Understanding the Difference Between SQL Dialects using DBMS Test Suites Report

ALBERT CERFEDA, KARLO PISKOR

Contents

24.05.2024

Eidgenössische Technische Hochschule Zürich
D-INFK Department
Switzerland

Author's address: Albert Cerfeda, Karlo Piskor.

1 ABSTRACT

This project tries to explore the variability among SQL dialects across different Database Management Systems (DBMSs). Given the distinct purposes of these systems, the SQL they utilize diverges significantly, leading to the concept of "SQL dialects". These dialects come with their own additional unique set of traits and extra features, but also might differ in the implementation of the base SQL features. In this project we aim to elucidate the main aspects: the inconsistencies in SQL features across dialects and the nature of these variations. To achieve this, the study leverages the individual test suites of each DBMS. Each test suite is designed to thoroughly encompass all the attributes and functionalities inherent to a specific SQL dialect. Applying these test suites, which are originally tailored for a particular DBMS, to an alternative DBMS, allows us to evaluate the distinctions in the feature sets offered by each DBMS but also their respective implementations. The analysis of these test suites will reveal the extent of SQL query interoperability between various dialects, providing insights into the complexities of SQL implementation and design across diverse database management systems.

2 INTRODUCTION

2.1 Motivation

Our project aims to determine and evaluate the slight variations and intricacies of SQL dialects across different Database Management Systems (DBMS). Such information is critical for developers, database administrators, and academics. We can identify the following salient points behind the motivation of our project:

- **Educational Value:** For those learning about databases, a clear understanding of the variations in SQL syntax and functionality across different systems can provide deeper knowledge and a robust educational foundation.
- **Development value:** The understanding gained through a system like ours enables developers to perform informed decisions and to develop application in a conscious manner, by utilizing the unique features of each DBMS effectively. This inherently results in more effective software systems and possibly prevents unnecessary resource overhead by avoiding the deployment of underutilized functionalities.
- **Research:** This study contributes to the ongoing research in the field of data management systems by providing an automated comparison of SQL dialects, potentially identifying areas where DBMSs can evolve. Researchers can use this tool for performing empirical studies and further expand the tool to adapt to new developments in the field.

2.2 Approach

Every DBMS is supported by a unique testing infrastructure that is designed to accommodate its special features and quirks. The proposed methodology entails integrating with the diverse testing environments and consolidating them into a unified and standardized framework. In order to conduct a comprehensive examination of the distinctions and intricacies of each SQL dialect, we proceed to carry out cross-evaluation by executing the testing statements of each DBMS against all others. Our methodology aims to ensure a comprehensive and unbiased evaluation of the DBMSs. In order to achieve this objective, our testing infrastructure will implement each DBMS in a segregated environment, resembling current software development methodologies.

3 RELATED WORK

SQLancer

https://github.com/sqlancer/sqlancer
SQLancer is an automated tool utilized for the purpose of testing Database Management Systems (DBMS) in order to identify logic errors that have the potential to generate result sets that are wrong. The process consists of two stages: firstly, the database is filled with mocked data, and secondly SQL queries are executed in order to detect any anomalies in the resulting database state are identified.

While SQLancer shares common ground with our project in which it aims to test each DBMS individually, its scope is limited solely on each individual DBMS, and doesn't therefore perform cross-evaluation like we are planning to do. In our situation, we aim to distinguish the variations in SQL dialects among different DBMS. Nevertheless, SQLancer's testing and fault identification methods can still be utilized to compare different implementations.

Translating between SQL Dialects for Cloud Migration

arxiv.org
This paper explores ways to bridge the gap between DBMS hosted on-premise compared to those hosted in the cloud such as AWS or Azure. It explores the approaches of Manual Rule Creation, Imitation Learning and Large Language Models. While we do not aim directly to bridge the gap of the different dialects in this project, such approaches might help us however to expand the test suites for each DBMS or create new ones should we have only limited tests available.

SQLFuzz

github.com/andygrove/sqlfuzz
SQLFuzz is a utility command line tool to generate SQL queries. Using this tool might be useful for our project to extend the test suites with more test cases to use on other databases.

SQLsmith

github.com/anse1/sqlsmith
Similar to SQLFuzz we can utilize SQLsmith to generate large amounts of queries specifically for PostgreSQL and SQLite which can be added to extend the existing test suites.

SparkFuzz: Searching Correctness Regressions in Modern Query Engines

ir.cwi.nl
SparkFuzz utilizes a similar approach to what we are aiming to do in this project. The system focuses on conducting SQL fuzzing on SparkSQL queries and subsequently comparing the query results when executed on Apache Spark, utilizing a PostgreSQL instance as the oracle. Prior to executing the generated queries, SparkFuzz initializes the database by randomly generating database schema and by populating it with fake data.

4 APPROACH

In our infrastructure, it is important that each DBMS is isolated and easily resettable. To this end we can deploy some DBMS under Docker containers. We're going to set up an automated system that takes the test suite designed for each database management system (DBMS), and run them on all the other DBMSs. Our goal is to make sure the comparisons we make are reliable and fair. To do this, we'll look closely at the queries in the test suites to find and replace any elements

that could give unpredictable results, like the `RAND()` function, or time-related functions such as `CURRENT_TIMESTAMP()` or `NOW()`, with constant values.

Beyond addressing specific functions, we'll adjust the configuration of each DBMS (e.g timezone configuration) in order to make them as alike as we can, in an effort to remove any biases in our evaluation that might arise from different configurations. By taking these steps we remove confounding factors, ensuring that any differences we observe in the test outcomes are genuinely due to the inherent characteristics and behaviors of the DBMSs themselves.

The main component of our infrastructure will be developed using Python. It will be responsible for connecting to the DBMSs, performing static analysis on the test suites, execute the queries, evaluate the results and produce the artifacts to be used in the final report. Python's extensive selection of database drivers, coupled with the utilization of Docker for deploying the DBMSs, allows us to compare and analyze an expansive array of database management systems with relative ease.

The objective of the project is to evaluate a minimum of four widely used Database Management Systems (DMBS): PostgreSQL, SQLite, ClickhouseDB, and DuckDB. This will enable us to conduct a comprehensive and exhaustive comparison. If one of the primary DBMS presents substantial difficulties or complete obstacles, we retain the choice to simplify the test environment or substitute it with an alternative DBMS.

After an analysis of the main proposed DBMSs for this project we can identify two main approaches in which these open source projects implement their own testing suites. The first one consists in generating SQL queries by employing some form of templating using a proprietary format, resulting in a convoluted and somewhat unintuitive testing process that can be challenging to work with for developers unfamiliar with the format. For example, after close inspection of the MySQL and MariaDB testing suite we notice how their testing suites are written in Perl and use their own format for specifying queries, hinting at a system where the suite reads each test file to generate the actual SQL queries. This SQL generation process makes it very hard to interface MySQL and MariaDB's testing suite with other database management systems.

The second approach consists in a much more straightforward testing suite consisting of a simple collection of SQL queries and their associated expected output. PostgreSQL, SQLite, ClickHouse and DuckDB rely on such a system. Their testing suites consist therefore of `query.sql` and `result.txt` tuples, allowing us to gather the queries to execute against other DBMSs and comparing results with relative ease.

5 TEST CASE EXTRACTION

5.1 Overview

As mentioned in section 4 we implement our infrastructure in Python, and DBMS which are harder to isolate and reset are deployed with Docker containers.

An obvious obstacle is that each DBMS employs its own testing suite and own format for specifying test cases and results. We therefore propose the implementation of our own testing infrastructure which carries out 3 phases.

(1) Test case ingestion: Our parser reads and translates each test case from the various DBMSs into one unified format. For DBMS like SQLite which uses its own proprietary format, the parser can run the testing suites and capture the SQL queries that get submitted to the database. For the other DMBSs which use a more standard test case format consisting of simple SQL statements and their expected output we simply read and parse the queries. Additionally some DBMS such as SQLServer or AzureSQL which are proprietary and closed source do not have a publicly available testing suite. It may be possible to substitute these

by using some publicly available test suites, but at the moment we haven't yet found one that is as comprehensive and thorough as other DBMS.

(2) Test case labelling: We identified 3 common types of tests: tests that check for a specific database state after executing queries, tests that expect the server to raise an error/exception, and tests that check for a specific output from the server's socket.

(3) Verification of correctness: To verify the correctness of our parser, we then run the parsed tests in our format against their respective database management systems. We therefore expect to receive the same identical output as to running the native testing suites.

5.2 Testing suites formats

To evaluate the difference in SQL Dialects we looked at each of the DBMS public repositories to identify which projects provide test suites that include SQL Query testing.

5.2.1 SQLite. SQLite's testing infrastructure is implemented in the TCL scripting language. Out of all the chosen DBMSs, this was the harder one to process.

```
do_execsql_test affinity2-100 {
  CREATE TABLE t1(
    xi INTEGER,
    ...
  );
  INSERT INTO t1(xi,...) VALUES(1,...);
} {}
```

Fig. 1. Example SQLite TCL test

Manually evaluating tests in this format obviously proved to be very difficult. Therefore, we made changes to the `tester.tcl` script so that whenever a test query is submitted, it is also logged in the regular test output log for future retrieval.

We included the puts statements, which facilitated the identification of each test case and its related SQL query from the log during the execution of the SQLite test suite. After the test suite has been completed, we are left with a 600MB .log file that we need to process through. This process though is rather trivial. After performing some test case selection, we reduce the selected tests to approximately 200 test cases. This reduction is achieved by eliminating tests that either invoke internal tester functions or excessively reset the database, since each reset takes several seconds.

```
-- Clean up in case a prior regression run failed
SET client_min_messages TO 'warning';
DROP ROLE IF EXISTS regress_alter_table_user1;
RESET client_min_messages;

CREATE USER regress_alter_table_user1;
...
```

Fig. 2. Example Postgres test file

Postgres. PostgreSQL instead proved to have one of the easiest to parse testing suites. Notice how each SQL query can be divided by the semicolon ";" [Figure 2]. We therefore make sure during our parsing phase to not consider the semicolon character as part of the SQL string.

Every file represents one test case that may contain multiple SQL statements. While parsing the tests we noticed that Postgres, as well as SQLite, sometimes use the dollar sign $ in order to

reference variables defined inside their testing scripts. We opted to eliminate such tests test cases, as well as some other ones which were using the `copy` command to populate the tables. We opted to use tests that either do this already in SQL or do not require data from the start.

```
# test double -> hugeint casts
statement ok
CREATE TABLE working_doubles(f DOUBLE);
CREATE TABLE broken_doubles(f DOUBLE);

query I
SELECT f::HUGEINT::DOUBLE FROM working_doubles
----
10.0
-10.0

query II
SELECT typeof(48329048239081049812 0984), 48329048239081049812 0984
----
DOUBLE 48329048239081049812 0984.0
```

Fig. 3. Example DuckDB test file

DuckDb. DuckDB was also one of the more challenging tasks to extract the test cases as they use a custom format to store SQL queries together with the expected results [Figure 3]. We noticed that the tests are divided between *statement* and *query* types. *Statement* tests are either `statement ok` or `statement error` which check for a correct execution and the emission of an exception respectively. `Query` tests instead are instead for checking the result set returned by the DuckDB server. `Query` tests are much more complicated than the basic statements. The second argument to a `Query` command indicates the amount of columns and the data types of the result set (i.e IT - 1 integer and 1 timestamp column).

Clickhouse. Similarly to PostgreSQL the test suite consists of multiple `.sql` files each containing multiple SQL queries. We only needed to split the SQL queries by a semicolon.

```
SET send_logs_level = 'fatal';

SELECT * FROM system.numbers LIMIT 3;
SELECT sys_num.number FROM system.numbers AS sys_num WHERE number > 2 LIMIT 2;
SELECT number FROM system.numbers WHERE number >= 5 LIMIT 2;
SELECT * FROM system.numbers WHERE number == 7 LIMIT 1;
```

Fig. 4. Example Clickhouse test file

5.3 Initial approach

The primary objective was to establish a unified JSON format containing, for each test case, a list of SQL queries and their corresponding outcomes. This would facilitate running the queries on different DBMSs and comparing the results. However, during the extensive development of parsers for SQLite and DuckDB, it became apparent that completing a full parser for the SQL queries, as well as parsing the expected results from the *.txt* files, was unfeasible within the project's timeframe. This challenge was compounded by the presence of heterogeneous data across databases and the varying formats of results for each DBMS. Additionally, some DBMS's testing suites are no running tests on the full fledged system and therefore also closely replicating the setup in which the testing suite is in was very hard.

5.4 New approach

As explained, the approach of also parsing the expected results proved to be far more time-consuming than initially anticipated. Consequently, we would have needed to drastically reduce our test size, perform a significant amount of work manually, or limit the number of DBMSs included.

To move forward, we made the following assumption: the tests written for each respective DBMS are correct. This means that if a query is expected to succeed, it will indeed succeed, and if it is expected to fail, it will fail.

The evaluation will be done in multiple steps:

- Create a parser for each DBMS to extract the SQL queries
- Store the queries in a unified JSON format
- Run each query for all the DMBS we are testing
- Get first results for all queries and narrow down selection of tests
- Rerun the evaluation with the narrowed down selection of queries

5.5 Unified test format

```
{
  "name": "Simple SELECT",
  "tests": [
    {
      "name": "",
      "query": "SELECT 1;"
    }
  ]
}
```

test1.json

```
{
  "name": "big5.sql",
  "tests": [
    {
      "query": "drop table t\u04f8",
      "name": "big5.sql"
    }
    ...
  ]
}
```

test_big5.json

Fig. 5. Showcase of our testing format

Figure 5 illustrates the format of the test files generated during the parsing phase. Although the project requirements suggested defining two separate SQL files—one for populating the database and another for running the queries—we opted to encapsulate each test within its own JSON file. The database is still populated by executing the queries sequentially, in the same order intended by the native testing suite. This approach ensures that if the original tests were meant to populate the database, our sequential format accomplishes the same, thus providing a basis for comparing Data Manipulation Language (DML) statements as well.

5.6 Test selection

As previously mentioned we applied an aggressive test selection process for tests whose results were influenced by certain arguments or other factors specific to the quirks of the testing suites. Essentially, we focused on analyzing the simpler queries, which still provided highly satisfactory results and yielded significant insights into the SQL dialects.

5.7 Execution and Interpretation of results

- **Execution Process**: Once we have unified the test suites into one format we run each test case originally designed for a specific DBMS on all the other supported DBMSs in our study. We meticulously record the outputs and log any discrepancies, such as differences in the data types or format of results.
- **Result Analysis**: We classify each executed test as either passed or failed. We then utilize such information to identify differences in the feature set and intricacies of the SQL dialects. Since we produce about 7'000 test cases, labelling test cases is very important for identifying patterns during our analysis. For example if we see a lot of failing cases labelled with

"materialized views" we can infer how the DBMS probably does not support materialized views, which is a feature specific to PostgreSQL.

Before we can run the final tests, we realized that in many cases (especially Clickhouse) has often a low percentage of compatibility when it comes to creating tables. Since many tests depend on some specific table existing and will fail if the creating of the table did not succeed. While there are many other dependencies that might cause other tests to fail, the missing table was by far the biggest issue. For ClickhouseDB we need to specify an ENGINE for the table, so the issue was adding this for the ClickhouseDB runns and removing for other DBs when we ran a query from Clickhouse.

The following Python code achieved this fix:

```
# If clickhouse is the target db check if the query contains create table
# and replace with create temporary table
# This is a workaround for many tests from other dbs failing in clickhouse
if target_db == "clickhouse":
    if "CREATE TABLE" in query:
        query = query.replace("CREATE TABLE", "CREATE TEMPORARY TABLE")
elif current_db == "clickhouse":
    if "CREATE TABLE" in query:

        # remove ENGINE = up to the next semicolon
        # this makes create table likely to pass in other dbs
        if "ENGINE =" in query:
            parts = query.split("ENGINE =")
            if len(parts) > 1:
                query = parts[0] + parts[1][parts[1].index(";") + 1:]
```

Fig. 6. Fix for ClickhouseDB for creating a table

6 EXPERIMENTAL RESULTS

Our main program gathers the tests produced by the parser in a standardized format and performs cross-evaluation by running the queries on every DBMS. The results for each query are saved as a Pandas DataFrame to facilitate easy comparison. The DataFrame includes the shape of the result sets and any exceptions that were emitted. After all the selected tests are run, we visualize a side-by-side comparison on a web dashboard.

The dashboard provides a general overview of how many tests executed without exceptions returned by the database. The most interesting results were obtained when the source database success rate was close to 100% (excluding test cases specifically designed to fail). In most cases, a low success rate for the source database was due to either faulty tests (caused by the parser being unable to parse some intermediate steps) or tests requiring a setup that was almost impossible for us to replicate, as mentioned in the previous section.

Report for sqlite

Test Name	Queries	Source Success	postgresql Success	postgresql Success Match	postgresql Shape Match	postgresql Result Match	clickhouse Success	clickhouse Success Match	clickhouse Shape Match	clickhouse Result Match	duckdb Success	duckdb Success Match	duckdb Shape Match	duckdb Result Match
upfrom4	55	50/55 (90.91%)	42/55 (76.36%)	41/55 (74.55%)	39/55 (70.91%)	38/41 (92.68%)	7/55 (12.73%)	12/55 (21.82%)	5/55 (9.09%)	5/12 (41.67%)	41/55 (74.55%)	42/55 (76.36%)	7/55 (12.73%)	5/42 (11.90%)
func7	72	72/72 (100.00%)	39/72 (54.17%)	39/72 (54.17%)	39/72 (54.17%)	38/39 (97.44%)	62/72 (86.11%)	62/72 (86.11%)	62/72 (86.11%)	56/62 (90.32%)	57/72 (79.17%)	57/72 (79.17%)	57/72 (79.17%)	54/57 (94.74%)
upfrom1	95	95/95 (100.00%)	64/95 (67.37%)	64/95 (67.37%)	64/95 (67.37%)	64/64 (100.00%)	27/95 (28.42%)	27/95 (28.42%)	2/95 (2.11%)	0/27 (0.00%)	63/95 (66.32%)	63/95 (66.32%)	8/95 (8.42%)	7/63 (11.11%)
unionall	128	123/128 (96.09%)	65/128 (50.78%)	70/128 (54.69%)	70/128 (54.69%)	70/70 (100.00%)	4/128 (3.12%)	9/128 (7.03%)	5/128 (3.91%)	5/9 (55.56%)	71/128 (55.47%)	76/128 (59.38%)	18/128 (14.06%)	13/76 (17.11%)
delete	64	61/64 (95.31%)	14/64 (21.88%)	17/64 (26.56%)	17/64 (26.56%)	16/17 (94.12%)	3/64 (4.69%)	6/64 (9.38%)	3/64 (4.69%)	3/6 (50.00%)	14/64 (21.88%)	17/64 (26.56%)	7/64 (10.94%)	6/17 (35.29%)
fkey6	73	63/73 (86.30%)	28/73 (38.36%)	30/73 (41.10%)	30/73 (41.10%)	30/30 (100.00%)	0/73 (0.00%)	10/73 (13.70%)	10/73 (13.70%)	10/10 (100.00%)	21/73 (28.77%)	27/73 (36.99%)	8/73 (10.96%)	8/27 (29.63%)
insert	67	61/67 (91.04%)	23/67 (34.33%)	29/67 (43.28%)	26/67 (38.81%)	24/29 (82.76%)	7/67 (10.45%)	13/67 (19.40%)	6/67 (8.96%)	6/13 (46.15%)	19/67 (28.36%)	25/67 (37.31%)	8/67 (11.94%)	6/25 (24.00%)
incrblobfault	13	13/13 (100.00%)	0/13 (0.00%)	0/13 (0.00%)	0/13 (0.00%)	0/0 (0.00%)	0/13 (0.00%)	0/13 (0.00%)	0/13 (0.00%)	0/0 (0.00%)	3/13 (23.08%)	3/13 (23.08%)	0/13 (0.00%)	0/3 (0.00%)
skipscan2	67	67/67 (100.00%)	43/67 (64.18%)	43/67 (64.18%)	43/67 (64.18%)	43/43 (100.00%)	0/67 (0.00%)	0/67 (0.00%)	0/67 (0.00%)	0/0 (0.00%)	43/67 (64.18%)	43/67 (64.18%)	1/67 (1.49%)	1/43 (2.33%)
orderby3	22	22/22 (100.00%)	22/22 (100.00%)	22/22 (100.00%)	22/22 (100.00%)	22/22 (100.00%)	2/22 (9.09%)	2/22 (9.09%)	0/22 (0.00%)	0/2 (0.00%)	22/22 (100.00%)	22/22 (100.00%)	14/22 (63.64%)	14/22 (63.64%)
window3	1486	1458/1486 (98.18%)	1018/1486 (68.51%)	993/1486 (66.82%)	993/1486 (66.82%)	993/993 (100.00%)	1/1486 (0.07%)	28/1486 (1.88%)	27/1486 (1.82%)	27/28 (96.43%)	1274/1486 (85.73%)	1249/1486 (84.05%)	1218/1486 (81.97%)	703/1249 (56.29%)
vtab_err	2	1/2 (50.00%)	0/2 (0.00%)	1/2 (50.00%)	1/2 (50.00%)	1/1 (100.00%)	0/2 (0.00%)	1/2 (50.00%)	1/2 (50.00%)	1/1 (100.00%)	0/2 (0.00%)	1/2 (50.00%)	1/2 (50.00%)	1/1 (100.00%)
walslow	23	22/23 (95.65%)	11/23 (47.83%)	12/23 (52.17%)	12/23 (52.17%)	12/12 (100.00%)	7/23 (30.43%)	8/23 (34.78%)	1/23 (4.35%)	1/8 (12.50%)	13/23 (56.52%)	14/23 (60.87%)	1/23 (4.35%)	1/14 (7.14%)

Fig. 7. Report snippet 1

Nr	Query	Source Shape	Source Result	Source Exception	Source Success	postgresql Success	postgresql Values Equal	sqlite Success	sqlite Values Equal	clickhouse Success	clickhouse Values Equal
3	CREATE UNIQUE INDEX uidx ON integers (i)	[0, 1]	Count		True	True	False	True	False	False	False
4	INSERT INTO integers VALUES (3, 4), (2, 5)	[1, 1]	Count: 0 → 2		True	True	False	True	False	True	False
5	SELECT * FROM integers	[2, 2]	i, j: 0 → 3, 4; 1 → 2, 5		True	True	True	True	True	True	True
6	INSERT INTO integers VALUES (6, 6), (3, 4);	None		Constraint Error: Duplicate key "i: 3" violates unique constraint. If this is an unexpected constraint violation please double check with the known index limitations section in our documentation (https://duckdb.org/docs/sql/indexes).	False	False	True	False	True	True	False
7	INSERT INTO integers VALUES (NULL, 6), (NULL, 7)	[1, 1]	Count: 0 → 2		True	True	False	True	False	True	False
8	UPDATE integers SET i=77 WHERE i IS NULL	None		Constraint Error: PRIMARY KEY or UNIQUE constraint violated: duplicate key "77"	False	False	True	False	True	False	True

Fig. 8. Report snippet 2

Nr	Query	Source Shape	Source Result	Source Exception	Source Success	postgresql Success	postgresql Values Equal	sqlite Success	sqlite Values Equal	clickhouse Success	clickhouse Values Equal
3	CREATE UNIQUE INDEX uidx ON integers (i)	[0, 1]	Count		True	True	False	True	False	False	False

DB	Success	Error	Result	Shape	Data types	Shape Equal	Columns Equal	Dtypes Equal	Values Equal
duckdb	True		Count	[0, 1]	Count int64				
postgresql	True			None	None	False	False	False	False
sqlite	True			None	None	False	False	False	False
clickhouse	False	:HTTPDriver for http://localhost:8123 returned response code 400) Code: 62. DB::Exception: Syntax error: failed at position 43 ('FORMAT') (line 2, col 2): FORMAT Native. Expected one of: OR, AND, IS NOT DISTINCT FROM, IS NULL, IS NOT NULL, BETWEEN, NOT BETWEEN, LIKE, ILIKE, NOT LIKE, NOT ILIKE, REGEXP, IN, NOT IN, GLOBAL IN, GLOBAL NOT IN, MOD, DIV, TYPE, GRANULARITY, end of query. (SYNTAX_ERROR) (version 24.4.1.2088 (official build))		None	None	False	False	False	False

Fig. 9. Report snippet 2

The example view in Figure 7 helps us to quickly identify either such broken tests that we exclude from further evaluation, or to spot errors in our test parser.

In the view in Figure 8 we notice the `test_unique_temp.test` and all the SQL statements contained withing the file and their outcome. Notice for example how for query #6 and #8 the expected result does not match and therefore the statement is marked as failed. To view in detail how all DBMS performed on a specific query we can click the query for a detailed view.

By analyzing the passing/failing tests we can draw some conclusions on the functionalities and behaviour of each DBMS.

6.1 Syntax

Postgres supports table inheritance through the INHERITS keyword, and from some tests it is obvious that it is the only DBMS to support it, in which all other DBMS return an error upon encountering the INHERITS keyword. Source [`create_misc.sql`]

While developing our parser, we encountered an interesting error that was due to us overlooking comments when parsing tests, which led to us discovering that every DBMS, for the exception of SQLite, accepts queries that are empty (or that just contain a comment).

Furthermore, when running SELECT queries, if we select a column more than once Clickhouse doesn't show the duplicated column while DuckDB opts to add a suffix I eg. `_01`, `_02` etc to make each column identifier unique.Another very interesting fact that arises from our analysis is that when using the `ORDER BY` keyword, PostgreSQL doesn't exhibit case sensitivity when sorting, while all other DBMS clearly sort not based on the alphabetical order but rather on the character value, resulting therefore in capital letters getting sorted in front of small letters. Finally using `GROUP BY` and `SORT BY` together with a expression to be evaluated yields very inconsistent results. Source [`select_implicit.sql` - **??**], On the other hand, UNIONS worked very consistently across DBMSs.

A DuckDB feature that stands out is the ability to perform array indexing inside of queries. This curious feature is unsupported by all other DBMS. [`test_list_index.sql`].

Queries which alter the DB/table schema usually return consistent results, although there are some specific which don't work which often arise when modifying some restricted schema which is usually DBMS specific. (e.g. `pg_user`, `pg_database`). Source [`alter.sql`]

6.2 Data Definition Language (DDL)

From the tests we noticed that the DBMSs don't all return the same data when executing a successful DDL query (e.g `CREATE` query), while some return results that are in a different shape.

Generally, Postgres and SQLite tend to return the exact same results. Obviously, the `SERIAL` data type is a Postgres-specific test, leading Duckdb and Clickhouse to return a syntax error on tests containing that keyword [`plpython_schema.test`]. On the same note of Postgres-specific features, the other DBMS clearly do not support table partitioning and adding comments on columns like PostgreSQL does, therefore we are unable to utilize the `PARTITION` and `COMMENT` keywords [`partition_aggregate.sql`]. On the matter of data types, we noticed that the `BLOB` data type is only supported by SQLite and DuckDB.

6.3 Built-in functions

Clickhouse doesn't handle UTF-8 characters at all, and it is very apparent from some tests which show that Clickhouse does not handle well the variable nature of the UTF-8 encoding. Clickhouse unlike every other DBMSs also has issues in returning the correct length of complex unicode strings that contain special characters and/or emojis.

When running some native Clickhouse tests, we noticed that Clickhouse features a plethora of timestamp-related functions, which are not offered in every other DBMS. We therefore have a wide array of datetime tests failing because such functions are defined only in Clickhouse [`parse_datetime_besteffort_or_null_emptystring.sql`].

6.4 Boolean handling

Postgres and Clickhouse are the only ones that have passing queries that use the `bit` built-in function. In particular, Sqlite and Clickhouse want a different format for specifying binary strings. [`bit.sql`]. Regarding SQLite, we noticed that the `BOOL` data type doesn't exist, and therefore it uses integers for representing binary values. Our testing infrastructure performs implicit casting when comparing values, therefore the test still passes [`bool.sql`]. Additionally all DBMS support the `ILIKE` comparison operator while SQLite doesn't.

6.5 Error handling

A very curious test instance regarding error handling surfaced when running a test which performs division by 0. We can see from the tests that division by 0 is not allowed by Postgres, while it is allowed by other DBMSs. When casting the result to an Integer in a second moment, most DBMS emit then an error.

6.6 Test suite limitation

We build the application to be as expandable as possible. Adding new DBMS into the existing infrastructure would require minimal cost. The main time cost would come from the writing of the new parser for the new DBMS. Extending the runner to then evaluate the tests would be minimal.

7 FUTURE WORK

As mentioned before, the tests are not always perfect and may have dependencies that need to be resolved to successfully execute subsequent queries. Detecting and fixing issues after a failed test would be beneficial for thoroughly testing all DBMSs with all test suites. However, this would require an in-depth understanding of each query's purpose and potentially rewriting them. Another task would be to ensure that all tests pass on their own source DBMS, as some queries failed due to missing plugins or data imports that were executed by DBMS-specific test runners.

It would also be interesting to expand our test runner to include more test suites and additional DBMSs. This would allow for a broader comparison of SQL dialects and their differences.

8 ARTIFACTS

The public repository of the project is hosted on GitHub https://github.com/BattleRush/AST-SQLDialects-Checker together with the latest report HTML file and the files to reproduce the results.