain and allows the OS to verify data that the device drivers have processed. The motivations for this research and the Nooks project are incredibly similar–in PostgreSQL, people develop extensions very similarly to kernel device drivers. Additionally, this paper explains an architecture technique for solving this problem, which we can apply to DBMS extensibility.

2.2 Software System Composability

2.2.1 Database System Composability

Recently, DBMS developers have argued that the databases community should prioritize database composability in DBMS design [27, 90, 138]. DBMS composability refers to designing the DBMS as a collection of well-defined components. Then, instead of building a DBMS when a new use case is needed, one would construct one from these components. This method streamlines development processes and improves innovation. The shared insight of these papers is that the DBMS stack is naturally composable and consists of a few well-defined parts, such as the parser, the query planner, the query optimizer, the execution engine, and the data storage layer.

The arguments for database extensibility are similar to composability because both of them advocate for reusable components of DBMSs that developers select according to specialized uses. The design tenet of extensibility argues that given an existing system with full functionality, we should allow users to override or extend a subset of its features. On the other hand, the design tenet of database composability argues that the DBMS should consist of many components that other parts can replace. Therefore, the composability movement argues that all major components of the DBMS should be extensions. It focuses on the process of implementing full DBMSs, with the existing literature discussing how to implement reusable execution engines or query optimizers [10, 89, 169]. However, the usage of extensibility varies from this use case to modifying a tiny part of an existing DBMS for more minimal purposes.

2.2.2 Operating System Composability

One pivotal example of operating system composability is the Exokernel project [39]. The main idea behind the Exokernel is that it gives untrusted users access to the resource management of the operating system. It does this by disaggregating protection mechanisms from resource management. Major benefits of using the Exokernel include a significant performance increase (since applications are aware of their resource consumption) and easy prototyping of new ideas since it is easier to understand which resources a new idea is using. The ideals behind the Exokernel are relevant to DBMS extensibility because both the Exokernel project and DBMS extensibility have motivations of giving untrusted users access to a portion of their system. This research also provides a compelling argument for allowing extensions to override resource management for the DBMS.

2.3 Feature Interactions

In the software engineering field, feature interactions research focuses on understanding how two different features in a software system modify each other's behavior. This research is relevant to problems in the database system extensibility sphere because when we can view extensions as additional features to the existing DBMS, then use the ideas, techniques, and results found from their research and apply them to database extensibility.

To help explore the feature interaction research space, Meinicke et al. developed VarexJ [75], a dynamic analysis tool based on variability-aware execution that generates program traces of software running at all possible configurations, allowing researchers to analyze the runtime state. They used this tool on several medium-sized Java programs, including Eclipse Jetty [37] and Checkstyle [19], to estimate the effort required to perform configuration-complete analysis on these programs. They showed it was feasible to conduct this analysis since the configuration complexity was low enough. VarXplorer [168] improves on VarexJ to support visualizations of feature-interaction graphs based on VarexJ's program traces.

Additionally, research has shown that variational traces improve the user development experience via a user study done with an Eclipse plugin, which supplied this functionality [76]. Lastly, researchers have investigated database systems concerning their feature interactions. In 2019, using static analysis, Kolesnikov et al. performed a case study [66] on SQLite's feature interactions. They found that there were 39 unique control flow interactions in SQLite, with each interaction involving four to six features.

The currently available tools are not usable since they only work on software written in Java. Tools similar to these, which also run static or dynamic analysis determining how extensions interact with each other, could make extension development easier. This research also shows that it is possible to examine database systems based on their features, as indicated by SQLite's performance feature analysis.

Chapter 3

Overview of Extensibility

We will start by discussing our definition of extensibility. We will then discuss the types of extensions we found via our survey. Then, we define the set of qualitative properties about database extensions. Lastly, we will discuss the types of extensibility supported by the DBMSs we surveyed.

We define *extensibility* as the capability of extending the capabilities of the database system with custom software and an *extension* as an instance of the software used to supplement a database system's features. An extension can combine several types of extensibility to add functionality or enhance performance. The DBMS must make extensibility a first-class feature. For example, one must not use any naming hacks, unintended behaviors, or security breaches to add an extension to the DBMS. There should be an intended and defined method of extending the capabilities of the DBMS. For instance, if a developer hacks the DBMS by code injection to run their code instead of the core DBMS code, we do not consider this to be extensibility, but a security breach. However, if the DBMS explicitly claims that a mechanism allowing developers to override function execution is allowed, then we consider this as extensibility support.

Furthermore, adding an extension should not rewrite the core DBMS source code; it should only add its source code or modify symbols to link to the DBMS executable. This is the main difference between an extension and a fork of a system. Although both extensions and forks of DBMSs change the capabilities of a DBMS, an extension will do so without modifying the original DBMS source code. In contrast, a fork of a system modifies the source code, making the fork and the original system different systems.

Additionally, it should be possible to load multiple extensions to a DBMS simultaneously using an intended mechanism, such as an exposed API or SQL. On the other hand, combining multiple forks of a system without extensive effort is impossible.

3.1 Types of Extensibility

We define ten different types of extensibility.

3.1.1 User-Defined Types (UDTs)

UDTs allow developers to implement a custom type that users can add to the DBMS. UDTs can be combinations or aliases of already supported types in a system or custom-implemented types. In the case of custom-implemented types, the user provides a set of functions, such as comparators, access methods, and I/O methods, for the custom type and then loads the library containing these functions into the system. Many well-known DBMSs, including PostgreSQL, DuckDB, SQL Server, and Oracle RDBMS, support UDTs. Their many applications include the geospatial, artificial intelligence, genomics, and game development domains. Examples of UDTs include vector embedding [129], LiDaR point [106], and standardized address [175] types.

3.1.2 User-Defined Functions (UDFs)

UDFs allow developers to implement a custom function that users can add to the DBMS. DBMSs typically support UDFs in higher-level languages, such as PL/SQL or Perl, and in the language of the DBMS source code. These functions take in data returned from SQL queries or sub-queries as input and are executed by the DBMS's executor engine when a query runs. A subtype of UDF is a user-defined aggregate (UDA). UDAs are UDFs that take in multiple rows of data as input and return a single result. A different subset of a UDF is a user-defined operator (UDO) [192]. UDOs are operators that users can define to perform custom operations on user-defined types. Examples of operators include comparison or arithmetic operators. Almost every well-known DBMS supports user-defined functions, including PostgreSQL, DuckDB, SQLite, MySQL, and Redis. UDFs are powerful and have a variety of use cases. Most extensions that we evaluated utilize UDFs to provide a helpful utility, configure and initialize functionality implemented via other kinds of extensibility, or define routines required by custom types and other database objects. There are many extensions that utilize UDFs. Below are a few examples:

- SQLite extension `ieee754` [54] provides functions for converting floating-point numbers to their IEEE754 Binary64 floating-point representation.
- PostgreSQL extension `citus_columnar` [22] defines a function `columnar.columnar_handler` that defines its table access method functions.
- PostgreSQL extension `pg_stat_kcache` [121] exposes a UDF called `pg_stat_kcache_reset()`, which resets the statistics collected by it.

3.1.3 External Tables

External table extensibility allows users to interact with external data sources as if the DBMS stored them directly. The databases community also refers to external table extensions as connectors. Developers can use external table extensibility to support reading and writing data stored as CSVs, on the cloud, or in a different file format. Examples of connectors include PostgreSQL's foreign data wrappers [45] and several DuckDB extensions [33, 34]. Notably, DuckDB implements this external table extensibility by overriding the catalog. On the other hand, most extensions extend the execution engine or storage manager components to implement external table extensibility. Many well-known DBMSs, including PostgreSQL, DuckDB, MySQL, and

SQLite, support external table extensibility.

3.1.4 Utility Command Extensibility

Utility command extensibility allows developers to override or introduce new processing utility commands. These commands process the updating of database objects, schemas, permissions, or configuration settings. Utility command extensions also include extensions that provide logical decoding support. Logical decoding support allows extensions to extract WAL data and display it in an easy-to-read format. Utility command extensions can be minimal, such as the PostgreSQL extension `dont_drop_db` [31], which determines a list of databases that the DBMS should not drop. However, extensions that override utility commands have the potential to be extremely powerful. For example, the PostgreSQL Trusted Languages Extension (`pg_tle`) framework [188] overrides the general PostgreSQL utility command handling to change execution if a user installs an extension through their framework.

3.1.5 Parser Extensibility

Parser extensibility allows developers to change or augment the parser. The parser of a DBMS is responsible for validating the syntax of the SQL query and interpreting it into internal plan tree structures used by the query execution layer. Developers modify the parser by modifying parser source code or via query rewrite rules, enabling users to change queries that pattern match appropriately. Examples of parser extensions include MySQL's Rewriter Query Rewrite Plugin [157], which keeps a table of rules and applies them onto SQL queries at the parsing stage.

Modifying the parser provides three benefits. First, it would allow developers to supplement the query language with syntax appropriate for their extension. For example, users could support different syntaxes to support new features in the query processing layer. Second, developers employing parser extensibility can change the validation of SQL queries. For example, an extension using parser extensibility could reject queries from users if it deemed them a security threat to the DBMS. Third, users can support different SQL dialects or special syntax for their extensions. For example, PostgreSQL does not support parser extensibility, which results in extensions using UDFs to express language syntax instead of introducing language constructs.

3.1.6 Query Processing Extensibility

Query processing extensibility allows developers to modify or extend the planner, optimizer, or execution engine logic. Extensions that employ query processing extensibility fall into two categories. First, we have query statistics collector extensions. These extensions collect query execution metrics, such as performance, query information, table access, and plan representations. Then, they store them for users to examine via UDFs or tables or output them to logging. Examples of query statistics collector extensions include `pg_stat_statements` [123] and `pg_qualstats` [109]. Second, extensions modify the query processing layer to support additional features. These can be minimal features, but they can also result in an overhaul of the whole query processing layer. For example, Citus [21] and Timescale [187] leverage significant parts of the query processing extensibility infrastructure to turn PostgreSQL into an entirely new database

system while implementing their software as an extension. In particular, PostgreSQL has a very non-restrictive environment for query processing extension support. PostgreSQL allows users to completely rewrite both its planner and executor engine. Therefore, many extensions using query processing extensibility found in our survey are PostgreSQL extensions.

3.1.7 Storage Engine Extensibility

Storage engine extensibility allows developers to modify or extend the storage engine logic of a database system. The storage engine of the DBMS is responsible for efficiently managing the storage of data, both in memory and on disk. Extensions employing storage engine extensibility can either override the filesystem layer or the storage engine logic. Examples of extensions which use storage engine extensibility include the S3 Storage Engine [159], which makes the storage engine stores its data in S3, and `unix` [87], which overrides the SQLite filesystem API to support running SQLite on a Unix OS. Extensions utilizing storage engine extensibility can be beneficial, allowing users to store their data using different storage layouts or retrieve it via a different protocol. They can also allow for compatibility of the DBMS across other operating systems, especially on DBMSs with a filesystem layer.

3.1.8 Client Authentication Extensibility

Client authentication extensibility allows developers to modify or add to the client authentication logic of a database system. The client authentication logic is responsible for identifying the client, determining the resources within the database they access, and determining the level of permissions granted to the client. Client authentication extensions fall into several categories. First, some extensions internally change how the DBMS handles user passwords. Usually, this involves making the user's password more protected, although the MySQL Client Cleartext extension [23] ensures that passwords get sent as plain text to the server. The second type of extension changes the password rules within the DBMS. For example, the PostgreSQL extension `passwordcheck` [88] allows developers to modify the DBMS with additional password rules. Lastly, client authentication extensions can change users' privilege levels on database objects. For example, `set_user` [164] is a PostgreSQL extension that allows switching users and privilege levels.

3.1.9 Table and Index Access Methods

Table and index access methods extend the DBMS and provide additional ways of accessing data via different storage methods. These differ from storage engine extensibility, which allows developers to override some of the core DBMS's storage manager components directly. They also differ from external table extensions, which store table data in auxiliary storage and allow the user to interact with this data as if the main table stored it. Table access methods directly store data where the core DBMS would store data, just in a different manner than the default storage manager. An example of a table access method extension is Citus Columnar [22], which provides table columnar storage with projection pushdown and compression. Index access methods provide an additional index implementation to the DBMS and store their data within the internal

DBMS, making them different from external table extensions. PostgreSQL has leveraged the existence of index access method extensions to implement many new auxiliary indexes, such as `lsm3` [69] (log-structured merge tree index) and `bloom` [12] (bloom filter index). Some index access methods like `GIN` and `GiST` [47] are extensible for new user-defined types.

We acknowledge that access methods can have similar use cases to the external table and storage manager extensions. For example, `Citus` [21] includes a columnar storage implementation using table access methods. Its implementation was directly derived from `cstore_fdw` [25], which used external tables. A table access method intends to continue using the storage manager, buffer cache, and write ahead log of the DBMS while also supporting secondary indexes. Conversely, external tables are typically stored and modifiable outside the DBMS.

3.1.10 Catalog Extensibility

Catalog extensibility allows developers to modify or access database metadata directly. Catalog extensions can be helpful when there is an internal database system state that developers want to display to users. One example of a catalog extension is the MariaDB plugin `userstat` [193], which creates tables in the catalog storing statistics on user activity, client connections, index usage, and table usage. These extensions can keep track of and allow users to access query cache data, session-level state, and user-defined variables. Notably, many other types of extensibility can indirectly modify database metadata. For example, user-defined type and customized operator metadata are stored in the DBMS catalog, and utility command extensions can modify DBMS metadata by influencing the execution of schema and object modification commands. Our survey found that only MySQL directly supports pure database metadata extensions. However, PostgreSQL allows the extension to write to their catalog database [178]. This method effectively achieves the same effect as MySQL's extensibility.

3.2 Database Extension Properties

Developers implement extensions to enhance DBMS usage or add new features. In this section, we define the qualitative categorizations to improve our understanding of what attributes differentiate database extensions from one another. We will discuss these categorizations in the subsections below. s

3.2.1 Types of Extensibility

Developers typically utilize multiple types of extensibility to create an extension. Although it is possible to create an extension with just one kind of extensibility, it is relatively rare. An example of an extension that would utilize one type of extensibility is an extension that provides helpful arithmetic UDFs. Even extensions that seem like they would only utilize one type of extensibility usually do not. For instance, an extension that introduces a custom type will also introduce custom comparison operators. It is also possible to simultaneously leverage many kinds of extensibility to create a cohesive extension. Citus [21], an extension for distributed PostgreSQL, leverages six different types of extensibility. Overall, categorizing an extension by

the types of extensibility an extension allows us to identify the bare bones of its structure. It also allows us to understand the scope of an extension's impact across the DBMS.

3.2.2 Extending vs. Overriding the DBMS

The second categorization is whether the extension *extends* or *overrides* the DBMS. Notably, extensions that override the DBMS modify the core execution of the DBMS, while extensions that extend the DBMS do not. For instance, an extension that adds statistics collectors to the planner and executor of a DBMS overrides the DBMS. In contrast, an extension that introduces a user-defined type and custom operators on that type extends the DBMS. This categorization is helpful because it allows us to categorize an extension by the impact of its code on the DBMS.

3.2.3 State Modification

The third categorization to consider is which state that an extension modifies. When invoked, extensions can either modify (1) *no state*, (2) *database state*, or (3) *system state*. Notably, an extension can also modify multiple states simultaneously. We define database state as the data stored in the database itself. For instance, table data, relation metadata, and indexes all count as database state. We define the system state as the DBMS's internal data structures. If an extension has the potential to modify these internal data structures, we say that it modifies the DBMS state. Extensions that have access to the DBMS state can also choose to either read the state or modify it; in our categorization, types of extensions that have access to the state are all grouped in the same category, but when reviewing existing extensions and their behavior, it is essential to note this. This categorization allows us to understand an extension's impact on a DBMS's state: the data within the databases and the metadata stored during DBMS runtime.

3.2.4 Isolation

The fourth categorization to consider is whether an extension is isolated from other extensions, which means determining if an extension has the potential to modify other extensions' execution or output. For instance, if two extensions receive the same modifiable input data structures, one after the other, we could claim that these extensions are not isolated since the latter extension can change or even revert the internal data structure changes of the former extension. This categorization allows us to categorize whether extensions can affect each other's execution.

We note that determining whether extensions are isolated from one another depends on the implementation of extensibility support in the DBMS. For instance, when extension developers gain the ability to write their extensions in C/C++, it is easy for extension developers to develop hacks or even malicious behavior to modify the output of other extensions deliberately. However, we will base all future categorizations on excluding malicious or deliberate behavior. Our definition focuses on the isolation of extensions when developers are trying to implement their functionality without deliberately messing up the execution of other extensions.

System Name	Number of Extensions
PostgreSQL	375+
DuckDB	30+
SQLite	61+
MySQL	47+
Redis	57+

Table 3.1: Number of extensions per DBMS

	PostgreSQL	DuckDB	SQLite	MySQL	Redis
Functions	Yes	Yes	Yes	Yes	Yes
Types	Yes	Yes	No	No	Yes
External Tables	Yes	Yes	Yes	Yes	Yes
Utility Commands	Yes	No	No	No	No
Parser	No	Yes	No	Yes	No
Query Processing	Yes	Yes	No	No	No
Storage Engine	No	Yes	Yes	Yes	No
Access Methods	Yes	No	No	No	No
Client Authentication	Yes	No	No	Yes	No
Catalog	Yes	Yes	No	Yes	No

Table 3.2: Overview of types of extensibility supported by each DBMS

3.3 DBMSs that Support Extensibility

We discuss several systems that support extensibility and provide an overview of the types of extensibility they support and how the systems implement them. Table 3.1 notes how many extensions we were able to observe for each DBMS. Table 3.2 contains a general overview of the types of extensions each DBMS supports.

In the following subsections, we will provide a small overview of what each DBMS supports and the key differences we found within each DBMS's extensibility support.

3.3.1 PostgreSQL

Out of all the DBMSs we surveyed, PostgreSQL has the most support for different types of extensibility. PostgreSQL supports seven of the nine kinds of extensibility we identified. Due to this comprehensive extensibility support, PostgreSQL also has a thriving ecosystem of extensions. Our survey found over 1000 PostgreSQL extensions, whereas other DBMSs had less than 100 per system. PostgreSQL even packages itself with 50 extensions approved by the PostgreSQL development committee.

In particular, PostgreSQL has comprehensive support for query processing extensions. For example, PostgreSQL extensions can completely rewrite the planner. However, PostgreSQL also allows extensions to rewrite smaller components of the planner code. It is also possible to insert custom scanning or joining protocols via the planner extensibility mechanisms in PostgreSQL.

These lower-level, more minor in-scope planner extensibility mechanisms have also proven useful to inject monitoring into the planner. Additionally, PostgreSQL has the most exhaustive executor extension support and allows extensions to overwrite the executor engine fully or partially if needed. As a result of this extensive support, there are many query-processing extensions in PostgreSQL of varying degrees of complexity. PostgreSQL is the only DBMS we surveyed that supports utility command extensions.

3.3.2 DuckDB

Similar to PostgreSQL and MySQL, DuckDB has comprehensive support for different kinds of extensibility. For example, DuckDB supports optimizer extensibility that allows developers to add customized optimizer passes. DuckDB supports vectorized UDFs, which means that each invocation of the UDF processes a large batch of values (i.e., a vector). This technique allows for faster execution, one of the main reasons users do not utilize UDFs in their applications. Additionally, DuckDB enables developers to override its catalog implementation with a custom one. In two DuckDB extensions, `postgres_scanner` [33] and `sqlite_scanner` [34], this feature implements support for integrating PostgreSQL and SQLite tables into a DuckDB instance.

3.3.3 SQLite

SQLite is the only DBMS we surveyed supporting filesystem extensibility. This feature enables developers to override SQLite's default filesystem under the hood. Users can use this extensibility to optimize their filesystem layer for different workloads, support portability between different operating systems, and implement security features such as encryption. SQLite restricts overriding more than one filesystem simultaneously. Apart from the filesystem extensibility support, extensibility support in SQLite is minimal and not invasive compared to PostgreSQL, MySQL, and DuckDB.

3.3.4 MySQL/MariaDB

Compared to other DBMSs, MySQL has a thriving storage engine extensibility ecosystem. It has a selection of 9 storage engine extensions, including InnoDB [61], the memory storage engine [186], which supports main-memory execution, and the federated storage engine [184], which allows users to access remote MySQL databases. MySQL even supports joining tables backed up by two different storage engines. Besides these storage engines, MySQL 8.0 has 44 extensions packaged with the core DBMS software. However, even though it supports many extensions in its core system and contains support for many types of extensibility, MySQL lacks the large open-source ecosystem that PostgreSQL has.

MariaDB's extensibility mechanisms are identical to MySQL. However, MariaDB supports more extensions than MySQL. For instance, MariaDB 11.0 includes support for 22 storage engines. These storage engines are more specialized than MySQL's storage engines. Some of their applications include text search [78], S3 backup [158], highly parallel workloads [73], and flash-storage [79]. MariaDB 11.0 has 65 extensions packaged with the core DBMS software. Mari-

aDB's third-party extensibility ecosystem is minimal despite the effort to include more built-in extensions.

3.3.5 Redis

Redis has the least support for different types of extensibility. Of nine kinds of extensibility, Redis only supports functions, types, and external tables. However, despite this restrictive support, Redis has an active, thriving extensibility environment. Redis' extensibility environment contains 57 extensions, and its most popular extension is RedisSearch [155], a query and indexing engine built on top of Redis. Redis supports two types of UDFs. First, it allows for developers to write functions in Lua, which perform operations on Redis tables and auxiliary data processing. Second, it allows developers to code modules, which are extensions with an interface of loadable C UDFs built on top of Redis.

Chapter 4

Design Decisions for DBMS Extensibility

We discuss the design decisions included in the process of supporting extensibility. These design decisions are the (1) programming languages provided to extensions developers, (2) methods provided by the DBMS to help extensions development, (3) integration of extensions into the core system, and (4) installation techniques. Figure 4.1 provides a general overview of these design decisions. We elaborate on them in the following sections.

4.1 Programming Languages

The first is choosing the languages in which developers can write extensions. The DBMS can allow developers to write extensions in the language of the DBMS source code or support extensions written in a different, usually higher-level programming language. The benefits of allowing developers to write extensions in the source code's language include the support of more involved extensions. For instance, writing an extension for overriding a significant component of the DBMS, such as the planner or storage engine, is significantly more straightforward to support if developers write extensions in the same language as the source code. Additionally, given that database systems are usually written in lower-level languages, such as C/C++, writing extensions in the source code language may also result in a performance benefit. However, coding an extension in the source code language requires a lot of knowledge of the source code language and the internals of the specific DBMS. On the other hand, the DBMS can also allow developers to write extensions in a higher-level language, such as Perl, Python, or JavaScript. One benefit of this approach is that it results in less faulty code. This method allows unprivileged users to create extensions without accessing the database's internals. Lastly, it is also possible for a DBMS to support writing extensions (particularly UDFs) in SQL. One benefit of supporting SQL is that the DBMS can easily inline the function body of a UDF into a calling query.

4.2 Methods

The second design decision is determining the methods provided by the DBMS for extension development. The DBMS can choose to expose a well-defined API, where functions in this API are the only functions usable by developers. On the other hand, the DBMS can allow developers

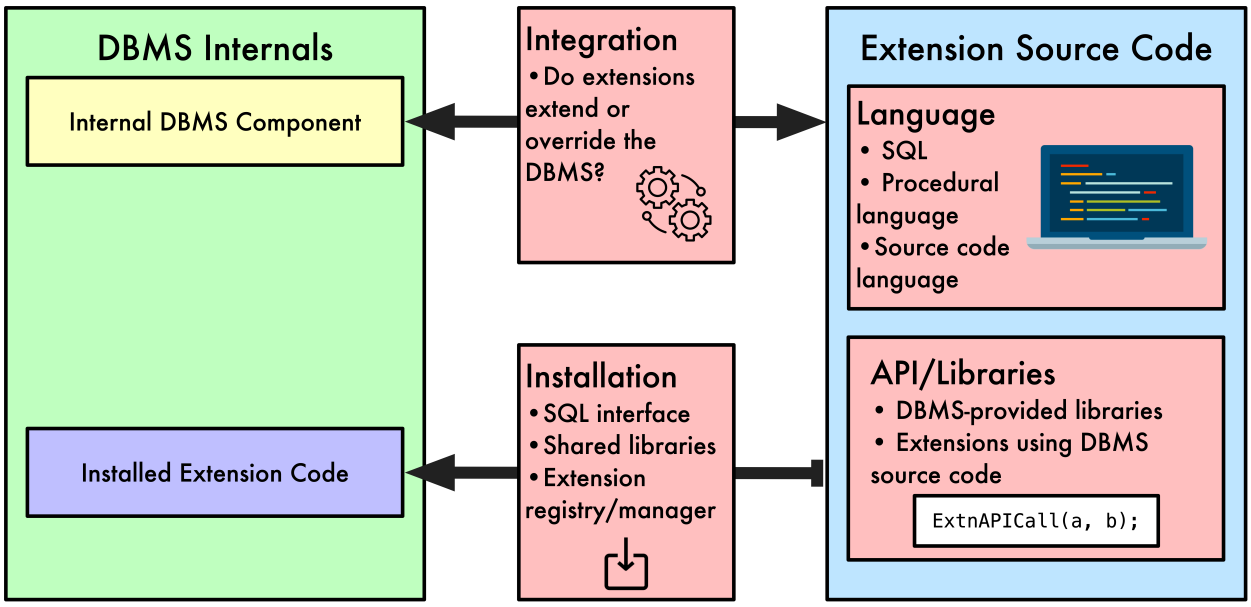

Figure 4.1: Overview of DBMS extensibility design decisions

to use any instance of the source code in their extension implementations, which is common when the source code is in C/C++. Providing a well-defined API limits the capabilities and scope of the extensions written, but it also prevents extensions from potentially breaking the DBMS's functionality. Allowing extension developers more freedom in using the source code as it suits them results in the opposite trade-offs.

4.3 Integration

The third major design decision is to consider how to integrate extensions within the DBMS. The DBMS can choose to do this in two manners. First, an extension can be an addition to a system, where the system execution is unchanged due to its addition unless the user is explicitly using the extension's functionality. Examples of this include adding a user-defined type or adding a user-defined function. Second, an extension can overwrite a specific system component's source code. For instance, a DBMS can provide hooks, which are function pointers at places in a DBMS's source code. An extension developer can then set the value of these hooks to point to the extension code. When the DBMS source code calls the hook, the developer has the guarantee that the DBMS will execute their code. Hooks can be a powerful extensibility method, as it allows extension developers to override significant components of a DBMS, such as the planner or storage engine.

4.4 Installation

The last design decision is to decide how extensions are installed and uninstalled within a system. A DBMS's most typical way to manage extensions is by providing an SQL interface. Users

can enter something similar to `LOAD EXTENSION_NAME` to enable an extension in a system and `UNLOAD EXTENSION_NAME` to ensure it is not enabled. These SQL statements have the prerequisite that the extension is already in DBMS-readable form (e.g., a shared or dynamic library). However, DBMSs will also sometimes provide an extension manager as a separate executable; instead of providing a SQL interface, one can pass in the extensions they would like to have integrated as command line arguments to this extension manager. A DBMS may also require a developer to register their extensions in their source code or edit a configuration file to enable an extension. The DBMS may utilize these methods to ensure an extension is fully integrated within a system. For instance, PostgreSQL extensions are required to provide a configuration file and a SQL script that runs when the extension is loaded and may optionally provide a shared library.

Chapter 5

Mechanisms of DBMS Extensibility

This section covers the mechanisms used to build DBMS extensions and provides an overview of the mechanisms each DBMS provides to support extensions developers.

5.1 Common Mechanisms Used to Build Extensions

In our survey, we also found five common mechanisms that DBMSs give to developers intended to simplify and enhance extension development.

5.1.1 Background Workers

Background workers are a supplied mechanism that allows extensions to spawn separate processes or threads that execute custom code. The DBMS actively manages background workers, which means they can be stopped, monitored, and restarted as needed. They have access to both DBMS memory and internal state. As a result, they can be leveraged in powerful extensions and can perform tasks like garbage collection (`pg_autovacuum` [93]), index structure background jobs (e.g., `lsm3` [69]), and periodic job scheduling (`pg_cron` [97]). Our survey found that background workers are supported by both PostgreSQL and MySQL.

5.1.2 Memory Allocation

Memory allocation mechanisms provide developers with a dynamic means of declaring memory for their extensions. The DBMS actively manages this memory and provides an accessible API for extensions to interact with it. Extensions use this auxiliary memory for a variety of purposes, including the support of auxiliary data structures for metadata storage (`RedisGraph` [156]), tables for storing statistics (`pg_stat_statements` [123]), or the storage for an in-memory column store (`imcs` [55]). Our survey found that both Redis and PostgreSQL support dynamic memory allocation for their extensions.

	PostgreSQL	DuckDB	SQLite	MySQL	Redis
Background Workers	Yes	No	No	Yes	No
Memory Allocation	Yes	No	No	No	Yes
Custom Configuration Options	Yes	No	No	Yes	Yes
Source Code	Yes	Yes	Yes	Yes	No

Table 5.1: Extensibility support mechanisms supported by DBMSs

5.1.3 Custom Configuration Options

Custom configuration options and user-defined variables empower extension developers to provide users with tailored control over their extensions. Developers achieve this by enabling users to pass variables or flags to the DBMS during startup or execution, allowing the extension to adapt its functionality based on specified settings. When extension developers want to utilize these customized options, they declare variables explicitly in the source code and then utilize their values in the code logic. This powerful customization feature can be used in a variety of smaller extensions, such as password validators and text search dictionaries. It can also be utilized in more involved extensions, such as `timescaledb` [187]. `timescaledb`, an extension allows users to treat PostgreSQL as a time series DBMS, offer users the flexibility to customize the behavior of their extensions through specified variables.

5.1.4 Source Code

Developers also use the source code of the DBMS to build extensions. Most of this is an artifact of the fact that the DBMSs we surveyed are implemented in C/C++, which easily allows developers to import source code headers. This feature is compelling, as users can now access and use the whole DBMS implementation as they please. Many developers import or directly copy DBMS types, functions, and classes and use them in their extensions. Although recognizing which functionality to use from the DBMS source code requires a deep understanding of the DBMS, using the source code effectively can reduce developer efforts.

5.2 Mechanisms Given by Database Systems that Support Extensibility

In this section, we discuss several well-known systems that support extensibility and provide an overview of the mechanisms they give developers to help them create extensions. First, Table 5.1 provides a general overview of the mechanisms each DBMS supports.

In the following subsections, we will provide a small overview of the mechanisms provided by each DBMS and the key differences we found within each DBMS's extensibility mechanism support.

5.2.1 PostgreSQL

By far, PostgreSQL provides the most extensive extensibility mechanism support. PostgreSQL provides a specialized background worker implementation designed to work with its process per-worker model and shared memory infrastructure. PostgreSQL also allows extensions to take advantage of their memory allocation interface by providing two hooks that extensions can override to request and initialize a fixed amount of shared memory. One of PostgreSQL's unique features is its customized configuration support. PostgreSQL allows users to edit configuration variables via SQL or by directly editing the configuration file to load the DBMS with these customized values. Allowing users to edit these files directly is unique to PostgreSQL.

Additionally, PostgreSQL provides essential building and testing infrastructure to developers. PostgreSQL's tooling helps users compile and test their extensions. PostgreSQL provides two tools: PGXS [133] and `pg_regress` [113]. With PGXS, users can write a simple Makefile to test and develop extensions against an installed PostgreSQL server. `pg_regress` enables users to conduct black box testing on their extensions to ensure robustness and compatibility with the PostgreSQL environment. Overall, these internal PostgreSQL tools significantly improve the efficiency of PostgreSQL extension development. Although there are a few other examples of PostgreSQL extensibility infrastructure tools, such as `pg_tle` [188] (support extension creation without superuser permissions)and `pgrx` [117] (a Rust framework for PostgreSQL extensions development). However, third-party developers have created these extensions, and they are not features of PostgreSQL's extensibility.

5.2.2 DuckDB

DuckDB's extensibility mechanism support is relatively limited, and it only allows users to import source code into its extensions. They also technically have a `malloc` wrapper that extensions could use. Still, we did not count this as memory allocation support since they do not explicitly provide this API to extension developers.

5.2.3 SQLite

SQLite's extensibility mechanism support is also relatively limited. SQLite only allows users to access the core DBMS source code and utilize it for extension development.

5.2.4 MySQL

MySQL provides a decently comprehensive set of extensibility mechanisms to developers. For instance, MySQL provides daemon extensibility, which allows developers to run logic adjacent to the DBMS within the DBMS. This extensibility form is similar to background workers' purpose, although it is supported differently within the DBMS. Additionally, MySQL allows extensions to define custom variables, which users can then set via SQL statements.

5.2.5 Redis

Redis provides a dynamic memory allocation API that allows extensions to allocate and free memory as they please. In addition to this memory allocation API, Redis provides an extensive utility API that allows extension developers to call Redis's internal commands from their extension code. Besides this utility API, Redis does not offer its core source code for extension developers. Instead, extension developers choose to build extensions using the key-value store as needed instead of directly modifying Redis.

Chapter 6

PostgreSQL Extensibility Analysis Framework

This section provides an overview of the analysis framework we developed to collect data on PostgreSQL's extensibility ecosystem. We evaluated PostgreSQL's extensibility ecosystem because it is the most prolific by far (3.1). PostgreSQL has many open-source, third-party-created extensions compared to the other DBMSs we surveyed. We evaluate PostgreSQL's extensibility ecosystem on three main facets: compatibility, source code quality, and extension API usage. Then, we determine whether there are any correlations between the source code quality or extension API usage and the compatibility results.

We developed `pgext_cli` to assist with our evaluation. To our knowledge, `pgext_cli` is the first automated analysis framework for PostgreSQL extensions. `pgext_cli` is implemented in Python. We run all of our extensibility analysis using extensions compatible with PostgreSQL 15.3 [143]. The framework supports 97 PostgreSQL extensions. We primarily collected the extensions used in our analysis from three main sources:

- PostgreSQL's `contrib` directory [24], which contains extension modules supported directly by the PostgreSQL core developers (once again, using version 15.3)
- Supported PostgreSQL extensions from AWS Relational Database Service [2], Google Cloud SQL [48], and Azure Database for PostgreSQL [6]
- Other popular and widely used extensions, over 2,000 Github Stars (e.g., Citus [21], TimescaleDB [187])

We wanted to ensure that we were running our analysis on widely used and reasonably popular extensions. All the extensions also had to be open-source to ensure that we could conduct our source code analysis. The framework includes four main components, which we describe below.

6.1 Extension API Information Analysis

`pgext_cli` includes a mode where we can extract the types of extensibility used within a PostgreSQL extension. As mentioned previously, we note that PostgreSQL supports seven types of extensibility: functions, types, access methods, external tables, client authentication, query

processing, and utility commands. pgext_cli's API information analysis tool can list the types of extensibility and the hooks used by each of the 97 PostgreSQL extensions.

pgext_cli extracts the extension repository code, then looks for keywords within the source code and SQL files. For instance, the analysis framework will check if the extension initializer function sets the hook global variables or if the source SQL code declares user-defined functions or types.

6.2 Extension Source Code Analysis

pgext_cli includes a mode that conducts source code analysis on the extensions. First, the source code analysis mode collects two metrics: the total lines of source code in the extension and the total lines of directly copied PostgreSQL source code. We used a static analysis tool called the PMD Copy/Paste Detector (CPD) [43] tool to determine the number of lines of directly copied PostgreSQL source code. The PMD CPD tool finds duplicate code blocks in codebases. In our analysis, we only kept track of code blocks with a minimum length of 100 tokens (where a token is the smallest unit of a programming language with meaning). Our tool also ensures that these copied code blocks are from the PostgreSQL 15.3 core DBMS codebase.

Second, the source code analysis mode analyzes the presence of versioning logic in the extension source code. Versioning logic is when the extension source code explicitly cases on the version of PostgreSQL in their source code. For instance, the developers may intend to call a different function when loading their extension onto a PostgreSQL 11 database instance versus a PostgreSQL 12 database instance. Extension developers write code that cases on the PostgreSQL version via C-style macros. We tracked whether the extension source code used version casing and, if so, how many lines of code were encapsulated between these versioning case statements.

6.3 Extension Compatibility Analysis

pgext_cli includes a mode that tests the compatibility of PostgreSQL extensions with each other. Given a set of extensions, the compatibility analysis framework will test extensions pairwise (where order matters) for basic compatibility. More concretely, the framework takes pairs of extensions, downloads and installs them onto the same PostgreSQL 15.3 instance, starts the database, and runs each extension's unit tests on this instance. Then, with both of these extensions installed, it runs pgbench [94], primarily as a basic check. If all of these checks pass, then the extensions pass this compatibility test and are compatible. At the end of its execution, the framework outputs the results of this compatibility testing.

6.4 Function Analysis

pgext_cli includes a mode that conducts static analysis on the extensions that entirely copied functions from the PostgreSQL source code into their codebases. The framework keeps track of the total number of function code lines for each of these extensions, then performs static analysis on the functions to determine if they are read-only or modified. It also keeps track of the number

of copied functions and the number of functions that modify state. We used Semgrep [162], a source code static analysis tool, to determine whether a function modifies DBMS state.

Chapter 7

Results

In this section, we present the main results that we collected through the `pgext_cli` framework. Additional results are located in the Appendix.

7.1 API Information Analysis

The API information analysis extracts the types of extensibility used within a PostgreSQL extension. For each hook, it also indicates whether the hook is utilized in our extension.

On average, PostgreSQL extensions utilized 1.59 types of extensibility per extension, although the maximum types of extensibility utilized in an extension was 6. Citus [21], which provides support for distributed PostgreSQL, utilized every type of extensibility in its implementation except for external tables. Figure 7.1 is a bar chart which shows how many extensions utilized how many API components in their implementations. We can see that most extensions only utilized about 1 to 2 different types of extensibility in their implementations.

For each type of extensibility that PostgreSQL supports, we also show the number of extensions which utilize this type of extensibility in Table 7.1. Here, we can see that user-defined functions are the most popular type of extensibility, with 83 extensions of 97 using it. Our analysis framework revealed that 33 of 97 extensions that we surveyed used hooks. Out of the hooks, the six most popular hooks were `shmem_startup_hook` [167], `shmem_request_hook` [166], `post_parse_analyze_hook` [150], `ExecutorStart_hook` [41], `ExecutorEnd_hook` [40],

Extensibility Type	Percentage of Extensions Which Utilized It
Functions	86%
Types	25%
Access Methods	4.1%
External Tables	5.1%
Client Authentication	7.2%
Query Processing	20%
Utility Command	12%

Table 7.1: Percentage of extensions using each type of extensibility

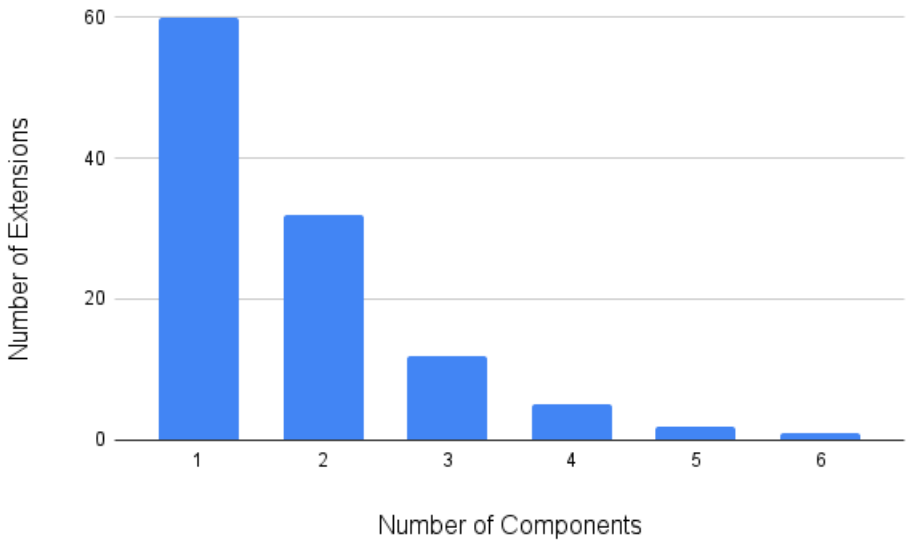

Figure 7.1: Number of components extensions utilized in their implementations

and `ProcessUtility_hook` [152]. The `shmem` hooks allow extensions developers to request and intiialize dynamic memory for storing extension metadata. The `post_parse_analyze_hook` allows the DBMS to secondarily parse a query. The `ExecutorStart_hook` and `ExecutorEnd_hook` allow the user to read or modify DBMS state before and after query execution, and the `ProcessUtility_hook` allows for users to override utility commands (e.g. `CREATE DATABASE`) and other query processing functionality.

7.2 Source Code Analysis

The source code analysis determines whether the extension copies core PostgreSQL source code and whether the extension uses versioning logic in its source code.

We determined that 41 of 97 extensions copied PostgreSQL source code in their codebases. For each of these extensions, we collected the percentage of of an extension's codebase that consisted of copied PostgreSQL DBMS code. The average percentage was 10.8%, and the maximum percentage was 75.5% percent. Figure 7.2 provides a histogram of these percentages.

We also determined that 38 of 97 extensions used versioning logic in their codebases. For each of these extensions, we collected the percentage of an extension's source code that was encapsulated between versioning logic. The average percentage was 7.0% percent, and the maximum percentage was 23.9% percent. Figure 7.3 provides a histogram of these percentages.

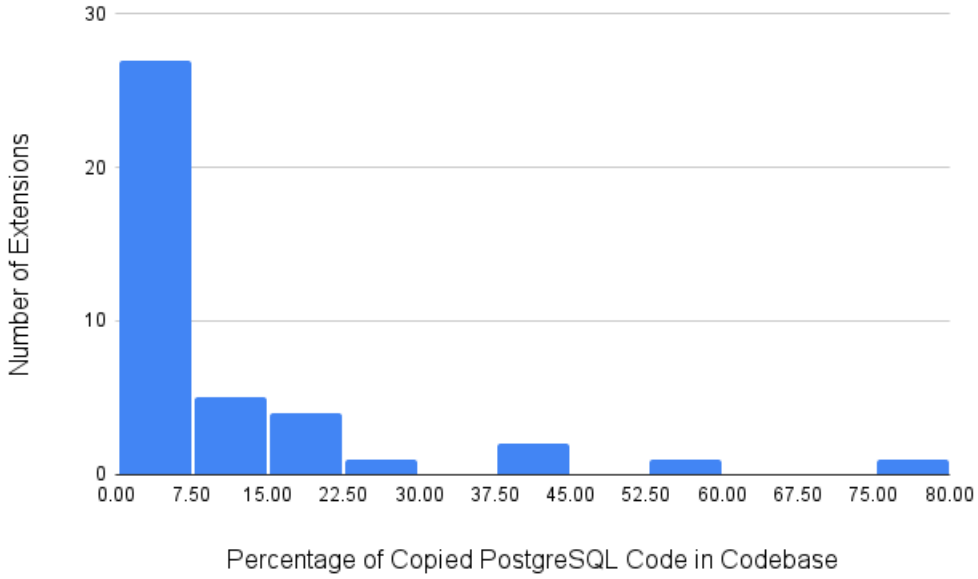

Figure 7.2: Percentage of copied PostgreSQL code in extensions

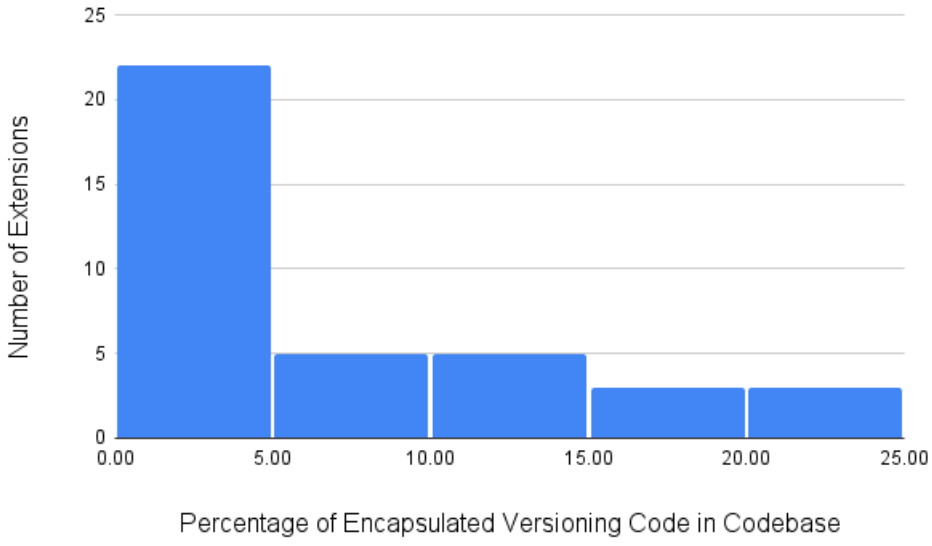

Figure 7.3: Percentage of encapsulated versioning code in extensions

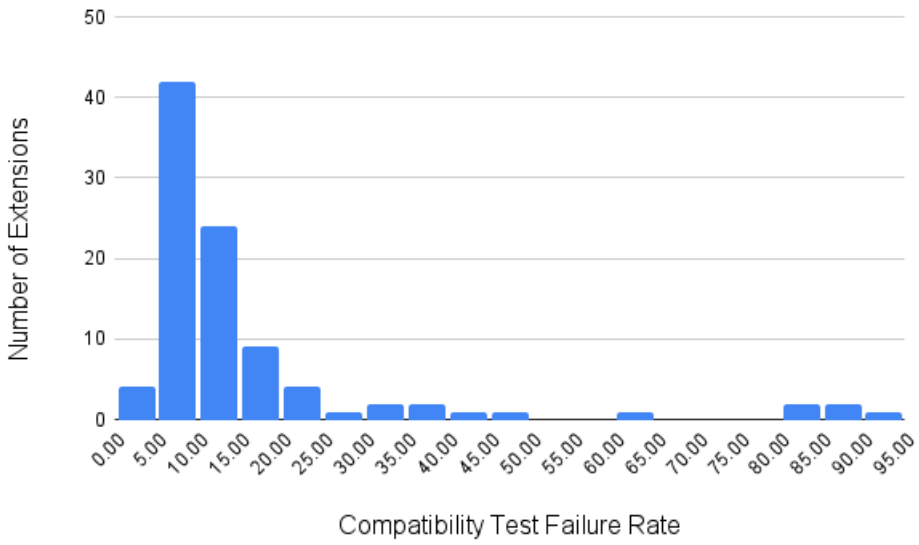

Figure 7.4: Compatibility test failure percentages

7.3 Compatibility Analysis

We ran compatibility testing on 97 different extensions. We kept track of the percentage of compatibility tests that failed for each of these extensions. The average compatibility test failure rate was 16.9%. The highest test failure rate was 91.6% (`pgextwtlist` [146]), and the lowest was 2.6% (`basebackup_to_shell` [7]). Figure 7.4 provides a histogram showing these test failure percentages.

7.4 Correlation Analysis

We wanted to see whether there was a significant difference in the compatibility of extensions that utilized certain types of extensibility, copied PostgreSQL source code, or used versioning logic. Therefore, we ran paired t-tests on the compatibility percentages. We split the percentages into two groups, based on whether the extensions:

- Used a certain hook in their source code
- Utilized a certain type of extensibility
- Copied code from the core PostgreSQL source
- Utilized versioning logic
- Had more than n lines of source code in their codebases ($n = 500, 750, 1000$)
- Whether an extension utilized strictly more than n types of extensibility (for $n = 1, 2, 3, 4$)

We found that there were four main factors with a p-value lower than 0.1, which means with at least a probability of 0.9, the average of the compatibility percentages of the two groups is significantly different. Therefore, we can claim that extensions with these factors are significantly

Extension Name	Copied Function LOC	Num. Functions	Num. State Modifying Functions
citus [21]	6281	119	31
cube [26]	776	34	2
orafce [85]	1031	38	3
pg_hint_plan [101]	1426	19	6
pg_ivm [102]	6653	113	29
pg_queryid [110]	34	1	1
pg_repack [114]	87	2	1
pg_stat_monitor [122]	64	2	2
pg_strom [91]	134	3	1
pg_tle [188]	873	15	1
pglogical [104]	267	5	3
seg [161]	771	34	2
timescaledb [187]	884	16	4

Table 7.2: Function analysis results

less compatible with other extensions than extensions without these factors. These factors are:

- Utilization of versioning logic ($p = 0.01$)
- Whether an extension utilized strictly more than n types of extensibility (for $n = 1, 2, 3, 4$, $p_1 = 0.08, p_2 = 0.07, p_3 = 0.07, p_4 = 0.08$)
- Had more than n lines of source code in their codebases ($n = 500, 750$, $p_{500} = 0.01, p_{750} = 0.09$)
- Usage of `ProcessUtility_hook` ($p = 0.08$)

7.5 Function Analysis

The function analysis mode runs static analysis on the fully copied functions from PostgreSQL source code. We conducted this analysis on 10 of 97 extensions, since these were the only extensions which copied whole functions from source code. We present our results in Table 7.2.

Chapter 8

Discussion

Ultimately, our survey and evaluation yielded many fruitful observations about the current database system extensibility ecosystem. We summarize these findings below in the following sections.

8.1 Extensibility Design Trade-offs

Our comprehensive survey identified two significant design trade-offs concerning extensibility within DBMSs. The first trade-off is the safety-flexibility trade-off. Within extensibility, safety guarantees include a guarantee that the DBMS will not crash or that an extension does not modify another extension's execution. SQLite demonstrates an example of this trade-off. Its filesystem extensibility support guarantees users can install only one filesystem extension simultaneously.

On the other hand, another example of a system opting for more flexibility over safety is when PostgreSQL allows an extension to disallow previously loaded extension's hooks from executing. For instance, Citus completely overrides the planner hook and ensures that no other previously loaded extension has access to the planner hook. Our survey found that PostgreSQL had the most flexible extensibility support, and Redis, a system where extension developers cannot modify source code, had the safest version of extensibility. To illustrate this trade-off, we provide Figure 8.1, which shows the five DBMSs we surveyed on the safety-flexibility spectrum.

The second trade-off we identified is the usability-conciseness trade-off. To support more concise or niche forms of extensibility, the DBMS will make their extensibility less accessible to developers without a deep understanding of the DBMS they are working on. For instance, some forms of planner extensibility within PostgreSQL are concise and niche (e.g., adding a new join path to the plan tree). This extensibility requires a deep knowledge of PostgreSQL's internal plan representation. However, in PostgreSQL, the executor hooks are designed with more usability in mind. There are four executor hooks: run at the beginning of an execution of a query plan, during execution (this hook overrides the executor's runtime completely), after the execution engine runs a query plan, and after the execution engine finishes running the last query plan. These hooks are more intuitive for DBMS developers without a deep understanding of PostgreSQL. However, this means it is impossible to make small changes to the executor engine code without rewriting these coarser hooks.

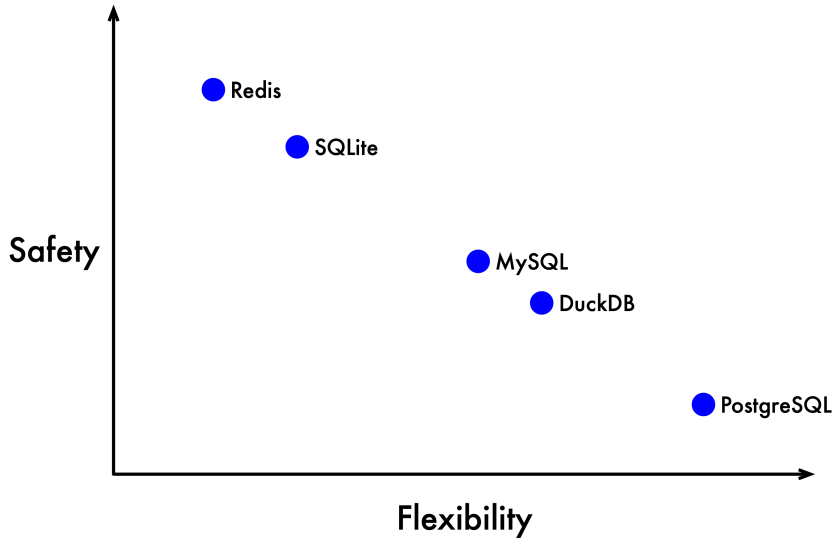

Figure 8.1: Safety vs. flexibility trade-off visualization

8.2 Extension Composability

Our survey also showed that DBMS extensibility was not inherently composable. There was no built-in way to reference another extension's code, components, or state from an extension. Rather, extension developers have to rely on deeply understanding the behavior of an extension and then code up their extension to match this behavior. However, this did not stop extension developers from setting other extensions as dependencies and utilizing the behavior of other extensions in their implementations. For instance, many PostgreSQL extensions [101, 110, 118, 121] rely on pg_stat_statements [123], a built-in PostgreSQL extension that tracks planning and execution statistics. These extensions usually interpret these statistics differently or add auxiliary statistics collection. Another example of this phenomenon is the collection of InnoDB-specific catalog extensions in MariaDB. InnoDB [57, 61] is a widely used storage engine plugin for MySQL and MariaDB. Both of these DBMSs include extensions that collect statistics related to InnoDB and store them in the catalog [56, 58]. These extensions store statistics about the transactions executing inside InnoDB, locks that InnoDB transactions have requested, and information about compressed tables. These examples provide a compelling argument to formally support database extension composability within commonly used DBMSs.

8.3 DBMS Extension Ecosystem Similarity

Analyzing extensions via our survey revealed that many extensions implemented for different DBMSs performed remarkably similar functions. We provide the following four examples of extensions with similar purposes.

1. Both MySQL and PostgreSQL support a DBMS job queuing extension. PostgreSQL's version of the job queue extension is `pg_cron` [97], while MySQL's version is `mysql_query_queue` [82]. Both extensions support user interaction via UDFs.

2. Both DuckDB and PostgreSQL support a SQLite external table extension [34, 172], which allows users to interact with a SQLite database in DuckDB/PostgreSQL. Most DBMSs supported external table extensions that processed other widely used DBMS file formats.
3. Both SQLite and PostgreSQL support vector embedding extensions [129, 173], which allow the user to perform vector similarity searches on their data. They implement these extensions slightly differently. PostgreSQL gives users a custom type and a custom HNSW index implementation. Since SQLite does not support UDTs, it provides a custom index implementation based on Faiss and allows users to insert their data as JSON or raw bytes into these tables.

A few high-impact, motivating examples exist where extension developers implement similar extensions in different systems. Ideally, there would be methods of writing extensions where developers would not have to write multiple extensions performing the same capabilities.

8.4 PostgreSQL's Popularity

Table 3.1 highlights that PostgreSQL's extensibility ecosystem is much more popular, well-used, and active than the other four DBMSs' extensibility ecosystems. We observe that a few factors contribute to PostgreSQL's immense popularity.

1. PostgreSQL provides significant flexibility to its developers. It is possible to almost completely override the core system and create a new DBMS by only using PostgreSQL's extensibility offerings. Unlike other DBMSs we surveyed, PostgreSQL has comprehensive support for query processing extensions. This flexibility allows developers to implement many kinds of extensions that may not be inherently doable in other DBMSs using PostgreSQL.
2. PostgreSQL provides many extensibility mechanisms to developers. In fact, 5.1 notes that PostgreSQL offers all the common mechanisms. PostgreSQL supports more comprehensive versions of these mechanisms than other database systems. For instance, their configuration option support allows for extensions to declare custom configuration variables that users can set in `postgresql.conf`, the main configuration file.
3. Compared to other extensions, PostgreSQL provides significant internal building and testing infrastructure via `pgxs`, their internal build system tool for extensions, and `pg_regress`, their internal black box testing tool for extensions. Most PostgreSQL developers utilize these tools when writing extensions.

8.5 PostgreSQL Analysis Discussion

Despite the PostgreSQL extensibility ecosystem's inherent dominance over every other extensibility ecosystem (in terms of its activity), our analysis framework highlighted some critical problems with it, mainly regarding extension compatibility. In the first subsection, we discuss the issues we found by manually analyzing the results of our compatibility analysis. In the second subsection, we highlight the results of our correlation analysis and explain them in context.

8.5.1 Qualitative Discussion

When manually inspecting the log files and output of the extension compatibility testing, we found several issues with PostgreSQL's extensibility ecosystem:

1. PostgreSQL does not have an extension manager. It relies on extensions to call the previous extension's installed hook if the hook exists. This means that extensions can prevent other extensions from executing their code. For example, Citus's [21] implementation uses this tactic. Citus overrides the planner and prevents previous extensions from running their planner code. Apart from this concern, an extension manager could fix ordering issues, where one ordering of extensions results in compatibility and another does not. It also allows for a more effortless experience of installing and uninstalling extensions. Lastly, some extensions require PostgreSQL with special compilation flags, and having an extension manager could ensure that this extension is only successfully loaded if PostgreSQL is installed with the correct configurations.
2. Although `pg_regress` is incredibly convenient and relatively easy for extension developers, its implementation has some problems. First, its only supported mode is black box testing. As a result, most extensions do not support unit testing and only have black box tests. Second, `pg_regress` fails tests even when it is not supposed to since it tests for correctness by comparing the text output of the SQL queries in the test file to an `.out` file, a text file with the correct output of these SQL queries. As a result, even if two extensions output the correct text for a query because the `.out` file does not take another extension's output into account, the test fails.
3. PostgreSQL should inform users via error messages if their extension fails due to incompatibility. Often, when extensions are incompatible, the outputted error messages are seemingly unrelated to the incompatibility. For instance, when `shared_ispell` and `plprofiler` are unsuccessfully loaded together, the error message in the log is "requested tranche is not registered", which alludes to an error in the shared memory hooks, but does not highlight the extension incompatibility as the problem.
4. PostgreSQL should determine when extension incompatibility is inherently unfixable. For instance, Citus and Timescale completely override the planner and change significant portions of the query plan's internal representation to cater to their purposes. However, users can install Citus and Timescale, even if their tests fail upon execution.

8.5.2 Correlation Discussion

The correlation analysis highlighted four key extension properties, resulting in a significantly higher compatibility test failure rate. We explain why these factors led to a higher failure rate below.

1. Extensions that utilize versioning logic in their source code have a significantly higher failure rate than extensions that do not because versioning logic introduces a significant amount of complexity that is hard to navigate. The versioning problem becomes even more tricky when considering PostgreSQL's dramatically changing core codebase. We discovered that most extensions case on recent versions of PostgreSQL, as opposed to older

ones. One of the reasons that versioning logic is tricky to handle is because extensions use some unspecified states of the core PostgreSQL code to implement their extension's logic. When those states are changed, their extension no longer functions correctly.

2. Extensions with more lines of code (over 500 and 750 LOC) and extensions that utilized more components have a significantly higher failure rate than those that did not. This phenomenon makes sense, as this increased failure rate is due to increased extension complexity. Shorter and more simple code is easier to understand and code correctly. Additionally, extension developers who code extensions that utilize more types of extensibility have to know how these components work with one another, which gets more difficult as they utilize more types of extensibility.
3. Extensions that use the `ProcessUtility_hook` have a significantly higher failure rate than extensions that do not use this hook. The `ProcessUtility_hook` allows users to override utility commands, or essentially every database command except SELECT, INSERT, UPDATE, and DELETE. Its calling location is also immediately after parsing, making it convenient for extension developers to insert extra logic before executing queries. The fact that the `ProcessUtility_hook` is extremely powerful, along with its prime location where extensions perform extra logic, seems to be why using this hook results in a significantly higher failure rate.

Overall, we see a few solutions to the PostgreSQL extensibility problems. First, PostgreSQL should reduce complexity in writing extensions. For instance, they can make extensions development agnostic to the version of PostgreSQL the extension runs on. However, the trade-off of this approach is that extensions are more restricted in their actions. Extension developers may be unable to utilize hacks based on PostgreSQL's internal state. They can also provide utility APIs based on the types of functions extensions tend to copy and easier ways of porting helpful functions besides copy-pasting them directly into the extension source code. Lastly, the `ProcessUtility_hook`'s great power, along with the power of other sledgehammer hooks, such as `planner_hook`, which can overwrite the whole planner, should not go unchecked. PostgreSQL should provide a utility tool that determines the extension's compatibility with the core DBMS and other extensions.

Chapter 9

Conclusion and Future Work

9.1 Future Work

There are several directions of future work that we could pursue due to this research endeavor. First, we could design methods of identifying whether PostgreSQL extensions affect each other's execution. We could start by viewing extensions and other main components of the core DBMS as features, then bring in the ideals of feature interaction research from the software engineering community as a starting point. Another exciting idea is to develop a better extensibility infrastructure that guarantees safe extensibility. For example, we could draw on ideals from the eBPF project, which uses sandboxing and verifying techniques to guarantee a semblance of safe extensibility. Lastly, we could develop extensibility that is portable between different database systems. One of our survey findings was that similar extensions existed across multiple DBMSs. Developing portable extensibility would increase the usage of extensibility in systems that are not just PostgreSQL and the impact of extensibility in the database systems sphere.

9.2 Conclusion

This thesis provides a comprehensive survey and evaluation of modern DBMS extensibility. We provide a taxonomy identifying ten types of DBMS extensibility, four common mechanisms provided to DBMS extension developers, four DBMS design decisions made about extensibility, and four main properties of DBMS extensions. We also offer a comprehensive analysis of PostgreSQL's extensibility ecosystem. To help with this analysis, we provide the first framework that runs evaluative analysis on PostgreSQL extensions. Through the results provided by our framework, we highlight that it sacrifices safety for high flexibility. This chosen trade-off is one of several reasons why it is the most popular environment for extension development. Overall, we hope this work can highlight existing problems of DBMS extensibility and motivate future research on improving DBMS extensibility.

Appendix A

Additional Results

In this chapter, we place useful data collected with `pgext_cli`.

A.1 API Information Analysis

Table A.1 shows the types of extensibility utilized for each extension.

Extension Name	Fns.	Types	Access Methods	External Tables	Client Auth	Query Processing	Utility Commands
adminpack [1]	Yes	No	No	No	No	No	No
amcheck [3]	Yes	No	No	No	No	No	No
anon [145]	Yes	Yes	No	No	No	No	No
auth_delay [4]	No	No	No	No	Yes	No	No
auto_explain [5]	No	No	No	No	No	Yes	No
basebackup_to_shell [7]	No	No	No	No	No	No	No
basic_archive [8]	No	No	No	No	No	No	No
bloom [12]	Yes	No	Yes	No	No	No	No
bool_plperl [13]	Yes	No	No	No	No	No	No
btree_gin [14]	Yes	No	No	No	No	No	No
btree_gist [15]	Yes	Yes	No	No	No	No	No
citext [20]	Yes	Yes	No	No	No	No	No
citus [21]	Yes	Yes	Yes	No	Yes	Yes	Yes
cube [26]	Yes	Yes	No	No	No	No	No
dblink [28]	Yes	Yes	No	Yes	No	No	No
decoderbufs [140]	No	No	No	No	No	No	No
dict_int [29]	Yes	No	No	No	No	No	No
dict_xsyn [30]	Yes	No	No	No	No	No	No
dont_drop_db [31]	No	No	No	No	No	No	Yes
earthdistance [35]	Yes	No	No	No	No	No	No
file_fdw [42]	Yes	No	No	Yes	No	No	No

fuzzystrmatch [46]	Yes	No	No	No	No	No	No
hll [148]	Yes	Yes	No	No	No	Yes	No
hstore [50]	Yes	Yes	No	No	No	No	No
hstore_plperl [51]	Yes	No	No	No	No	No	No
hstore_plpython3u [52]	Yes	No	No	No	No	No	No
hypopg [53]	Yes	No	No	No	No	Yes	Yes
imcs [55]	Yes	Yes	No	No	No	Yes	No
intagg [59]	Yes	No	No	No	No	No	No
intarray [60]	Yes	Yes	No	No	No	No	No
ip4r [62]	Yes	Yes	No	No	No	No	No
isn [63]	Yes	Yes	No	No	No	No	No
jsonb_plperl [64]	Yes	No	No	No	No	No	No
jsonb_plpython3u [65]	Yes	No	No	No	No	No	No
lo [67]	Yes	No	No	No	No	No	No
logerrors [68]	Yes	No	No	No	No	No	No
lsm3 [69]	Yes	No	Yes	No	No	Yes	Yes
ltree [70]	Yes	Yes	No	No	No	No	No
ltree_plpython3u [71]	Yes	No	No	No	No	No	No
mysql_fdw [81]	Yes	No	No	Yes	No	No	No
old_snapshot [83]	Yes	No	No	No	No	No	No
oracle_fdw [44]	Yes	No	No	Yes	No	No	No
orafce [84]	Yes	Yes	No	No	No	No	No
pageinspect [88]	Yes	No	No	No	No	No	No
passwordcheck [88]	No	No	No	No	Yes	No	No
pg_bigm [95]	Yes	No	No	No	No	No	No
pg_buffercache [96]	Yes	No	No	No	No	No	No
pg_cron [97]	Yes	No	No	No	No	No	No
pg_freespacemap [100]	Yes	No	No	No	No	No	No
pg_hint_plan [101]	No	No	No	No	No	Yes	No
pg_ivm [102]	Yes	No	No	No	Yes	No	No
pg_log_userqueries [105]	No	No	No	No	No	Yes	Yes
pg_partman [149]	Yes	Yes	No	No	No	No	No
pg_prewarm [107]	Yes	No	No	No	No	No	No
pg_proctab [108]	Yes	No	No	No	No	No	No
pg_qualstats [109]	Yes	Yes	No	No	No	Yes	No
pg_query_rewrite [112]	Yes	No	No	No	No	Yes	No
pg_queryid [110]	Yes	No	No	No	No	No	No
pg_querylog [111]	Yes	Yes	No	No	No	Yes	No
pg_repack [114]	Yes	Yes	No	No	No	No	No
pg_show_plans [119]	Yes	No	No	No	No	No	No
pg_similarity [120]	Yes	No	No	No	No	No	No

pg_stat_kcache [121]	Yes	No	No	No	No	Yes	No
pg_stat_monitor [122]	Yes	No	No	No	Yes	Yes	Yes
pg_stat_statements [123]	Yes	No	No	No	No	Yes	Yes
pg_strom [91]	Yes	Yes	No	Yes	Yes	Yes	No
pg_surgery [125]	Yes	No	No	No	No	No	No
pg_tle [188]	Yes	Yes	No	No	Yes	No	Yes
pg_trgm [127]	Yes	Yes	No	No	No	No	No
pg_variables [128]	Yes	No	No	No	No	Yes	No
pg_visibility [130]	Yes	No	No	No	No	No	No
pg_wait_sampling [131]	Yes	No	No	No	No	Yes	No
pg_walinspect [132]	Yes	No	No	No	No	No	No
pgaudit [92]	Yes	No	No	No	Yes	Yes	Yes
pgcrypto [98]	Yes	No	No	No	No	No	No
pgextwlist [146]	No	No	No	No	No	No	Yes
pgfincore [99]	Yes	No	No	No	No	No	No
pgjwt [103]	Yes	No	No	No	No	No	No
pglogical [104]	Yes	No	No	No	Yes	No	Yes
pgrouting [115]	Yes	Yes	No	No	No	No	No
pgrowlocks [116]	Yes	No	No	No	No	No	No
pgsentinel [118]	Yes	No	No	No	No	No	No
pgstattuple [124]	Yes	No	No	No	No	No	No
pgtap [126]	Yes	No	No	No	No	No	No
pgtt [147]	No	No	No	No	No	Yes	Yes
plpgsql_check [134]	Yes	No	No	No	No	No	No
plprofiler [135]	Yes	No	No	No	No	No	No
plproxy [136]	Yes	No	No	Yes	No	No	No
plv8 [137]	Yes	No	No	No	No	No	No
postgis [139]	Yes	Yes	No	No	No	Yes	No
postgres_fdw [141]	Yes	No	No	Yes	No	No	No
prefix [151]	Yes	Yes	No	No	No	No	No
rdkit [153]	Yes	Yes	No	No	No	No	No
seg [161]	Yes	Yes	No	No	No	No	No
sepgsql [163]	Yes	No	No	No	Yes	No	Yes
set_user [164]	Yes	No	No	No	Yes	No	Yes
shared_ispell [165]	Yes	No	No	No	No	No	No
spi [170]	Yes	No	No	No	No	No	No
sslinfo [174]	Yes	No	No	No	No	No	No
tablefunc [180]	Yes	Yes	No	No	No	No	No
tcn [181]	Yes	No	No	No	No	No	No
tds_fdw [182]	No	No	No	No	No	No	No
test_decoding [183]	No	No	No	No	No	No	No

timescaledb [187]	Yes	Yes	No	Yes	No	Yes	Yes
tsm_system_rows [189]	Yes	No	No	No	No	No	No
tsm_system_time [190]	Yes	No	No	No	No	No	No
unaccent [191]	Yes	No	No	No	No	No	No
uuid-ossp [194]	Yes	No	No	No	No	No	No
vector [129]	Yes	Yes	Yes	No	No	No	No
vops [195]	Yes	Yes	No	Yes	No	Yes	No
wal2json [196]	No	No	No	No	No	No	No
xml2 [197]	Yes	No	No	No	No	No	No

Table A.1: Types of extensibility utilized by each extension

A.2 Source Code Analysis

Table A.2 shows the results of our duplicated PostgreSQL code analysis. Table A.3 shows the results of our versioning code analysis. Notably, we do not include the results of extensions that did not have copied PostgreSQL code or versioning logic in their respective tables.

Extension Name	**Total LOC**	**Copied Postgres LOC**	**Percent Copied Postgres LOC**
bloom	1737	60	3.45
citus	236981	21418	9.04
cube	5683	2496	43.92
hstore	4471	57	1.27
hypopg	3576	76	2.13
orafce	22688	3242	14.29
pg_cron	4691	18	0.38
pg_hint_plan	6990	1532	21.92
pg_ivm	21260	15974	75.14
pg_log_userqueries	1479	52	3.52
pg_queryid	1065	36	3.38
pg_repack	5212	87	1.67
pg_stat_monitor	5127	589	11.49
pg_strom	68081	376	0.55
pg_tle	5281	2278	43.14
pg_trgm	5156	123	2.39
pgcrypto	11668	36	0.31
pglogical	26192	1243	4.75
plpgsql_check	19196	155	0.81
plv8	30819	59	0.19
seg	4705	2492	52.96
timescaledb	73501	2475	3.37

Table A.2: Source code analysis data

Extension Name	Total LOC	Versioning LOC	Percent Versioning LOC
anon	329	8	2.43
citus	236981	28650	12.09
decoderbufs	1520	7	0.46
dont_drop_db	195	24	12.31
hll	4214	332	7.88
hypopg	3576	853	23.85
imcs	15324	63	0.41
ip4r	8395	24	0.29
logerrors	776	20	2.58
lsm3	1301	40	3.07
mysql_fdw	9418	1034	10.98
oracle_fdw	9031	12	0.13
orafce	22688	1090	4.8
pg_bigm	1265	87	6.88
pg_cron	4691	86	1.83
pg_ivm	21260	316	1.49
pg_log_userqueries	1479	325	21.97
pg_partman	530	59	11.13
pg_proctab	1287	20	1.55
pg_qualstats	2616	253	9.67
pg_query_rewrite	897	90	10.03
pg_queryid	1065	2	0.19
pg_querylog	546	2	0.37
pg_repack	5212	37	0.71
pg_show_plans	630	25	3.97
pg_similarity	5066	95	1.88
pg_stat_kcache	1384	255	18.42
pg_stat_monitor	5127	1208	23.56
pg_strom	68081	30	0.04
pg_tle	5281	163	3.09
pg_variables	3904	169	4.33
pg_wait_sampling	1598	136	8.51
pgextwlist	1235	4	0.32
pgfincore	1168	9	0.77
pglogical	26192	692	2.64
pgsentinel	2265	391	17.26
pgtt	2031	247	12.16

plpgsql_check	19196	2955	15.39
plprofiler	2341	30	1.28
plproxy	5332	28	0.53
plv8	30819	1012	3.28
postgis	245474	11	0
prefix	1879	2	0.11
rdkit	447861	55	0.01
set_user	1163	161	13.84
shared_ispell	1100	29	2.64
tds_fdw	7087	524	7.39
timescaledb	73501	224	0.3
vector	7915	331	4.18
vops	8651	347	4.01
wal2json	3064	568	18.54

Table A.3: Versioning analysis data

A.3 Compatibility Analysis

Table A.4 provides the results of our compatibility data analysis. The Percent Not Compatible column is the number of compatibility pairing tests that failed over the total number of tests ran.

Extension Name	**Percent Not Compatible**
adminpack	5.73
amcheck	9.9
auth_delay	2.6
auto_explain	3.13
basebackup_to_shell	2.6
basic_archive	5.73
bloom	8.85
bool_plperl	10.94
btree_gin	14.06
btree_gist	16.15
citext	13.02
citus	30.73
cube	15.1
dblink	8.85
dict_int	8.85
dict_xsyn	9.38
dont_drop_db	5.21
earthdistance	82.29

file_fdw	12.5
fuzzystrmatch	8.85
hll	10.42
hstore	15.1
hstore_plperl	16.15
hstore_plpython3u	13.02
hypopg	9.38
imcs	12.5
intagg	5.73
intarray	13.02
ip4r	11.46
isn	21.35
jsonb_plperl	10.94
jsonb_plpython3u	10.42
lo	8.85
logerrors	10.42
lsm3	21.35
ltree	14.58
ltree_plpython3u	13.02
old_snapshot	6.77
orafce	7.29
pageinspect	9.9
passwordcheck	7.29
pg_bigm	21.35
pg_buffercache	6.77
pg_cron	39.06
pg_freespacemap	9.38
pg_hint_plan	21.35
pg_ivm	13.02
pg_log_userqueries	3.13
pg_partman	7.81
pg_prewarm	6.77
pg_proctab	6.77
pg_qualstats	12.5
pg_query_rewrite	46.35
pg_queryid	88.54
pg_repack	16.15
pg_show_plans	26.56
pg_stat_kcache	40.63
pg_stat_monitor	83.85
pg_stat_statements	19.27

pg_surgery	9.9
pg_tle	33.85
pg_trgm	14.58
pg_variables	9.38
pg_visibility	9.38
pg_wait_sampling	9.9
pg_walinspect	10.42
pgaudit	16.67
pgcrypto	9.38
pgextwlist	91.67
pgfincore	8.33
pgjwt	6.25
pgrowlocks	6.77
pgsentinel	37.5
pgstattuple	9.38
pgtap	6.77
pgtt	10.94
plprofiler	13.54
plv8	5.21
postgres_fdw	14.06
prefix	16.15
seg	16.67
shared_ispell	10.94
sslinfo	5.21
tablefunc	9.38
tcn	6.77
tds_fdw	64.58
test_decoding	8.33
timescaledb	87.5
tsm_system_rows	9.9
tsm_system_time	9.38
unaccent	9.9
uuid-ossp	7.29
vector	9.38
vops	10.42
wal2json	6.77
xml2	10

Table A.4: Compatibility data

Bibliography

[1] adminpack. https://www.postgresql.org/docs/current/adminpack.html, 2024. A.1

[2] Amazon RDS for PostgreSQL. https://aws.amazon.com/rds/postgresql/, 2024. 1, 6

[3] amcheck. https://www.postgresql.org/docs/current/amcheck.html, 2024. A.1

[4] auth_delay. https://www.postgresql.org/docs/current/auth-delay.html, 2024. A.1

[5] auto_explain. https://www.postgresql.org/docs/current/auto-explain.html, 2024. A.1

[6] Azure Database for PostgreSQL. https://azure.microsoft.com/en-us/products/postgresql, 2024. 6

[7] basebackup_to_shell. https://www.postgresql.org/docs/current/basebackup-to-shell.html, 2024. 7.3, A.1

[8] basic_archive. https://www.postgresql.org/docs/current/basic-archive.html, 2024. A.1

[9] D.S. Batoory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise. Genesis: an extensible database management system. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988. doi: 10.1109/32.9057. 2.1.1

[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230, 2018. 2.2.1

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995. ISSN 0163-5980. doi: 10.1145/224057.224077. URL https://doi.org/10.1145/224057.224077. 2.1.2

[12] bloom — bloom filter index access method. https://www.postgresql.org/docs/current/bloom.html, 2024. 3.1.9, A.1

[13] bool_plperl. https://pgpedia.info/b/bool_plperl.html, 2024. A.1

[14] btree_gin. https://www.postgresql.org/docs/current/btree-gin.html, 2024. A.1

[15] btree_gist. https://www.postgresql.org/docs/current/btree-gist.html, 2024. A.1

[16] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A database proxy that bounces with user-bypass. *Proc. VLDB Endow.*, 16(11):3335–3348, jul 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611530. URL https://doi.org/10.14778/3611479.3611530. 2.1.2

[17] Michael Carey and Laura Haas. Extensible database management systems. *SIGMOD Rec.*, 19(4):54–60, dec 1990. ISSN 0163-5808. doi: 10.1145/122058.122064. URL https://doi.org/10.1145/122058.122064. 2.1.1

[18] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M. Muralikrlshna. *The Architecture of the EXODUS Extensible DBMS*, pages 231–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. ISBN 978-3-642-84374-7. doi: 10.1007/978-3-642-84374-7_15. URL https://doi.org/10.1007/978-3-642-84374-7_15. 2.1.1

[19] Checkstyle. https://checkstyle.sourceforge.io/, 2024. 2.3

[20] citext. https://www.postgresql.org/docs/current/citext.html, 2024. A.1

[21] Citus. https://github.com/citusdata/citus, 2024. 3.1.6, 3.1.9, 3.2.1, 6, 7.1, 7.5, 1, A.1

[22] Citus Columnar. https://github.com/citusdata/citus/tree/main/src/backend/columnar, 2024. 3.1.2, 3.1.9

[23] Client-Side Cleartext Pluggable Authentication. https://dev.mysql.com/doc/refman/8.2/en/cleartext-pluggable-authentication.html, 2024. 3.1.8

[24] Contrib module. https://pgpedia.info/c/contrib-module.html, 2023. 6

[25] cstore_fdw. https://github.com/citusdata/cstore_fdw, 2024. 3.1.9

[26] cube. https://www.postgresql.org/docs/current/cube.html, 2024. 7.5, A.1

[27] Voltron Data. "the composable codex". https://voltrondata.com/codex, 2023. 2.2.1

[28] dblink. https://www.postgresql.org/docs/current/cube.html, 2024. A.1

[29] dict_int. https://www.postgresql.org/docs/current/dict-int.html, 2024. A.1

[30] dict_xsyn. https://www.postgresql.org/docs/current/dict-xsyn.html, 2024. A.1

[31] dont_drop_db. https://github.com/s-hironobu/dont_drop_db, 2024. 3.1.4, A.1

[32] DuckDB. https://duckdb.org/, 2024. 1

[33] DuckDB Postgres extension. https://github.com/duckdb/postgres_scanner, 2024. 3.1.3, 3.3.2

[34] DuckDB SQLite extension. https://github.com/duckdb/sqlite_scanner, 2024. 3.1.3, 3.3.2, 2

[35] earthdistance. https://www.postgresql.org/docs/current/earthdistance.html, 2024. A.1

[36] eBPF Documentation. https://ebpf.io/what-is-ebpf/, 2023. 2.1.2

[37] Eclipse Jetty. https://eclipse.dev/jetty/, 2024. 2.3

[38] ElasticSearch. https://www.elastic.co/elasticsearch, 2024. 1

[39] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917154. doi: 10.1145/224056.224076. URL https://doi.org/10.1145/224056.224076. 2.2.2

[40] ExecutorEnd_hook. https://pgpedia.info/e/ExecutorEnd_hook.html, 2024. 7.1

[41] ExecutorStart_hook. https://pgpedia.info/e/ExecutorStart_hook.html, 2024. 7.1

[42] file_fdw. https://www.postgresql.org/docs/current/file-fdw.html, 2024. A.1

[43] Finding duplicated code with CPD. https://pmd.github.io/pmd/pmd_userdocs_cpd, 2024. 6.2

[44] Foreign Data Wrapper for Oracle. https://github.com/laurenz/oracle_fdw, 2024. A.1

[45] Foreign Data Wrappers. https://wiki.postgresql.org/wiki/Foreign_data_wrappers, 2024. 3.1.3

[46] fuzzystrmatch. https://www.postgresql.org/docs/current/fuzzystrmatch.html, 2024. A.1

[47] GiST and GIN Index Types. https://www.postgresql.org/docs/9.1/textsearch-indexes.html, 2024. 3.1.9

[48] Google Cloud SQL. https://cloud.google.com/sql/postgresql, 2024. 1, 6

[49] The PostgreSQL Global Development Group. "postgresql 8.2.23 documentation". https://www.postgresql.org/docs/8.2/index.html, 2006. 2.1.1

[50] hstore. https://www.postgresql.org/docs/current/hstore.html, 2024. A.1

[51] hstore_plperl. https://pgpedia.info/h/hstore_plperl.html, 2024. A.1

[52] hstore_plpython3u. https://pgpedia.info/h/hstore_plpython.html, 2024. A.1

[53] HypoPG. https://github.com/HypoPG/hypopg, 2024. A.1

[54] ieee754. https://sqlite.org/src/file?name=ext/misc/ieee754.c&ci=trunk, 2024. 3.1.2

[55] imcs. https://github.com/knizhnik/imcs, 2024. 5.1.2, A.1

[56] Information Schema InnoDB Tables. https://mariadb.com/kb/en/information-schema-innodb-tables/, 2024. 8.2

[57] InnoDB. https://mariadb.com/kb/en/innodb/, 2024. 8.2

[58] InnoDB INFORMATION_SCHEMA Tables. https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema.html, 2024. 8.2

[59] intagg. https://www.postgresql.org/docs/current/intagg.html, 2024. A.1

[60] intarray. https://www.postgresql.org/docs/current/intarray.html, 2024. A.1

[61] Introduction to InnoDB. https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html, 2024. 3.3.4, 8.2

[62] IP4R. https://github.com/RhodiumToad/ip4r, 2024. A.1

[63] isn. https://www.postgresql.org/docs/current/isn.html, 2024. A.1

[64] jsonb_plperl. https://github.com/postgres/postgres/tree/master/contrib/jsonb_plperl, 2024. A.1

[65] json_plpython3u. https://github.com/postgres/postgres/tree/master/contrib/jsonb_plpython, 2024. A.1

[66] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. On the relation of external and internal feature interactions: A case study. Technical Report 1712.07440, arXiv, 12 2017. URL https://arxiv.org/abs/1712.07440. 2.3

[67] lo. https://www.postgresql.org/docs/current/lo.html, 2024. A.1

[68] logerrors. https://github.com/munakoiso/logerrors, 2024. A.1

[69] lsm3. https://github.com/postgrespro/lsm3, 2024. 3.1.9, 5.1.1, A.1

[70] ltree. https://www.postgresql.org/docs/current/ltree.html, 2024. A.1

[71] ltree_plpython. https://github.com/postgres/postgres/tree/master/contrib/ltree_plpython, 2024. A.1

[72] MacOS 11. https://support.apple.com/guide/deployment/

system-and-kernel-extensions-in-macos-depa5fb8376f/web, 2024. 2.1.2

[73] MariaDB ColumnStore (Analytics). https://mariadb.com/docs/columnstore/, 2024. 3.3.4

[74] Steven McCanne and Van Jacobson. 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*, USENIX'93, page 2, USA, 1993. USENIX Association. 2.1.2

[75] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 483–494, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970322. URL https://doi.org/10.1145/2970276.2970322. 2.3

[76] Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. Understanding differences among executions with variational traces. Technical Report 1807.03837, arXiv, 7 2018. URL https://arxiv.org/pdf/1807.03837.pdf. 2.3

[77] Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server, 2024. 1

[78] Mroonga. https://mariadb.com/kb/en/mroonga/, 2024. 3.3.4

[79] MyRocks. https://mariadb.com/kb/en/myrocks/, 2024. 3.3.4

[80] MySQL. http://www.mysql.com, 2024. 1

[81] MySQL Foreign Data Wrapper for PostgreSQL. https://github.com/EnterpriseDB/mysql_fdw, 2024. A.1

[82] MySQL Query Job Queue. https://github.com/adrpar/mysql_query_queue, 2024. 1

[83] old_snapshot. https://www.postgresql.org/docs/current/oldsnapshot.html, 2024. A.1

[84] Oracle. http://www.oracle.com, 2024. 1, A.1

[85] Orafce - Oracle's compatibility functions and packages. https://github.com/orafce/orafce, 2024. 7.5

[86] Sylvia L. Osborn and T. E. Heaven. The design of a relational database system with abstract data types for domains. *ACM Trans. Database Syst.*, 11(3):357–373, aug 1986. ISSN 0362-5915. doi: 10.1145/6314.6461. URL https://doi.org/10.1145/6314.6461. 2.1.1

[87] os_unix. https://sqlite.org/src/file?name=src/os_unix.c&ci=trunk, 2024. 3.1.7

[88] passwordcheck. https://www.postgresql.org/docs/current/passwordcheck.html, 2024. 3.1.8, A.1

[89] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment*, 15(12):3372–3384, 2022. 2.2.1

[90] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. The composable data management system manifesto. *Proceedings of the VLDB Endowment*, 16(10):2679–2685, 2023. 2.2.1

[91] PG-Strom. https://github.com/heterodb/pg-strom, 2024. 7.5, A.1

[92] pgAudit Open Source PostgreSQL Audit Logging. https://github.com/pgaudit/pgaudit, 2024. A.1

[93] pg_autovacuum. https://www.postgresql.org/docs/8.3/catalog-pg-autovacuum.html, 2013. 5.1.1

[94] pgbench. https://www.postgresql.org/docs/current/pgbench.html, 2024. 6.3

[95] pg_bigm. https://github.com/pgbigm/pg_bigm, 2024. A.1

[96] pg_buffercache. https://www.postgresql.org/docs/current/pgbuffercache.html, 2024. A.1

[97] pg_cron. https://github.com/citusdata/pg_cron, 2024. 5.1.1, 1, A.1

[98] pgcrypto. https://www.postgresql.org/docs/current/pgcrypto.html, 2024. A.1

[99] PgFincore. https://github.com/klando/pgfincore, 2024. A.1

[100] pg_freespacemap. https://www.postgresql.org/docs/current/pgfreespacemap.html, 2024. A.1

[101] pg_hint_plan. https://github.com/ossc-db/pg_hint_plan, 2024. 7.5, 8.2, A.1

[102] pg_ivm. https://github.com/sraoss/pg_ivm, 2024. 7.5, A.1

[103] pgjwt. https://github.com/michelp/pgjwt, 2024. A.1

[104] pglogical 2. https://github.com/2ndQuadrant/pglogical, 2024. 7.5, A.1

[105] pg_log_userqueries. https://github.com/gleu/pg_log_userqueries, 2024. A.1

[106] pgpointcloud. https://github.com/pgpointcloud/pointcloud, 2024. 3.1.1

[107] pg_prewarm. https://www.postgresql.org/docs/current/pgprewarm.html, 2024. A.1

[108] pg_proctab. https://github.com/markwkm/pg_proctab, 2024. A.1

[109] pg_qualstats. https://github.com/powa-team/pg_qualstats, 2024. 3.1.6, A.1

[110] pg_queryid. https://github.com/rjuju/pg_queryid, 2024. 7.5, 8.2, A.1

[111] pg_querylog. https://github.com/adjust/pg_querylog, 2024. A.1

[112] pg_query_rewrite. https://github.com/pierreforstmann/pg_query_rewrite, 2024. A.1

[113] pg_regress. https://github.com/postgres/postgres/blob/master/src/test/regress/pg_regress.c, 2024. 5.2.1

[114] pg_repack. https://github.com/reorg/pg_repack, 2024. 7.5, A.1

[115] pgRouting. https://github.com/pgRouting/pgrouting, 2024. A.1

[116] pgrowlocks. https://www.postgresql.org/docs/current/pgrowlocks.html, 2024. A.1

[117] pgrx. https://github.com/pgcentralfoundation/pgrx, 2024. 5.2.1

[118] pgsentinel. https://github.com/pgsentinel/pgsentinel, 2024. 8.2, A.1

[119] pg_show_plan. https://github.com/cybertec-postgresql/pg_show_plans, 2024. A.1

[120] pg_similarity. https://github.com/eulerto/pg_similarity, 2024. A.1

[121] pgstatkcache. https://github.com/powa-team/pg_stat_kcache, 2024. 3.1.2, 8.2, A.1

[122] pg_stat_monitor. https://github.com/percona/pg_stat_monitor, 2024. 7.5, A.1

[123] pg_stat_statements. https://www.postgresql.org/docs/current/pgstatstatements.html, 2024. 3.1.6, 5.1.2, 8.2, A.1

[124] pgstattuple. https://www.postgresql.org/docs/current/pgstattuple.html, 2024. A.1

[125] pg_surgery. https://www.postgresql.org/docs/current/pgsurgery.html, 2024. A.1

[126] pgTAP. https://github.com/theory/pgtap, 2024. A.1

[127] pg_trgm. https://www.postgresql.org/docs/current/pgtrgm.html, 2024. A.1

[128] pg_variables. https://github.com/postgrespro/pg_variables, 2024. A.1

[129] pgvector. https://github.com/pgvector/pgvector, 2024. 3.1.1, 3, A.1

[130] pg_visibility. https://www.postgresql.org/docs/current/pgvisibility.html, 2024. A.1

[131] pg_wait_sampling. https://github.com/postgrespro/pg_wait_sampling, 2024. A.1

[132] pg_walinspect. https://www.postgresql.org/docs/current/pgwalinspect.html, 2024. A.1

[133] pgxs. https://www.postgresql.org/docs/current/extend-pgxs.html, 2024. 5.2.1

[134] plpgsql_check. https://github.com/okbob/plpgsql_check, 2024. A.1

[135] plProfiler. https://github.com/bigsql/plprofiler, 2024. A.1

[136] plproxy. https://github.com/plproxy/plproxy, 2024. A.1

[137] PLV8. https://github.com/plv8/plv8, 2024. A.1

[138] Danica Porobic. Revisiting risc-style data management system design. In *CIDR*, 2019. 2.2.1

[139] PostGIS. https://postgis.net/, 2024. A.1

[140] Postgres Decoderbufs. https://github.com/debezium/postgres-decoderbufs, 2024. A.1

[141] postgres_fdw. https://www.postgresql.org/docs/current/postgres-fdw.html, 2024. A.1

[142] PostgreSQL. http://www.postgresql.org, 2024. 1

[143] PostgreSQL 15.3. https://www.postgresql.org/docs/release/15.3/, 2023. 6

[144] PostgreSQL 8.1.23 Documentation: Appendix E. Release Notes. https://www.postgresql.org/docs/8.1/release-8-1.html, 2005. 1

[145] PostgreSQL Anonymizer. https://gitlab.com/dalibo/postgresql_anonymizer, 2024. A.1

[146] PostgreSQL Extension Whitelist. https://github.com/dimitri/pgextwlist, 2024. 7.3, A.1

[147] PostgreSQL Global Temporary Tables. https://github.com/darold/pgtt, 2024. A.1

[148] postgresql-hll. https://github.com/citusdata/postgresql-hll, 2024. A.1

[149] PostgreSQL Partition Manager. https://github.com/pgpartman/pg_partman, 2024. A.1

[150] post_parse_analyze_hook. https://pgpedia.info/p/post_parse_analyze_hook.html, 2024. 7.1

[151] Prefix Range module for PostgreSQL. https://github.com/dimitri/prefix, 2024. A.1

[152] ProcessUtility_hook. https://pgpedia.info/p/processutility-hook.html, 2024. 7.1

[153] RDKit. https://github.com/rdkit/rdkit, 2024. A.1

[154] Redis. https://redis.io/, 2024. 1

[155] RediSearch. https://github.com/RediSearch/RediSearch, 2024. 3.3.5

[156] RedisGraph. https://github.com/RedisGraph/RedisGraph, 2024. 5.1.2

[157] Rewriter Query Rewrite Plugin Reference. https://dev.mysql.com/doc/

refman/8.0/en/rewriter-query-rewrite-plugin-reference.html, 2024. 3.1.5

[158] S3 Storage Engine. https://mariadb.com/kb/en/s3-storage-engine/, 2024. 3.3.4

[159] S3 Storage Engine. https://mariadb.com/kb/en/s3-storage-engine/, 2024. 3.1.7

[160] P. Schwarz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the starburst database system. In *Proceedings on the 1986 International Workshop on Object-Oriented Database Systems*, OODS '86, page 85–92, Washington, DC, USA, 1986. IEEE Computer Society Press. ISBN 0818607343. 2.1.1

[161] seg. https://www.postgresql.org/docs/current/seg.html, 2024. 7.5, A.1

[162] Semgrep. https://semgrep.dev/, 2024. 6.4

[163] sepgsql. https://www.postgresql.org/docs/current/sepgsql.html, 2024. A.1

[164] set_user. https://github.com/pgaudit/set_user, 2024. 3.1.8, A.1

[165] Shared ISpell Dictionary. https://github.com/postgrespro/shared_ispell, 2024. A.1

[166] shmem_request_hook. https://pgpedia.info/s/shmem_request_hook.html, 2024. 7.1

[167] shmem_startup_hook. https://pgpedia.info/s/shmem_startup_hook.html, 2024. 7.1

[168] Larissa Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Almeida. 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*, pages 59–66, 02 2018. ISBN 978-1-4503-5398-4. doi: 10.1145/3168365.3168376. 2.3

[169] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. Orca: A modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 337–348, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2595637. URL https://doi.org/10.1145/2588555.2595637. 2.2.1

[170] spi. https://www.postgresql.org/docs/current/contrib-spi.html, 2024. A.1

[171] SQLite. https://www.sqlite.org/index.html, 2024. 1

[172] SQLite Foreign Data Wrapper for PostgreSQL. https://github.com/pgspider/

sqlite_fdw, 2024. 2

[173] sqlite-vss. https://github.com/asg017/sqlite-vss, 2024. 3

[174] sslinfo. https://www.postgresql.org/docs/current/sslinfo.html, 2024. A.1

[175] stdaddr. https://postgis.net/docs/manual-2.4/stdaddr.html, 2024. 3.1.1

[176] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, page 340–355, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897911911. doi: 10.1145/16894.16888. URL https://doi.org/10.1145/16894.16888. 2.1.1

[177] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, page 102–107, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 9781450378062. doi: 10.1145/1133373.1133393. URL https://doi.org/10.1145/1133373.1133393. 2.1.2

[178] System Catalogs. https://www.postgresql.org/docs/current/catalogs-overview.html, 2024. 3.1.10

[179] System Extensions. https://developer.apple.com/documentation/systemextensions, 2024. 2.1.2

[180] tablefunc. https://www.postgresql.org/docs/current/tablefunc.html, 2024. A.1

[181] tcn. https://www.postgresql.org/docs/current/tcn.html, 2024. A.1

[182] TDS Foreign data wrapper. https://github.com/tds-fdw/tds_fdw, 2024. A.1

[183] test_decoding. https://www.postgresql.org/docs/current/test-decoding.html, 2024. A.1

[184] The FEDERATED Storage Engine. https://dev.mysql.com/doc/refman/8.0/en/federated-storage-engine.html, 2024. 3.3.4

[185] The Linux Documentation Project. https://tldp.org/, 2020. 2.1.2

[186] The MEMORY Storage Engine. https://dev.mysql.com/doc/refman/8.0/en/memory-storage-engine.html, 2024. 3.3.4

[187] TimescaleDB. https://github.com/timescale/timescaledb, 2024. 3.1.6, 5.1.3, 6, 7.5, A.1

[188] Trusted Language Extensions for PostgreSQL. https://github.com/aws/pg_tle, 2024. 3.1.4, 5.2.1, 7.5, A.1

[189] tsm_system_rows. https://www.postgresql.org/docs/current/tsm-system-rows.html, 2024. A.1

[190] tsm_system_time. https://www.postgresql.org/docs/current/

tsm-system-time.html, 2024. A.1

[191] unaccent. https://www.postgresql.org/docs/current/unaccent.html, 2024. A.1

[192] User-Defined Operators. https://www.postgresql.org/docs/current/xoper.html, 2024. 3.1.2

[193] User Statistics. https://mariadb.com/kb/en/user-statistics/, 2024. 3.1.10

[194] uuid-ossp. https://www.postgresql.org/docs/current/uuid-ossp.html, 2024. A.1

[195] vops. https://github.com/postgrespro/vops, 2024. A.1

[196] wal2json. https://github.com/eulerto/wal2json, 2024. A.1

[197] xml2. https://www.postgresql.org/docs/current/xml2.html, 2024. A.1