SparkFuzz: Searching Correctness Regressions in Modern Query Engines

Bogdan Ghiț
bogdan.ghit@databricks.com
Databricks Inc.

Nicolas Poggi
nicolas.poggi@databricks.com
Databricks Inc.

Josh Rosen
joshrosen@databricks.com
Databricks Inc.

Reynold Xin
rxin@databricks.com
Databricks Inc.

Peter Boncz
peter.boncz@cwi.nl
Centrum Wiskunde & Informatica

ABSTRACT

With more than 1200 contributors, Apache Spark is one of the most actively developed open source projects. At this scale and pace of development, mistakes are bound to happen. In this paper we present SparkFuzz, a toolkit we developed at Databricks for uncovering correctness errors in the Spark SQL engine. To guard the system against correctness errors, SparkFuzz takes a fuzzing approach to testing by generating random data and queries. SparkFuzz executes the generated queries on a reference database system such as PostgreSQL which is then used as a test oracle to verify the results returned by Spark SQL. We explain the approach we take to data and query generation and we analyze the coverage of SparkFuzz. We show that SparkFuzz achieves its current maximum coverage relatively fast by generating a small number of queries.

ACM Reference Format:
Bogdan Ghiț , Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. 2020. SparkFuzz: Searching Correctness Regressions in Modern Query Engines. In *Workshop on Testing Database Systems (DBTest'20), June 19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3395032.3395327

1 INTRODUCTION

Early data analytics frameworks such as MapReduce enabled users to simplify the execution of their big data workloads by means of a powerful, but low-level procedural programming interface. To cope with this limitation, systems such as Hive [24], Impala [19], and Spark SQL [13] expose relational interfaces to big data applications, thus providing richer automatic optimizations. As a result, the design of mechanisms for improving the performance of data analytics systems is an active research area both in academia and industry [15, 16, 25]. With an increasingly complex architecture, such systems are difficult to test with good coverage in practice. Developers are at risk to incorporate bugs, which may not only negatively impact the system performance, but may also alter the correctness of the results. In this paper we present the design and implementation of SparkFuzz, a toolkit for automatically generating SQL test cases which consist of random data and queries.

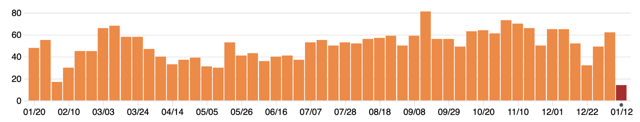

Figure 1: The distribution of code contributions in the Apache Spark open source project in the trailing year.

With powerful processing features and simple programming interface catalyzing its wide adoption, Spark has recently become the de facto framework for big data analytics [21]. Figure 1 shows that the Spark open source code base changes at a pace of tens of commits per day and so, mistakes are bound to happen. To guard the framework against errors, developers add unit tests which often results in a significant engineering effort. Spark has roughly the same amount of source and testing code. The effectiveness of such tests is however relatively low because they are restricted to specific operations on fixed inputs which cannot cover all possible code paths. Furthermore, data analytics frameworks are also prone to relatively high variability when the input dataset changes [17]. Therefore, standard testing techniques fail to capture data-dependent runtime interactions in these frameworks.

Catalyst, the Spark query optimizer, employs pattern-matching to express composable rules in a Turing-complete language, while offering a general framework for transforming trees. Catalyst modifes the user queries through tree transformations which are called rules. Such rules are grouped into multiple batches which are executed until the query plan reaches a fixed point – the tree stops changing after applying the same set of rules. Combining the supported set of rules in different ways typically diversifies the generated code paths and uncover regressions that may remain hidden otherwise. Testing all possible combinations of rules is however a daunting task for developers, and so the existing testing framework in Spark only includes tests with all implemented rules enabled.

Even worse, data analytics frameworks are notoriously difficult to setup because they expose many configuration parameters which add an exponentially growing number of code paths. As a result, the typical testing matrix when developing a new feature suddenly becomes a multi-dimensional testing space. Unlike other data processing frameworks, Spark also exposes an extensive API with more than 200 SQL operators. In order to address these challenges we

DBTest'20, June 19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8001-0/20/06…$15.00
https://doi.org/10.1145/3395032.3395327

want to employ fuzzing which is a well-known technique for improving testing coverage in software systems. With fuzzing we are able to (re-)generate multiple queries and input datasets of different sizes. In contrast, unit tests only support a few queries running on a relatively small dataset.

We propose SparkFuzz, an automatic test case generator for the Spark SQL engine. SparkFuzz provides correctness guarantees by checking the results of a test case against the PostgreSQL reference database implementation [5]. At the core, SparkFuzz consists of two complementary mechanisms. Firstly, it enables automatic generation of columnar-oriented datasets by randomly sampling across the supported data types to generate table schemas and then filling those tables with random data. Secondly, SparkFuzz employs a recursive SQL model to construct a query profile with features consisting of operators and clauses. All features are annotated with weights used to calculate their probability of being sampled during the test case generation. SparkFuzz is an external tool that connects to a running Spark instance to load the generated data and execute random queries. Finally, SparkFuzz can also randomize the configuration space of a Spark deployment by randomly toggling optimization rules and sampling over valid parameter ranges.

In this paper we make the following contributions:

- We design SparkFuzz, a toolkit that automates the discovery of bugs in Spark by randomly generating data and queries. SparkFuzz takes a *data-before-query* generation approach and enables two testing modes by comparing query results either against a reference database implementation or against different instances of Spark SQL.
- We deploy SparkFuzz in production and show that it achieves maximum coverage after generating a small number of queries. Being able to produce more diverse test cases in a shorter time is promising and recommends SparkFuzz for testing code during development.

2 BACKGROUND

Many SQL testing techniques are based on comparison tests, which compare the system-under-test output with a reference result. A well-tested relational database management system is SQLite which employs a comprehensive list of testing techniques to achieve reliability and robustness [11]. In this section we present an overview of the most common techniques used in SQL testing.

Fuzz Testing. In general, fuzzing is an automated software testing technique that provides random input data and monitors whether the tested program manifests unexpected behaviours such as crashes, exceptions, and invalid outputs. A fuzzer can be either generation-based [18] or mutation-based [20] depending on whether the input data is generated from scratch or by modifying a given input. Whereas SQL fuzzers such as RQG [6] need to be aware of their input and program structure, more general-purpose fuzzers may completely treat the testing environment as a black-box [3, 4].

Anomaly Testing. Anomaly tests seek to verify that the system exhibits a correct behavior even in situations when something goes wrong. Whereas, a system may behave correctly on well-formed inputs, often it is more difficult to respond and operate properly to invalid inputs. For instance, one may want to check whether the database system is able to gracefully handle out-of-memory

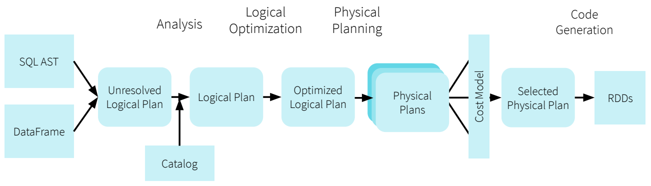

Figure 2: Catalyst has a complex design with pipelined stages that transform the query from a generic tree representation to RDDs.

errors, to respond safely to failed I/O operations, or to recover from operating-system crashes at runtime [11].

Boundary Value Testing. SQLite has well-defined operation limits such as the maximum number of columns a table may have or the maximum length of a SQL statement. Similarly, Spark has a large set of configuration parameters that have specific value domains. However, brute-force testing is often impractical because some input spaces are way too large to test exhaustively. Whereas brute-force may still be useful in specific cases such as testing every expression with simple literals of every input type, large classes of inputs are redundant because they behave equivalently. In contrast, boundary value testing aims to cover extreme cases of very large or very small inputs, including empty sets, infinity, null, and NaN. Additional tests may go beyond the defined limits and verify that the system correctly reports errors [4, 11, 14].

3 CHALLENGES AND GOALS

In this section, we explain how Catalyst transforms a SQL query in order to prepare it for execution on a cluster with many machines. Furthermore, we identify some of the most important dimensions in assessing correctness of a SQL engine.

3.1 Catalyst Overview

A Spark SQL query has an abstract representation called *query plan* which is converted through a sequence of transformations into a binary that can be distributed and executed on a cluster. These transformations are performed by Catalyst, which is a highly extensible framework that enables the addition of new optimizations. Figure 2 shows that Catalyst creates two types of query plans. The logical plan provides a high-level representation of the type of computation we want to perform on the input dataset. In particular, the logical plan denotes the join operator on two tables, but avoids defining how to perform the actual operation. This decision is deferred to physical planning during which the logical plan is annotated with specific instructions to execute the computation. For instance, if we join two tables one of which has less than 10 MB, the join operator is executed as a broadcast-hash join.

Catalyst drives the query from a basic abstract syntax tree to execution in three phases, i.e., analysis, optimization, and planning, each of which is a potential source of errors. The analyzer translates unresolved attributes and relations to fully-typed objects using a query session catalog. The analyzed logical plan feeds the optimizer which applies sets of optimization rules using a fixed-point policy until the plan stops changing. Similarly, the planner converts the logical plan into a physical plan through a set of planning strategies that determine how each computation will be executed. Spark takes a data-centric approach to query execution by running an additional stage to generate Java bytecode that is tailored to the physical plan of the query. In this way, each query has a fixed compiled plan that is used to process each row of the input dataset.

3.2 Assessing Correctness

The main challenge in assessing correctness is making sure we have a reference query execution that we know with high fidelity it behaves well. We use PostgreSQL as our test oracle in assessing the correctness of Spark SQL. Although PostgreSQL may also have bugs we are not aware of, it is a mature open source database system that is ANSI SQL compliant. Thus, we consider it is a relatively low chance to encounter test cases that provide identical but incorrect answers in Spark and PostgreSQL. In this section we present our goals by identifying several aspects that may impact the correctness of Spark SQL. In order to define correctness of Spark SQL we consider the following dimensions:

- **Correct answer.** Our primary goal is to validate the Spark SQL query answer with the one returned by the reference database. In order to have an accurate comparison between the results, we need to make sure our comparator captures any differences between the SQL dialects used by Spark SQL and PostgreSQL. Moreover, we further want to be able to validate whether the query returns the correct answer for different input data, Spark parameter values, operating systems, and cluster configurations.

- **Useful crashes.** Another goal we set is to make sure that each query executes without crashing at runtime. Thus, invalid queries should result in a clear analysis exception showing the reason of failure. We want Spark to reject invalid queries early on during analysis and avoid their execution.

- **Performance.** Finally, a more subtle goal is to catch queries that regress because they are not properly optimized due to internal bugs. For instance, bugs in Catalyst may lead to incorrect code generation when employing optimizations such as dynamic partition pruning [16]. Other interesting examples are bugs that cause the optimizer either to reach its iteration limit or to generate code that fails to compile because of exceeding JIT limits.

4 SOURCES OF ERRORS

Whereas most of the errors in software systems come from bugs introduced by developers, we argue that popular open source projects such as Spark are at risk of experiencing errors from other sources as well. Such errors may hinder both the developer productivity and the agile development of the system, and so we want an automatic way to guard Spark against them. Based on our experience with the Apache Spark open source project, we identify several sources of errors some of which are unrelated to Spark code changes:

- **Merge conflicts**. In a source-controlled project such as Spark, a developer typically contributes by creating new branches of the repository and working independently possibly for an extended period of time. At the same time, other developers may change some parts of the same code. This often results in merge conflicts which are typically resolved manually by a different person who is also a commiter or a reviewer of the project. Lack of exhaustive testing may lead to bugs caused by manual and automated merges. This problem is exacerbated in the case of *forks* which are full long-running versions of the project that might be merged at a later stage while both repositories evolved differently.

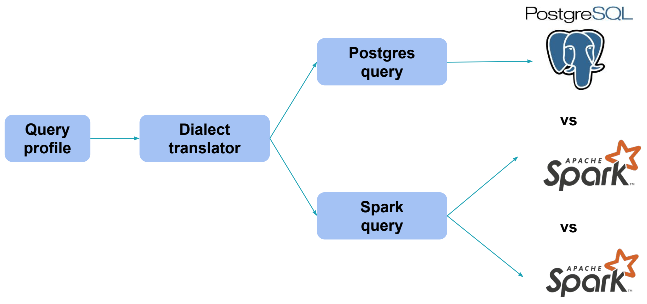

Figure 3: SparkFuzz enables two operation modes in which we validate query results either with PostgreSQL or with a different Spark version.

- **Code refactoring**. Code refactoring is typically a maintenance task that requires re-organizing parts of the project or features with the goal of simplifying the existing code. However, developers that perform refactoring may need to modify a large surface of the software but without having a deep understanding of the entire codebase. In this way they can incorporate bugs which may remain uncovered without adding unit tests that target the interactions between the modified components. For instance, reorganizing Spark's memory management API [8] introduced more virtual function calls which resulted in a performance regression.

- **Semantic changes**. In a fast-paced development environment, we often compromise standard compliance for fast, iterative development. However, lack of full compliance with the SQL standard may prevent some users to migrate their SQL workloads on Spark. As a consequence, the framework may face frequent changes of the API semantics for better alignment with the SQL standard. In particular, Spark minor release 2.4.0 corrected the behavior of the HAVING clause in the absence of a GROUP BY operator by considering it a global aggregate instead of a local aggregate [9].

- **External library updates**. The project incorporates dependencies to a myriad of external libraries that are necessary to manage internal operations from network communication to data compression. Because these libraries are updated and maintained externally, new releases may introduce unexpected bugs. For example, in the `spark-avro` data source package, an upstream library deviated from the schema in the specification, thus leading to validation exceptions on previously working code [7].

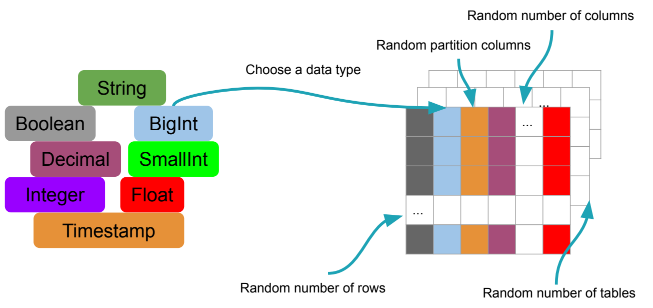

Figure 4: SparkFuzz generates input data by randomly constructing a table schema and then inserting rows with random data that match the schema.

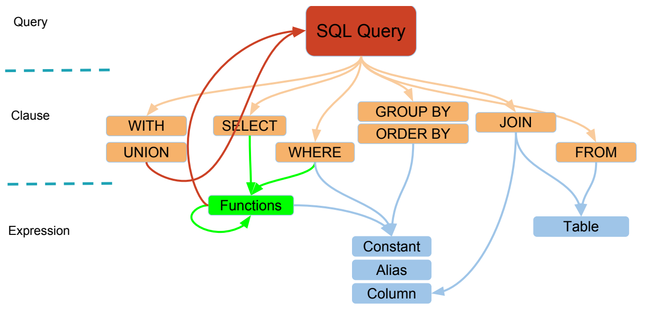

Figure 5: SparkFuzz uses a recursive SQL model to generate queries by randomly selecting clauses and expressions.

5 THE SPARKFUZZ FRAMEWORK

In this section we present the design of SparkFuzz, a framework for testing the correctness of Spark SQL against a reference database such as PostgreSQL. We present two complementary components for generating random datasets and queries.

5.1 Design Overview

Figure 3 highlights the two operation modes supported by SparkFuzz. In particular, we can compare the query result of Spark SQL versus a reference database system such as PostgreSQL that is assumed to be correct and would trigger a manual verification in case of a mismatch. This way of testing provides the strongest correctness guarantees, but it restricts the coverage of the generated queries to the set of features of the dialect implemented by the reference system. Another way of testing correctness is to compare the query results on multiple instances of the Spark SQL framework. In particular, we can compare the latest stable version with the development branch, different previous releases, or the same release with different framework configurations.

A straightforward technique which allows rapid testing of many queries is to first produce a database with randomly generated tables and then construct multiple queries over those tables. In this way we may miss deep data-dependent execution bugs because complex queries are likely to generate empty result sets on naively generated data. However, we prefer this approach because it's simple and offers the ability to increase operator coverage by quickly generating thousands of queries.

5.2 Data Generation

SparkFuzz supports generating datasets in different file formats such as Parquet, Delta, CSV, and ORC which can be set by the user or randomly selected by SparkFuzz. Figure 4 shows the steps we take to generate random datasets. Firstly, SparkFuzz randomly selects the number of tables to populate the dataset with. Whereas for most query patterns we only need a few tables, having a large number of tables may be required if we seek to generate queries that consist of many join operations. Secondly, we set the dimension of each table by randomly choosing the number of rows and the number of columns. Based on the number of columns of a given table SparkFuzz constructs a random schema using a predefined set of supported data types and fills each row of the table with random values according to this schema. Finally, SparkFuzz may randomly select partitioning columns in some of the generated tables.

```
SELECT
  GREATEST(COALESCE(t2.bigint_c5, 525),
  COALESCE(MIN(t2.bigint_c5), 116))
    AS int_col,
  COALESCE(MAX(MIN(-554)) OVER (),
    -654, -342) AS int_c1,
  t2.bigint_c5,
  (MAX(t1.decimal_c4)) != (t2.bigint_c5)
    AS boolean_col
FROM table_2 t1
LEFT JOIN table_6 t2 ON
  (t2.decimal0_c12) = (t1.decimal_c4)
GROUP BY t2.bigint_c5
```

Listing 1: An example of query generated with SparkFuzz.

5.3 Query Generation

In order to generate a query we need to define a query profile that includes all possible SQL features than can be used. At a high-level, a SQL query may consist of one or multiple clauses such as SELECT, FROM, GROUP BY, or UNION. In turn, each clause has one or multiple expressions that may include constants, columns, or functions. As Figure 5 shows, this model is recursive because multiple functions may be nested, whereas some clauses may include sub-queries. The query profile is extensible and allows adding new functions without changing the basic structure of the model.

Each clause and operator in the query profile is assigned a fixed probability that represents its chance of being selected when generating a query. For instance, to blacklist an operator during query generation we can set its weight 0 and so, it will never be selected. SparkFuzz sets a probability of 1 to the mandatory query clauses, i.e., SELECT and FROM. Furthermore, SparkFuzz requires setting additional inter-dependent weights to select a specific join operator from all possible join types (e.g., inner, left, or right). Another example of operators that have inter-dependent weights are the different classes of functions that can be employed (e.g., aggregation, analytic, or basic). Whereas the model can be used to control the logical

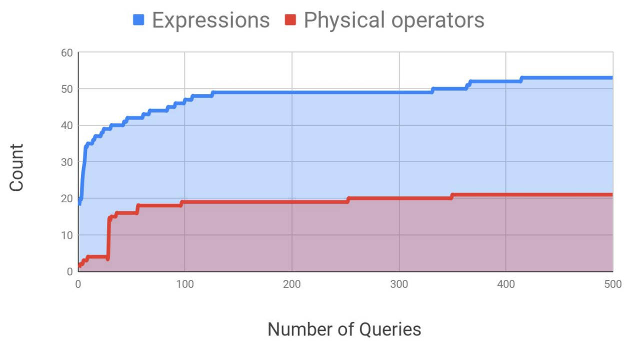

Figure 6: The evolution of the expression and physical operator coverage during an experiment with 500 queries generated and executed sequentially by SparkFuzz.

plan of the generated query, it has no control over the physical plan. To diversify the generated physical plans, SparkFuzz needs to randomize Spark configuration parameters. In particular, changing the maximum size in bytes for broadcasting a table may result in sets of queries that have different fractions of broadcast-hash joins and sort-merge joins.

In addition to the query profile, we use a dialect translator that specializes the generated query to a particular SQL dialect. We need such a translator to address differences between Spark SQL which is based on the Hive dialect and PostgreSQL which has its own SQL dialect. An example of a randomly generated query with SparkFuzz is shown in Listing 1. The query incorporates several SQL features and the attribute names in the query denote the table data types.

6 EXPERIMENTAL RESULTS

In this section, we assess the effectiveness and coverage of SparkFuzz on a typical setup that we have used to test and verify SQL correctness in Spark. We implemented our SparkFuzz prototype on top of Impala Random Query Generator [2] which we extended with the specific features of the Spark dialect and system configuration.

```
SELECT
COALESCE(COALESCE(foo.id, foo.val), 42)
FROM foo
GROUP BY
COALESCE(foo.id, foo.val)
```

Listing 2: The minimized SparkFuzz query that uncovered an incorrect expression simplification inside the GROUP-BY clause applied during optimization by Catalyst.

Within a year, SparkFuzz uncovered more than 10 analysis errors, 10 runtime crashes, and 20 wrong results. A recent bug we have found with SparkFuzz is an aggregation query in which the optimizer incorrectly simplifies the expression inside a GROUP-BY clause. As a result, the expression inside the SELECT clause is referencing neither a grouping nor an aggregate expression. Listing 2 shows the minimized version of the generated SparkFuzz query after optimization. We notice that the nested COALESCE expression

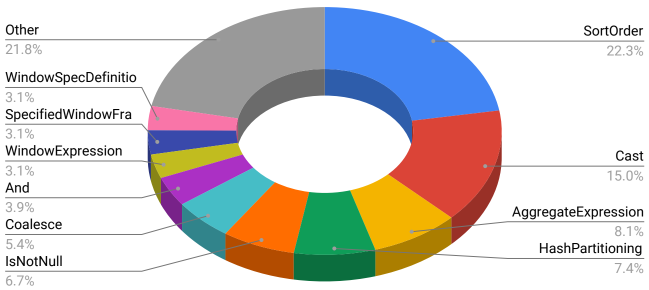

Figure 7: The distribution of Spark SQL expressions in a 500 query workload generated by SparkFuzz.

in the GROUP-BY clause is flattened by Catalyst, thus resulting in an analysis exception [10].

To analyze the operation of SparkFuzz, we deploy Spark (version 2.4) on a single node that runs the driver next to a PostgreSQL installation (version 9.5.19). Whereas in benchmarking it's important to run representative long-running queries, when testing correctness we need to maximize the number of queries we run within a relatively short amount of time. Therefore, in this experiment we use a single-node which is both cost-effective and sufficient for testing correctness. Furthermore, we generate a relatively small dataset with only 5 tables each of which has a 5-column wide schema and 5 rows. We executed similar test cases in terms of configuration and setup for the majority of bugs we found with SparkFuzz.

We generate 500 queries with SparkFuzz which we execute sequentially both on Spark and PostgreSQL while automatically comparing and reporting the results. Figure 6 shows how the number of distinct operators selected by SparkFuzz during query generation increases over time as more queries are generated. We find that running more queries over time increases the operator coverage. Nevertheless, SparkFuzz is able to achieve its current maximum coverage relatively fast after the first 100 queries are generated. In particular, SparkFuzz has a maximum operator coverage of roughly 50 expressions and 20 physical operators. We have restricted the current coverage of SparkFuzz to a common set of operators with the test oracle. Although obtaining the maximum operator coverage does not guarantee the correctness of the system, the metric can be used as a stop condition for limiting the total test time.

Figure 7 depicts the most frequent expressions generated by SparkFuzz excluding attribute references and aliases. We found sorting, casting, and aggregations to be the most popular expressions in our randomly generated workload. Because Spark is a general processing engine, it supports a larger number of SQL functions and provides extra functionality when compared to ANSI SQL. Therefore, in order to remain compatible with PostgreSQL we limited the coverage by restricting the query profile to a common set of operators and expressions. However, we aim to further increase the coverage of SparkFuzz through the alternative operation mode in which we compare results against different Spark versions. Although this mode provides weaker correctness guarantees, we will be able to generate random queries using the complete set of supported functions in Spark.

7 RELATED WORK

Whereas a wide array of prior work has focused on testing correctness of software systems [14, 22], there is little work on assessing the correctness of modern SQL engines such as Spark. In this section, we present an overview of existing fuzzing tools and approaches that are used to test SQL correctness.

One such approach is Apollo [18], which has been recently proposed to test standard database implementations such as SQLite and PostgreSQL against performance regressions when the system is upgraded to a more recent release. The Microsoft SQL Server group proposed RAGS [23] an automated testing framework for exploring functional bugs in database systems. RAGS operates by generating SQL statements through stochastic construction of parse trees based on the database schema. In contrast, Snowtrail [27] and Oracle SQL Performance Analyzer [26] define workloads for testing the system performance by monitoring if the execution exceeds a baseline performance threshold. Closest to our work are Impala RQG [2] and SQLsmith [1] which are SQL-aware fuzzers used to test Impala and CockroachDB, respectively. Both tools use PostgreSQL as an oracle to validate results, but they have limited coverage and are prone to false positives because of dialect differences.

In SparkFuzz we pregenerate the input tables and later on we construct arbitrary queries matching the input schemas of those tables. Another approach to data and query fuzzing is to generate queries before input data [12]. With this technique input tables are generated in such a way that certain queries over them will return a specified number of results. For instance, we can consider a join query and then generate two input tables so that the join returns a non-empty result set. A more advanced application of this technique is to define different cardinality constraints over different sub-plans of the query and generate tables that satisfy all constraints. However, this approach is time-consuming and may result in a small set of queries that satisfy all constraints.

8 CONCLUSIONS

In this paper we introduced SparkFuzz, a toolkit for testing the correctness of Spark SQL by means of random data and query generation. SparkFuzz first takes random schemas to generate input tables of random sizes, and employs a recursive query model with features that have fixed probabilities to construct queries. SparkFuzz validates the correctness of Spark SQL versus a reference database implementation such as PostgreSQL. Thus, SparkFuzz executes each generated query both on Spark SQL and PostgreSQL in order to detect differences in the result sets returned. We demonstrated that SparkFuzz achieves good coverage with relatively small query sets and input data. With SparkFuzz we uncovered tens of bugs that include analysis exceptions, runtime crashes, and incorrect results.

In future work, we want to use operator coverage metrics to limit the search space in SparkFuzz when generating queries. In this way, SparkFuzz can be used more frequently in testing code changes at the granularity of commits rather than releases. In addition, we aim to reduce the dependency on the test oracle and focus more on comparison between different versions of the Apache Spark framework. Thus, instead of using PostgreSQL to validate results, we will maintain a history of results for the same query and input data on different versions of Spark.

REFERENCES

[1] https://cockroachlabs.com/blog/sqlsmith-randomized-sql-testing.
[2] https://github.com/apache/impala.
[3] https://jepsen.io.
[4] https://github.com/google/oss-fuzz.
[5] https://postgresql.org.
[6] https://launchpad.net/randgen.
[7] https://issues.apache.org/jira/browse/SPARK-25002.
[8] https://issues.apache.org/jira/browse/SPARK-25317.
[9] https://issues.apache.org/jira/browse/SPARK-25708.
[10] https://issues.apache.org/jira/browse/SPARK-25914.
[11] https://sqlite.org/testing.html.
[12] A. Arasu, R. Kaushik, and J. Li. Data Generation Using Declarative Constraints. *ACM SIGMOD*, 2011.
[13] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational Data Processing in Spark. *ACM SIGMOD*, 2015.
[14] B. Beizer. *Software Testing Techniques*. Dreamtech Press, 2003.
[15] B. Ghit and D. Epema. Better Safe than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks. *ACM HPDC*, 2017.
[16] B. Ghit and J. Sompolski. Dynamic Partition Pruning in Apache Spark. *Spark+AI Summit Europe*, 2019.
[17] B. Ghita, D. Tome, and P. Boncz. White-box Compression: Learning and Exploiting Compact Table Representations. *CIDR*, 2020.
[18] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *VLDB Endowment*, 13(1):57–70, 2019.
[19] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. *CIDR*, 2015.
[20] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.
[21] S. Salloum, R. Dautov, X. Chen, P. Peng, and J. Huang. Big Data Analytics on Apache Spark. *Journal of Data Science and Analytics*, 2016.
[22] S. K. Singh and A. Singh. *Software Testing*. Vandana Publications, 2012.
[23] D. R. Slutz. Massive Stochastic Testing of SQL. *VLDB*, 98:618–622, 1998.
[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *VLDB Endowment*, 2(2):1626–1629, 2009.
[25] A. Uta, B. Ghit, A. Dave, and P. Boncz. Low-Latency Spark Queries on Updatable Data. *ACM SIGMOD*, 2019.
[26] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL Performance Analyzer. *IEEE Data Eng. Bull.*, 31(1):51–58, 2008.
[27] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee. nowtrail: Testing with Production Queries on a Cloud Database. *DBTest*, 2018.