Difference Between SQL Dialects Using DBMS Test Suites

JIAQI ZONG* and XIAOYUAN JIN*, ETH Zurich, Switzerland

We studied the difference among the four SQL dialects, DuckDB, PostgreSQL, MySQL, and ClickHouse, by their official testsuites. We first analysed and extracted 18253 testcases from them and executed the testcases mutually, resulting in 12 pairs of test result. We compared the 12 pairs and produced valuable insights among different SQL products.

CCS Concepts: • **Information systems** → **Relational database model**.

Additional Key Words and Phrases: SQL Dialects, DBMS, Test Suites

ACM Reference Format:
Jiaqi Zong and Xiaoyuan Jin. 2024. Difference Between SQL Dialects Using DBMS Test Suites. 1, 1 (June 2024), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the area of database management systems (DBMS), Structured Query Language (SQL) is the cornerstone of data manipulation and querying. However, different platforms have been designed to more or less extend the standard SQL, for example, by adding new features, which has given rise to a phenomenon known as "SQL dialects", i.e., variations of SQL customized to the unique functionality and design choices of a particular DBMS. These dialects often differ in the features they support and their implementations, leading to challenges in cross-platform database interoperability or optimization.

This project aims to dissect and understand the nuances of these SQL dialects through a comparative analysis of four DBMSs: *PostgreSQL, DuckDB, MySQL, and ClickHouse*. Each database system is designed to meet different operational needs, such as Online Analytical Processing (OLAP) for analyzing complex query operations and Online Transaction Processing (OLTP) for managing transaction-oriented applications. These varying requirements have led to the development of unique dialects of SQL to suit the strengths and architectures of the respective DBMSs. In this project, by looking into these dialects in pairs, we will not only highlight their similarities and differences, but also reveal the design philosophies that led to them.

Thus, the core objective of our investigation is to uncover: (1) The SQL features that exhibit inconsistencies across the chosen four SQL dialects. (2) The variations in how these SQL features are implemented. (3) The rationale behind the design differences in these SQL dialects.

To achieve this, we will design a DBMS test suite, which is a structured collection of SQL queries designed to cover SQL functionality comprehensively and to test different specific SQL dialects. These test suites should be an ideal resource for verifying the compatibility of SQL queries across different DBMSs. Our approach consists of:

*Both authors contributed equally to this research.

Authors' address: Jiaqi Zong, jizong@student.ethz.ch; Xiaoyuan Jin, xiaojin@student.ethz.ch, ETH Zurich, Rämistrasse 101, Zurich, Zurich, Switzerland, 8032.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
© 2024 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
ACM XXXX-XXXX/2024/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(1) **Test Suite Extraction**: automatically extracting test suites from the selected DBMSs, PostgreSQL, DuckDB, MySQL, ClickHouse, and ensuring each test case encompasses the database setup, the SQL query statements for evaluation and also the expected results.

(2) **Pairwise Execution**: structure the selected DBMSs into six pairs and automate the execution of test suites from one system (DBMS A) on another (DBMS B), observing the query results to determine their compatibility.

(3) **Categorize results**: classify the outcomes into three distinct categories by comparing them to the expected results from the base system (DBMS A). (i) **ERROR**: Features are supported by DBMS A but not by DBMS B. (ii)**SAME**: Features are supported and behave identically in both DBMS A and B. (iii) **DIFFERENT**: Features are supported by both DBMS A and B, but their behaviors differ.

(4) **Statistical Analysis**: conduct a detailed analysis for each category by: (i) Identifying the SQL features involved in the test cases. (ii) Performing statistical analysis by computing percentages of three categories and highlighting intriguing examples. (iii) Providing an investigation of the reasons behind the differences in SQL feature support and implementation across DBMSs.

Through this methodology, we will not only quantify the compatibility of SQL functionality across different systems, but also reveal the reasons why certain SQL functionality is implemented in different ways. This exploration not only promises to enhance our understanding of DBMS design principles but also to inform better practices for database schema design, query optimization, and system selection tailored to application-specific requirements.

In summary, through these efforts, we hope to deepen our understanding of the nuances of SQL dialects and may contribute to the development of more robust, interoperable, and efficient DBMSs in the future.

2 RELATED WORK

2.1 Standard SQL and Dialects

Since 1970s, SQL has become the de facto standard for database communication. However, the existence of SQL dialects arises due to various factors such as historical context [1], vendor-specific [4] optimizations [2], and evolving requirements. Dialects emerge as different vendors implement SQL with their own optimizations, extensions, and interpretations, leading to variations in syntax, features, and behavior. These differences make achieving a universal SQL standard challenging, as reconciling diverse requirements and priorities across vendors can be complex.

Therefore, we have to study the features of different SQL dialects case by case. The purpose of this project is not to come up with a universal SQL standards, but to study the difference between varies dialects by integrating heterogeneous testing suites.

2.2 Testing Framework

Database testing framework or testing suite is a critical component in ensuring the reliability, robustness, and overall quality of a DBMS. Database testing frameworks are designed to validate the functionality, performance, and stability of a DBMS. These frameworks typically include a variety of tests that are automated and run against the database to ensure that it behaves as expected under different conditions and workloads.

Normally the testing consists of unit testing to verify individual components, integration testing to ensure proper interaction between these components, system testing to examines the entire DBMS under various conditions, and finally acceptance testing, including user acceptance, performance, and security testing. There are different methods of testing, such as regression testing, coverage testing, mutation testing, fuzzing test, as well dynamic analysis and static analysis [3].

In this project we only focus in the first phase testing, especially those related to SQL logic, because it is the intersection among the user-level functions of different database systems, while the other parts varies largely according to different implementations. In the approach chapter we will dive into the testing framework of each database system and discuss about possibilities of extracting and integrating heterogeneous testcases.

3 APPROACH

The main structure of our proposed DBMS test suite is described in the introduction part. We focus on the individual features of each SQL dialect system and the possibilities about how to extract and integrate heterogeneous test frameworks.

3.1 PostgreSQL

PostgreSQL provides a comprehensive test suite along with the source code repository, including regression test, isolation test, modules test, coverage test. We will focus on the regression test that is mainly for testing SQL logic. Normally we execute gmake installcheck to run the test. From the bottom level, the regression test consists of .sql files and .out files in directory src/test/regress

Test Suite Extraction: (1) Each .sql file in the sql directory corresponds to an individual test and we need to detect and extract **db.sql** code from the .sql files. Through some static analysis, we found that normally the setup code is at the beginning of the file. (2) After extracting setup code, the rest part of the .sql file is the **test.sql**. (3) Each test has a corresponding .out file in the directory called expected, which exactly works as a **result.txt** for matching the output of sql execution.

3.2 DuckDB

DuckDB use an extended version of the SQL logic test suite for testing plain SQL, which is adopted from SQLite. The test is driven by duckdb/test/unitest.cpp and from the bottom level, each test is a single self-contained file with a .test extension name. The test describes a series of SQL statements, together with either the expected result, a statement ok indicator, or a statement error indicator.

Test Suite Extraction: (1) Each .test file is a series of related SQL statements for one specific topic. Each statement consists of sql commands and corresponding expected result. We can extract the beginning of the series of statements to get the **db.sql**. (2) The test sql and result are separated by dashes. After extracting setup code, we can easily extract the test statements from the test-result pairs to get **test.sql** and **result.txt**.

3.3 MySQL

MySQL test suite is part of the source code repository, hosted in the mysql-test directory on GitHub. This directory contains a structured collection of test cases (.test files) and their expected outputs (.result files), carefully designed to validate MySQL's SQL processing capabilities. The organizational framework of this test suite can help us systematically extract and analyze SQL queries and their expected behavior.

Test Suite Extraction: (1) The initial phase involves parsing the .test files to extract SQL statements related to database creation and initial data population, and aggregated them into a **db.sql** file. (2) After the database setup, the body of the .test file is extracted, which contains a number of SQL queries and commands. This set of SQL statements reflects the SQL dialect of MySQL and is compiled into a **test.sql** file. (3)At last, extract the expected outcomes of the test queries from the corresponding .result files. These expected results, matched against the queries in test.sql, are compiled into the **result.txt** file.

When extracting the MySQL test suite, the main difficulty lies in the processing of .test files and .result files. In automated scripts, we need to distinguish read-only queries and write queries, ignore comments and blank lines, capture execution results, etc.. This is one of the difficulties of our work.

3.4 ClickHouse

ClickHouse test suite is accessible through its source code repository on GitHub, within the test/queries directory. The repository contains a well-organized collection of test cases, including .sql files for the SQL statements for the test and .reference files for the expected output. This setting facilitates an in-depth examination of ClickHouse's handling of SQL queries.

Test Suite Extraction: (1) The goal of the initial extraction phase is to extract files related to starting the database schema and initialization data from the .sql file. These prepared SQL statements are compiled into the **db.sql** file, laying the foundation for the subsequent testing phase. (2) Then directly copy the .sql files in the test/queries directory to the **test.sql** file. (3) The final step involves extracting the expected results from the .reference file and aligning them with the query in test.sql to get **result.txt**.

ClickHouse's test cases are designed with an analytical focus, often involving complex queries and large data sets. Therefore, our scripts must be adept at handling these complexities, including parsing and executing multi-statement transactions, and accurately matching query results to expected results.

4 IMPLEMENTATION AND EXPERIMENT

The implementation of the project can be divided into three main parts: dataset extraction, test execution, result analysis. (1) The dataset extraction methods depend on the structure of the test suites data from each dialect, which can be very different among different dialects. (2) The test execution methods depend on what kind of APIs the DBMS provide. There are interfaces for command line or specific languages such as C++ or Python. Therefore, we need to implement an execution framework for each SQL dialect. (3) The result analysis framework can be shared by all dialects that support the uniform three-file-testcase protocol.

4.1 Dataset extraction

4.1.1 PostgreSQL. PostgreSQL test suites consists of sql files as pure queries and txt files as expected result. We focus on read-only queries like SELECT and each select query is converted into the test.sql file, taking all the write queries before this query from the beginning as the db.sql file. The expected result is simply the output of execution of PostgreSQL with additional information which requires further normalization for comparing with other SQL dialects.

4.1.2 DuckDB. DuckDB test suites follows the sqllogictest, originally published by sqlite with complete documentation at https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki that explains how to parse and interpret the testcase file. There are several records in each file separated by empty lines. A record can be a statement, a query or a control record. We identify and convert each query to the test.sql file, taking all the statement records shows before the current query from the beginning as the db.sql file. The result right follows after the query and will be either empty or the form of tab separated table values which is easy to be parsed.

4.1.3 MySQL. MySQL test suites rely on a series of .test and .result files to test the correctness of the database. Though not directly written in SQL, these files contain plain SQL commands and are possible to be translated into .sql files by extracting the SQL lines. After doing so, we can use a similar strategy as above for these .sql files and divide them into db.sql and test.sql.

Initially, we filtered and removed unwanted elements such as comments and specific non-SQL lines, ensuring that the code is clean and relevant to the test scenarios. Key operations include the removal of comments starting with –error (indicating that the following queries are incorrect) up to the next blank line, single-line comments, and commands like connect, disconnect, and specific SQL commands that are not part of the main testing logic.

The next significant step is the reconstruction of database objects. We ensured that before any CREATE statement (for tables, views, events, functions, databases, etc.), there is a corresponding DROP statement if the object already exists. This approach ensures a clean state for each test, preventing conflicts due to pre-existing database objects.

After that, we addressed specific patterns in the SQL commands. For example, to streamline and enhance the efficiency of our test suite, we eliminated tests for certain functionalities, such as transactions and custom functions. We also modified quotes and split the quires into individual statements to isolate and sequence operations.

4.1.4 ClickHouse. ClickHouse test suites are divided into two categories: stateful tests (requires preloading data) and stateless tests (often creates small synthetic datasets on the fly, within the test itself). Processing the stateless type is relatively simple, only the query statement needs to be recognized by a script and put into test.sql, and the rest of the statements that manipulate the data can be put into db.sql. For stateful types, the preloaded data comes from the publicly available Yandex Metrica datasets, which contain anonymized web analytics data with hits (hits_v1) and visits (visits_v1). Due to the large amount of data, we extracted this part of the data loading to a high level instead of putting it into the db.sql of each test case. Meanwhile, the smaller db.sql units are not empty, as there are also some test cases that involve operations such as creating and dropping new tables.

However, as the preloaded data contained a large amount of ClickHouse-specific nested data structures, it is unsuitable for the other three DBMSs. Nevertheless, we proceeded with the test set extraction. And given its relatively small data size, we ultimately decided to exclude the stateful testcases.

To extract the stateless testcases, we removed any residual comments that might obscure the SQL commands and fixes unterminated strings that could lead to errors during execution. Also, by employing regular expressions to identify and manipulate SQL statements, the script ensures that all necessary modifications are made before the statements are divided into db.sql and test.sql. This includes ensuring proper spacing around SQL keywords to avoid syntactical errors and adjusting string terminations to prevent runtime exceptions.

4.2 Test Execution

In the official testing framework of the DBMS, they might use different APIs such as original C or C++ interface to execute the test queries. In our test system, we use Python as a driver to execute tests for all the four dialects, as all of them have complete Python interfaces.

We execute db.sql and test.sql in each test case in turn and then compare the results to the original result.txt from the current test case.

4.3 Result analysis

The most difficult part of result analysis is to find a uniform way to represent the output of each execution. Most of the workload actually lies in the former two parts, that is how to interpret and save the expected or executed result for each dialect in a uniform format. Concerning we run all the four dialects through python API which finally returns the result as a python object, we decide to use the object serialized as a string for comparison.

Note that the pairs are directed because running data A on dialect B is different from running data B on dialect A, so there are 12 directed pairs for analysing. The analysis includes two parts,

SQL Dialect	PostgreSQL	DuckDB	MySQL	ClickHouse
#Testcases	608	8685	1899	7061

overall metrics like accuracy and special case analysis where we discover the feature difference between dialects.

4.4 Experimental settings

During extracting, we observed the testcases carefully and filtered out many testcases containing SQL features that other dialects will not support. Such as the nested data structure of ClickHouse. We also removed testcases that uses transaction which relies on a undividable series of queries. Finally, as shown in the table, we totally extracted 18253 testcases from the four testsuites after filtering.

Concerning that the experiments are not performance sensitive, the tests will be conducted on two different Apple Silicon machines to speed the experiments up. To make it easier to reproduce the results, we use docker-compose to build all the software environment.

5 EXPERIMENTS RESULTS

5.1 PostgreSQL vs Others

5.1.1 Running ClickHouse testcases on PostgreSQL. There are 99 sames and 6823 errors. Most of the errors are ClickHouse features not supported by PostgreSQL, such as the `ENGINE` and other commands; some of them are syntax errors, such as `argument of CASE/WHEN must be type boolean`; there are also datatype problems such as array syntax in `SELECT [-1, 1 + 1];`

There are 139 differences, which can be divided into three situations. (1) Boolean values are represented by 0/1 in ClickHouse while in PostgreSQL they are represented by True/False; (2) Result of floating point computation may differ due to precision problem; (3) Hexadecimal such as `\x30` will escape to ASCII `P` in ClickHouse while remains `\x30` in PostgreSQL;

5.1.2 Running DuckDB testcases on PostgreSQL. There are 1772 sames and 6697 errors. Most of the errors are DuckDB features not supported by PostgreSQL, such as function `range()` and other commands; some of them are syntax errors such as the json object format.

There are 415 differences which are quite different and it is hard to classify them into simple categories: (1) Most of them are data type related problem: result of floating point computation may differ due to precision problem. The representation of floating points also varies; (2) Algorithmic problem: \ is treated as integer division in DuckDB but in PostgreSQL it is treated as floating division and the result is implicitly converted into a float; (3) Bit type: PostgreSQL treat `SELECT ('0101011'::BIT);` as 0 while DuckDB treat it as real bits; (4) Time interval type behavior: for query `SELECT '1.5 CENTURY'::INTERVAL;` PostgreSQL returns 54750 days while DuckDB returns 54000 days. We can also infer that they store date data in a different way and precision according to other date related tests; (5) ENUM type behavior: for query `select * from person where current_mood < 'sad';` PostgreSQL does not return anything because it does not treat ENUM types as comparable values like DuckDB; (6) Unicode support: it seems that PostgreSQL does not support unicode and simply ignores everything; (7) Array unnest: for query `SELECT UNNEST(ARRAY[[1, 2], [3, 4]]::VARCHAR[]);` PostgreSQL returns `('1',), ('2',), ('3',), ('4',)` while DuckDB returns `('[1, 2]',), ('[3, 4]',)`; (8) For query `SELECT 'a'::BYTEA::VARCHAR;` the two dialect also react in different ways; (9) Some orders of results are different when sorting order is not specified. We can infer they probably have different default order related to high-performance storage; (10) Join duplication problem: on query

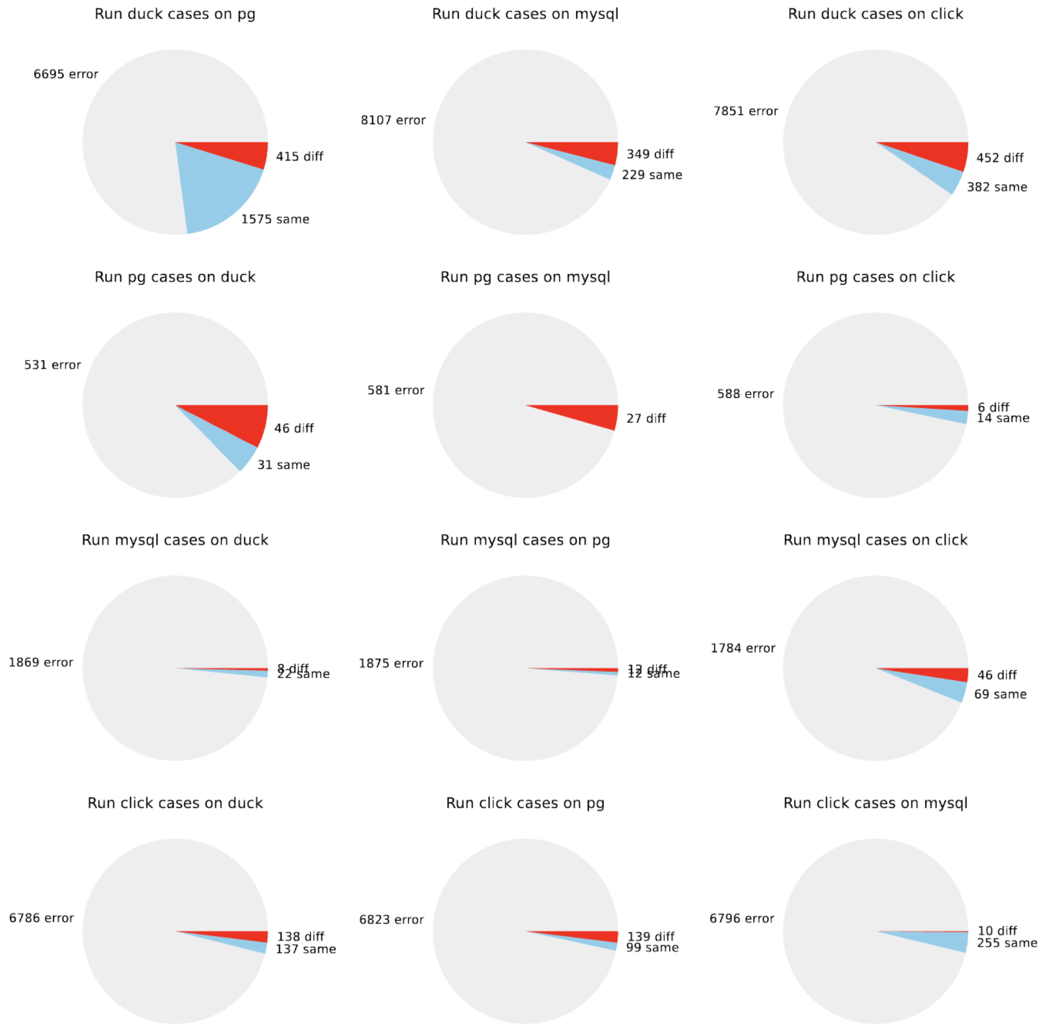

Fig. 1. Experiments result statistics

`SELECT * FROM left_table SEMI JOIN right_table USING (a);` PostgreSQL returns one row more than DuckDB; (11) Overflow number: `select '1e308'::float;` DuckDB take it as inf while PostgreSQL does not; (12) Explain command: the two dialect behave different when explaining their query plan; (13) Generate series: for query `SELECT generate_series(1, 6, 2);` they return the same content in different format. DuckDB returns an array object and PostgreSQL returns a series of rows; (14) Format: PostgreSQL accept the format function but does not behave correctly, such as returning empty for query `SELECT format('{}', 'hello'), format('{}: {}', 'hello', 'world');` (15) String pattern matching: DuckDB support stronger pattern matching for strings given the Similar command, such as `SELECT 'aaa' SIMILAR TO '[a-z][a-z].*';` (16) String split: PostgreSQL accepts split command but does not behave correctly, such as `select split_part('a,b,c',NULL,1);`

(17) pg_catalog: It is surprised to see that DuckDB even prepared tests specifically for PostgreSQL, such as `SELECT viewname, viewowner FROM pg_views WHERE viewname='v1';`

5.1.3 Running MySQL testcases on PostgreSQL. There are 1875 errors and 12 same cases. Most of the errors are MySQL features not supported by PostgreSQL, such as `ENGINE` and `SET NAMES utf8mb3;` some of them are syntax errors such as `@` commands.

There are 12 differences, which can be roughly devided into: (1) Integer division problem: MySQL treat \ as floating division and implicitly convert the result; (2) Statement code: `1 + /*!00000 2 */ + 3 /*!9999` only MySQL will parse the 2 and include it into the addition returning result as `10`; (3) Boolean values are represented by 0/1 in MySQL while in PostgreSQL they are represented by True/False; (4) XOR / exponent: MySQL treat ^ as XOR while PostgreSQL treat it as exponent operator, e.g., `select 3 ^ 11, 1 ^ 1, 1 ^ 0, 1 ^ NULL, NULL ^ 1;`

5.2 DuckDB vs Others

5.2.1 Running MySQL testcases on DuckDB. There are 22 sames and 1869 errors. Most of the errors are MySQL features not supported by DuckDB, such as `ENGINE` and `SET NAMES utf8mb3;` some of them are syntax errors such as multi-drop problems `Can only drop one object at a time.`

There are 8 differences, (1) Statement code: already mentioned before in PostgreSQL vs MySQL; (2) Algorithmic related : similar to the XOR / exponent problem; (3) Boolean values are represented by 0/1 in MySQL while in DuckDB they are represented by True/False; (4) Floating precision problem.

5.2.2 Running ClickHouse testcases on DuckDB. There are 137 sames and 6786 errors. Most of the errors are ClickHouse features not supported by DuckDB, such as `ENGINE`, or missing built-in tables such as `system.numbers;`

There are 138 differences, most of them belongs to the first category (Boolean representation): (1) Boolean values are represented by 0/1 in ClickHouse while in DuckDB they are represented by True/False; (2) ClickHouse has strong support for array such as `SELECT [1, 2][3], [1, NULL, 2][4], [('1',` while DuckDB simply return None; (3) Hexadecimal such as `\x30` will escape to ASCII `P` in ClickHouse while remains `\x30` in DuckDB;

5.2.3 Running PostgreSQL testcases on DuckDB. There are 31 sames and 531 errors. Most of the errors are PostgreSQL features not supported by DuckDB, such as `INHERITS`, `PARTITION`, `ON` for creating index, or some missing types such as `TEXT;`

There are 46 differences, (1) Type char(n): if the result is shorter than size n, DuckDB will return a truncated string, while PostgreSQL returns strings with padded blank chars. We can infer that DuckDB tends to optimize the space and PostgreSQL prefer time optimization. (2) Different behavior such as deduplication when grouping; (3) Floating precision problem; (4) Octal number system support: DuckDB returns 0 for query `SELECT 0o1000000000000000000000;` while it is indeed a non-zero number.

5.3 MySQL vs Others

5.3.1 Running PostgreSQL testcases on MySQL. There are 581 errors. We mainly focused on those caused by syntax and after filtering, 94 syntax errors were left. Most of the rest are due to built-in database objects and creation failures (due to inconsistent db.sql syntax). MySQL and PostgreSQL each have many specific syntax, as some examples given here. (1) Type casting: In PostgreSQL, `::numeric` is used to cast values to the numeric type. While in MySQL, `CAST()` is used instead. (2) Full join: PostgreSQL supports FULL JOIN directly while MySQL use a combination of LEFT JOIN and RIGHT JOIN with a UNION. (3) Various `@ operators` that are not supported by MySQL.

There are 27 differences and can be divided into 2 categories. (1) Different boolean values representations, as `0/1` in MySQL while `True/False` in PostgreSQL. (2) Different number representations. MySQL often returns binary representations of literals, and PostgreSQL returns the integer value or a decimal representation directly. For large values, PostgreSQL use `Decimal` type type to represent, while MySQL uses binary representation. Both DBMSs handle negative binary literals correctly, but the returned format differ slightly.

5.3.2 *Running DuckDB testcases on MySQL.*

There are 8107 errors. Apart from inevitable in-built objects-related errors, DuckDB supports the `IS DISTINCT FROM` clause, window functions like `ROW_NUMBER()` without needing an explicit window frame, and recursive common table expressions (CTEs) (e.g., `WITH RECURSIVE cte AS (...)`). It also includes advanced SQL functions like `array_sum`, the `EXCLUDE` clause in window functions, and specific data types such as `HUGEINT`. These features can cause the two DBMSs differ from each other.

There are 349 differences. Here we list some examples: (1) Generated Columns. DuckDB supports generated (computed) columns directly in the `CREATE TABLE` statement, meaning some are computed based on the expressions provided. While MySQL requires explicit declaration of generated columns using the `GENERATED ALWAYS` AS syntax. (2) Date arithmetic. They use different week formats, with DuckDB adhering to international standards. Additionally, DuckDB supports cast function in date calculate, whereas MySQL will directly return None.

5.3.3 *Running ClickHouse testcases on MySQL.*

There are 6796 errors. As mentioned above, here we use the stateless testcases in ClickHouse which are highly associated with the in-built tables and databases of ClickHouse. After filtering syntax errors, we have 3769 remaining. MySQL and ClickHouse have significant differences in syntax. Here we list a few examples: (1) Differences in operators, such as '==' in ClickHouse and '=' in MySQL. (2) Alias within Expression, such as `SELECT (dummy AS x) - 1` is not supported. (3) Unsupported functions such as `arrayExists`. (4) Unsupported array notation in SELECT.

There are 10 differences. (1) Number Representation: ClickHouse often uses floating-point numbers and uses scientific notation with large and small numbers, while MySQL uses precise decimal types. MySQL does not interpret `\x` as an escape sequence for hexadecimal characters, whereas ClickHouse does. (2) Different behaviors of function: Especially for function `greatest()` in our case, both DBMSs supports it, but ClickHouse returns as an integer, while MySQL returns as a floating-point number. Also, MySQL's `position` function is not standard SQL while Clickhouse can correctly return index. (3) Differences in ||: In MySQL, || is interpreted as a logical OR operator unless the `PIPES_AS_CONCAT` SQL mode is enabled, while ClickHouse uses || for string concatenation.

5.4 ClickHouse vs Others

5.4.1 *Running PostgreSQL testcases on ClickHouse.*

There are 588 errors. Among them, 319 syntax errors can be roughly divided into 3 categories. (1) Unsupported functions such as `UNNEST()` and unsupported operators such as <@, <<, <<|, ~=, etc.; (2) Unsupported `'VALUES'` clauses; (3) Unsupported `'LATERAL'` keyword, as ClickHouse joins typically require all joining tables to be independent or use subqueries that do not refer to any external columns from preceding tables.

There are 6 differences and can be divided into 3 categories: (1) Four of them are caused by different binary literal support. Firstly, ClickHouse directly supports binary literals, while PostgreSQL would typically require a decimal representation or casting from a string. Secondly, ClickHouse's use of unsigned integers allows it to represent 2ˆ63 directly within its native integer types, whereas PostgreSQL defaults to a numeric type for numbers that exceed bigint. (2) Configuration options.

Especially in this case, `extra_float_digits` in PostgreSQL can directly affect the textual representation of numbers. (3) Differences in scientic notion. Infinity is displayed as `Inf` in ClickHouse and `Infinity` in PostgreSQL.

5.4.2 Running DuckDB testcases on ClickHouse. There are 7851 errors. DuckDB supports several features that are not available in ClickHouse. For instance, DuckDB allows chained type casts (e.g., `SELECT 15::SMALLINT::BIT;`), and full joins. Additionally, DuckDB includes specific functions like `pg_size_pretty`, supports the `IS DISTINCT FROM` clause, and has a flexible array syntax (e.g., `SELECT ARRAY[1, 2, 3];`). It also allows multiple values in a single `INSERT` statement (e.g., `INSERT INTO table VALUES (1, 'a'), (2, 'b');`) and has comprehensive support for the `UUID` data type. These features highlight some of the key syntax and functional differences between DuckDB and ClickHouse.

There are 452 differences. (1) Different behaviors in functions, like `from_base64()`, `DATE_ADD()`, `ceil()`, etc.. (2) Unsupported operators. Take `//` as an example, DuckDB interprets it as integer division and returns two rows with `0`, while ClickHouse does not support it. Instead, it uses `/` for both integer and floating-point division. And `//` operator is treated as a comment indicator, resulting in a single row with `0`. (3) Differences in scientic notion. "NONE" is displayed as `nan` in ClickHouse and `None` in DuckDB. (4) Different boolean values representations, as `0/1` in ClickHouse while `True/False` in DuckDB. (5) Type casting. DuckDB supports chaining type casts, while Clickhouse does not support type cast.

5.4.3 Running MySQL testcases on ClickHouse. There are 1784 errors. ClickHouse and MySQL have significant syntax differences, making it challenging to successfully create the required tables for testing in `db.sql`. Most errors are caused by "table doesn't exist" or "missing columns.". Apart from that, most of the syntax errors are cause by MySQL's `IGNORE INDEX`, which is not supported in ClickHouse.

There are 46 differences. (1) Unsupported configurations. Lots of `SET` are not supported in ClickHouse and they can easily lead to differences in return. (2) Differences in data interpretation as mentioned above, basically lying in handling very large and small numbers and decimal representations. (3) In MySQL, `concat('a','b')` matches `concat('ab','')`, but in ClickHouse, this match does not occur as expected, resulting in `NULL`. (4) MySQL's comment parse as mentioned above in 5.1.3.

6 CONCLUSION

The most valuable part of this work is the comparison among the four SQL dialects. We gained many insights by comparing the outcome of the various testcases. The differences can be roughly divided into two categories: format difference and content difference. Obviously content differences are more valuable, through which we can discover the bottom level design of DBMS products, some OLAP DBMS may focus on optimizing computing speed such as ClickHouse, while other OLTP DBMS will prefer optimizting query latency.

From the perspective of software testing, we also learned what is a good testsuite through this project. Obviously DuckDB is has an outstanding testsuite equipped with comprehensive testcases, and a rigorous but at the same time very friendly testing framework. The DuckDB testsuite has the highest coverage among many aspects of a DBMS, and provides the highest-quality differences that reveals many insights when applying to other dialects. It is hard to develop testsuite for large software but it is necessary and will benefits us with a robust software and also countless new insights for development.

REFERENCES

[1] C. J. Date and Hugh Darwen. 1993. A Guide to the SQL Standard: A User's Guide to the Standard Relational Language SQL. https://api.semanticscholar.org/CorpusID:227287683

[2] Jim Gray. 1993. The Benchmark Handbook for Database and Transaction Systems. https://api.semanticscholar.org/CorpusID:260927671

[3] Dorota M. Huizinga and Adam Kolawa. 2007. Automated Defect Prevention. https://api.semanticscholar.org/CorpusID:107945788

[4] Jim Melton and Alan R Simon. 1993. *Understanding the new SQL: a complete guide.* Morgan Kaufmann.