Vrije Universiteit Amsterdam Universiteit van Amsterdam

Master's Thesis

A Testing Strategy for Hybrid Query Execution in Database Management Systems

Author: Florian Gerlinghoff (2734858)

1st supervisor:	prof. dr. Peter Boncz
daily supervisor:	Boaz Leskes (MotherDuck)
2nd reader:	dr. Natalia Silvis-Cividjian

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

September 24, 2023

Abstract

Powerful personal computing devices give rise to a new model of query processing in database management systems: hybrid query execution. In this model, the execution of a single database query can be distributed between a client and a server. Because the client can take part in the processing as well, local resources can be harnessed more efficiently and computation can be moved close to the data. Hybrid query execution has been implemented based on DuckDB, an analytical, embeddable database system.

In this master's thesis, we describe the development of a test suite for verifying the functional correctness of the hybrid-query-execution implementation. We used an existing test suite for DuckDB as a starting point and progressively enabled the SQL-based tests to target the new query processing model. Our effort was directed by regularly measuring the code coverage achieved by the test suite, thereby identifying new problems to work on.

Through this systematic approach, we increased the code coverage from 25% to around 80%. Our test suite serves as a valuable tool for validating the correct operation of hybrid query execution in DuckDB.

Contents

List of Figures **v**

List of Tables **vii**

List of Listings **ix**

Acronyms **xi**

1 Introduction **1**

2 Background and Related Work **5**

2.1 Relational Database Management Systems 5

2.1.1 The Relational Model 6

2.1.2 Relational Algebra 7

2.1.3 Types of Workload 8

2.2 Query Processing in RDBMSs 10

2.2.1 Components of an RDBMS 10

2.2.2 How a Query Is Processed 11

2.2.3 Parallel Query Execution 16

2.3 DuckDB 18

2.4 Software Testing 18

2.4.1 Functional Software Testing 19

2.4.2 Manual and Automated Testing 20

2.4.3 Black-Box and White-Box Testing 20

2.4.4 Code Coverage 21

2.4.5 Testing Levels 22

2.5 Testing of Database Systems 22

3 The DuckDB Test Suite **25**
3.1 Correctness Testing 25
3.1.1 SQLLogic Tests 26
3.1.2 C++ Tests 28
3.1.3 Fuzzing 28
3.2 Compatibility Testing 29
3.3 Performance Testing 29
3.4 Code Quality Measurements 30

4 Hybrid Query Execution **31**
4.1 Advantages and Disadvantages of Existing DBMS Architectures 31
4.1.1 Distribution of Query Processing 32
4.1.2 Comparison of Client-Server and Single-Node Architectures 33
4.2 A Motivating Example 35
4.3 High-Level Overview 36
4.4 Implementation in DuckDB 37
4.4.1 Binding 38
4.4.2 Planning 38
4.4.3 Execution 40

5 Foundational Considerations for the Test Suite Design **41**
5.1 The Trade-Off Between Risk and Costs 41
5.2 Reusing an Existing Test Suite 42
5.3 Modifications Made by Hybrid Query Execution 43
5.3.1 Modes of Testing 44
5.3.2 The Test Server 44
5.4 Design of Experimental Evaluation 45
5.4.1 Metrics for the Effectiveness of a Test Suite 46
5.4.2 Measuring the Code Coverage 46

6 Leveraging DuckDB's Test Suite for Hybrid Query Execution **49**
6.1 Initial Situation 50
6.2 Using a Feature-Complete Test Runner 51
6.3 Getting All Test Scripts to Run 53
6.4 Disabling Tests That Are Expected to Fail 54
6.5 Aggregating an Overview of Test Failure Causes 55

6.6 Handling Multiple Connections 58
6.7 Check for Correct Errors on Expected Failures 59
6.8 Running Tests in Parallel 60
6.9 Summary 60

7 Discussion and Future Work **63**
7.1 Concluding Code Coverage Analysis 63
7.1.1 Subsisting Coverage Holes 63
7.1.2 Impact of Unsupported Functionality on Code Coverage 66
7.2 Code Coverage to Measure Test Effectiveness 67
7.3 Use of the Test Suite in Practice 68
7.4 The Local Test Mode 69
7.5 Using an External Test Runner 69

8 Conclusion **71**
8.1 A Final Look at Our Research Questions 71
8.2 Future Improvements to the Test Suite 73
8.2.1 Testing “Full Hybrid” Execution 73
8.2.2 More Diverse Workloads 74
8.2.3 Testing Non-Functional Requirements 75

References **77**

List of Figures

2.1 Two relations in a database of a fictional airline reservations system 7
2.2 Row-oriented storage layout 9
2.3 Column-oriented storage layout 9
2.4 Steps that a DBMS takes to process a query 12
2.5 Logical query plans for the SQL query in Listing 2.1 13
2.6 Physical query plan for the SQL query in Listing 2.1 14
2.7 Pipelines in the query plan from Figure 2.6 17

3.1 Overview of tests contained in the DuckDB test suite 25

4.1 Architectural styles from the viewpoint of distributing database application and DBMS 32
4.2 Processing of an exemplary query of the airline's data analyst 36
4.3 High-level visualization of the hybrid-execution model 36
4.4 Logical query plan for Listing 4.1 with different placements of operators . . 39

6.1 Visualization of the line coverage of physical operators after step $\mathbf{S_2}$ 57
6.2 Line coverage after each step $\mathbf{S_0}$–$\mathbf{S_4}$ 61
6.3 The line coverage achieved and time needed for a test run as a function of the number of test scripts that are executed 62

7.1 Visualization of the line coverage of physical operators after step $\mathbf{S_5}$ 64
7.2 Coverage in local test mode when excluding test scripts related to unsupported aspects 67

List of Tables

2.1 Comparison of OLTP and OLAP workloads 10

4.1 Comparison of the single-node and typical client-server DBMS architecture 34

6.1 C++ files in *duckdb/src/execution/operator* with a line coverage less than 50% 58

6.2 Line coverage after each step $\mathbf{S_0}$–$\mathbf{S_4}$ 61

6.3 Overview of the number of executed tests and the time required for this for steps $\mathbf{S_0}$–$\mathbf{S_5}$ 62

List of Listings

2.1 Example query to find the flight number of Mr. Springer 12
2.2 Pseudocode implementation of the `next` method of a filter operator 15

3.1 Example of an SQLLogic test script 26
3.2 Testing the ACIDity of DuckDB transactions in an SQLLogicTest script. . . 27

4.1 Example SQL query that is executed hybridly 37

6.1 Code to illustrate how test cases were defined 53

LIST OF LISTINGS

Acronyms

ADBC	Arrow Database Connectivity. 29
API	application programming interface. 18, 29
CPU	central processing unit. 9, 14–17, 34
CSV	comma-separated values. 35, 56, 60, 68
DBMS	database management system. v, vii, 5, 6, 8, 10, 12, 16–18, 22, 23, 25, 28–36, 41–45, 47, 49, 56, 65, 71, 74, 75
DDL	data definition language. 6
DML	data manipulation language. 7, 37, 39
DQL	data query language. 7, 37
HQE	hybrid query execution. 37, 38, 41–44, 47, 49–56, 59, 60, 65, 66, 69, 74, 75
HTTP	Hypertext Transfer Protocol. 51
I/O	input/output. 14
IMDb	Internet Movie Database. 30
ISO	International Organization for Standardization. 19
JDBC	Java Database Connectivity. 29
LDBC	Linked Data Benchmark Council. 30
NUMA	non-uniform memory access. 17
ODBC	Open Database Connectivity. 29
OLAP	online analytical processing. vii, 8–10, 15, 18, 28, 30

OLTP online transaction processing. vii, 8–10, 26

RDBMS relational database management system. 5, 6, 10, 11, 18, 22, 23, 33, 38, 49

SIMD single instruction, multiple data. 15

SQL Structured Query Language. v, ix, 2, 5–7, 10, 11, 13, 14, 22, 23, 25–27, 33, 36–38, 41–44, 49–56, 58, 65, 66, 69, 71, 73, 75

SSD solid-state drive. 11

TPC Transaction Processing Performance Council. 28, 30, 75

1 Introduction

Data has become a crucial asset for many companies worldwide. So crucial, in fact, that it has even been called the "oil of the 21$^{\text{st}}$ century". This prompted companies to collect more and more data, for example, to aid business decision making or to be able to apply advanced technologies like machine learning. All this data, of course, has to be processed at some point.

Database research has been at the forefront of this development and contributed to the rise of big data systems like Apache Hadoop and Apache Flink, or more recently cloud-based data warehouses such as Amazon Redshift, Google BigQuery and Snowflake.

But for many purposes, this massive amount of data might not even be necessary. For many purposes, only the most recent data might be useful or the totality of data could be summarized, shrinking the data volume that needs to be processed immensely. And what remains might fit onto a single machine. In short, laptops and other personal computers, powerful as they are nowadays, might be able to answer many of the questions from the data locally. No complex distributed system needed. This is the central thesis behind the dictum "Big Data Is Dead"[1] and calls for rethinking how data processing is done.

It is the motivation for the emergence of a new way of query processing: *Hybrid query execution*[2]. This novel execution model taps into the resources available on local computers to enhance the performance and reduce the cost of processing. While in a traditional client-server architecture, the client only uses a thin connector to the database server, in a hybrid-execution architecture, a complete database system is available locally. This enables the client to participate in the processing of queries and the computation of results, putting its own compute resources to use.

[1] `https://motherduck.com/blog/big-data-is-dead/`

[2] In the following, we refer to this model simply as *hybrid execution*.

1. INTRODUCTION

DuckDB, an analytical, embeddable database system, can run on a variety of platforms, from smartphones to laptops to big servers, and comes with a small footprint and good performance. Therefore, it is well-suited to be used in a hybrid-execution architecture. And in fact, a company called MotherDuck[3] has implemented hybrid execution on top of DuckDB.

While, in a world of “Big Data Is Dead”, less data is sufficient to answer questions, analyze trends, or help with decision-making, this does not derogate the importance of the data or the conclusions drawn from it in any way. Consequently, it is essential that database systems process this highly important data correctly and reliably, providing accurate results to queries. Faults could have dire consequences, ranging from incorrect decision-making to the erosion of customer trust due to frequent errors.

To become confident that a software system works correctly or, if not, to identify and fix existing problems and raise the quality of the system, software testing is an essential tool. Especially a new technology like hybrid execution should be thoroughly tested, making sure that everything works in practice as intended.

In this research, we develop a test suite for the implementation of the hybrid-execution model by MotherDuck, which is based on DuckDB. To reduce the likelihood of bugs staying undiscovered and gain high confidence in the system, our primary goal is to achieve a wide coverage of the system's functionality while minimizing the development effort.

DuckDB uses an extensive test suite of SQL-based tests that are not coupled to internals of DuckDB, but specifies SQL statements and expected results as a higher level of abstraction. Therefore, it is portable between different database systems. This includes MotherDuck, since the expected result does not change when switching to the hybrid-execution model. We want to build upon this test suite and leverage it to test hybrid execution, giving us quickly a set of tests that encompass a substantial portion of our system's surface area. This leads us to the following research questions:

RQ1 How can the DuckDB test suite be adapted to test our DuckDB-based implementation of hybrid execution?

RQ2 How effective are these adaptations?

We use code coverage as an evaluation and guidance tool. It helps us to identify weaknesses in the test suite and directed our efforts to address these shortcomings. This approach gives us valuable insight into the strengths and limitations of using code coverage in this context.

[3] `https://motherduck.com/`. The work in this thesis was done while the author was at MotherDuck.

In this thesis, we describe our stepwise approach of developing this test suite and applying code coverage to evaluate each step. We were able to increase the overall coverage from 44% to 84% and to develop a test suite that is used often and extensively in practice, validating each and every change made to the system. But before describing how we got there, we set the scene.

In Chapter 2, we take a look at the inner workings of database systems and present one specific one called DuckDB. We also provide information about software testing and the different approaches that exist in this area. In Chapter 3, we go on to describe the test suite that is used to ensure that DuckDB works correctly and with high performance. This test suite is the basis for our following work. Chapter 4 portrays hybrid query execution and explains how it is implemented with DuckDB. These chapters provide the foundational information for our own research.

In Chapter 5, we motivate why we base the development of a test suite for hybrid execution on an existing test suite. Afterwards, we arrive at the gist of this master's thesis: We describe how we developed a comprehensive test suite for hybrid execution in Chapter 6 and evaluate the implementation steps we took. We discuss our approach in Chapter 7. Finally, we conclude the thesis in Chapter 8 and end with ideas for future work.

2

Background and Related Work

This chapter provides a background on database systems and software testing. We begin with an overview of relational database management system (RDBMS), highlighting their components and describing how SQL queries are processed. We then present DuckDB, the RDBMS based on which hybrid query execution has been implemented. Finally, we turn to software testing, exploring its significance in ensuring the quality of software systems, including DBMSs. We categorize different approaches of software testing and give an overview of various code coverage metrics which are also used to determine the effectiveness of tests. In the last section, we give a brief account on the testing of DBMSs.

2.1 Relational Database Management Systems

Every more-than-simplistic software application has to deal with persistent data. It needs to store information that users entered into the system, data from events that were triggered, or results from the processing of outside data sources. And even though all this data could potentially be stored in normal files on the computer's file system, this way of storage would soon become inconvenient and inefficient to work with [81, pp. 5–7]. One of several drawbacks that arises from storing data in a flat file system is that program-specific code is required to access the data instead of being able to use an abstract and standardized query language. Furthermore, when the application does not take measures against it, data can easily become redundant and consequently inconsistent, conflicts can arise from concurrent access to files, and data can quickly get lost when the application crashes.

For that reason, many software applications rely on a database management system (DBMS) to handle the access and manipulation of data, along with all the aforementioned concerns. From the perspective of other applications, a DBMS provides an interface to databases, i.e., (possibly very large) collections of interrelated data. The implementation of the DBMS functionalities is hidden behind this interface. Hence, applications do not need to be concerned with storage organization, data consistency, integrity, security, etc. [81, p. 2].

Definition 1. *A database management system (DBMS) is a software-based system that maintains databases for other applications and makes the data available to them for reading and writing.*

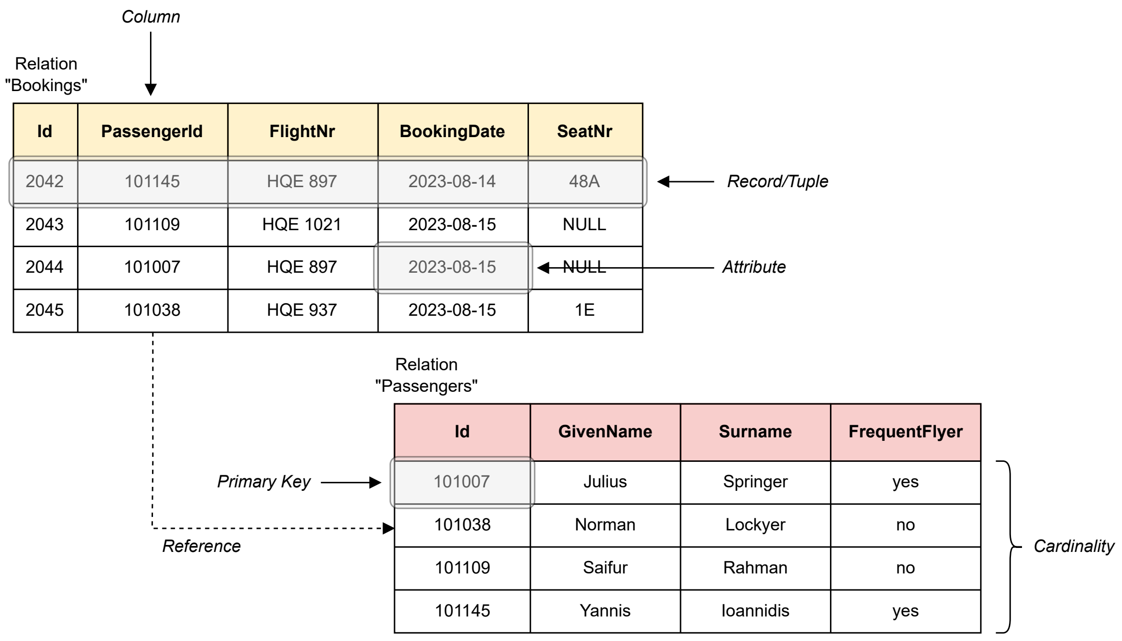
2.1.1 The Relational Model

The abstract interface to the data provided by a DBMS is called the *data model*. It describes the data itself, the values that are valid, as well as relationships within the data [81, p. 8]. One of the main data models is the relational model, which was first presented in a 1970 paper by Codd [11] and found widespread adoption in the decades that followed [27, p. 3]. The relational model organizes the data into relations (or tables), which themselves consist of columns. A column is of one particular data type. A relation contains records (or tuples) with attributes that correspond to its columns. Each record in a relation is unique and can be identified by a set of attribute values. The corresponding attributes form the unique primary key. Records can reference other records, potentially in other relations, by this primary key [81, p. 8]. A visualization of two relations, one containing data about bookings made in an airline reservations system and the other storing information about passengers, is shown in Figure 2.1. A DBMS that uses the relational model is called relational database management system (RDBMS).

Most RDBMSs use the Structured Query Language (SQL) [41] to make it possible to store and retrieve data while working on the abstraction level of the relational model [81, p. 13]. Application programmers and other users do not have to be concerned about how operations are accomplished or about their performance, but only declare *what* they want to do. The job of the RDBMS is to convert these abstract operations to efficient instructions for the operating system.

Definition 2. *A relational database management system (RDBMS) is a DBMS that is based on the relational model. It often employs SQL as a language to query and manipulate data.*

SQL itself consists of various sublanguages. The data definition language (DDL) is used to create database objects such as tables, thereby defining the shape of the data in a database. Part of the DDL are statements like `CREATE` or `DROP TABLE`. With the



Id	PassengerId	FlightNr	BookingDate	SeatNr
2042	101145	HQE 897	2023-08-14	48A
2043	101109	HQE 1021	2023-08-15	NULL
2044	101007	HQE 897	2023-08-15	NULL
2045	101038	HQE 937	2023-08-15	1E

Id	GivenName	Surname	FrequentFlyer
101007	Julius	Springer	yes
101038	Norman	Lockyer	no
101109	Saifur	Rahman	no
101145	Yannis	Ioannidis	yes

Figure 2.1: Two relations in a database of a fictional airline reservations system

data manipulation language (DML), data is written to or deleted from a database. The `INSERT INTO` and `UPDATE` statements are a part of this sublanguage. Lastly, to retrieve data from a database, a data query language (DQL) is used to write `SELECT` queries.

2.1.2 Relational Algebra

Queries on relational databases can conceptually be expressed with the operations of the relational algebra. SQL is based on this algebra. The operations are defined over one or two input relations and produce a new relation as output. They are [81, pp. 47–57]:

- Selection $\sigma_{\varphi}(R)$: Selecting only the tuples of relation R that satisfy the predicate φ
- Projection $\Pi_{a_1,...,a_n}(R)$: Produces a new relation with the tuples from R, but only keeps attributes $a_1, ..., a_n$. Duplicates are removed
- Rename $\rho_{a/b}(R)$: Produces a relation with the same tuples as R, but with attribute a renamed to b
- Union $R \bigcup S$: From two relations R and S with the same attributes, produces a new relation that contains the tuples of both. Duplicates are removed
- Set difference $R - S$: From two relations R and S with the same attributes, produces a new relation that contains all tuples from R that are not in S
- Cartesian product $R \times S$: Produces a relation that concatenates every tuple from R with every tuple from S

Extensions of the relational algebra define operations for purposes like arithmetic calculations or aggregations.

From the aforementioned operations, join operations can be derived that combine data from two relations. For example, in Figure 2.1, to see which passenger, referred to by their name, booked which flight, the `Bookings` and `Passengers` relations have to be linked. This is done by combining tuples from the `Bookings` relation whose `PassengerId` is equal to the `Id` of a tuple in `Passengers`.

The *theta-join* combines tuples from two relations if they satisfy a given predicate θ. It is defined as $R \bowtie_\theta S = \sigma_\theta(R \times S)$. The *natural join* $R \bowtie S$ produces a relation in which tuples of R and S are combined if and only if the values of their shared attributes are equal.

In the previous example, to join the `Bookings` and `Passengers` relations, a theta-join can be used with $\theta := (\texttt{Bookings.PassengerId} = \texttt{Passengers.Id})$. This operation is also called *equi-join*, since the join predicate is an equality of two attributes.

2.1.3 Types of Workload

Database systems are mainly used in two different ways [75, p. 847]: Firstly, they facilitate business operations by managing data from business transactions. Individual business transactions are usually small and only retrieve or update a few tuples in the database, but a large number of them has to be processed during everyday operation [75, p. 847]. This type of usage is called online transaction processing (OLTP). For example, in the airline reservation system shown in Figure 2.1, when a customer books a new flight, a new tuple is added to the `Bookings` relation without touching any other tuple. The DBMS must be able to process such modifications fast and handle a large quantity of them simultaneously.

The second usage scenario is to use a DBMS as decision support system. That is, data is used to recognize trends, identify business problems, guide decision making, or draw conclusions [14, p. 689]. This type of workload is called online analytical processing (OLAP). For example, to achieve full utilization of their airplanes, an airline might want to establish for each flight the average number of passengers that did not show up, canceled very late, or could not board because the plain was already full. Such a query would touch a lot of tuples from the database and use more complex operations such as aggregations and groupings, but only read a small subset of the attributes of each tuple. Furthermore, those analytical queries are triggered only occasionally.

The different types of workload result in different data access patterns. During transactions in OLTP, a handful of records are accessed completely. For example, a complete

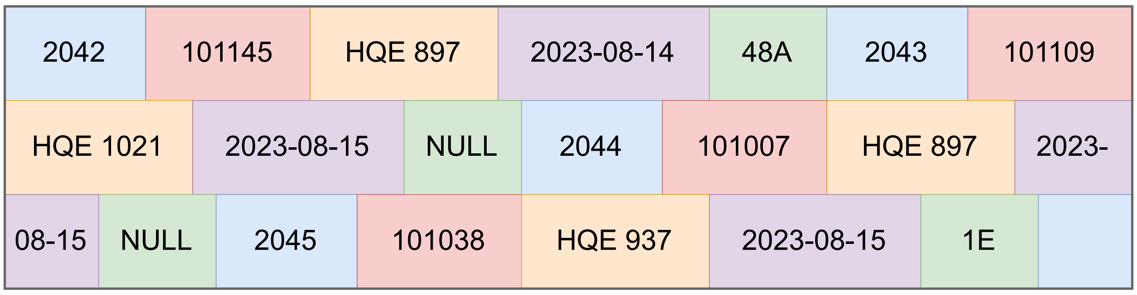


Figure 2.2: Row-oriented storage layout for records from the `Bookings` relation

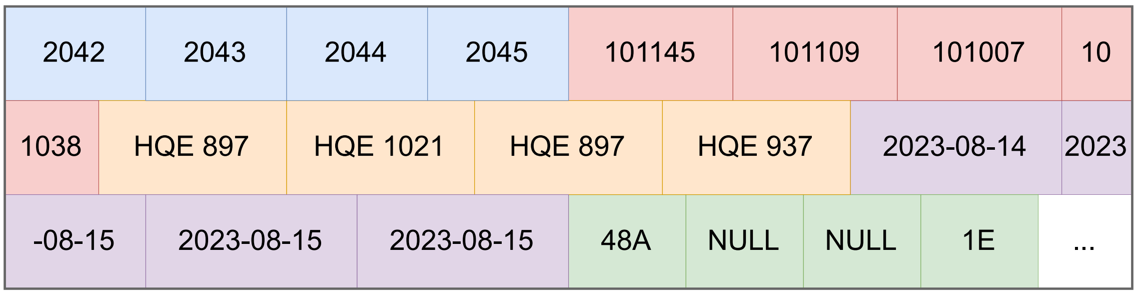


Figure 2.3: Column-oriented storage layout for records from the `Bookings` relation

record is inserted into the `Bookings` relation when a customer books a flight. To allow fast access to an entire record, all attributes that belong to one record are stored together. In Figure 2.2, this *row-oriented* storage model is shown for the `Bookings` relation from Figure 2.1.

OLAP queries, on the other hand, often access a lot of records, but only use a subset of their attributes. To attain sequential memory access, which is faster than random memory access when reading large data sets, data is often stored in a *column-oriented* fashion. The column-oriented storage model is visualized in Figure 2.3. It is also called the *decomposition storage model* [12] because records are broken down into their attributes. The CPU can read and aggregate the values of one column first and then jump to the next column of interest. This way of processing uses the cache more efficiently, allows for faster disk access, and can utilize CPU prefetching. Furthermore, data can be compressed better because columns of the same data type are stored sequentially. Therefore, the likelihood is higher that data is homogeneous or even consists of similar values for each tuple [1]. This way of storing data was pioneered by MonetDB [6; 37].

Table 2.1 summarizes the differences between OLTP and OLAP.

	OLTP	**OLAP**
Query frequency	Many queries executed simultaneously	Only few queries executed in a period of time
Data volume	Queries access or modify only a small number of tuples	Queries read many tuples, potentially from multiple sources
Response time	Queries should finish promptly	A high response time is acceptable
Operation types	Mostly simple operations such as `SELECT`, `INSERT`, but also `JOIN`, often in database transactions	Read-heavy, more complex queries with joins and aggregations
Storage model	Row-oriented	Column-oriented

Table 2.1: Comparison of OLTP and OLAP workloads

2.2 Query Processing in RDBMSs

When software applications want to read data from a database, they send SQL queries to an RDBMS and get back the query results. What happens behind the scenes is of no concern for the application and hidden behind the abstract DBMS interface.

In this section, though, we take a look at the inner workings of an RDBMS and see how a query gets processed. This knowledge is the foundation to understand how hybrid query execution, a new processing model, is implemented, which we explain in Chapter 4. We start with the high-level architecture of an RDBMS and continue to dissect the query processing flow. We finish off this section with an overview of how the processing of queries can be parallelized.

2.2.1 Components of an RDBMS

A DBMS broadly consists of three main subsystems: the query processor, the storage manager, and the transaction manager [81, p. 20].

The *query processor* receives an SQL query from the DBMS client, which is subsequently parsed and compiled into a query plan. A *query plan* describes the operations needed to calculate the desired result [68]. Finally, the query plan is evaluated and the result is returned to the client [2]. We cover the query processor in more detail in Section 2.2.2.

The *storage manager* is needed if the size of the processed data exceeds the available memory capacity or when data needs to be persisted. Using operating system primitives, its task is to efficiently store, organize, and retrieve data from physical storage such as

SSDs [81, pp. 18–19]. It makes database pages available to the query processor for reads and writes through the *buffer pool*, a part of the computer's memory [30]. The storage manager can also assume many other responsibilities and become closely interwoven with the query processor (see [32] for a summary).

The third subsystem is the *transaction manager*. Database transactions, i.e., multiple operations that form a single logical unit, must conserve four properties [34]:

- Atomicity: Either all operations of a transaction are executed completely or none at all
- Consistency: The complete execution of an transaction leaves the database in a consistent state, i.e., no database constraints are violated
- Isolation: Multiple concurrent transactions must not interfere with each other. The final state of the database must be equivalent to a state that would have been reached if all transactions were executed sequentially
- Durability: The result of completed transactions are permanently stored and are effective even in the face of system crashes or malfunction

The task of the transaction manager is to ensure that transactions are executed with these four *ACID* properties. In particular, the transaction manager handles the recovery from failures and concurrent access to databases [81, pp. 20–21].

2.2.2 How a Query Is Processed

In the following paragraphs, we describe a representative process of how a query is processed. Figure 2.4 shows a visualization of this process.

Parsing and Binding An RDBMS receives a query usually as SQL text, which can easily be processed by humans, but not by machines. Thus, the first step during query processing is to bring the query into a machine-processable form. The lexer deconstruct the query string into tokens from which the parser creates a parse tree. Afterwards, the parse tree is validated for syntactical and semantical correctness. This validation involves the binding of identifiers for tables, views, columns, etc. that are referenced in the query to the actual database objects (name resolution) [81, p. 689].

Planning The system subsequently transforms the parse tree into a *logical query plan*. It is a tree of relational algebra operations [81, p. 689; 68] that represents the data flow for the execution of the query [52].

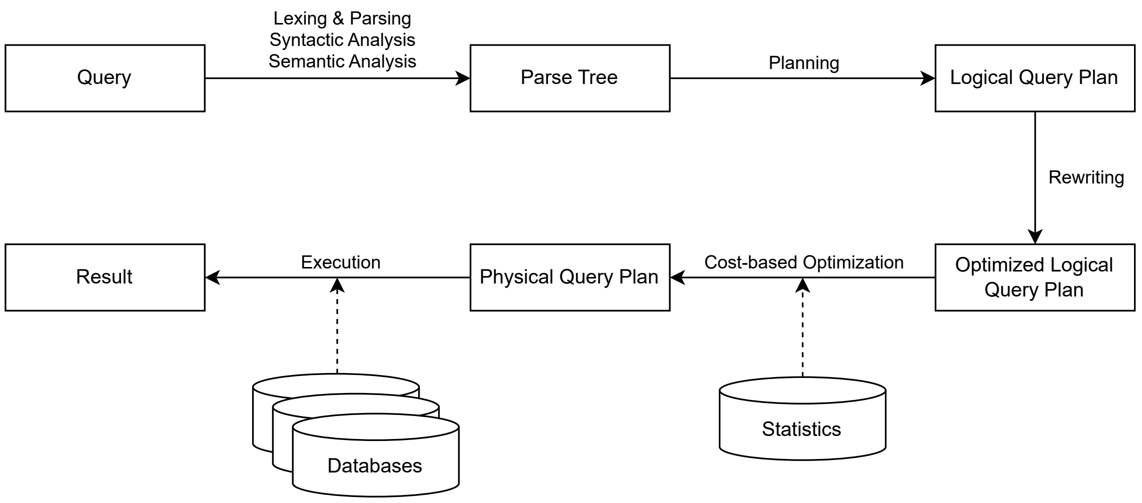


Figure 2.4: Steps that a DBMS takes to process a query

We take the query in Listing 2.1 as a running example. It queries the relations depicted in Figure 2.1.

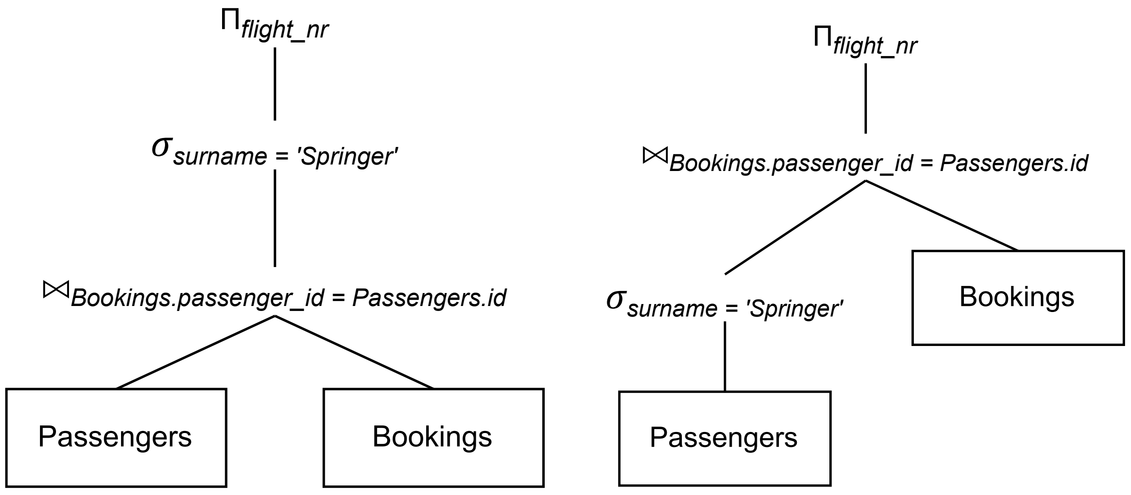
```
SELECT flight_nr
FROM Bookings
JOIN Passengers ON Bookings.passenger_id = Passengers.id
WHERE surname = 'Springer';
```

Listing 2.1: Example query to find the flight number of Mr. Springer

Figure 2.5a shows the logical query plan that is produced for this query: First, the two tables `Bookings` and `Passengers` are joined, the result relation is filtered to only include tuples where the value of the `surname` attribute equals “Springer”, and finally a projection is done to reduce the relation to its `flight_nr` column.

Rewriting In the following step, the logical query is rewritten into an optimized but semantically equivalent form following a set of rewrite rules. For example, subqueries are translated into joins [52], selections/filters are moved closer to the data source, and views are unfolded and substituted with their definition [67].

The query plan in Figure 2.5a is transformed in this step to the one shown in Figure 2.5b. A transformation known as filter pushdown is used: The `Passenger` relation is filtered first before it is joined with `Bookings`. This leads to better performance since fewer tuples have to be processed during the join, which is a comparatively expensive operation.


Π_{flight_nr}
$\sigma_{surname = 'Springer'}$
$\bowtie_{Bookings.passenger\_id = Passengers.id}$
Passengers
Bookings
Π_{flight_nr}
$\bowtie_{Bookings.passenger\_id = Passengers.id}$
$\sigma_{surname = 'Springer'}$
Bookings
Passengers

(a) Unoptimized logical query plan generated **(b)** Optimized query plan after filter pushdown

Figure 2.5: Logical query plans for the SQL query in Listing 2.1

Cost-Based Optimization Many relational algebra operations can be executed in different ways. For example, an equi-join on the join attribute a, which produces all combinations of tuples from two relations R and S were the values for a are equal, could, among others, be executed as [54]:

- Nested-loops join, where for every tuple $t_R \in R$ in an outer loop, every tuple $t_S \in S$ is checked in an inner loop for $\Pi_a(t_R) = \Pi_a(t_S)$.
- Sort-merge join, where R and S are in a first step sorted by a and then sequentially scanned, adding groups of matching tuples to the result relation.
- Hash join, where first a hash table is constructed for the values of a from relation R, which is then probed with the tuples of S.

However, there is no single "best" join algorithm, but the optimal choice depends on factors like the size of the relations or whether they are already sorted. The job of the query optimizer is to select a proper implementation for each of the relational algebra operation in the logical query plan. The low-level implementations are called *physical operators.*

The query optimizer produces a *physical query plan*, also known as *query execution plan.* This plan not only specifies the physical operators to run, but also which indexes should be utilized during execution, the order in which to join relations, and other low-level details [81, pp. 690–691]. The final physical query plan is selected on the basis of its minimum

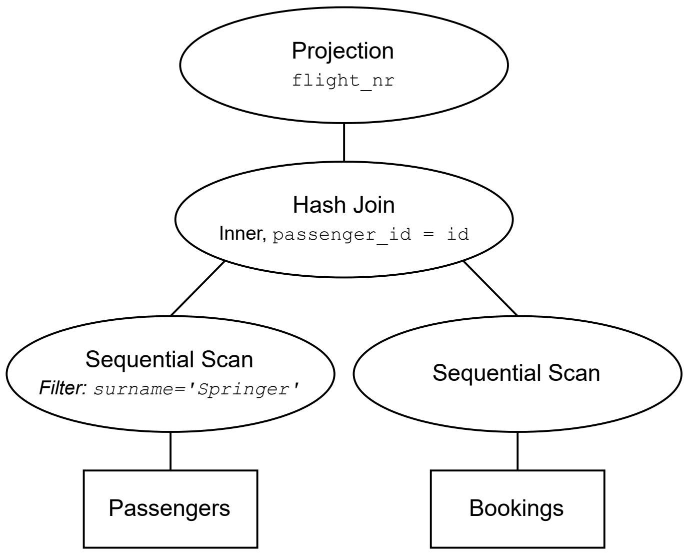


Figure 2.6: Physical query plan for the SQL query in Listing 2.1

estimated costs according to a cost model. It can consider, for example, the expected CPU utilization or I/O costs [67; 59]. The costs are mainly influenced by the cardinalities of intermediate results, which are estimated using statistics about the data [49]. The query optimizer explores alternatives by enumerating possible plans in the search space and comparing their estimated costs [67].

For the logical query plan shown in Figure 2.5b, the query optimizer might choose a hash join and produce the physical query plan in Figure 2.6.

Execution Finally, the physical query plan is evaluated by the query execution engine either by interpreting the physical operators in the execution plan iteratively (Volcano-style model [33]) or by further compiling the execution plan to machine code.

When interpreting a query, the parent operators call a function on their children to retrieve the next tuple. This can be seen in Listing 2.2. The many function calls, but also other factors like frequent metadata lookups, lead to significant overhead of query interpretation, especially when the CPU rather than I/O is the bottleneck [81, p. 733].

Query compilation [58] reduces the CPU cost of processing, because multiple functions can be combined by the compiler and metadata lookups can be done at compile time. However, additional time is needed to compile queries, hence this approach has a disadvantage for small queries [25]. Generated code can also impede the debuggability of the system [44].

The interpretation overhead in the Volcano-style model can be amortized by using *vectorized execution*. In this execution model, operators do not work on individual tuples,

```
std::unique_ptr<Tuple> next() {
  while (t = child.next()) {
    if (predicate(t)) {
      return t;
    }
  }
  return nullptr;
}
```

Listing 2.2: Pseudocode implementation of the `next` method of a filter operator

but blocks of thousands of tuples called *vectors* [7]. Not only does this approach reduce the interpretation overhead, for example by reducing the number of function calls or branches, but it also enables CPUs to exploit opportunities for parallelism, loop unrolling and other optimizations, and can better leverage SIMD instructions [46]. With vectorized execution, the performance for OLAP workloads is on par compiled query execution [44].

Conceptually, the evaluation of an query operator tree starts at its leaf nodes by reading data from sources. The tree is traversed upwards as child operators move the result tuples they produce to their parent operators until the final result is materialized at the root of the tree [81, pp. 724–725].

One variant for implementing query execution is to *materialize* the result of each operator. That is, the complete result relation is constructed, potentially stored on disk, and then read by the next operator in the chain [81, pp. 724–725]. Alternatively, an evaluation method called *pipelining* can be used to reduce the amount of data that needs to be stored during processing. In this model, an operator can process tuples from its child operators and pass already computed results on to its own parent. It does not have to wait for all results to be computed. Therefore, intermediate results do not have to be materialized[1], which can lead to more efficient execution [81, pp. 725–726].

Two different implementations can be distinguished: pull-based and push-based pipelining. In the *pull-based* (or *demand-driven*) model, an operator gets a request from its parent to compute the next tuple, fetches itself the next tuples from its children, computes the result tuple, and returns it to its parent. Listing 2.2 demonstrates this with pseudocode for the implementation of a `next` function for a filter operator that emits tuples that satisfy a given predicate. This function would be called by the parent's `next` function.

[1]There are some operators like joins or set difference that require the materialization of intermediate results.

In the *push-based* (or *producer-driven*) model, each operator computes tuples eagerly when it receives input from its child operators and puts them into its result buffer. Parent operators continue processing when new tuples are in that buffer [81, pp. 726–727; 66].

2.2.3 Parallel Query Execution

Modern hardware comes with many CPU cores, which DBMSs aim to utilize through parallel query execution. This can happen through interquery parallelism, i.e., executing multiple queries at the same time, or intraquery parallelism where operators of the same query are executed in parallel [31, pp. 127–128; 81, p. 1039].

Types of Intraquery Parallelism A first opportunity for intraquery parallelism can be realized by decomposing physical operators to work on distinct sets of tuples in parallel [81, p. 1040]. For example, there are parallel implementations of various sort and join algorithms [31]. Hence, for queries on large data sets with a modest number of operations, multiple cores can work on the same operation—or even a sequence of operations—in parallel and do not sit idle.

Aside from that, different physical operators can be evaluated at the same time, either because they do not depend on each other or by using pipelining [81, pp. 1052–1055]. Pipelining we have already described briefly in Section 2.2.2.

An execution plan can be represented as a tree of physical operators. A pipeline is a sequence of physical operators where the output of one operator is the input to the next. The pipeline ends in a pipeline breaker. These are operators that need to process all tuples before forwarding them to the next pipeline. Hence, one pipeline has to wait for another pipeline to finish, and the flow is interrupted. Examples for pipeline breakers are sorts, aggregations, hash joins since a hash table has to be constructed for one of the joined relations, and the final result materialization.

In a pipeline, child and parent operators can run on different threads. Parallelization can be achieved because a parent operator can start to process tuples that it receives from its children even before the latter finished completely [81, p. 1053]. Additionally, independent operators can run in parallel on different threads.

Figure 2.7 indicates the pipelines for the execution plan in Figure 2.6. The pipeline breaker for the left (red) pipeline is the join of the `Passenger` and `Bookings` relations. The pipeline breaker for the right (blue) pipeline is the final result materialization. The two sequential scans can be executed in parallel on different threads because they are independent from each other.

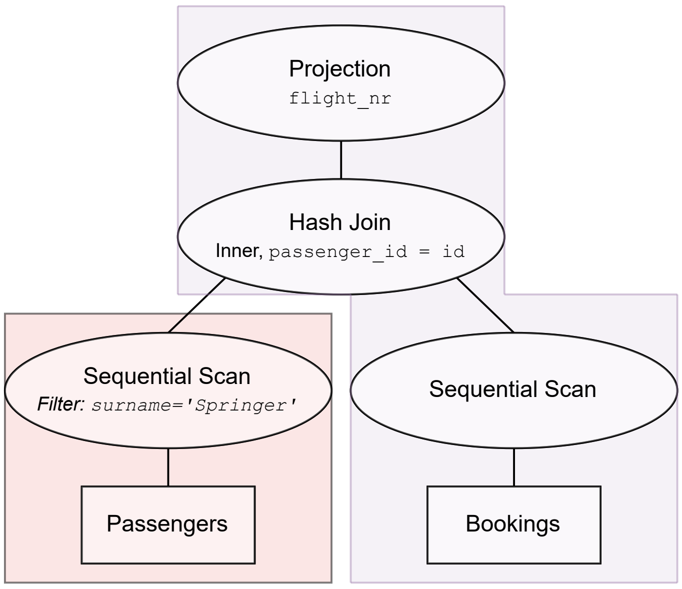


Figure 2.7: Pipelines in the query plan from Figure 2.6

To achieve an even higher degree of parallelism, it is possible to execute many instances of a single pipeline in parallel on distinct subsets of tuples of the input relations. For example, when the `Passenger` relation contains one million tuples, the table scan and filtering can be distributed across 10 threads, each processing a data segment of 100,000 tuples.

Morsel-Driven Parallelism *Morsel-driven parallelism* is a kind of pipelined parallelism that achieves a better division of work between available CPU cores and is suitable for NUMA memory designs [48]. The input data to a pipeline, coming from child operators or data stores like tables, is split into small segments called *morsels*. A dispatcher schedules pipeline jobs on worker threads to operate on a morsel until they reach the next pipeline breaker. At this point, the intermediate results are materialized.

The scheduling mechanism takes into consideration the number of available cores. Since a thread processes one pipeline job with the associated morsel at a time and then requests the next unit of work from the dispatcher, this transition provides an opportunity to dynamically adjust the number of threads used for processing. Hence, the number of threads can flexibly be adjusted to changes in the degree of parallelism, even while a query is executed.

To implement morsel-driven parallelism, all physical operators must support it. One DBMS which uses this kind of parallelism is DuckDB, which we describe in the next section.

2.3 DuckDB

DuckDB [73] is a RDBMS that is optimized for analytical (OLAP) workloads. Thanks to its easy setup and user friendliness, but also good performance and scalability [69, 09:15–12:51, 14:51–15:49], it has become a popular tool among data scientists [71, 21:26–23:19].

DuckDB runs embedded in a host application. This arrangement enables fast data transfer (or even data sharing) between the host application and the DBMS, and simplifies the installation for users since no setup of a database server is required [18]. Instead, DuckDB is linked into the host application, which can then communicate with the DBMS using its C/C++ API. Bindings for other programming languages are available, too [17]. Especially important to data scientists, DuckDB can conveniently be used from Python and R.

DuckDB, by and large, follows the separation of components described in Section 2.2.1, and queries traverse the stages described in Section 2.2.2.

The parser for DuckDB is based on the one for PostgreSQL[2] and produces a parse tree composed of DuckDB-specific nodes. The storage format is column-oriented to better serve analytical workloads. Furthermore, DuckDB uses an interpreted, vectorized query execution engine together with push-based pipelining and morsel-driven parallelism.

New functionalities can be added to DuckDB and existing can be modified by loading extensions. Extensions can change the parsing behavior, create new database operators, append new optimizer rules, or add new storage locations. They are a powerful way to make DuckDB itself more powerful and the cornerstone to implement hybrid query execution (see Chapter 4).

2.4 Software Testing

In today's digital age, software systems are prevalent and deeply integrated in domains ranging from health care to transportation to communication. Due to their pervasive and sometimes critical role, these systems are expected to be of high quality. However, the view of what quality encompasses exactly can vary. For users, it usually means that "a product meets user needs and expectations" [56, p. 520]. For software engineers, quality might involve building a product with good internal properties [56, p. 520] such as low coupling, high cohesion, and other indicators of manageable complexity [22, Part III].

[2] `https://github.com/pganalyze/libpg_query`

The ISO norm 25010 [39] describes eight categories of quality characteristics for software systems, addressing both the users' as well as the internal view. They encompass the aspects of functional suitability, performance/efficiency, compatibility, usability, reliability, security, maintainability, and portability. If they are important to stakeholders, the quality characteristics transition to explicit requirements. Through software testing, development teams can assess if a system meets its requirements.

Software testing is the process of systematically executing a software system under controlled conditions to evaluate and assess specific quality aspects [42, def. 3.4272]. Various types of tests are employed to target the characteristics described in ISO 25010. For instance, performance testing assesses whether the software can work within constraints for time and resource consumption [42, def. 3.2873], security testing evaluates how well security objectives are met [42, def. 3.3661], and usability testing focuses on the system's fitness for use [42, def. 3.4458].

Definition 3. *Software testing is the process of executing a system under specified conditions to evaluate if quality requirements are met*

2.4.1 Functional Software Testing

One facet of the quality characteristic termed functional suitability is *functional correctness*, i.e., how accurately the system provides the correct result or does the correct action [39]. The correct or expected behavior for a given input is outlined in functional requirements.

Software errors occur when there is a difference between the observed and expected behavior of the software, or when it transits into an erroneous state [42, def. 3.1441]. They can arise from a misunderstanding of requirements or failing to translate them into an appropriate specification [29]. For example, a requirement can be inappropriate for the use case it tries to solve because no actual users were involved in the analysis process [88], or the requirements might be incomplete [23]. But oftentimes, errors are introduced during the implementation phase—after the specification has already been written. When this happens and the software does not meet the specification, it has one or more defects, also called bugs [42, def. 3.1081; 29].

Functional software testing verifies that the software meets this specification and can thus help to ensure that no software defects are present. Complementary approaches to achieve the same goal include code reviews, program proving [5, pp. 16–17], and static code analysis ("linting") [82, pp. 257–258]. This kind of functional testing is called *static*, whereas *dynamic* testing involves executing the program with a given combination of inputs. In this work, we are only concerned with dynamic testing.

2.4.2 Manual and Automated Testing

In the field of functional software testing, again different approaches can be distinguished. A first division can be made between manual and automated testing. During manual testing, a human tester performs actions on the software that were previously defined during the design of test cases and looks out for suspicious behavior [5, p. 22]. However, this approach is costly and takes a long time. It may also be less reliable than the alternative [55, p. 128]: automated testing. This approach utilizes software tools to execute tests and compare actual outcomes with the expected ones [5, pp. 22–23]. It makes it possible to run many tests frequently, reliably, and at low costs. A major advantage of this approach is that developers can start the tests and get the results back shortly after, which makes the feedback loop between implementation and testing very short. As discussed, for example, in [76; 43], manual testing can still offer advantages over automated testing such as lower effort to develop tests, higher flexibility, and that different kinds of bugs are found. In any case, automated testing will be the sole object of our interest in this work.

2.4.3 Black-Box and White-Box Testing

Another distinction can be made between black-box and white-box testing. These categorize different strategies to select test data, i.e., the inputs to the software under test for a particular test case.

To be sure that all defects were found by tests, all possible inputs and input combinations need to be checked. Otherwise, there is no way to be sure that the software does not contain code that produces the wrong result for some specific input [55, pp. 8–9]. However, this strategy known as exhaustive testing is infeasible even for small programs. Therefore, in practice, representative inputs must be chosen that give enough confidence that the software works correctly for all inputs [5, pp. 12–13].

Black-box testing describes one category of heuristics to select representative input combinations. Here, test data is selected without regard for the software's internal structure or source code. Instead, only the specification is used for this purpose [5, p. 14].

An example for a simple black-box heuristic is equivalence partitioning. With this technique, input values which, according to the specification, are processed in the same way, are grouped together. For each group, one test case is created with one representative value as input [5, p. 27].

For white-box testing, on the other hand, testers have knowledge of the software's internals. Test cases are designed by analyzing a program's logic and deriving test data that causes this logic to be executed. For this reason, this approach is also called structural testing. However, the expected behavior against which to compare the system's actual behavior is given by the specification, not the source code [5, p. 86]. The aim is to derive test cases that examine most, if not all, of the code, as defined by a code coverage criterion (see Section 2.4.4). The ideal, though usually impractical state of a white-box test suite is to cover all code paths through an application [55, p. 10].

Myers, Sandler, and Badgett recommend to use the black-box approach first to design the majority of test cases and then use the white-box approach to find test scenarios that are not yet covered [55, p. 42].

2.4.4 Code Coverage

Code coverage is a measure of the extent to which the source code of a software system is executed by tests. Suppose that a program consists of two independent functions f_1 and f_2 and that there exists one test case that calls f_1. The *function coverage* is then 50% because one of the two functions was executed.

Looking at the source code as a set of functions gives a very macroscopic view of how well the program is tested. But if f_1 itself is made up of 100 lines of code, but the test case only ever executes 20 of them and then returns early, it is questionable whether the test even covers f_1 adequately.

For this reason, various coverage criteria were defined at a more fine-grained level. Besides function coverage, some other basic coverage criteria are listed below.

- Statement coverage: The percentage of statements, mostly corresponding to one line, that were executed at least once during a test run [5, pp. 86–87].
- Branch coverage: Control structures like `if`, `while` or `switch` create branches in the control-flow graph of the application. That is, based on a condition, the execution jumps either to one or another statement. Branch coverage represents the percentage of edges that were taken in the control flow graph during a test run [5, p. 99]. For example, a branch coverage of 100% can only be achieved when the test suite executes both the `true` and `false` case for every conditional statement.
- Condition coverage: While branch coverage only looks at the branches that were taken, condition coverage additional checks that every possible boolean combination of conditional statements were tested [60]. For example, if the source code contains

`if (a && b)`, all four combinations of `a` and `b` each being `true` or `false` need to be tested to achieve a condition coverage of 100%.

- Path coverage: This coverage criterion looks at the possible execution paths from a program's entry point until it terminates and denotes the percentage of execution paths taken during a test run [5, p. 113]. However, this criterion is more of theoretical nature as no tool exists to measure the path coverage in complex software programs [5, p. 114].

2.4.5 Testing Levels

Depending on the granularity of the test, different testing levels can be distinguished. Unit tests exercise small components of the system [55, p. 85]. The "units of work" are usually (public-facing) functions. The tests run in isolation from other components and outside dependencies, which allows them to run fast and deterministically. However, defects that occur during the interaction of several components can still go unnoticed. To find these is the objective of tests at higher levels.

Unit tests are usually written by software developers and can be created with both the black-box and white-box approach.

At the opposite end of the testing spectrum, end-to-end tests (also called system tests) validate the system as a whole. These tests mirror real-world scenarios more closely and use real dependencies.

2.5 Testing of Database Systems

The academic literature on the functional testing of RDBMSs primarily revolves around automated black-box testing of the whole system. In this section, we give a succinct overview of research on this topic.

In database systems, it is difficult to test the individual components separately since they are all intertwined and interact in complex ways with each other [21; 35]. Nevertheless, there is some work in this area. For example, the testing strategy described in [13] targets directly the transaction manager, the one in [87] the query optimizer. However, these tests are never unit tests in a classical sense, but use the SQL interface to interact with the DBMS and do therefore still exercise the whole system. One reason for this is that the internal interface of DBMS components can be quite complex, which makes the generation of test input data challenging[3]. Moreover, using SQL, a standardized language supported by many RDBMSs, makes it possible to test many different database systems with a single tool.

With SQL, there are infinitely many possible queries that a DBMS could process. Manually created test suites can only cover a small, though mostly representative subset of this wide input space. Therefore, an active area of research is random query generation, also called fuzzing, to find new errors. These randomly generated test cases are end-to-end tests: they send a query to the DBMS and validate the result.

For random query generation, three problems have to be solved:

- The data generation problem: Generating a database against which to run queries. The values in the database should follow a representative distribution [8; 74]. An alternative to using synthetic data is to rerun real-world queries [90; 80]
- The query generation problem: Generating a query that is syntactically valid but also semantically for a given database [91; 50]
- The test oracle problem: Determining whether the result returned by the DBMS is correct [77; 79; 78]. This task is simple when tests are manually created since humans can determine the expected result from the specification. But to automatically find the correct result for a given query, one, in fact, would have to implement a DBMS. This is why one common test oracle is indeed to use another DBMS as a reference.

To maximize the value of each test query, internal validation can be applied to find bugs in the system [47].

Fuzzing is one pillar of the testing strategy for DuckDB, too. Additionally, the correctness of DuckDB is checked using a large number of manually written tests as well as internal validation during the test runs. In the next chapter, we describe the DuckDB test suite in more detail.

[3]Internally, DBMS vendors might still write tests against internal interfaces, but do not publish papers on this topic.

3

The DuckDB Test Suite

For DuckDB, an extensive test suite is in place. It is used to test the correctness and performance of the DBMS as well as the integration into various programming languages and the interoperability with other DBMSs. Additional checks were established to ensure good code quality. Figure 3.1 shows an overview of the test suite, which we describe in more detail in the following sections. We outline the test suite for version 0.8.1 of DuckDB.

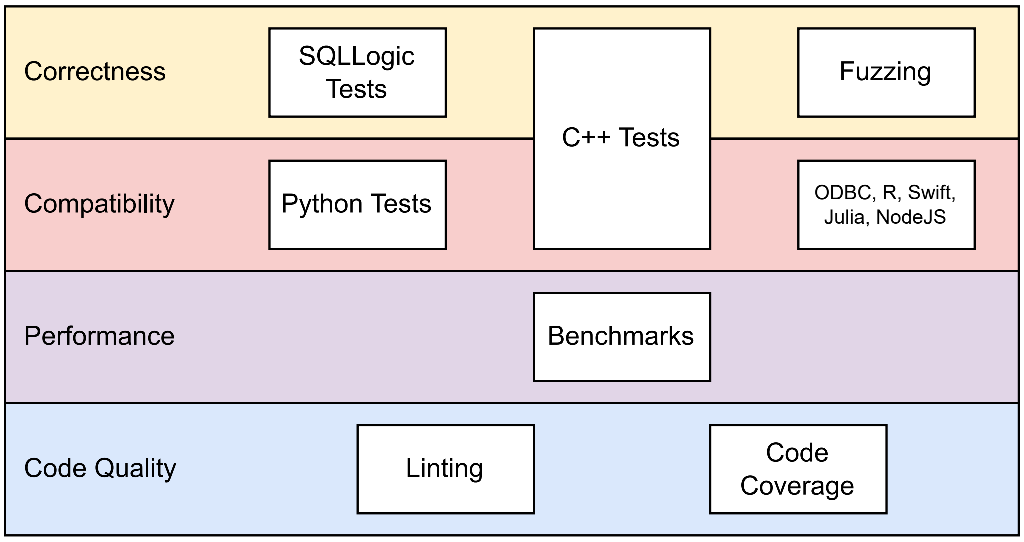


Figure 3.1: Overview of tests contained in the DuckDB test suite

3.1 Correctness Testing

To verify the correctness of the system, the DuckDB developers use a mix of C++- and SQL-based unit tests as well as several fuzzing tools.

```
# SQL statement that is expected to succeed.
# A new database table is created in the process.
statement ok
CREATE TABLE integers (i INTEGER)

# The database table is filled to prepare
# for the following query.
statement ok
INSERT INTO integers VALUES (1), (2)

# The query is expected to return the two values
# inserted previously.
query I
SELECT * FROM integers
----
1
2
```

Listing 3.1: Example of an SQLLogic test script

3.1.1 SQLLogic Tests

The SQL-based tests are written as scripts in the *SQLLogicTest* format. The SQLLogic test framework[1] was initially designed and implemented for SQLite, a popular open-source, embeddable OLTP database system. It consists of a set of test scripts in combination with a program, the test runner, that can execute these scripts. Each test script contains a list of SQL statements that are executed sequentially. Additionally, it is possible to specify the result that is expected from running a query. Listing 3.1 shows an example of an SQLLogic test script.

The developers of DuckDB follow a test-driven development style, which means that they write tests for the feature they want to implement before the actual code [70, 29:39–30:36]. This has led to the creation of a rich set of test scripts for all the SQL features implemented in DuckDB. Furthermore, when a new bug is discovered, a regression test is created by first reproducing the bug in an SQL test if possible.

Besides checking normal query results, SQLLogic tests are also used to validate the transaction semantics. Transactions, which can consist of several SQL statements, are expected to be executed as one logical operation and isolated from other simultaneous

[1]`https://www.sqlite.org/sqllogictest`

```
statement ok
CREATE TABLE integers (i INTEGER)

statement ok con1
BEGIN TRANSACTION

statement ok con1
INSERT INTO integers VALUES (1), (2)

# On the second connection, the inserted values should
# not yet be visible for the second connection
query I con2
SELECT * FROM integers
----

statement ok con1
COMMIT

# After committing the transaction, the
# inserted values should become visible
query I con2
SELECT * FROM integers
----
1
2
```

Listing 3.2: Testing the ACIDity of DuckDB transactions in an SQLLogicTest script. `con1` and `con2` are the identifiers for two different connections to the same database.

transactions. In Listing 3.2, an SQLLogic test script is shown that verifies that the values inserted into a table during a transaction are only made visible after this transaction is committed. The test relies on multiple DuckDB connections that access the same underlying database.

`EXPLAIN` queries in SQL produce a textual representation of the query plan for a given inner query, but do not actually run that query. DuckDB uses `EXPLAIN` queries in SQLLogic test scripts to check that the query optimizer follows the right rules. In the test scripts, the textual query plan is matched against a regular expression, which is useful to see if specific operators are present in the plan.

There are SQLLogic test scripts in the test suite that execute the well-known TPC-H[2] and TPC-DS[3] benchmarks for OLAP database systems. The benchmarks contain queries related to a specific business scenario. The queries work on a large volume of data and some are of high complexity, which makes them a handy tool to assess the overall condition of the DBMS.

To get the most out of each test, some internal verification steps are taken when DuckDB runs a test script [70, 19::04–26:05]. For example, when the query plan is created or modified, every operator in the new plan is serialized and then deserialized again. By ensuring that the result of this procedure is equivalent to the original statement, the tests can show that the serialization functionality works correctly [72]. A similar check is done to verify that copying a statement does not change its value. Another technique used in the test suite is to run the query with no optimizations, without operator caching, or as prepared statement. The invariant that must hold is that all these changes must not change the query result. That is, for instance, the optimized and unoptimized versions of a query must produce the same output.

Version 0.8.1 of DuckDB was tested with 2623 SQLLogic test scripts [4].

3.1.2 C++ Tests

To have more control over how tests are executed, some of DuckDB's tests are written in C++. One example is a test that kills the process running DuckDB to later verify that the database can be recovered. Other C++ tests are used, among other things, to execute multiple queries in parallel, to make sure that databases are stored correctly, or to test internal utility functions. In version 0.8.1, the test suite contained 40 of those tests.

3.1.3 Fuzzing

The SQLLogic tests make sure that typical workloads are executed correctly, but they only cover a small and fixed subset of possible inputs. Hence, DuckDB uses random query generation, or fuzzing, as an additional testing tool. Because fuzzers tend to find bugs in an unpredictable manner, they are continuously checking DuckDB in the background. Three fuzzers are hosted via Google OSS-Fuzz[5]. These are AFL++, libFuzzer and Honggfuzz, which are aimed at detecting memory corruption vulnerabilities in a variety of programs.

[2]`https://www.tpc.org/tpch/`
[3]`https://www.tpc.org/tpcds/`
[4]`https://github.com/duckdb/duckdb/tree/6536a772329002b05decbfc0a9d3f606e0ec7f55/test/`
[5]`https://github.com/google/oss-fuzz`

Two fuzzers that are specifically targeted DBMSs are SQLLancer[6] and SQLsmith[7]. DuckDB provides means to start these two fuzzers, but does not run them continuously at the time of writing.

3.2 Compatibility Testing

Users can interact with DuckDB through a variety of interfaces. First and foremost, there is the C/C++-API. The test suite includes C++ tests that make sure that basic functionalities such as querying data, executing utility functions, changing the configuration, or creating user-defined functions work correctly via this interface. Similar tests are done for the other programming languages for which DuckDB provides bindings: Julia, NodeJS, Python, R, and Swift.

For Python, extra care has to be taken because DuckDB is deeply integrated with the rest of this ecosystem. For example, DuckDB relations can be converted into Pandas[8] data frames and vice versa. Furthermore, DuckDB is compatible with the Python Database API[9]. There are tests to test that DuckDB is well integrated with these other Python tools.

Applications can also connect to DuckDB via ODBC, a standardized API for DBMSs, JDBC, which resembles ODBC but for the Java programming language, and ADBC, a database API on top of Apache Arrow[10]. DuckDB has tests in place to verify that the basic connection setups work correctly. Because these interfaces are also used by other DBMSs and database tools, it makes it possible to reuse many of their tests. For example, DuckDB runs some of the tests from the nanodbc library[11].

DuckDB provides compatibility with the SQLite DBMS, so that DuckDB can be used as a drop-in replacement. There are tests that verify that clients can access DuckDB via this interface, too.

3.3 Performance Testing

DuckDB regularly runs tests to quickly discover any regressions in query processing performance. During a performance test, several benchmarks are run multiple times and

[6] `https://github.com/sqlancer/sqlancer`
[7] `https://github.com/anse1/sqlsmith`
[8] `https://pandas.pydata.org/`
[9] `https://peps.python.org/pep-0249/`
[10] `https://arrow.apache.org/`
[11] `https://nanodbc.github.io/nanodbc/`

the average execution time is calculated. If the difference to the previous run is greater than 10%, a regression is reported.

DuckDB uses a rich set of benchmarks, from small microbenchmarks that only execute a few small commands to standardized ones that are also used by other DBMS vendors. The latter include:

- TPC-H[2] and TPC-DS[3], industry-standard benchmarks for performance that were developed by the Transaction Processing Performance Council (TPC)
- the H2O.ai benchmark [20] which focuses on the performance of `GROUP BY` and `JOIN` statements
- the Join Order Benchmark [49] which uses data from the IMDb

Additionally, DuckDB supports the following benchmarks, which, however, are not run continuously as part of the regression tests:

- ClickBench[12], a benchmark developed by ClickHouse (another DBMS for OLAP processing) which is based on real data from a web analytics platforms
- the Social Network Benchmark of the Linked Data Benchmark Council (LDBC) [85], which contains OLAP queries on a social graph
- the Train Benchmark [84] for queries on railway-related graphs

3.4 Code Quality Measurements

Automatic static testing is performed for every code change in DuckDB. First, the source code is automatically checked to meet the coding standards. Second, the code coverage is collected when running the suite of correctness tests. A report is created about files for which the coverage is too low.

[12]https://github.com/ClickHouse/ClickBench

4

Hybrid Query Execution

Hybrid query execution is a novel query processing model for distributed databases that aims to make efficient use of the resources available on a DBMS client and reduce the amount of data that is exchanged over the network. It promises to lead to better performance for queries that involve local data as well as common data science workflows.

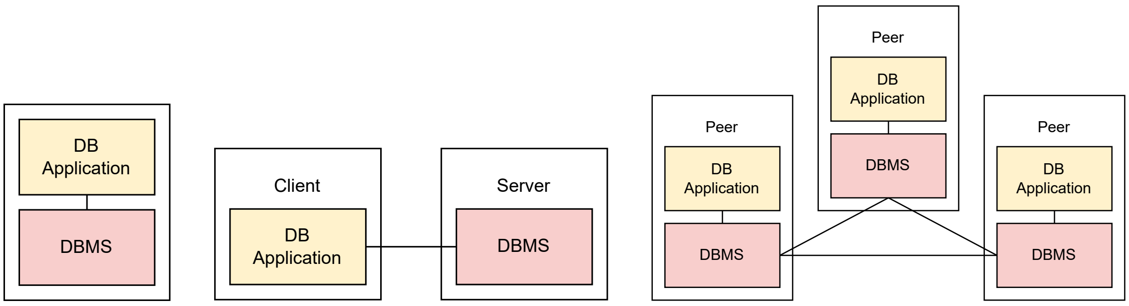
By embedding a full DBMS into the client, the processing of a single query can be done partly on the client and partly on the server (intraquery distribution), depending on where the processing is most efficient. This is often determined by where the processed data is located: If, for example, all data resides remotely, the server can do all the processing, as in a traditional client-server architecture. But when both local and remote data is involved in the same query, both nodes can participate and exchange only small, processed result relations rather than raw data.

In this chapter, we start with a comparison of two architectures that resemble the two extreme ends of hybrid execution: a typical client-server architecture where the bulk of the processing happens on the server, and the single-node architecture where a single node does all the processing locally. After giving a motivating example to demonstrate the potential benefits of the hybrid execution model, we go on to first give a high-level and then a detailed overview of it.

4.1 Advantages and Disadvantages of Existing DBMS Architectures

There is no one-size-fits-all database system that can serve all needs of the very different kinds of database users [83]. This argument can be extended to the architectural level of DBMSs from the viewpoint of *where* query processing happens: Different architectural styles come with different benefits and drawbacks which might fit one database application but not another. In the following section, we mainly introduce and compare two different styles of architectures, the single-node and the client-server architecture.

4.1.1 Distribution of Query Processing



(a) Single-node **(b)** Client-server **(c)** Peer-to-peer

Figure 4.1: Architectural styles from the viewpoint of distributing database application and DBMS

When looking on the architectures of DBMSs from the viewpoint of *where* the processing happens, three architectural styles can be distinguished, as shown in Figure 4.1. We will consider two layers: the DBMS and the database applications that communicate with it. We are primarily interested in the components included in the architecture and the connections between them, giving each architectural style its unique structure.

The first is what we call the *single-node* architectural style (Figure 4.1a). Here, the database application and the DBMS run on the same node and communicate through a local connection. All data is stored in a single location.

Second, there is the *client-server* architecture (Figure 4.1b). In this architecture style, the database application and the DBMS run on different nodes called *client* and *server*, respectively. They communicate via a network. The data is stored only on the server. The one logical server could also be comprised of many physical servers between which data and processing are distributed. But from our architectural viewpoint, we consider the server as a single node.

For completeness, we also want to mention the *peer-to-peer* (P2P) architectural style (Figure 4.1c), in which both data and processing are distributed among *equitable* peers. That is, each peer can request other peers, but can also be requested, to participate in the processing of queries and transfer of data. In the following section, however, we will only focus on a comparison between the single-node and client-server architectural styles.

4.1.2 Comparison of Client-Server and Single-Node Architectures

In a typical client-server architecture for a relational database management system (RDBMS), the client is *thin*. This means that even though the client can cache data or perform other small tasks such as consistency checking, the bulk of the processing happens on the server side. The client, for the most part, only sends SQL queries to the server and receives back the result [62, pp. 20–21], but only after having to wait for the network round trip to finish.

But oftentimes, the clients themselves possess plentiful compute resources: Modern desktop computers, laptops, and even smartphones have powerful processors. Since the clients do not participate in the query processing, these resources go unused.

A single-node architecture gives a different picture. Here, the "client"[1] performs the entire processing and can thus utilize its resources to the fullest. There is also no communication over wide area networks happening between the database application and the DBMS that could cause additional delays, assuming that all data is already available locally. Accordingly, for the processing of small and medium-sized data sets for which the local compute resources suffice, DBMSs which are built for single-node architectures can enable a faster user-perceived query execution. This holds especially true in regions with no high-speed internet connection.

Single-node DBMSs have another advantage from a monetary perspective. The computers on which they run are already paid for or would have been bought anyway since they are used for many other tasks aside from hosting a DBMS. By contrast, in a client-server architecture, the DBMS has to be hosted on some server. This could be a dedicated server, possibly in a data center, but it has to be bought, installed, and maintained. This burden can be lifted thanks to cloud computing, where servers are fully managed by cloud providers. But this does not come for free either: Customers of cloud providers pay for storage, bandwidth and compute in fine-grained units (usage-based pricing) [26, pp. 14–15]. Those costs are not incurred in a single-node architecture. In summary, installing the DBMS on existing local computers can reduce the capital and the operational expenditures.

Cloud computing comes with the additional disadvantage that data is stored in a third-party's data center [81, p. 995], while single-node systems come with privacy by design.

[1]The term "client" is not completely accurate since in a single-node architecture there is only this single node, the host of the DBMS. Nonetheless, we will call this node the "client" to emphasize that, for a DBMS user, it plays the same role as a client in a client-server architecture.

	Local-only	**Client-server**
Communication overhead	No network communication for query processing itself	Network communication between client and server to exchange queries and results
Access to remote data	Remote data has to be downloaded first	Data is stored in cloud data center or can be downloaded via high-bandwidth connection
Capital expenditure	Hardware is already paid for	Might require upfront investment in server hardware
Operational expenditure	Low	Server costs in data center or cloud
Privacy	Data is stored on trusted client	When using cloud computing, data is stored at third-party side
Resources	Bounded CPU and memory	Scalable CPU and memory; elasticity
# Users	Typically single-user system	Multi-user system

Table 4.1: Comparison of the single-node and typical client-server DBMS architecture

However, running all queries locally, as in a single-node architecture, comes with disadvantages, too. For example, the data that is analyzed is often stored at a remote location such as cloud storage. When users want to analyze the data on a local computer, the data has to be downloaded first. This process could take quite a long time, depending on the available bandwidth, and can occasion costs.

Moreover, single-node systems can only use the compute resources that are available locally, while running a system in a data center offers scalability and elasticity, i.e., the ability to quickly adjust the amount of resources in use [81, p. 992]. However, the need for the availability of almost unbounded resources is questioned by the claim “Big Data Is Dead” (see Chapter 1).

Database servers are designed and provisioned to support many users [81, p. 962]. Single-node systems, on the other hand, are typically designed for a single user. This means that it is more difficult to work on the same data.

Table 4.1 summarizes the comparison between single-node and client-server DBMSs based on the dimensions mentioned before. In the following sections, we introduce *hybrid query execution*, an architecture that can be seen as a middle ground between the single-node and client-server models, offering the benefits of a cloud-based DBMS while utilizing the compute resources that are available locally.

4.2 A Motivating Example

The idea of *hybrid query execution*, or *hybrid execution* for short, is to move the compute close to the data, thereby offering better performance while better utilizing locally available resources.

Better performance is hoped to be achieved by reducing the amount of data that is transferred via a network. Say that a data analyst at a big airline company runs a query against a relation that contains all the ticket sales of the past week. The database is located in the airline's cloud-based data warehouse. Since the holiday season is just around the corner, the `Bookings` relation contains a lot of tuples. But the analyst is only interested in the few bookings made by passengers known for skiplagging, i.e., ending the trip already on a stopover and not showing up to the flight to their booked end destination. They have previously downloaded a CSV file from their airline alliance's web portal containing identification features of such customers. Here, too, the analyst is only interested in a small subset of the data, specifically the information about customers that booked a flight within the last year. After applying this filter, the CSV file is way smaller than the `Bookings` relation.

If the analyst uses a DBMS that supports hybrid execution, the query processing might take place as follows (depicted in Figure 4.2): First, the local CSV file will be read into a temporary relation and filtered using the DBMS running on the client. The result relation will then be uploaded to the server. Simultaneously, the `Bookings` relation will be filtered for bookings from the past week on the server, which runs close to the data warehouse. When the skiplagging data from the client arrives, it is joined with the filtered `Bookings` relation. Finally, only the tuples of the small result relation will be transferred back to the client.

Thanks to hybrid execution, neither was it necessary to upload the whole CSV file to the server nor did the client have to fetch a large `Bookings` relation. Processing was done as much as possible close to the data and the amount of data transferred via the network was thereby reduced.

Now imagine that the analyst wants to further filter the results, sort them, or apply projections on them. Since the result relation is now locally available, these operations can be done entirely on the client, without requiring any communication with or processing

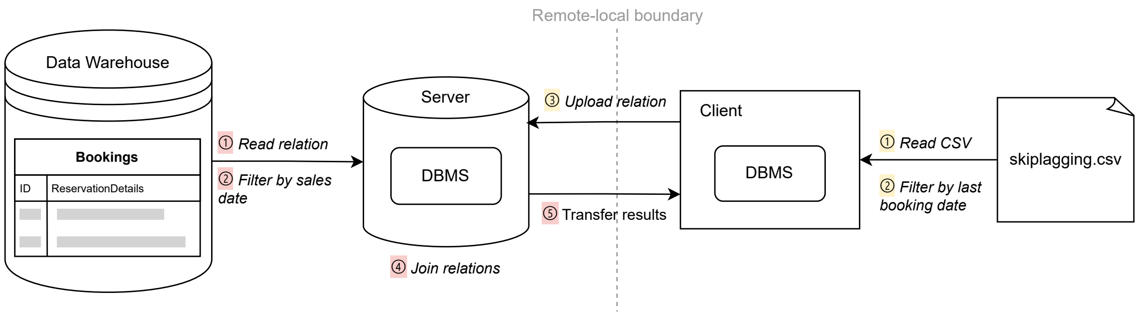


Figure 4.2: Processing of the exemplary query of the airline's data analyst. Client and server process parts of the query concurrently and exchange data to synchronize.

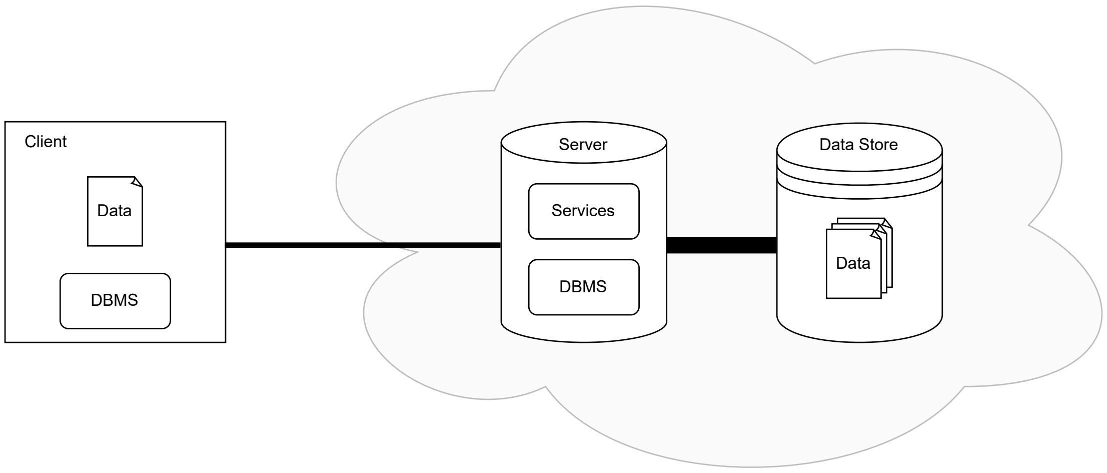


Figure 4.3: High-level visualization of the hybrid-execution model

on the server. Hence, there is no network delay and the operations can be computed in almost no time.

4.3 High-Level Overview

In the hybrid-execution model, the system is composed of two high-level logical components, a client and a server, and both have a fully-fledged DBMS at their disposal. This is shown in Figure 4.3. Hence, both the client and the server can take part in processing an incoming SQL query. When the client receives a query, its query optimizer decides where to execute which parts of the query.

Take the query in Listing 4.1 as an example. It shows the flights that persons with the surname "Springer" have taken. Two relations are joined in the process, but they reside in

```sql
SELECT Bookings.flight_nr
FROM remote_db.Bookings
JOIN local_db.Passengers ON Bookings.passenger_id = Passengers.id
WHERE Passengers.surname = 'Springer';
```

Listing 4.1: Example SQL query that is executed hybridly

different locations: one locally, the other remotely.

Suppose that the `Bookings` relation is stored in cloud storage and has a large cardinality. The `Passengers` relation, on the other hand, may be large, too, but the selection (`WHERE`) operation is very restrictive and lets only few tuples pass through. For that reason, the query optimizer decides to keep the `Bookings` relation on the server and instead transfer the filtered `Passengers` relation from the client to the server for the `JOIN` that follows. The following query is run on the client:

```sql
SELECT id FROM local_db.Passengers WHERE surname = 'Springer'.
```

The client then transfers the much smaller result relation to the server. There, the intermediate results from the client are joined with `Bookings` and the final projection is applied:

```sql
SELECT flight_nr
FROM remote_db.Bookings
JOIN (SELECT * FROM GET_CLIENT_RESULTS()) ON passenger_id = id.
```

As a final step, the results from the server have to be sent back to the client. In the following section, we dissect the hybrid query processing process in more detail.

4.4 Implementation in DuckDB

Hybrid execution has been implemented on top of DuckDB. We describe DuckDB and its query processing flow in Chapter 2. In DuckDB, extensions can be loaded that make modifications to the phases of the query processing flow. Hybrid query execution was implemented in such an extension, which we call the HQE extension, in combination with an implementation of the server[2]. In this section, we describe what was added to the phases of binding, planning, and execution. We focus primarily on DQL queries and only write briefly about DML statements.

[2]The HQE extension is separate from DuckDB and resides in a different codebase which is developed by a company called MotherDuck. DuckDB, on the other hand, is open source but maintained by DuckDB Labs.

The hybrid-execution model enables a different way of processing queries, but the queries themselves stay the same. Hence, no changes to the parser were needed.

4.4.1 Binding

The catalog in an RDBMS stores information about objects that are part of a database, together with associated metadata. Each catalog corresponds to one database. The catalog keeps track of the schemas, tables, and views defined in a database, indexes that the system or users have created, columns contained in tables and views, and many other things. Notably, it also includes statistics about the database objects, such as the number of rows or the distribution of values in a table. The RDBMS stores information about catalogs themselves in the system catalog.

After an SQL query has been successfully parsed, catalog information is used to resolve identifiers of database objects referenced in the query (see Section 2.2.2). In the hybrid-execution model, those database objects can reside on the local computer but also in the cloud. Therefore, a special catalog was implemented that stores information about remote database objects and keeps it up-to-date. The synchronization is especially important in multi-user scenarios where catalog information can be changed by other parties.

Catalogs in DuckDB can be modified with SQL. For example, new schemas, tables, views, and other database objects are created when users issue the corresponding statements like `CREATE SCHEMA`. With hybrid execution, these operations are executed purely on the server when they affect remote databases. The client cannot manipulate the local catalog information for remote database objects directly, but only when it receives updated information from the server. This prevents the client and server from getting out of sync.

The new catalog must also provide information about the location of databases to the remote-local planner as well as statistics to DuckDB's query optimizer.

4.4.2 Planning

The aim of hybrid execution is to perform the computation close to where the data is and reduce the amount of data transferred over the network. Through the catalog extension, information is available about the location of each database. When reading from local files, the initial processing is set to happen on the client. For remote files such as those stored on AWS, it should instead take place on the server, since it usually has a better network connection and is often physically closer to the data.

Hence, developers of the HQE extension cannot unrestrainedly make changes to DuckDB, though it is possible to contribute to the project.

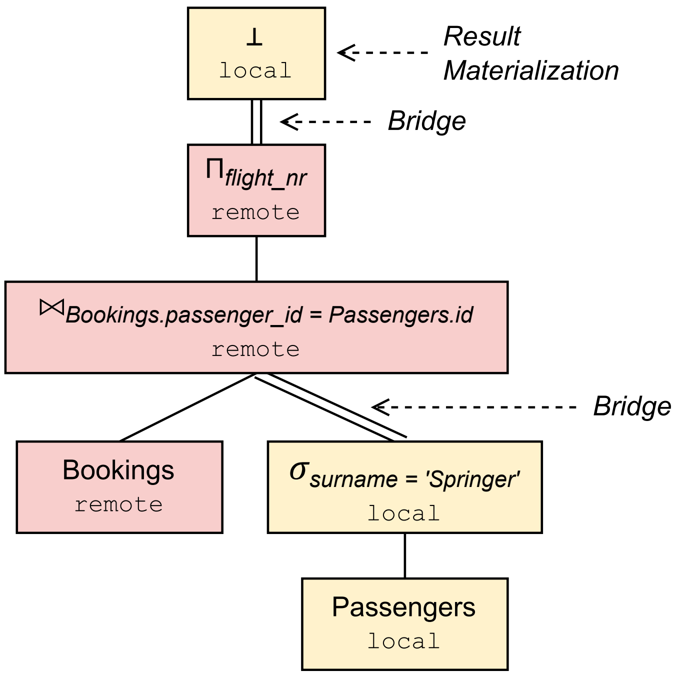


Figure 4.4: Logical query plan for Listing 4.1 with operations placed either locally or remotely

Starting with this information, the *remote-local planner* traverses the optimized query plan that DuckDB produced from the leaf nodes to the root and decides for every operator if it should be placed on the client or the server. Table functions, the leaf nodes in the operator tree, are placed close to the data, as described before. Other operators are usually placed wherever its leftmost child is placed. This child is in general the one that is expected to produce the largest intermediate result.

Some of DuckDB's operators share state with each other. The remote-local planner has to make sure that those operators are co-located on the same node.

For binary operators, it is possible that one child is evaluated locally, but the other remotely. Then the result of one of the children has to be transferred to the client or the server, respectively, depending on where the parent operator runs. A new logical query operator called a *bridge* is introduced that facilitates this data transfer.

If the original root of the query operator tree is executed remotely, an additional operation $\perp$ has to be inserted on top of the tree to send the result relation to the client, where it is finally materialized. For DML statements that target remote databases, the data transfer happens in the other direction: The client sends tuples to the server. Those data transfers are initiated by the bridge operator, too.

In Figure 4.4, the logical query plan for the query in Listing 4.1 is shown. The operations are marked with the location where they are executed. Because the `Bookings` relation has a larger estimated cardinality than the filtered `Passengers` relation, the theta-join is also done remotely. The $\perp$ operation had to be inserted at the top of the operator tree to transfer the results back to the client through a bridge.

4.4.3 Execution

DuckDB uses morsel-driven parallelism (see Section 2.2.3). Therefore, query operator trees are split into pipelines which are scheduled and executed for segments of the input data. Pipelines start at a data source and end in a pipeline breaker, also called *sink*.

These sinks are also the places where, in the hybrid-execution model, a data transfer between client and server can take place with the logical bridge operator. Bridges separate *fragments* from each other. These are groups of one or more connected pipelines which are executed entirely on the same node until their intermediate result is transferred to the other node.

When the query execution starts, the fragments that are marked as to be executed remotely are sent to the server. There they are scheduled, executed, and their results transmitted to the client. The fragments have already been parsed on the client side, so the server can skip this step of processing.

During execution, a bridge plays two roles: On the node that sends the data, it is a data sink which marks the end of a pipeline. On the receiving side, it starts a new pipeline as a data source, and the tuples are further processed when they arrive. Bridges are used both for the server sending data to the client and the other way round.

Unlike other sinks, intermediate results do not have to be completely written to the disk. Instead, tuples are buffered and sent in batches over the network.

5

Foundational Considerations for the Test Suite Design

The goal of this thesis is to devise a strategy to test the functional correctness of the DuckDB extension for hybrid query execution (HQE) (see Section 4.4). In this chapter, we motivate why we use DuckDB's own test suite as a starting point and portray what characteristics of hybrid execution should be tested. We end with an overview of test evaluation methods and explain why we chose code coverage as metric.

5.1 The Trade-Off Between Risk and Costs

Functional testing is an important activity during the development of software to make sure that it functions as intended. During dynamic testing, the software under test is executed with selected inputs and the actual outcomes (return values, output, state changes, calls to outside dependencies, etc.) are verified to match the expected ones. We have already explored different testing techniques in Section 2.4.3. They are used to make sure that many code paths through the system are exercised and the relevant input combinations are tested to give enough confidence that all functionality works correctly. These different techniques exist because only exhaustive testing, i.e., testing all possible combinations of inputs to the application, can consistently find all possible defects in a program [29; 5, pp. 11–12]. But exhaustive testing is not feasible in all but a few cases [5, pp. 1, 12]. For example, there are infinitely many valid SQL queries that can be sent to a DBMS, not even to mention the many different states (databases, tables, values, etc.) and configuration options that influence the query result. Therefore, pragmatic heuristics for software tests are needed to demonstrate with some certainty that the system is of high quality and give enough confidence that the risk of software failures is low enough.

This points to an inherent trade-off that has to be made between the costs of testing and the risk that the software manufacturer is willing to accept [5, pp. 3–4]. New tests need to be developed and maintained, and every test run takes some time[1]. Yet, they can ever only show that a system *has* defects, never prove that it is free of them. This means that there are diminishing returns on the investment in software tests [5, p. 4].

Still it holds true that more—and more effective—testing reduces the chance that defects remain undetected [82, p. 20]. By conducting a higher number of sufficiently rigorous tests, a larger portion of the codebase can be executed, potentially uncovering more bugs and thereby enhancing the overall confidence in the software's reliability and quality. Especially when there is a low-cost way to significantly increase the number of tests, it will likely pay off to go down this path.

5.2 Reusing an Existing Test Suite

DuckDB already has an extensive test suite in place, which we outline in Chapter 3. Our objective is to reuse these existing tests and extract the most value from them when testing the functional correctness of the HQE extension. This is captured in our research question **RQ1** on how to adapt the DuckDB test suite for hybrid execution.

We focus only on functional correctness and leave out all concerns about reliability and robustness, performance, or security. The tests in the DuckDB test suite that are mainly used to test the functional correctness are the SQLLogic tests.

With the SQLLogic framework, the whole DBMS is seen as a black box. Tests do not target any specific internal code of the system directly. Instead, all interactions are expressed in SQL, with implementation details assumed to be unknown to the test author.

SQLLogic tests are end-to-end tests, exercising the entire DBMS. Since the components of a DBMS, which we describe in Section 2.2.1, exhibit intricate interactions between each other and often come with a rather complex stateful interface, it is difficult to test them in isolation. SQLLogic tests are written from a user's perspective, so they are still able to trigger all code paths that are executed in real scenarios while simplifying the writing of tests thanks to a higher level of abstraction: All a test author needs to specify is the SQL statement and possibly the expected result.

[1]The execution time for automated tests is comparably low, though, and it is in general not a problem to run these tests very often.

The abstraction from implementation details makes it possible to reuse SQLLogic tests for other DBMSs. DuckDB based its test suite on the tests of SQLite. Now we are about to reuse DuckDB's tests for a hybrid-execution test suite. The results that are expected for an SQL query are the same for vanilla DuckDB and DuckDB with hybrid execution, because the placement of the computation should not affect what is computed.

This approach gives us a comprehensive test suite early on and with comparatively low effort, ensuring a certain level of quality and giving developers the confidence to change and deploy the software with assurance.

In general, end-to-end tests tend to be slower than component tests. For a good adoption of the tests, we need to ensure that they are executed fast and deterministically.

5.3 Modifications Made by Hybrid Query Execution

To implement hybrid execution on top of DuckDB, the HQE extension hooks into the different phases of the query processing flow (see Section 4.4). We want the tests to demonstrate that DuckDB together with this extension still produces correct results. Therefore, it is beneficial when tests cover a big surface area of DuckDB's public interface and hence exercise a big part of DuckDB's functionality.

Hybrid execution transforms DuckDB from a single-node DBMS to a system with two distributed nodes. Therefore, data can be placed in two different locations. Furthermore, since there is now a network between these two nodes, communication issues can occur.

In its SQLLogic tests, DuckDB does not have the notion of different data locations. Instead, every database is local on the client[2]. This assumption does not hold anymore for hybrid execution. In this case, a database can reside either locally or remotely.

To start with, the tests for hybrid execution should verify that the correct result is returned no matter if a particular database is residing on the client or the server. This leads us to the creation of different test modes, which we expound in Section 5.3.1.

In this way, tests can exercise the query planner which decides where to execute a query based on the data location. The additions in the binding and execution phases that stem from the HQE extension can be tested in particular when interactions between the client and the server take place.

Hybrid execution introduces problems that originate from the fact that it makes DuckDB a distributed system. However, these problems mostly fall into the category of reliability errors and are thus outside of the scope of this thesis. We also do not address tests for any additional features that can be built on top of hybrid execution, such as multi-user support or collaborative data sharing, but only pay attention to the execution model itself.

5.3.1 Modes of Testing

Hybrid execution introduces the concept of database location which is not present in DuckDB itself. To incorporate this concept, the test suite for hybrid execution should confirm that:

a) Executing SQL statements against a local database still works as expected and is not affected by the HQE extension.
b) Executing SQL statements against a remote database yields the expected result.
c) Executing SQL statements that use data from both local and remote databases yields the expected result as well.

The last goal was not achieved in this thesis, but some ideas for it are discussed in Section 8.2.1.

For the other two goals, each SQLLogic test script is run in two test modes: Local and remote[3]. Because SQLLogic tests are written in SQL, it is possible to use them like this with different DBMS configurations.

In the *local* test mode, the client is not connected to a server and all queries run against a local in-memory database.

The *remote* test mode is used to check the execution of queries on the server. For this, the client first has to connect to a server and create a remote database to work on. But to speed up the tests, we do not want to connect via a real network. Instead, the server runs in the same process as the client.

It is the remote test mode in particular that helps to validate the implementation of the hybrid execution model since it is here that client-server interactions take place.

5.3.2 The Test Server

Automated tests should be fast, isolated and deterministic so that developers can run them often, identify errors easily, and trust their results (no false positives) [61, p. 6]. Using a real network for the communication between the DBMS client and server in the tests would go against these objectives, as it would be too slow and error-prone. Therefore, we use a fake implementation [53, p. 134] of the actual server in the remote test mode[3].

[3]The test modes and test server implementation were not done as part of this thesis, but were already completed by other developers previously.

This fake server runs in the same process as the test runner, and newly created databases are stored in memory. At the communication boundary, every remote procedure call is instead converted to a local function call. Hence, there is close to zero communication overhead, and no network problems can affect the test result.

A new server instance is spun up for every SQLLogic test script so that all tests run isolated from each other and do not share any server state.

Having the client and server running in the same process greatly aids debuggability, too. Developers can trace the whole execution on a single machine without having to correlate information from multiple nodes. Furthermore, it allows for fine-grained control of the server execution. For example, to test how the client can handle error conditions, a test can be created in which the server is shut down after a short period of time.

However, a fake implementation of the server comes with downsides, too. Since the real implementation is not fully exercised by the tests, those tests may not find all the problems in the production code. Moreover, the fake implementation has to be kept up-to-date with the real server, or otherwise both will diverge and the fake implementation resembles the real one less and less.

The wish for deterministic execution when testing distributed database systems is not new. FoundationDB is built around a deterministic core which abstracts from sources of non-determinism such as the clock, network, or disk. Tests are executed in a deterministic discrete-event simulator. This not only helps to execute queries deterministically, but can also be used, for instance, to see how the system behaves in the face of erroneous conditions [92].

The developers of the transactional TigerBeetle DBMS built deterministic simulators for the network, storage, and other external dependencies, too. They run a fuzzer inside this simulator to find problems which then can be easily reproduced [19].

5.4 Design of Experimental Evaluation

Software tests should be able to find many possible defects and hence reduce the risk of bugs staying undiscovered. To evaluate whether the steps to adapt DuckDB's SQLLogic tests have the desired effect, we want to conduct measurements. A metric for these measurements should be simple, consistent, and relevant. Code coverage is a widely-used metric that fulfills these requirements and is thus used for our evaluation, too.

5.4.1 Metrics for the Effectiveness of a Test Suite

Code coverage is a metric to describe how much of a system's code gets exercised during a test run (see Section 2.4.4). Although it does not express directly how "good" a test suite is, high code coverage is a necessary condition of a good test suite: A test can only find a bug in a particular piece of code or on some code path when this piece or path is actually executed.

Other metrics have also been proposed to measure the effectiveness of a test suite. Hutcheson defines the test effectiveness as the ratio of bugs found by tests to the total number of bugs found (by tests and users) [36, p. 191]. It is also possible to use metrics of software quality as indication for the test suite effectiveness. One such metric is the reliability and corresponding measures of the time between failures or the failure count [28]. Other authors have proposed to use a mutation score [16; 9] to determine test quality [86; 64]. For this, an artificial defect is implanted into a program, producing a mutant. Then, the mutants that are discovered by the test suite are counted. Hence, this metric does not only capture if the code that contains an artificial defect is executed, as would code coverage do. It also indicates whether test oracles are able to identify a failure [86].

But none of these metrics is as easily quantifiable as code coverage. Especially measuring the number of bugs or time of production failures assumes long-standing production experience to collect these metrics, which is not available in our case.

Mutation testing comes with an inherent complexity for large software systems, since there are a lot of possible mutations for every part of the code. There have been attempts to apply this kind of testing to large, but they come with their own limitations. In [4], the authors at Facebook used machine learning to find mutants that reflect typical errors made by programmers. But this approach requires a big effort to set up. At Google, the mutation tests are run only incrementally and not for the whole codebase [65]. However, we want to measure the effectiveness of the whole test suite at every step.

Because code coverage is easy to measure and there are many tools to help with this, we use this metric to measure the effectiveness of our test suite.

5.4.2 Measuring the Code Coverage

We use the source-based code coverage tool[4] that is integrated with LLVM, the backend for the Clang compiler. It instruments the program with counters at various points, such as the start of a function or statement, to count the number of times this function or

[4]`https://clang.llvm.org/docs/SourceBasedCodeCoverage.html`

statement is executed during a test run. This instrumentation comes with low overhead and also works on optimized code. A code coverage report is then generated from the counter values with LCOV[5].

LCOV can visualize the function, line, and branch coverage. However, the branch coverage numbers are not always accurate because conditional branches can also be created for error propagation, and in general it does not work well with optimized code [63]. Therefore, we use the line coverage as metric to measure the test suite effectiveness.

The codebase of the HQE extension consists of DuckDB itself[6] as well as additional code to implement hybrid execution. However, what complicates the code coverage measurement for DuckDB is that the same code is shared by the client- and server-side DuckDB instance. In other words, no matter whether a DuckDB function is called by the client or by the server, it is considered to be covered by a test in any case. But it would be interesting to know what portion of the code is executed by the DBMS on the client and how much on the server. We can get an approximate answer to this question by using the two different test modes.

In the **local test mode**, only a very small part of the hybrid-execution code is needed; the execution mostly happens in the DuckDB codebase. Since one objective of the local test mode is to verify that the core DuckDB functionality is not affected by the HQE extension, it is sufficient to measure the line coverage for:

- Only the code of DuckDB.

On the other hand, for the **remote test mode**, we want to know how much of the hybrid execution-specific code is tested. But we are also interested in how much of DuckDB's code gets covered since we want to ensure that the available DuckDB functionalities also work in combination with hybrid execution. Therefore, we look at the line coverage for:

- Only the code for hybrid execution.
- Only the code of DuckDB.

In the next chapter, we chronicle our effort to use the DuckDB tests for hybrid execution and report on the code coverage achieved with the local and remote test mode, respectively.

[5] `https://github.com/linux-test-project/lcov`

[6] Concretely, DuckDB's source code is integrated as a git submodule. It is not under our direct control.

6

Leveraging DuckDB's Test Suite for Hybrid Query Execution

When creating a test suite, one has to find a balance between its thoroughness, the cost of running it (including the test runtime) and the implementation effort. As motivated in Chapter 5, we therefore want to find ways to leverage the extensive DuckDB test suite to test hybrid query execution (HQE).

Using tests that are written in SQL comes with the advantage that they are not coupled to internals of any RDBMS, but only use its standardized public interface. For a given statement, the expected result is mostly[1] defined by the SQL specification [41]. On top of the standardized SQL, many database vendors add custom features with a corresponding syntax and semantics to their particular DBMS. This leads to the creation of different dialects of SQL.

When SQL-based test scripts only specify the statements to execute and the expected results, they can validate the correctness of every DBMS that speaks the SQL dialect that is used in the tests. The only ingredient of the test suite that has to change is the test runner—the program that reads the SQL statements from the test scripts, sends them to the DBMS under test, and asserts that the actual and expected results match.

The mode of processing, whether with hybrid execution or not, should not change the result of queries. This fact, together with some adjustments of the test runner, allowed us to repurpose the SQLLogic tests from DuckDB to test hybrid execution.

In this chapter, we engage with the two research questions that we pose in Chapter 1. We attempt to answer **RQ1** by chronologically documenting the steps we took to adapt

[1]Some small aspects are left unspecified. One example is the sort order of `NULL` values, i.e., if they appear first or last in a sorted column [40, §10.10 Rule 3].

DuckDB's SQLLogic tests to validate the HQE extension. To easily refer back to them later, every step is given an identifier $\mathbf{S_n}$. Our aim was to use the existing tests as much and effectively as possible. This meant in particular increasing the amount of code that is executed by the tests.

We start by analyzing the situation we found at the beginning of our work and identify problems that prevented some tests from being run, therefore keeping the code coverage low. In the next step, we describe how we tackled one of these problems and again look out for new ones. We repeat this process for all the steps that follow.

To see how effective these steps were (**RQ2**) and to inform the next step to take, we regularly measure and analyze the code coverage that the tests achieve. Specifically, we collect the line coverage for the following parts of the codebase (see Section 5.4.2):

1. The core DuckDB code, executed in the local test mode ("Local, DuckDB")
2. The core DuckDB code, executed in the remote test mode ("Remote, DuckDB")
3. The code for hybrid execution, executed in the remote test mode ("Remote, HQE")

We contrast the coverage with the time that is required to run all tests in both the local and remote test mode.

6.1 Initial Situation ($\mathbf{S_0}$)

DuckDB's SQL-based test suite consist of SQLLogic test scripts. They are executed and their results validated by a test runner.

At the beginning of our work, the SQLLogic tests from DuckDB were already used to test the HQE extension. The scripts were executed by a custom test runner that uses the Catch2 test framework[2].

The test framework provides a main entry point into the test program and offers the infrastructure that is needed to write unit tests. For example, it defines ways to create new test cases or write assertions. A Catch2-based program can automatically discover and execute the test cases, verify that the assertions hold, and generate a report about a test run. Every test case is identified by a unique name and can be marked with several tags. This makes it possible to select which tests to run.

The SQLLogic test scripts were sorted into different test groups, which correspond to the directories in which the test scripts were located. For every group, a single Catch2 test case was created in which all the test scripts in the group were executed one by one by

[2]`https://github.com/catchorg/Catch2`

the custom test runner. However, this test runner came with major limitations, as we will explore in the next section.

In the local test mode, only four SQLLogic tests scripts were executed—of the around 2600 SQLLogic test scripts from the DuckDB test suite (see Chapter 3). This resulted in a line coverage of 21% for the DuckDB codebase. In the remote test mode, 101 test scripts covered 39% of DuckDB's source code and 50% of the hybrid execution code. The average execution wall-clock time[3] calculated from five test runs with both tests modes was 1.7 s.

	Local, DuckDB	Remote, DuckDB	Remote, HQE	Execution time
$\mathbf{S_0}$	21%	39%	50%	1.7 s

To increase the amount of code that is executed by the tests and hence increase the chances of finding bugs in the HQE extension, we wanted to increase the number of test scripts that are run. But before attempting this, as a stepping stone to subsequent improvements, we first replaced the custom test runner.

6.2 Using a Feature-Complete Test Runner ($\mathbf{S_1}$)

Initially, a custom test runner was used to execute SQLLogic test scripts. However, a major drawback of this test runner was that it did not support all the commands that are used in the test scripts. For example, a test script can contain a `require` command which instructs DuckDB to load an extension, or a `loop` command to execute some statements multiple times. The custom test runner failed to interpret these commands. Hoping to get more test scripts to run successfully, the first step we took was to replace it with DuckDB's own test runner. Executing more test scripts would also lead to a higher code coverage.

Of the 1997 test scripts, 825 contained test commands that were unsupported. Sometimes, but not always, this led to a test failure, for example, because loop variables could not be interpreted. In other cases, the command was just ignored. This was the case, for instance, for `require` commands: The specified extension would just not be loaded. However, this could still lead to wrong results or failures in subsequent SQL statements. For example, there are SQL queries that read data from a remote source. They rely on an extension to create a HTTP connection. If this extension was not loaded, those queries would fail.

For DuckDB, an alternative test runner had already been implemented, also based on the Catch2 test framework. Because it is developed side-by-side with DuckDB and is targeted

[3]All experiments were done on a MacBook Pro (2020) with 8-core M1 chip and 16 GB of memory, using the GNU Time (`time`) utility

specifically towards executing its SQLLogic tests, this test runner does support all the test commands used in the test scripts. And since it is developed in lockstep with the rest of DuckDB's test suite, it will support all test features that may be added in the future as well. Aside from being able to run more DuckDB tests, an additional benefit of using this test runner is that all test commands are also available for use in new tests written specifically for the HQE extension. Therefore, we replaced the original test runner with DuckDB's one.

The DuckDB test runner was not designed with extension in mind. But to test hybrid execution, some custom initialization logic has to run before each test. In particular, the client DuckDB instance has to load the HQE extension and possibly connect to a test server. Before being able to replace the test runner, we therefore had to create an extension point in DuckDB's test runner that derived classes can utilize.

Thereafter, two new test runner classes—one for each test mode—were created that derive from DuckDB's test runner. Both test runners load the HQE extension into DuckDB before running any tests, but only the one for the remote test mode connects to a server and creates an initial remote database against which to execute all subsequent SQL statements.

A set of SQLLogic test scripts was previously excluded from running because they did not support some test command. With the new test runner, all these could now be enabled. But during a test run still only 117 test scripts were executed in total: five in the local and 112 in remote test mode.

Since the number of executed test scripts did not increase by much, the improvements in code coverage are also marginal. The local tests covered 23% (+2%) of the DuckDB codebase. In the remote test mode, the coverage of the hybrid execution code stayed the same with 50%, but there was a small increase in the line coverage of DuckDB up to 40% (+1%). The execution wall-clock time stayed roughly the same with 1.8 s (+0.1 s) as average of five test runs.

	Local, DuckDB	Remote, DuckDB	Remote, HQE	Execution time
$\mathbf{S_1}$	23%	40%	50%	1.8 s

Although using an alternative test runner did not affect the code coverage numbers much, it was the foundation on which subsequent changes could be made. The reason that only 12 additional test scripts were executed, despite there being 825 that contain unsupported test commands, is a result of how the test cases were organized. In the next section, we describe how we changed the structure of the test suite to remedy this problem.

6.3 Getting All Test Scripts to Run (S_{2a})

SQLLogic test scripts in DuckDB are assembled into groups based on the directory in which they are located. To give an example, all test scripts that are targeted at testing the logic of `INSERT INTO` are grouped together. We found that for each group, one Catch2 test case was created in which all test scripts in this group were executed sequentially. This is exemplified in Listing 6.1.

```
TEST_CASE("Insertion Tests Remote") {
    auto test_group = "insert-into";
    auto directory = std::filesystem::path {duckdb_test_root + test_group};
    auto test_scripts = GetTestScriptsInDirectory(directory);

    for (auto& script : test_scripts) {
        RemoteTestRunner::ExecuteFile(script);
    }
}
```

Listing 6.1: Code to illustrate how test cases were defined

The Catch2 framework automatically discovers and executes the test cases defined in the program. The issue, though, was that a test case terminates as soon as the first assertion fails. Hence, whenever there is one assertion failure in one of the test scripts, all remaining test scripts in the group are not even attempted to be run. Assertions fail, for example, when an unexpected SQL query result is produced by DuckDB or when an error occurs during the processing of a statement.

At the time we faced this issue, a lot of test scripts were still not run successfully because a few SQL statements were not supported by the HQE extension, the test runner still failed to execute a small fraction of the test scripts, and because some actual bugs have crept into the code. This meant that many test cases were terminated early, too, and many test scripts were never run.

To increase the total number of scripts that the test runner executes, every test script had to run independently of the others. We therefore enclosed every test script in its own test case. Hence, even if the execution of one test script is unsuccessful and the corresponding test case would thus fail, this does not impact the execution of other test scripts.

Another benefit of this change is that test scripts can be called individually. With the Catch2 framework, it is possible to specify exactly which test cases to run when the test program is started [57]. In particular, a test case can be specified by its name. Since a test case now maps to exactly one test script, developers can conveniently run a single test script. This makes it especially easier to debug individual tests.

One problem still remained that caused other test cases to not be executed: invalid or abnormal states. There were some tests that gave rise to segmentation faults and others that provoked the AddressSanitizer to abort the execution. The entire test program would stop after receiving those signals. Tests in this category had to be debugged or, as a first aid, disabled for test runs to complete.

Since after taking these steps all test scripts were now executed, successfully or not, the scale of test failures became apparent. Before continuing to collect the code coverage, we first had to make sure that a complete test run finishes successfully.

6.4 Disabling Tests That Are Expected to Fail (S_{2b})

At this stage of development of the HQE extension, there were still some SQL features that were not yet supported and known bugs in the system that still needed to be fixed. We go into more details about this in Section 6.5. Because it was finally possible to execute all test scripts after changing the structure of the test suite (see Section 6.3), many test cases did fail, which led the test runner to produce an overwhelming output. Therefore, we temporarily disabled the tests that were expected to fail.

A good regression test suite should give developers timely feedback on the correctness of new code *they* wrote. For instance, after fixing a bug, all it should take is to push a button to verify that a) the bug is indeed fixed and b) no other defects were introduced along the way. The same goes for new features: developers should get from the test suite the confidence to make any necessary changes while still being certain that they do not break any other part of the system. For a test suite, this implies that it should be "green by default". This means that all enabled tests shall pass when they are run against the current codebase, with no failures or errors, and that a test failure should indicate a recent mistake made by the developer who observes it. A test suite that does not live up to this standard has difficulty earning the trust of developers and might more often be ignored than used.

To provide developers immediately with a tool to increase confidence in the code as a whole and the changes they make, we wanted to get the SQL-based test suite to a green state as fast as possible while still running most of the tests. This meant that we had to ignore some bugs for the time being by disabling the test scripts that found them. Moreover, we also disabled tests that required yet unsupported SQL features. At the cost of having an incomplete test suite, it gave developers the opportunity to run the majority of tests and quickly find defects they might have introduced.

Therefore, we added a list of excluded test scripts. These test scripts are skipped during a normal test run. However, it is still possible to call them directly, for example, to debug and fix the root cause due to which they are disabled in the first place.

After we got the test suite to a green default state again, we were able to collect new code coverage numbers. 82% (+59%) of the lines in DuckDB's codebase were covered by 1716 test scripts in the local test mode. In the remote test mode, 745 test scripts were executed, resulting in a line coverage of 66% (+26%) for DuckDB and 65% (+15%) for the hybrid execution code. But since a lot more tests were executed, an average test run took 70 s (+68 s).

	Local, DuckDB	Remote, DuckDB	Remote, HQE	Execution time
$\mathbf{S_2}$	82%	66%	65%	70 s

The coverage for the hybrid execution source code is higher, though, when taking tests that were specifically written for hybrid execution into account. With these tests, the line coverage stood at 82%. But our primary focus will stay on DuckDB's existing tests.

A challenge which the new exclusion list was to keep it up-to-date and remove test scripts from the list whenever corresponding features are added or bugs are fixed. Another possible concern was that developers just exclude newly failing tests instead of finding and fixing the root causes of the failure. But this worry did not materialize.

Lastly, this long list of test names could be very confusing and difficult to work with. To alleviate this problem, we categorized the failing tests by the error message they generated. Still, it could be hard to see which test scripts exactly were already excluded and in which category to put test scripts that were newly added to the exclusion list. But at least this categorization was very helpful in creating an overview of unsupported SQL features and known bugs.

6.5 Aggregating an Overview of Test Failure Causes

The test scripts in the exclusion list are classified by error category. This made it possible to assemble an overview of why tests failed and hence which features are still unsupported, where known bugs are, and what it would take to resolve these issues. This overview was then used to prioritize the continuing work on the HQE extension.

Often, the error messages of failing tests indicated that some of the SQL features that are supported by DuckDB and are hence used in the SQLLogic tests were not yet supported by the HQE extension. The reason was that a new database catalog had been implemented, which did not yet allow to create new database objects of types such as sequences, indexes or functions, or provide statistics for them.

Another major gap was the missing support for multi-statement transactions. All statements in a transactions should be executed by the DBMS as a single logical unit whose results only are made visible to other concurrent transactions after it is committed [34]. However, with the HQE extension, transactions were not always atomic and isolated, and rolling back changes often did not work.

Lastly, some other functionality such as `PIVOT` statements[4] or server-side `PRAGMA` statements[5] also did not work. Together with other developers, these problems were—and some still will be—tackled step-by-step.

A second approach to identify which features were not supported is by taking a look at the code coverage achieved in the remote test mode. Since all SQL statements are eventually translated to physical operators, it is possible to detect unsupported features by looking at the coverage of these operators. This is shown in Figure 6.1.

In Table 6.1, the files with a line coverage of less than 50% are listed. It also contains the following three files listed: `physical_create_function.cpp`, `physical_create_index.cpp` and `physical_create_type.cpp`. This shows that with this method, too, we identified that `CREATE INDEX`, `CREATE FUNCTION`, etc. were not supported. However, while we can say for certain that SQL features that correspond to C++ files that were not covered were unsupported, this implication only goes in one direction: Some files corresponding to unsupported features actually *were* partially covered, for example, because DuckDB called the functionality internally.

In Table 6.1, some C++ files responsible for handling the reading of data from CSV files are listed as having low coverage. This was because all tests in this category were disabled since many of them caused a segmentation fault. The problem was fixed later and the coverage of these files went up as well.

This analysis showed that an important factor in increasing the code coverage even more was to fix bugs and add support for new DuckDB features. But there were still some things left to do on the test runner.

[4]`https://duckdb.org/docs/sql/statements/pivot.html`
[5]`https://duckdb.org/docs/sql/pragmas.html`

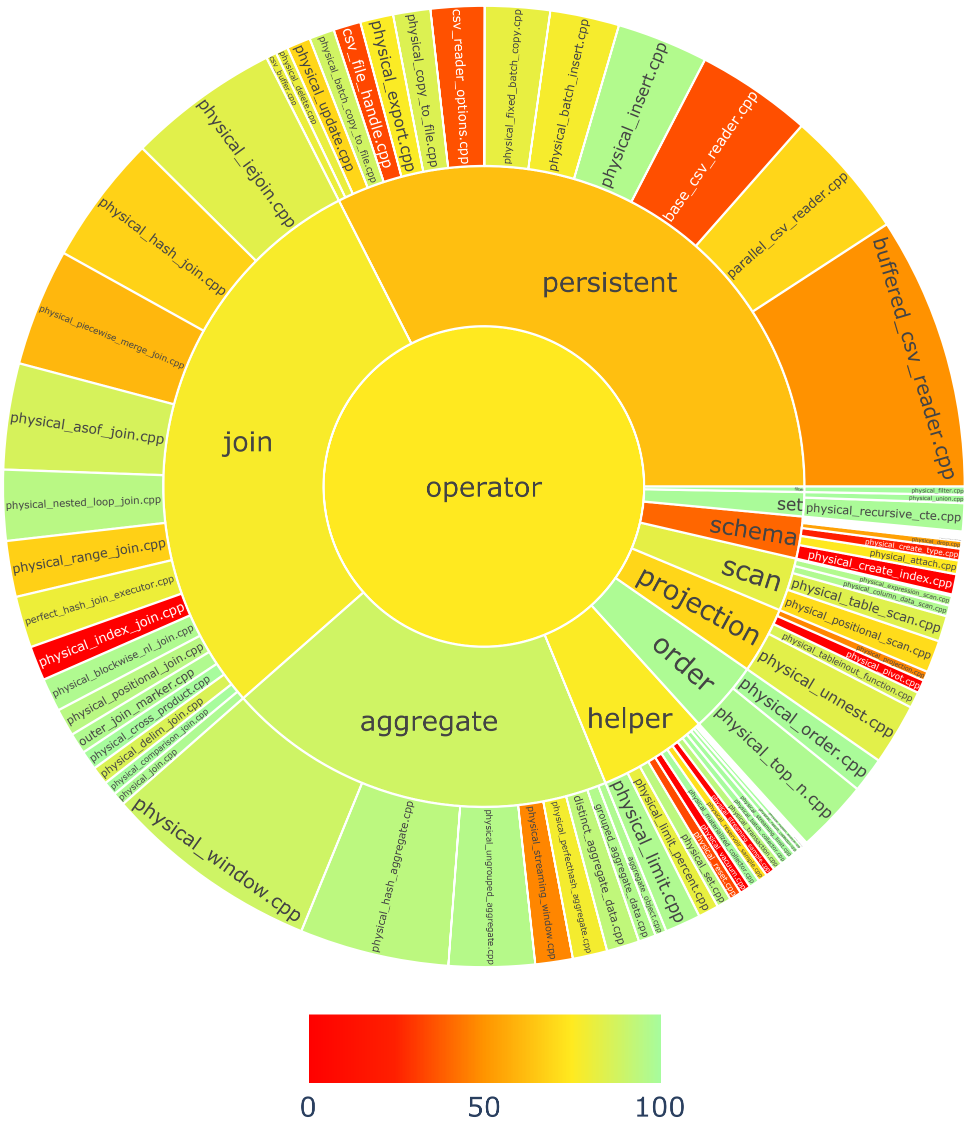


Figure 6.1: Visualization of the line coverage of source files in *duckdb/src/execution/operator* in the remote test mode after step $\mathbf{S_2}$. The area of each section corresponds to the number of lines of code.

File name	Line Coverage
operator/aggregate/physical_streaming_window.cpp	46.9%
operator/helper/physical_pragma.cpp	0.0%
operator/helper/physical_reset.cpp	34.1%
operator/helper/physical_streaming_sample.cpp	0.0%
operator/helper/physical_vacuum.cpp	0.0%
operator/join/physical_index_join.cpp	0.0%
operator/persistent/base_csv_reader.cpp	35.2%
operator/persistent/buffered_csv_reader.cpp	49.6%
operator/persistent/csv_file_handle.cpp	33.3%
operator/persistent/csv_reader_options.cpp	35.5%
operator/projection/physical_pivot.cpp	0.0%
operator/projection/physical_projection.cpp	46.9%
operator/schema/physical_create_function.cpp	0.0%
operator/schema/physical_create_index.cpp	0.0%
operator/schema/physical_create_type.cpp	22.5%

Table 6.1: C++ files in *duckdb/src/execution/operator* with a line coverage less than 50%

6.6 Handling Multiple Connections (S_3)

To test database transactions, it is possible to create multiple connections to the same DuckDB instance in a SQLLogic test script. All its SQL statements are still executed sequentially, but some are sent over one and some over another connection. Importantly, changes that are made in transactions on one connection should not be visible to the other connections before the transaction is committed.

The problem with tests in the remote test mode was that newly created database connections did not run SQL statements against the remote test database. Instead, DuckDB's client-side database was used by default. Therefore, all statements on new connections were executed as if in the local test mode.

To solve this, we changed the default database used by new connections to a remote database at the beginning of each test run. Only after this change, some issues with the transaction processing in the remote test mode became visible. They were previously hidden because transactions work correctly on local databases.

This change actually forced us to disable some tests that were now failing due to transaction issues. However, since the last measurement of the code coverage, other issues had been resolved and some additional tests could be enabled. In the local test mode, 1870 test scripts gave a line coverage of 84% (+2%) for DuckDB. In the remote test mode, the line coverage from 1326 test scripts was 75% (+9%) for the DuckDB and 70% (+5%) for the hybrid execution source code. With 85% (+3%), the code coverage for the hybrid execution implementation would again be higher when additionally considering the tests written specifically to test this aspect. An average test run took 96 s (+26 s).

	Local, DuckDB	Remote, DuckDB	Remote, HQE	Execution time
S_3	84%	75%	70%	96 s

6.7 Check for Correct Errors on Expected Failures (S_4)

In Section 6.4 we introduced the list of test scripts that are expected to fail and therefore excluded from the standard test run. One problem with this list is that test scripts have to be manually taken off this list whenever a bug is fixed or a previously unsupported feature gets implemented.

To automatically find out if tests still fail for the expected reason, we first linked the expected error message to every test script on the list. This was made easier thanks to the existing grouping of test scripts by failure category. Then we implemented a tool that runs the excluded tests and asserts that the produced error messages match the expected ones.

After running this tool, we could enable around 200 additional tests for which the root cause due to which they were disabled was fixed. In addition, we included 580 test scripts in the test run that are part of the DuckDB test suite, but not in the test directory on which we initially focused our attention. A test run did thus consist of 4031 test cases: 2216 SQLLogic test scripts were executed in the local test mode, 1815 in remote mode.

However, even though many more test scripts were used, the code coverage only improved slightly. The coverage of DuckDB in the local test mode stayed at 84%. In the remote test mode, the coverage for DuckDB was now at 78% (+3%). For the hybrid execution code, it even decreased to 64% (-6%). The reason for this was that new features were added to the HQE extension, but those are never exercised by DuckDB tests. Instead, they are tested separately in SQLLogic and C++ tests.

	Local, DuckDB	Remote, DuckDB	Remote, HQE	Execution time
S_4	84%	78%	64%	136 s

A test run now took already 136 s (+40 s). This could become an annoyance for developers that want fast feedback from the tests. Hence, we set out to improve the test running speed.

6.8 Running Tests in Parallel (S_5)

The Catch2 test framework provides the facilities to split the test suite into chunks of a certain size and run only one chunk. To deal with the variance in the time it takes to execute individual test scripts, the scripts are pseudo-randomly assigned to chunks. Otherwise, tests that tend to take longer than the average, such as tests that read from CSV files, would be clustered together.

Each chunk of test scripts is executed in parallel by a job, created using the GNU `parallel` tool[6]. With eight jobs, each covering 12.5% of the tests, on a laptop with eight cores, the test execution time was reduced by 62% down to 52 s (-84 s).

6.9 Summary

For our analysis, we make distinctions for the two different test modes. For the local test mode, the code coverage of the DuckDB code was captured ("Local, DuckDB"). For the remote test mode, we collected the code coverage of the DuckDB codebase ("Remote, DuckDB") and the HQE extension codebase ("Remote, HQE").

Through our work, we were able to more than double the overall code coverage from 40% to 83%. Since we focused on making the most of the existing DuckDB tests, the increase was most pronounced for the coverage of the DuckDB codebase: In the local test mode, the coverage increased from 21% to 84%; from 39% to 78% it climbed for the remote test mode.

We took the following steps to be able to use the majority of DuckDB's SQLLogic tests to run with hybrid execution:

- $\mathbf{S_0}$: Initial situation before the thesis-related work started
- $\mathbf{S_1}$: Switching to DuckDB's own test runner that supports all test commands
- $\mathbf{S_2}$: Running test scripts individually and enabling most of them
- $\mathbf{S_3}$: Handling multiple database connections in the same test script; enabling tests for added features/fixed bugs

[6]`https://www.gnu.org/software/parallel/`

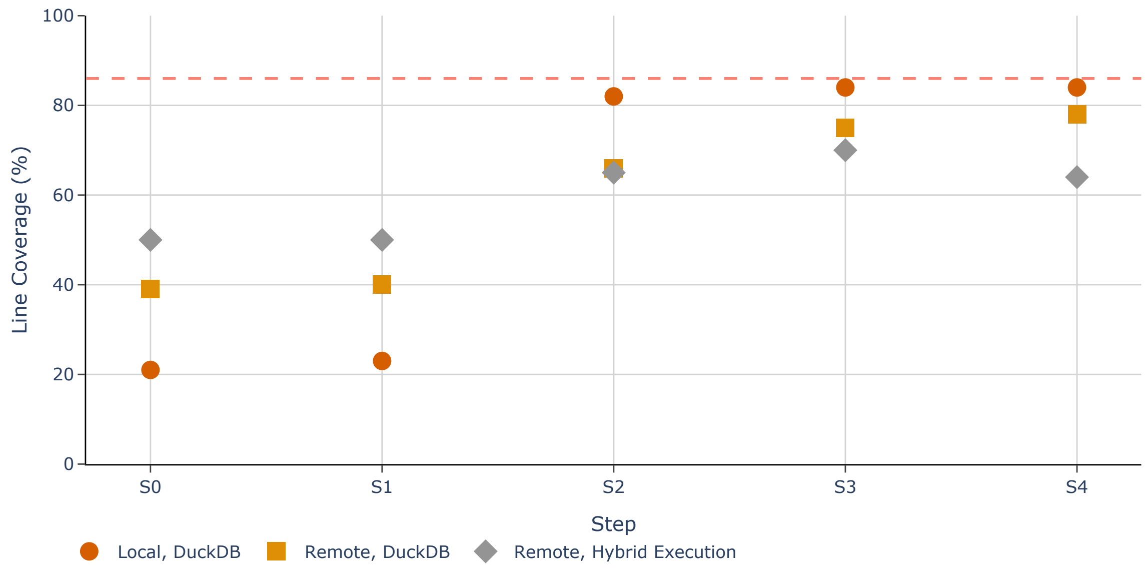


Figure 6.2: Line coverage after each step $\mathbf{S_0}$–$\mathbf{S_4}$

- $\mathbf{S_4}$: Using a tool to check which disabled tests can be enabled; including tests outside of the original test script directory
- $\mathbf{S_5}$: Parallelization of the test execution

Table 6.2 summarizes the collected code coverage for every implementation step (the coverage is the same for $\mathbf{S_4}$ and $\mathbf{S_5}$). The same data is visualized in Figure 6.2. There it can be seen that the biggest leap regarding code coverage was made for the tests in the local test mode. Furthermore, step $\mathbf{S_2}$, running test scripts independently and enabling most of them, was by far the most successful in terms of the increase in coverage.

	$\mathbf{S_0}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$	$\mathbf{S_4}$
Local, DuckDB	21%	23%	82%	84%	84%
Remote, DuckDB	39%	40%	66%	75%	78%
Remote, Hybrid execution	50%	50%	65%	70%	64%

Table 6.2: Line coverage after each step $\mathbf{S_0}$–$\mathbf{S_4}$

The higher test coverage was a result of enabling more and more SQLLogic test scripts with each step. However, when the code coverage was already high, just running more test scripts did not improve the coverage much further. This can be seen, for example, when comparing steps $\mathbf{S_2}$ and $\mathbf{S_4}$ in Tables 6.2 and 6.3. Even though 500 additional test scripts are executed after $\mathbf{S_4}$, the code coverage in the local test mode is only 2% higher—at the cost of a much higher execution time.

	S_0	S_1	S_2	S_3	S_4	S_5
Test execution time	1.7 s	1.8 s	70 s	96 s	135 s	52 s
Number of test scripts (local)	4	5	1716	1870	2216	2216
Number of test scripts (remote)	101	112	745	1326	1815	1815

Table 6.3: Overview of the number of executed tests and the time required for this for steps S_0–S_5

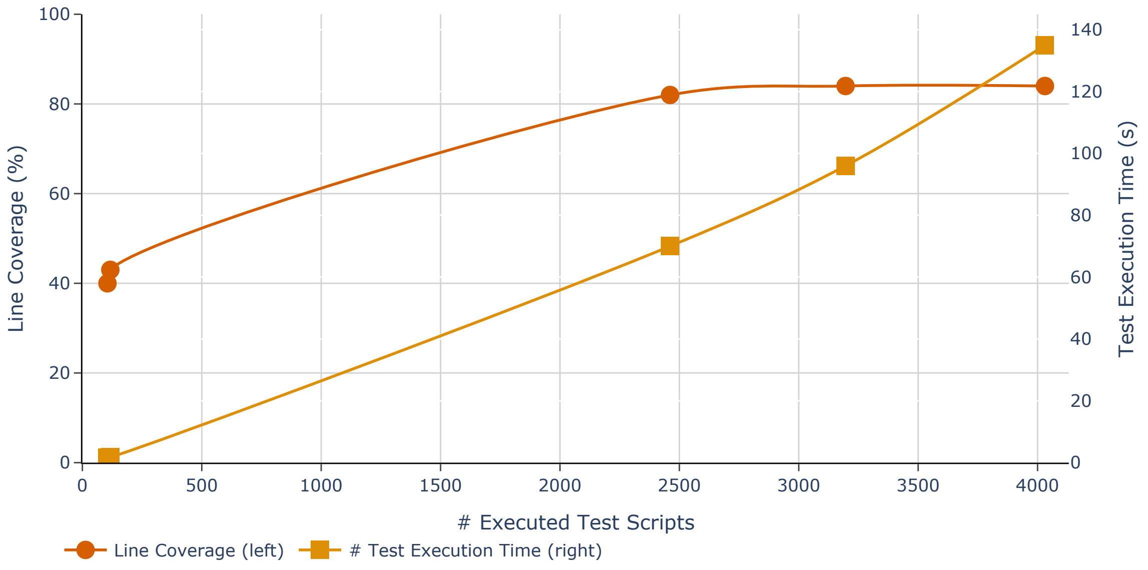


Figure 6.3: The line coverage achieved and time needed for a test run as a function of the number of test scripts that are executed

Those diminishing returns are also reflected in Figure 6.3. It shows the total coverage for the DuckDB source code from the local and remote test mode combined. While the execution time increases steadily and unrestrained with the number of executed test scripts, the coverage converges asymptotically to around 85%.

The execution time was reduced only once parallelization was added (see Table 6.3).

7

Discussion and Future Work

To create an extensive test suite for hybrid execution while keeping the implementation effort low, we wanted to leverage the existing SQLLogic tests in DuckDB's test suite (**RQ1**). We chronicle the actions we undertook in Chapter 6. In the end, we used most of these tests and executed them in the local and remote test modes, respectively.

We used code coverage as a metric to measure the effectiveness of our actions (**RQ2**). From the measurements it became apparent that not all steps were equally effective, even if put in relation with the time needed to implement them. Specifically towards the end, when the code coverage was already quite high, only small leaps were made. This could be ascribed to a focus that shifted, for instance, to improving the developer-friendliness of the tools. On the other hand, many new test scripts were added even during the last steps, which we expected to lead to better outcomes. Hence, a critical look must be taken at the metric we used.

Before doing this, we want to take another look at where the code coverage stands after the final step and identify holes in our test suite that are still left open.

7.1 Concluding Code Coverage Analysis

To identify holes in the code coverage, we reinstantiate the visualization from Section 6.5 for the last measured code coverage.

7.1.1 Subsisting Coverage Holes

Figure 7.1 already has a lot more green in it compared to Figure 6.1 and only a few parts are marked red, showing that most of the source files are now covered to a sufficient degree.

Only four physical operators are left with a code coverage of less than 50%. These are:

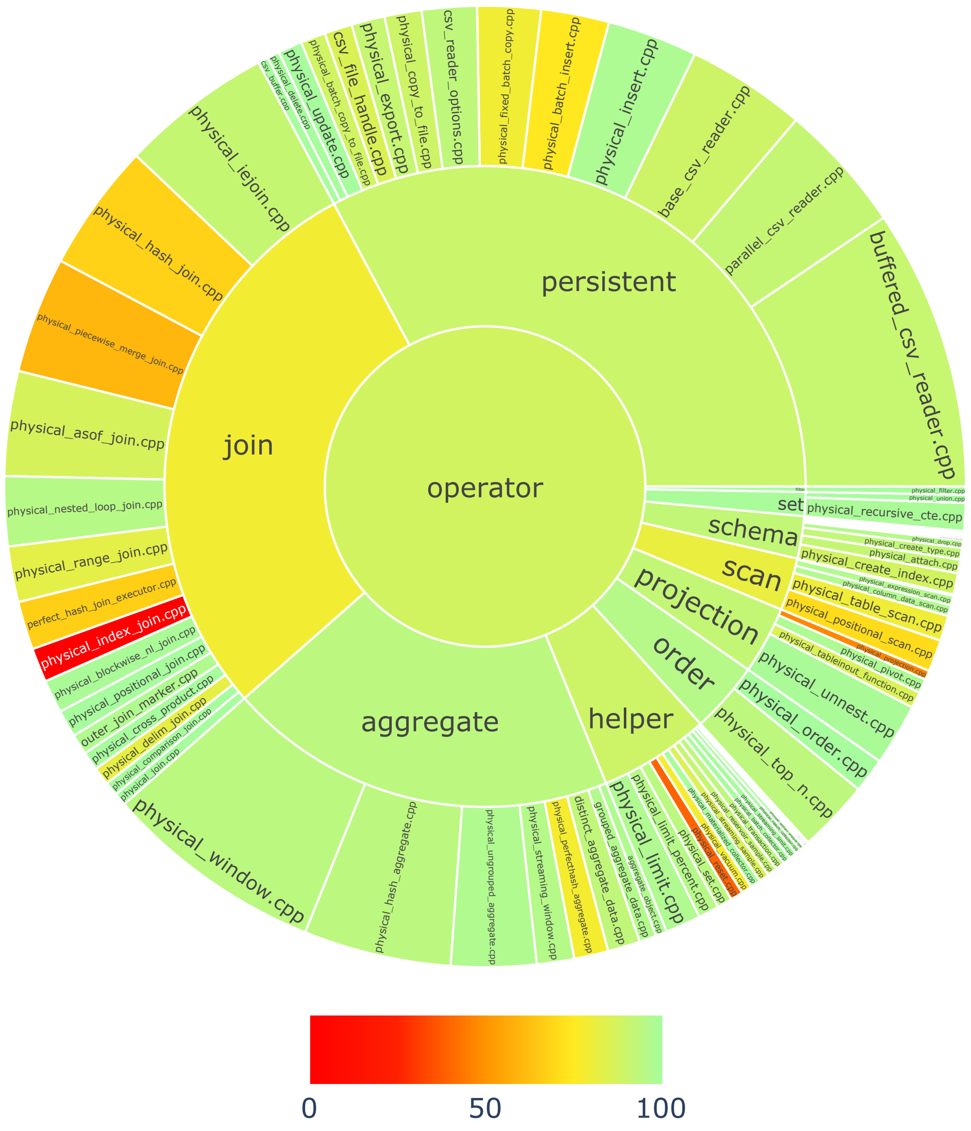


Figure 7.1: Visualization of the line coverage of source files in *duckdb/src/execution/operator* achieved in the remote test mode after step $\mathbf{S_5}$. The area of each section corresponds to the number of lines of code.

- operator/helper/physical_load.cpp: 0.0%; for loading other extensions
- operator/helper/physical_reset.cpp: 39.5%; for resetting configuration values
- operator/join/physical_index_join.cpp: 0.0%; for index joins, but indexes are not yet supported in the HQE extenison
- operator/projection/physical_projection.cpp: 46.9%; join projections are not covered, but neither are they in the original DuckDB codebase

When taking a closer look at which lines exactly are not covered in these but also in other files, it appears that many error paths are not tested. That is, there are not many SQLLogic tests that validate that exceptions, in particular internal ones, are thrown. However, many of the internal exceptions seem to exist for the purpose of defensive programming, checking conditions that are usually not triggered by interactions via the public interface of the DBMS. Therefore, not having them covered is acceptable.

Other coverage holes are a result of functionality not being used for hybrid execution. For example, the code in DuckDB that converts data into the Apache Arrow format is not hit by any test because the HQE extension internally uses the Protobuf format instead. A second example is all the code that is concerned with printing information such as the progress of query processing on the terminal. The SQLLogic tests do not validate that the rendering works correctly; DuckDB uses C++ tests for this purpose. Similarly, many `ToString` and other helper methods are never executed by any test.

These are some of the reasons why even in the local test mode, where after step $\mathbf{S_4}$ only six of the 2222 test scripts were disabled, the line coverage did not exceed 84%. DuckDB has additional tests aside from the SQLLogic tests that help them to cover 93% of the lines in their codebase. However, looking at the coverage of the source files does not clearly indicate what can be improved about the test suite. Instead, it requires to take a long, hard look at every file that is not well covered and determining if the not-covered functions are indeed relevant and worth testing.

For the hybrid-execution codebase, we were presented with a similar situation. Lines that were not covered often corresponded to error-handling code that stems from a defensive programming style. Other files contained features that are not strictly part of the implementation of hybrid execution, such as an AI-powered assistant for writing SQL and analyzing data. As such, they were not covered by the hybrid-execution tests. However, the code coverage pointed out some legitimate but small holes in the test suite as well.

7.1.2 Impact of Unsupported Functionality on Code Coverage

From our earlier results in Tables 6.2 and 6.3 we can see that adding more and more test scripts did not increase the code coverage much further in the local test mode. One reason for this is that a line that is hit a single time during a test run weighs the same as a line that was hit thousands of times. Hence, even though some lines might get executed a lot more and possibly as part of more diverse code paths, the coverage is only concerned with the number of lines hit. A (hypothetical) tool that measures the path coverage attained by a test run might thus show improved metrics for steps $\mathbf{S_3}$ and $\mathbf{S_4}$.

Still we wanted to see what code coverage could be reached in the remote test mode by adding the tests for features that are still unsupported. For this, we use the local test mode with its code coverage of 84.0% as baseline. For each set of features, we disabled the corresponding tests in the local test mode. The difference in code coverage to the baseline is the increase in coverage we can expect in the remote test mode when implementing the corresponding features and enabling those tests.

- When excluding the 204 tests related to `CREATE INDEX`, `CREATE SCHEMA` and other SQL statements that are not yet support by the catalog of the HQE extension, the coverage drops to 82.8% (-1.2%).
- When excluding the 17 test scripts that fail in the remote test mode because some operators cannot be serialized and transmitted to the server, the resulting coverage would be 83.7% (-0.3%).
- When excluding the 104 test scripts that are disabled because they fail for transaction-related problems, the coverage surprisingly stays at 84.0% (±0%). One explanation for this is that all the transaction-related DuckDB code is already executed by single-statement transactions or transactions against a DuckDB catalog, both which also work in the remote test mode.

In Figure 7.2, the minor differences are visualized. These experiments show the limitations of code coverage as a metric for measuring the quality and effectiveness of a test suite. The tests for database transactions are undeniably useful, as are the tests for the other features, but their usefulness is not reflected in this metric. Adaptations like weighing in the number of hits per line might improve the metric in this regard, but increase its complexity and make it more difficult to understand.

Code coverage was a valuable metric for us since it led us to raise the number of DuckDB test scripts to run, which increases the chances of finding bugs and resulted in a test

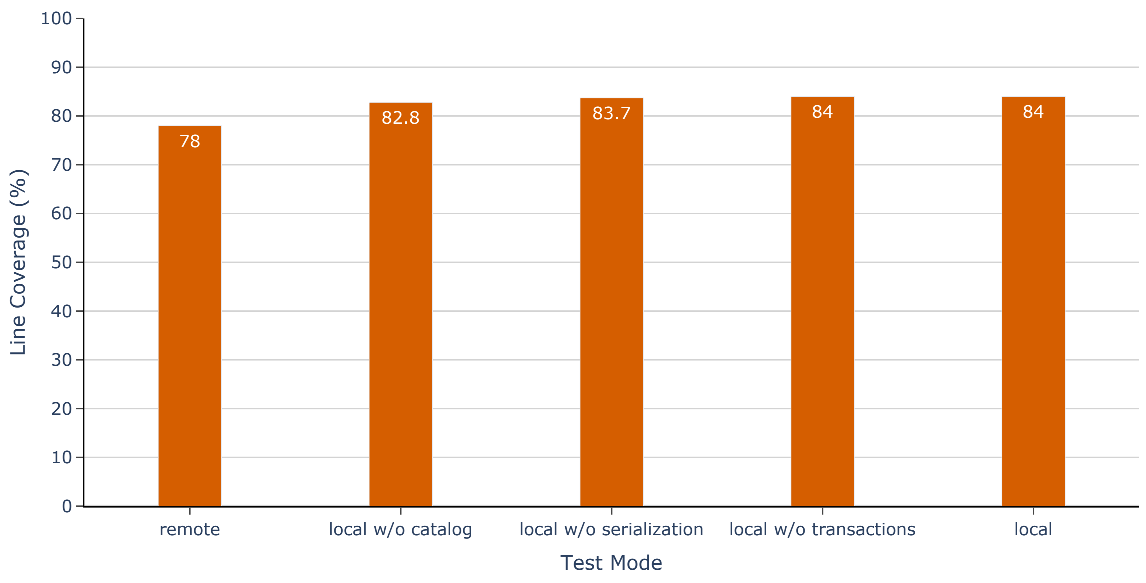


Figure 7.2: Coverage in local test mode when excluding test scripts related to unsupported aspects

suite that can give developers the confidence that executing queries remotely indeed works correctly. However, an even easier metric that could have helped us to achieve the same goal is simply the number of executed test scripts. This is a stance that other researchers take, too: In the next section, we provide a brief and incomplete summary of the debate on the correlation between code coverage and test quality.

7.2 Code Coverage to Measure Test Effectiveness

Code coverage is a metric widely used in practice to inform testers if the test suite can detect most of the software defects. And even though it cannot make a point about the quality of tests directly, increasing the coverage is hoped to increase the quality of the tests as well. However, this correlation between code coverage and test quality is hotly debated.

Del Frate et al. observed that an "[i]ncrease in code coverage is likely to increase reliability" [15], but the correlation between both aspects varied greatly between the tested programs. The data of Frankl and Iakounenko showed that "the likelihood of detecting a fault increased sharply as very high coverage levels were reached" [24]. And Kochhar, Thung, and Lo even reported a "statistically significant correlation between code coverage and bug kill effectiveness" [45] for real bugs. On the other hand, Cai and Lyu found that, even though code coverage results in good fault detection capabilities on error-paths in the code, this correlation is "weak in normal testing" [10]. Summaries of more work in this area can be found in [51] and [38].

In a large study on 31,000 test suites for five different programs, Inozemtseva and Holmes concluded that a higher effectiveness of a test suite can be better explained by its size. They only found a "low to moderate correlation between the coverage of a test suite and its effectiveness when its size is controlled for" [38]. This finding aligns with our observation that the attempt to solely increase the number of executed tests would have steered our work in a similar direction as did code coverage and would have been even easier to measure.

In conclusion, we have the impression that we are hitting the limitations of the code coverage metric. It is still possible to improve the tooling around it and, for example, rely on branch instead of line coverage to get a more granular picture or exclude error branches from the measurement to reduce the noise in the results. But despite this, other metrics, such as a mutation score (see Section 5.4.1) or real errors encountered during the use of the system, might provide more insightful information about possible improvements of the test suite.

These metrics can also make a statement about the quality of test assertions. That is, by testing if actual bugs are found, they can check if the tests actually test the right things. Test coverage is not able to provide this information [64].

Even though it might not be perfect, code coverage is still a useful metric, especially when writing new code. Since it is easy to collect automatically, we can continue using it without much effort.

7.3 Use of the Test Suite in Practice

The test suite that we describe in this thesis is actively used in practice by MotherDuck. The SQLLogic tests are automatically executed in local and remote test mode for every GitHub pull request, so that potential bugs are detected before the changes are merged into the main branch. They are also a vital part of the local development workflow, giving fast feedback on the correctness of recent code changes. Due to the high test coverage that was achieved, the test suite gives confidence that no existing functionality is broken and that the software is most likely to continue to work as it should.

The tests helped to identify problems with the existing code, too. To give an example, the tests helped to catch bugs where passing invalid arguments to a function that reads a CSV file would result in an internal error, which transitions the database into an invalid state, forcing the user to restart the DuckDB process or reloading the web app. The summary of the reasons for test failures that we describe in Section 6.5 was an invaluable tool to get an overview of what functionality is fully supported by MotherDuck.

With the test suite, updates of external dependencies can quickly be validated. This is especially important for DuckDB. MotherDuck tries to always support the latest released version of DuckDB. Since DuckDB is developed in the open, recent changes can already be integrated with MotherDuck, which prompts early adjustments on the side of MotherDuck and generates feedback for the developers of DuckDB. The SQLLogic tests are a very good indicator of what is working and what is not after updating to a new DuckDB version.

7.4 The Local Test Mode

One of the modes in which test scripts are executed is the local test mode, which runs SQL statements against a local database without connecting to the server. The general idea of this test mode was to verify that loading the HQE extension into DuckDB does not produce any correctness regressions. However, we are grappling with the question of whether this mode offers any substantial benefits.

Although the local test mode has been helpful in identifying certain bugs, it is unclear whether it represents a realistic scenario, because the typical workflow of users involves directly connecting to the server.

The cost of the local test mode, beside longer test execution times, is that connections to the server have to be avoided during the test run. Solely for the purpose of testing, this has necessitated inserting specialized code in SQL functions that would normally establish such a connection. As such, the perceived value of the local test mode has diminished.

7.5 Using an External Test Runner

The first step $\mathbf{S_1}$ of our work was to use DuckDB's test runner. The code for it is not under our direct control but part of the DuckDB codebase. This separation poses certain challenges, particularly in terms of modifying the test runner to suit our specific needs. In order to implement changes, we have to create extension points within the test runner and upstream these modifications to DuckDB.

The alternative process would be to fully integrate the test runner code into our codebase. But this implies that changes to the test runner on the DuckDB side have to be laboriously merged.

Hence, despite some limitations, the use of an external test runner is by and large beneficial. In particular, since the test runner is maintained as part of DuckDB, the compatibility with the test suite is ensured while reducing our own workload.

8

Conclusion

In this master's thesis, we developed a test suite for hybrid query execution. Hybrid query execution is a novel form of query processing that utilizes the resources available on local computers by allowing them to participate in the query processing, which would otherwise take place purely on a server. Parts of a single query can be executed on the local database client while other parts are processed on the server, aiming to bring the computation close to the data. Hybrid query execution has been implemented on top of DuckDB, an embeddable, analytical DBMS.

8.1 A Final Look at Our Research Questions

Our goal was to create a test suite that can ensure high software quality of and inspire confidence in this implementation. Rather than designing such a test suite from scratch, we investigated how we can leverage an existing test suite, namely the one for DuckDB (**RQ1**). This allowed us to quickly attain a large number of tests and achieve a high coverage of important DBMS functionality. While it is possible that tests designed from first principles might have proven more targeted, finding bugs that are more specific to hybrid query execution itself and with fewer tests, achieving a comparable coverage in the same time frame would have been challenging.

The existing DuckDB test suite consists mainly of SQL-based test scripts that are interpreted by a test runner. In the new test suite, these test scripts are executed in two modes. In the first mode, SQL statements are run against a local database, in the second mode against a remote database on the server. This helps to verify that hybrid query execution functions correctly regardless of where the data is placed. However, a limitation of our approach is that these two modes do not encompass all possible modes of execution.

In particular, more complex data exchange patterns between client and server are currently not tested.

Our work in adapting the test suite for hybrid query execution involved a series of steps, all of which we describe in Chapter 6. Two key steps in this process included replacing a custom test runner with DuckDB's own test runner and enabling many test scripts by executing them independently from each other. Furthermore, we were able to create an overview of known errors along the way.

We wanted to know how effective our steps to adapt the DuckDB test suite were (**RQ2**). For this, we measured the percentage of lines of code executed (covered) during a test run. Our implementation steps were guided by measuring the code coverage achieved in the previous step and, in general, identifying actions that can increase the coverage further.

In the process, we increased the line coverage for the entire codebase from 44% to 84%. In the local test mode, the coverage of the DuckDB code even jumped from 21% to 84%. The test suite provides great value in practice, too. The tests are executed whenever new changes are integrated into the main branch, giving developers—thanks to the high code coverage—the confidence they need to quickly iterate and improve the existing software. Bugs can be detected early in the process, which keeps the cost of fixing them low and raises the quality of the whole system.

Using the metric of code coverage led us mainly to get more and more test scripts to run. Hence, we might have achieved similar results simply by aiming to increase the size of the test suite. The metric also caused a considerable amount of noise and failed especially at the end to capture the progress made by enabling more test scripts, even though the new tests can arguably find different errors than other tests. A limitation is that we did not measure any other metric that expresses the effectiveness of our tests.

Overall, the strategy to build on top of an existing test suite proved successful, since it gave us a high number of tests and good test coverage quickly and with reasonable effort. It helped to identify some bugs and raise the quality of the system. A big success was especially that we could aggregate an overview of existing bugs and unsupported features. This greatly helped to prioritize the work of the team.

Our work revolved around testing the correctness of the hybrid-execution implementation. Other aspects like reliability or performance were outside the scope of this thesis but are very important work, too. Future work should be informed by real-world experience with the test suite, and an assessment of what is missing or how production bugs could have been prevented. Still, we propose some improvements in the next section.

8.2 Future Improvements to the Test Suite

High code coverage could be achieved by only having a single type of tests. But there is value in executing various types of them, since each comes with strengths and weaknesses. In the following sections we discuss future improvements we can imagine for the test suite for hybrid query execution, also going beyond our focus on functional correctness.

8.2.1 Testing “Full Hybrid” Execution

In our test suite, we run test scripts in two test modes: local and remote. This means that all data for a test is usually in a single place. But one important use of hybrid execution is to combine data from local and remote sources, something that is not tested by our current test suite.

For hybrid execution to operate as intended, not only do client and server have to work correctly, but their interplay must also proceed properly. However, most interactions between both nodes are rather simple in the remote test mode: they just exchange data at the beginning or the end of the query processing.

A third test mode could promote more complex interactions by executing SQLLogic test scripts “fully hybridly”. For this, the location of newly created database objects would have to alternate between the client and the server. For example, the first table created during a test could be stored in a local database, the second table in a remote one.

However, for this test mode to generate insightful results, it requires test scripts that combine data from different sources. In terms of SQL, it requires tests that use binary operations such as cartesian products, `JOIN`, or the set operations `UNION`, `INTERSECT` and `EXCEPT`. One input to this operation then has to come from a local, the other from a remote database. Hybrid execution would add a bridge operator to the query plan for the purpose of data transfer.

A quick analysis of the DuckDB test suite showed that there are around 350 files that meet these requirements, containing in total approximately 1500 relevant SQL statements. But most of these statements have a very similar shape, such as an equi-join of two tables.

To give confidence that the client and server work smoothly together, many *different* interactions should be tested. In other words, tests should produce many query plans that have different numbers and combinations of bridges for the data transfer. For example, a query during which data is simply shipped from the server to the client should work just as correctly as a query with a lot of back and forth between both nodes.

It is unclear if the DuckDB tests are the right starting point for these tests, since they were not developed with data placement in mind. Hence, they are also not designed to exhibit complex client-server interactions. A better way to test those might then be to write tests from scratch that lead to the desired behavior.

8.2.2 More Diverse Workloads

A relatively static test suite is an excellent tool for regression testing, i.e., finding errors that were introduced by recent changes. But some bugs might go into the system undiscovered and lurk there for a long time, never found by the tests since the tests always execute the same code.

Fuzzing is a testing method in which queries are randomly generated and the DBMS is automatically inspected for unexpected behavior. Usually, the generation is directed toward covering a large share of the source code [89; 3]. By continuously fuzzing the system, a much bigger part of its surface area can be examined. Therefore, bugs can be found that stayed hidden from the normal test suite.

DuckDB already uses randomized tests, and this would be a nice addition to the test suite for the HQE extension, too. But in addition, the randomly generated queries can also be executed in different modes such as remotely or "fully hybridly". Since a query result should remain the same regardless of where the data is located, the test oracle problem (see Section 2.5) is straightforward to solve. A test oracle is some mechanism or method to determine if the result produced by the DBMS is actually correct or not, something that is not immediately clear when issuing random queries. But under the assumption that core DuckDB processes queries correctly, one can compare the results of the different test modes including the local one to identify problems.

Fuzzing can generate complex queries, but they will likely be different from the workloads that the system encounters in production. It would therefore be valuable to test the system under realistic conditions. One way to do this is to rerun production workloads and verify that the results did not change. Snowflake [90] and Treasure Data [80] use previously captured workloads for their testing.

These workloads are not only helpful to find correctness bugs, but also to identify performance problems caused by recent changes.

8.2.3 Testing Non-Functional Requirements

In this thesis, we were only concerned with testing for correctness. However, this is only one of many quality attributes of a software system, some of which we mentioned in Section 2.4. Other quality requirements that are definitely important for DBMSs are performance and reliability.

DuckDB regularly runs a representative set of tests, including the TPC-H and TPC-DS benchmarks. When the difference in the execution time to the previous run is above a certain threshold, the problem is reported.

Something similar can also be conceived for the HQE extension. But the additional advantage of running a cloud-based service is that all server-side queries are executed in one central logical location. Hence, anonymized real-world workloads can be used to measure the improvement or decline of the system's performance.

One method to test the reliability of the system is to simulate faults and observe if the system can tolerate them. For example, by using an artificial network between the DBMS client and server in the tests, it is possible to inject network errors such as a high delay or high packet-loss rate. Additionally, since the server can be fully controlled during the test execution, it is also possible to examine how the whole system behaves when the server "unexpectedly" terminates and if SQL statements are still processed successfully.

A third reliability test method is stress testing: the load that the system has to handle is increased to a very high point while monitoring the resource utilization of the server and other parts of the system. For example, many complex SQL statements could be issued simultaneously. To run those tests, an environment that matches the production environment closely is desirable to exclude the influence of non-relevant factors and identify real issues.

Some of this work has already started, and we are looking forward to finding more errors in more areas before our users do.

References

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. “Column-Stores vs. Row-Stores: How Different Are They Really?” In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, BC, Canada: ACM, 2008, pp. 967–980 (9).

[2] Anastasia Alamaki and Ippokratis Pandis. “Query Processor.” In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, 2016, pp. 1–2. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_676-2` (visited on 06/16/2023) (10).

[3] Jinsheng Ba and Manuel Rigger. “Testing Database Engines via Query Plan Guidance.” In: *2023 IEEE/ACM 45th International Conference on Software Engineering*. ICSE ’23. IEEE, May 2023, pp. 2060–2071 (74).

[4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. “What It Would Take to Use Mutation Testing in IndustryA Study at Facebook.” In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’21’. Virtual Event, Spain, May 2021, pp. 268–277 (46).

[5] Ralf Bierig, Stephen Brown, Edgar Galván, and Joe Timoney. *Essentials of Software Testing*. Cambridge University Press, Aug. 19, 2021 (19–22, 41, 42).

[6] Peter Boncz. “Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.” PhD thesis. Universiteit van Amsterdam, May 2002 (9).

[7] Peter Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution.” In: *Proceedings of the 2005 CIDR Conference*. CIDR 2005. 2005 (15).

[8] Nicolas Bruno and Surajit Chaudhuri. “Flexible Database Generators.” In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB ’05. Trondheim, Norway: VLDB Endowment, 2005, pp. 1097–1107 (23).

[9] Timothy A. Budd, Richard J. Lipton, Richard A. DeMillo, and Frederick G. Sayward. *Mutation Analysis*. Tech. rep. Apr. 1979 (46).

[10] Xia Cai and Michael R. Lyu. “The Effect of Code Coverage on Fault Detection under Different Testing Profiles.” In: *Proceedings of the 1st International Workshop on Advances in Model-Based Testing.* A-MOST ’05. St. Louis, Missouri: ACM, 2005, pp. 1–7 (67, 68).

[11] Edgar F. Codd. “A Relational Model of Data for Large Shared Data Banks.” In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387 (6).

[12] George P. Copeland and Setrag N. Khoshafian. “A Decomposition Storage Model.” In: *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’85. Austin, Texas, USA: ACM, 1985, pp. 268–279 (9).

[13] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. “Differentially Testing Database Transactions for Fun and Profit.” In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* ASE ’22 35. Rochester, MI, USA: ACM, Oct. 10, 2022, pp. 1–12 (22).

[14] Christopher J. Date. *An Introduction to Database Systems.* 8th ed. Addison Wesley, July 22, 2003 (8).

[15] Fabio Del Frate, Praerit Garg, Aditya P. Mathur, and Albert0 Pasquini. “On the correlation between code coverage and software reliability.” In: *Proceedings of Sixth International Symposium on Software Reliability Engineering.* ISSRE ’95. Toulouse, France: IEEE, 1995, pp. 124–132 (67).

[16] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer.” In: *Computer* 11.4 (Apr. 1978), pp. 34–41 (46).

[17] DuckDB Contributors. *Client APIs Overview.* DuckDB Foundation. URL: `https://duckdb.org/docs/api/overview` (visited on 08/07/2023) (18).

[18] DuckDB Contributors. *Why DuckDB.* DuckDB Foundation. URL: `https://duckdb.org/why_duckdb` (visited on 08/07/2023) (18).

[19] Phil Eaton and Joran Dirk Greef. *We Put a Distributed Database In the Browser And Made a Game of It!* July 11, 2023. URL: `https://tigerbeetle.com/blog/2023-07-11-we-put-a-distributed-database-in-the-browser/` (visited on 08/08/2023) (45).

[20] Tom Ebergen. *The Return of the H2O.ai Database-like Ops Benchmark.* DuckDB Foundation. Apr. 14, 2023. URL: `https://duckdb.org/2023/04/14/h2oai.html` (visited on 08/02/2023) (30).

[21] Mostafa Elhemali and Leo Giakoumakis. “Unit-Testing Query Transformation Rules.” In: *Proceedings of the 1st International Workshop on Testing Database Systems.* DBTest ’08 3. Vancouver, BC, Canada: ACM, June 13, 2008, pp. 1–6 (22).

[22] David Farley. *Modern Software Engineering. Doing What Really Works to Build Better Software Faster.* Pearson Education, Limited, 2022 (18).

[23] Donald Firesmith. "Are Your Requirements Complete?" In: *The Journal of Object Technology* 4.1 (Jan. 2005), pp. 27–43 (19).

[24] Phyllis G. Frankl and Oleg Iakounenko. "Further Empirical Studies of Test Effectiveness." In: *SIGSOFT Softw. Eng. Notes* 23.6 (Nov. 1998), pp. 153–162 (67).

[25] Henning Funke, Jan Mühlig, and Jens Teubner. "Efficient Generation of Machine Code for Query Compilers." In: *Proceedings of the 16th International Workshop on Data Management on New Hardware.* DaMoN '20. Portland, OR, USA: ACM, 2020, 6 (14).

[26] Borko Furht. "Cloud Computing Fundamentals." In: *Handbook of Cloud Computing.* Ed. by Borko Furht and Armando Escalante. Springer, 2010. Chap. 1 (33).

[27] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems. The Complete Book.* 2nd ed. Pearson, Jan. 11, 2011 (6).

[28] Amrit L. Goel. "Software Reliability Models: Assumptions, Limitations, and Applicability." In: *IEEE Transactions on Software Engineering* SE-11.12 (Dec. 1985), pp. 1411–1423 (46).

[29] John B. Goodenough and Susan L. Gerhart. "Toward a Theory of Test Data Selection." In: 10.6 (Apr. 1975), pp. 493–510 (19, 41).

[30] Goetz Graefe. "Buffer Pool." In: *Encyclopedia of Database Systems.* Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, Dec. 5, 2016, pp. 1–3. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_682-2` (11).

[31] Goetz Graefe. "Query Evaluation Techniques for Large Databases." In: *ACM Computing Surveys* 25.2 (June 1993), pp. 73–169 (16).

[32] Goetz Graefe. "Storage Manager." In: *Encyclopedia of Database Systems.* Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, Dec. 8, 2016, pp. 1–6. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_678-2` (11).

[33] Goetz Graefe and William J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search." In: *Proceedings of the Ninth International Conference on Data Engineering.* Vienna, Austria: IEEE Computer Society, Apr. 19, 1993, pp. 209–218 (14).

[34] Theo Härder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery." In: *ACM Computing Surveys* 15.4 (Dec. 1983), pp. 287–317 (11, 56).

[35] Sung Hsueh and Arvind Ranasaria. “Cross Feature Testing in Database Systems.” In: *Proceedings of the 1st International Workshop on Testing Database Systems*. DBTest ’08 4. Vancouver, BC, Canada: ACM, June 13, 2008, pp. 1–5 (22).

[36] Marnie L Hutcheson. *Software Testing Fundamentals*. Nashville, TN, USA: Wiley, Apr. 11, 2003 (46).

[37] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. “MonetDB: Two Decades of Research in Column-oriented Database Architectures.” In: *IEEE Data Engineering Bulletin* 35.1 (Mar. 2012), pp. 40–45 (9).

[38] Laura Inozemtseva and Reid Holmes. “Coverage is Not Strongly Correlated with Test Suite Effectiveness.” In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 435–445 (68).

[39] *ISO/IEC 25010:2011. Systems and software engineering Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models*. Standard. International Organization for Standardization, Mar. 2011 (19).

[40] *ISO/IEC 9075-2:2016. Information technology Database languages SQL Part 2: Foundation (SQL/Foundation)*. Standard. International Organization for Standardization, Dec. 2016 (49).

[41] *ISO/IEC 9075:2023. Information technology Database languages SQL*. Standard. International Organization for Standardization, June 2023 (6, 49).

[42] *ISO/IEC/IEEE 24765:2017. Systems and software engineering Vocabulary*. Standard. International Organization for Standardization, Sept. 2017. URL: `https://standards.iso.org/ittf/PubliclyAvailableStandards/c071952_ISO_IEC_IEEE_24765_2017.zip` (visited on 07/26/2023) (19).

[43] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. “Software Test Automation in Practice: Empirical Observations.” In: *Advances in Software Engineering* 2010 (Jan. 2010) (20).

[44] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. “Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask.” In: *Proceedings of the VLDB Endowment* 11.13 (2018), pp. 2209–2222 (14, 15).

[45] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. “Code coverage and test suite effectiveness: Empirical study with real bugs in large systems.” In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. SANER. Montreal, QC, Canada, 2015, pp. 560–564 (67).

[46] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, and Alfons Kemper. “Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines.” In: *The VLDB Journal* 29 (2020 2019), pp. 757–774 (15).

[47] Allison Lee, Mohamed Zait, Thierry Cruanes, Rafi Ahmed, and Yali Zhu. “Validating the Oracle SQL Engine.” In: *Proceedings of the Second International Workshop on Testing Database Systems.* DBTest ’09 4. Providence, Rhode Island: Association for Computing Machinery, 2009 (23).

[48] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. “Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age.” In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’14. Snowbird, UT, USA: Association for Computing Machinery, 2014, pp. 743–754 (17).

[49] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. “How Good Are Query Optimizers, Really?” In: *Proceedings of the VLDB Endowment* 9.3 (Nov. 2015), pp. 204–215 (14, 30).

[50] Yu Liang, Song Liu, and Hong Hu. “Detecting Logical Bugs of DBMS with Coverage-based Guidance.” In: *31st USENIX Security Symposium.* USENIX Security 22. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 4309–4326 (23).

[51] Michael R. Lyu. “Software Reliability Engineering: A Roadmap.” In: *Future of Software Engineering.* FOSE ’07. Minneapolis, MN, USA, 2007, pp. 153–170 (68).

[52] Volker Markl. “Query Processing (In Relational Databases).” In: *Encyclopedia of Database Systems.* Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, 2016, pp. 1–6. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_296-2` (11, 12).

[53] Gerard Meszaros. *xUnit Test Patterns. Refactoring Test Code.* Boston, MZ, USA: Addison-Wesley, 2007 (44).

[54] Priti Mishra and Margaret H. Eich. “Join Processing in Relational Databases.” In: *ACM Computing Surveys* 24.1 (Mar. 1992), pp. 63–113 (13).

[55] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* 3rd ed. John Wiley & Sons, Nov. 8, 2011 (20–22).

[56] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance. Theory and Practice.* Wiley, Feb. 7, 2008 (18).

[57] Phil Nash. *Command line: Specifying which tests to run.* Catch2. URL: `https://github.com/catchorg/Catch2/blob/v2.x/docs/command-line.md#specifying-which-tests-to-run` (visited on 08/09/2023) (53).

[58] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware.” In: *Proceedings of the VLDB Endowment* 4.9 (June 2011), pp. 539–550 (14).

[59] Thomas Neumann. “Query Optimization (in Relational Databases).” In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, 2016, pp. 1–7. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_293-2` (14).

[60] Srinivas Nidhra and Jagruthi Dondeti. “Black Box and White Box Testing Techniques - A Literature Review.” In: *International Journal of Embedded Systems and Applications* 2.2 (June 2012), pp. 29–50 (21).

[61] Roy Osherove. *The Art of Unit Testing. With examples in C#*. 2nd ed. New York, NY, USA: Manning, Dec. 2013 (44).

[62] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. 4th ed. Springer Nature Switzerland, 2020 (33).

[63] Pavlech Juraj and Commentators. *Confused with branch coverage*. Linux Test Project on GitHub. URL: `https://github.com/linux-test-project/lcov/issues/75` (visited on 08/12/2023) (47).

[64] Goran Petrovi, Marko Ivankovi, Gordon Fraser, and René Just. “Does Mutation Testing Improve Testing Practices?” In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE ’21. Madrid, Spain: IEEE Press, Nov. 5, 2021, pp. 910–921 (46, 68).

[65] Goran Petrovi, Marko Ivankovi, Gordon Fraser, and René Just. “Practical Mutation Testing at Scale: A view from Google.” In: *IEEE Transactions on Software Engineering* 48.10 (Oct. 2022), pp. 3900–3912 (46).

[66] Evaggelia Pitoura. “Pipelining.” In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, 2016, pp. 1–2. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_872-2` (16).

[67] Evaggelia Pitoura. “Query Optimization.” In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, 2016, pp. 1–2. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_861-2` (visited on 06/16/2023) (12, 14).

[68] Evaggelia Pitoura. “Query Plan.” In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York City, NY, USA: Springer, 2016, pp. 1–2. URL: `https://link.springer.com/referenceworkentry/10.1007/978-1-4899-7993-3_864-2` (visited on 06/16/2023) (10, 11).

[69] Mark Raasveldt. *DuckDB The SQLite for Analytics*. CMU Database Group. Apr. 20, 2020. URL: `https://www.youtube.com/watch?v=PFUZlNQIndo` (visited on 08/07/2023) (18).

[70] Mark Raasveldt. *DuckDB Testing - Present and Future*. DBTest Workshop. June 17, 2022. URL: `https://www.youtube.com/watch?v=BgC79Zt2fPs` (visited on 07/26/2023) (26, 28).

[71] Mark Raasveldt. *State of the Duck DuckCon #3 (San Francisco)*. DuckDB. June 29, 2023. URL: `https://www.youtube.com/watch?v=LlkEna0kzdk` (visited on 08/07/2023) (18).

[72] Mark Raasveldt and Contributors. *Pragmas - DuckDB*. DuckDB Foundation. URL: `https://duckdb.org/docs/sql/pragmas.html` (visited on 07/26/2023) (28).

[73] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database.” In: *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, The Netherlands). SIGMOD ’19. New York, NY, USA: ACM, June 2019 (18).

[74] Tilmann Rabl and Meikel Poess. “Parallel Data Generation for Performance Analysis of Large, Complex RDBMS.” In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. DBTest ’11 5. Athens, Greece: ACM, 2011 (23).

[75] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd ed. McGraw-Hill, Aug. 14, 2002 (8).

[76] Rudolf Ramler and Klaus Wolfmaier. “Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost.” In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. AST ’06. Shanghai, China: Association for Computing Machinery, 2006, pp. 85–91 (20).

[77] Manuel Rigger and Zhendong Su. “Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction.” In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: ACM, Nov. 8, 2020, pp. 1140–1152 (23).

[78] Manuel Rigger and Zhendong Su. “Finding Bugs in Database Systems via Query Partitioning.” In: *Proceedings of the ACM on Programming Languages* 4.211 (OOPSLA Nov. 13, 2020), pp. 1–30 (23).

[79] Manuel Rigger and Zhendong Su. “Testing Database Engines via Pivoted Query Synthesis.” In: *14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’20. USENIX Association, Nov. 2020, pp. 667–682 (23).

[80] Taro L. Saito, Naoki Takezoe, Yukihiro Okada, Takako Shimamoto, Dongmin Yu, Suprith Chandrashekharachar, Kai Sasaki, Shohei Okumiya, Yan Wang, Takashi Kurihara, Ryu Kobayashi, Keisuke Suzuki, Zhenghong Yang, and Makoto Onizuka. "Journey of Migrating Millions of Queries on The Cloud." In: *Proceedings of the 2022 Workshop on 9th International Workshop of Testing Database Systems*. DBTest '22. Philadelphia, PA, USA: ACM, June 17, 2022, pp. 10–16 (23, 74).

[81] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill, Feb. 19, 2019 (5–7, 10, 11, 13–16, 33, 34).

[82] Andreas Spillner and Tilo Linz. *Software Testing Foundations. A Study Guide for the Certified Tester Exam*. Ed. by Michael Barabas and Christa Preisendanz. 5th ed. Rocky Nook, Sept. 28, 2021 (19, 42).

[83] Michael Stonebraker. "One Size Fits All: An Idea Whose Time Has Come and Gone." In: *Communications of the ACM* 51.12 (Dec. 2008), p. 76 (31).

[84] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. "The Train Benchmark: cross-technology performance evaluation of continuous model queries." In: *Software & Systems Modeling* 17.4 (Jan. 2017), pp. 1365–1393 (30).

[85] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. "The LDBC Social Network Benchmark: Business Intelligence Workload." In: *Proceedings of the VLDB Endowment* 16.4 (Dec. 2022), pp. 877–890 (30).

[86] Valerio Terragni, Pasquale Salza, and Mauro Pezzè. "Measuring Software Testability Modulo Test Quality." In: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 241–251 (46).

[87] Florian M. Waas, Leo Giakoumakis, and Shin Zhang. "Plan Space Analysis: An Early Warning System to Detect Plan Regressions in Cost-Based Optimizers." In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. DBTest '11 2. Athens, Greece: ACM, June 13, 2011, pp. 1–6 (22).

[88] Gursimran Singh Walia and Jeffrey C. Carver. "A Systematic Literature Review to Identify and Classify Software Requirement Errors." In: *Information and Software Technology* 51.7 (July 2009), pp. 1087–1109 (19).

[89] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. "Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP. Madrid, Spain: IEEE, May 2021, pp. 328–337 (74).

[90] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D. Viglas, and Allison Lee. “Snowtrail: Testing with Production Queries on a Cloud Database.” In: *Proceedings of the Workshop on Testing Database Systems*. DBTest ’18 4. Houston, TX, USA: ACM, June 15, 2018, pp. 1–6 (23, 74).

[91] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. “SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback.” In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. Virtual Event, USA: ACM, Oct. 30, 2020, pp. 955–970 (23).

[92] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. “FoundationDB: A Distributed Unbundled Transactional Key Value Store.” In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. Virtual Event, China: ACM, 2021, pp. 2653–2666 (45).