DeSQL: Interactive Debugging of SQL in Data-Intensive Scalable Computing

SABAAT HAROON, Virginia Tech, USA
CHRIS BROWN, Virginia Tech, USA
MUHAMMAD ALI GULZAR, Virginia Tech, USA

SQL is the most commonly used front-end language for data-intensive scalable computing (DISC) applications due to its broad presence in new and legacy workflows and shallow learning curve. However, DISC-backed SQL introduces several layers of abstraction that significantly reduce the visibility and transparency of workflows, making it challenging for developers to find and fix errors in a query. When a query returns incorrect outputs, it takes a non-trivial effort to comprehend every stage of the query execution and find the root cause among the input data and complex SQL query. We aim to bring the benefits of step-through interactive debugging to DISC-powered SQL with DeSQL.

Due to the declarative nature of SQL, there are no ordered atomic statements to place a breakpoint to monitor the flow of data. DeSQL’s automated query decomposition breaks a SQL query into its constituent sub-queries, offering natural locations for setting breakpoints and monitoring intermediate data. However, due to advanced query optimization and translation in DISC systems, a user query rarely matches the physical execution, making it challenging to associate subqueries with their intermediate data. DeSQL performs fine-grained taint analysis to dynamically map the subqueries to their intermediate data, while also recognizing subqueries removed by the optimizers. For such subqueries, DeSQL efficiently regenerates the intermediate data from a nearby subquery’s data. On the popular TPC-DC benchmark, DeSQL provides a complete debugging view in 13% less time than the original job time while incurring an average overhead of 10% in addition to retaining Apache Spark’s scalability. In a user study comprising 15 participants engaged in two debugging tasks, we find that participants utilizing DeSQL identify the root cause behind a wrong query output in 74% less time than the de-facto, manual debugging.

CCS Concepts:
• Software and its engineering → Software testing and debugging;

Additional Key Words and Phrases: Debugging, SQL, data-intensive scalable computing

ACM Reference Format:

1 INTRODUCTION

The prevalence of large-scale data has escalated the need for easy-to-use and efficient data-intensive scalable computing (DISC) systems that are capable of handling and processing large amounts of data. For instance, the SQL [2] front-end of Apache Spark [2] and Hive [36] provides a simple yet powerful way to process large amounts of data in parallel, distributing its processing across multiple nodes in a cluster. Such DISC-backed SQL systems support complex query development using both relational and dataflow operators. Legacy SQL queries can take advantage of DISC systems, where new users can write declarative queries instead of complex MapReduce [11] programs.

Authors’ addresses: Sabaat Haroon, Virginia Tech, , USA, sabaat@vt.edu; Chris Brown, Virginia Tech., USA, dcbrown@vt.edu; Muhammad Ali Gulzar, Virginia Tech., USA, gulzar@cs.vt.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART35
https://doi.org/10.1145/3643761
Debugging SQL queries that return an unexpected output is challenging for developers on DISC systems [4]. Despite SQL’s extensive usage, there is currently no technique available for interactive debugging of SQL queries akin to using “gdb” for C programs. A typical debugging practice in SQL is trial and error debugging that requires recurring query execution to manually extrapolate the intermediate data of an SQL query [4, 13, 15]. Users often resort to manually breaking down a query into constituent parts called subqueries, as shown in Figure 1, and running them in isolation to diagnose the root cause of a problem—a prohibitively time-consuming and expensive procedure [13]. On top of that, DISC-specific challenges further complicate SQL debugging. These systems ingest datasets that reside and are processed on distributed remote nodes, posing additional hurdles in understanding the intermediate results. DISC systems also introduce additional layers of code transformers and query optimizers to an already complicated system stack. The numerous processing and scheduling phases a SQL query goes through make it even more difficult to envision its true physical execution, leading to a conceptual gap between a user’s understanding of the query and physical query processing [2]. To facilitate such a debugging practice, we must provide an efficient and intuitive way to interact with the fine-grained intermediate data of a query.

We present DeSQL, an interactive step-through debugging technique for DISC-backed SQL queries. This approach allows users to inspect constituent parts of a query and their corresponding intermediate data interactively, similar to watchpoints in gdb-like debuggers. DeSQL advances SQL query debugging in three steps. First, it automatically decomposes a given query into fine-grained constituent subqueries, as shown in Figure 1, by using static query analysis—eliminating manual human effort. During query execution, it identifies the intermediate data corresponding to the extracted subqueries without requesting additional data processing jobs. Lastly, when inspecting the constituent parts of a query, the corresponding identified data is lazily delivered to the user from remote DISC nodes, providing a complete view of the query execution for an interactive debugging experience. Collectively, the three contributions can significantly reduce the effort to debug a faulty SQL query.

DeSQL is designed on the following key ideas. 1) Manually devising constituent subqueries of a given user query can be error-prone, time-consuming, and may not reflect the complete progression of data through atomic operations. During the SQL query parsing stage, DeSQL uses SQL grammar production rules\(^1\) to identify the portion of the query that is optional ([where-clause]) or alternate (cond | cond OR cond), to methodologically emit all possible subqueries against a given query; 2) DeSQL maps observed intermediate data in Spark’s Resilient Distributed Dataset (RDD) tasks to individual subquery by applying taint analysis on query components during the scheduling phase; 3) during query execution, DeSQL leverages existing RDD abstraction of Spark to intercept debugging data in a completely unobtrusive way to minimize runtime overhead; and 4) because of the query optimizer, not all data required to debug a query is captured during the execution.

\(^1\)DeSQL uses SQL BNF grammar from ISO/IEC 9075-2:2003 - Database Language SQL (SQL-2003) SQL/Foundation [33]
DeSQL analyzes the physical and logical plan of the query and identifies the closest subquery with data to regenerate the data for the current subquery.

We first evaluate DeSQL on 10 TPC-DS [28] queries with large-scale data, comparing its performance, overhead, and scalability against standard baselines. Second, we conduct an extensive user study to find qualitative evidence of DeSQL’s usability and efficacy. DeSQL incurs 10% overhead over vanilla Spark when enabled, and it offers a complete debugging view of all subqueries in only 13.8% of the manual debugging time. Even though DeSQL requires modification in vanilla Spark, DeSQL retains the scale-out and scale-up properties of Spark, posing minimal impact when debugging is not required. Our user study with 15 participants tasked with resolving two debugging scenarios demonstrates that when compared to participants engaged in manual SQL query debugging, those leveraging DeSQL achieve an average time reduction of 74% in localizing bugs within the queries. To the best of our knowledge, DeSQL is the first fully functional interactive debugger for SQL running on the DISC system. DeSQL is currently designed for the Apache Spark ecosystem; however, its underlying techniques can easily be ported to other DISC-based SQL engines. DeSQL is publicly available at https://github.com/SEED-VT/DeSQL.git

2 BACKGROUND: APACHE SPARK SQL

Apache Spark is a large-scale data processing framework that runs on a cluster computing environment. A user can write MapReduce dataflow applications in Scala, Java, and Python using Spark’s Resilient Distributed Dataset abstraction. Spark also offers a SQL [2] front-end, which allows users to write SQL queries and execute them within the Apache Spark ecosystem, delivering both scalability and ease of use to DISC users. Underneath, Spark uses a state-of-the-art query optimization engine, Catalyst [2].

When a user submits an SQL query, Spark SQL first parses the query into an Abstract Syntax Tree (AST). This process also resolves any reference to tables already loaded in Spark. Using AST, Spark builds a logical plan of the query and applies a series of rule-based optimizations to the plan. These optimizations include constant folding, predicate push-down, projection pruning, null propagation, and boolean expression simplification. The resulting optimized logical plan is then translated into one or more physical plans using Spark’s RDD-based operations such as map and filter. Using cost-based estimation, Spark selects a suitable plan to generate Java code that becomes part of an RDD’s user-defined function (UDF). Finally, the generated code, a sequence of RDDs, is submitted to the job scheduler to decompose into RDD-based tasks, which are then sent to the remote workers for distributed data processing. Each task in the submitted job computes its local output and stores it in the local memory-based storage. Outputs from tasks are either (1) consumed by the downstream tasks on any worker in the cluster or (2) gathered together to form the final output of the user-submitted query.

3 RUNNING EXAMPLE

Inspired from prior work [23], the following running example demonstrates debugging challenges in DISC-based SQL queries and the benefits of using DeSQL. Suppose a user has access to a database with four tables: names and addresses of people, information on their visits to bars, their preferred drinks, and the prices of drinks in each bar. Figure 2 shows sample rows from each relation. The user...
wants to find bars that serve beers liked by more than three patrons. To retrieve such information from the database, they write a SQL query using Spark SQL, as shown below.

```
SELECT * FROM Serves INNER JOIN Likes ON Serves.beer = Likes.beer GROUP BY Serves.bar
HAVING COUNT(Likes.patron) > 3
```

The query first selects data from two tables, Serves and Likes, and joins them on the common attribute beer. The result is grouped by the bar attribute in the Serves table. A HAVING clause is applied that counts the number of times patrons like each beer and filters the bars that have less than or equal to three likes for any beer. The user submits this query to the Spark SQL cluster and receives the output in Table 1.

Upon inspection, the user finds the output to be incorrect since very few patrons like beers that the bar The Edge offers. This error occurs because the query does not incorporate cases when the same patron likes multiple beers at the same bar. For example, if a bar serves two beers and the same customer likes those beers, the total count will be inflated, leading to incorrect results. In such cases, the query may overestimate the number of distinct patrons. However, the user is unaware of the source of the problem.

**Limitations of Existing Debugging Practices.** A common debugging practice is to start inspecting the input datasets and then apply each subquery in isolation to understand the progression of data transformation through the original query. This approach is inherently expensive since the user must run six subqueries, which may take several hours on large-scale data. Additionally, not all subqueries (e.g., `SELECT * FROM Serves INNER JOIN Likes ON Serves.beer = Likes.beer GROUP BY Serves.bar`) map to a true physical execution step due to the `PushDownPredicate` query optimization phase. BigDebug [17] offers simulated breakpoints and on-demand watchpoints in Apache Spark, but only in the physical layer, giving no information on what physical operation maps to which SQL operation. It also incurs infeasible overheads—capturing a mere 8% of the data for only one subquery incurs a 30% overhead, leading to prohibitively expensive and unscalable SQL debugging.

**DeSQL’s Contributions.** The user decides to use DeSQL by making the following code change.

```scala
+ val deSqlContext = new DeSqlContext()
+ val debuggerResults = deSqlContext.enable(query, sparkSession)
```

Once the job is finished and the user requests to debug the query, DeSQL lists all possible constituent subqueries representing every logical stage in the query execution. A user can select a subquery that triggers DeSQL to collect the corresponding remote debug data from data nodes.

![Table 1. Output of the motivating example](image1)

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
<th>name</th>
<th>beer</th>
</tr>
</thead>
<tbody>
<tr>
<td>JJ Pub</td>
<td>Amstel</td>
<td>3.50</td>
<td>Ben</td>
<td>Amstel</td>
</tr>
<tr>
<td>The Edge</td>
<td>Amstel</td>
<td>2.00</td>
<td>Ben</td>
<td>Amstel</td>
</tr>
<tr>
<td>ToT</td>
<td>Dixie</td>
<td>2.50</td>
<td>Ben</td>
<td>Dixie</td>
</tr>
</tbody>
</table>

![Table 2. Correct Output for motivating example](image2)

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
<th>name</th>
<th>beer</th>
</tr>
</thead>
<tbody>
<tr>
<td>JJ Pub</td>
<td>Amstel</td>
<td>3.50</td>
<td>Ben</td>
<td>Amstel</td>
</tr>
<tr>
<td>ToT</td>
<td>Dixie</td>
<td>2.50</td>
<td>Ben</td>
<td>Dixie</td>
</tr>
</tbody>
</table>

Fig. 3. Debugging data of constituent subqueries of motivating example by DeSQL
DeSQL: Interactive Debugging of SQL in DISC

Subject SQL Query

Query Decomposition

Data Capture Points

Missing Data Regenerator

Logical Plan

Spark Plan

Generated Code

RDDs

DeSQL Output

Fig. 4. Apache Spark SQL workflow with DeSQL. DeSQL modules are represented by Green boxes.

Figure 3 shows the debug output for each subquery. DeSQL allows the user to seamlessly move between different constituent subqueries and request the associated debug data. This imitates interactive breakpoint debugging, enables a close inspection of the intermediate query data, and facilitates isolating the root cause of the problem. In the example, the debug data for subquery SELECT * FROM Serves INNER JOIN Likes ON Serves.beer = Likes.beer shows that after applying JOIN, the output contains many duplicate patrons’ names which are subsequently counted in the final aggregation, COUNT. This visibility into the intermediate data improves the user’s understanding and helps pinpoint the error. The user did not include DISTINCT before performing a count in the HAVING operation, which counts each patron only once. After the query correction, the user gets the correct output as shown in Table 2.

4 APPROACH

DeSQL enables step-through, interactive debugging of SQL queries running on DISC systems in a post-mortem fashion. Its underlying methodology centers around three phases: (1) automated query decomposition, (2) identification and tagging of relevant intermediate data, and (3) data regeneration. DeSQL delivers breakpoint debugging to DISC-backed SQL by decomposing a user-submitted query into subqueries. During query optimization and physical plan generation, DeSQL performs fine-grained tracing to map a physical plan node (Spark’s RDD-based task) to concise locations in the logical plan of the user-submitted query, identifying where to insert data monitoring agent against a subquery. After the query finishes, DeSQL offers functionality to select a subquery in the list of all possible subqueries and, if selected, it triggers a data gathering phase to retrieve the corresponding debug data collected at individual remote executors. Lastly, for subqueries without matching intermediate data, DeSQL locates the closest subquery with debug data and regenerates the debug data. Figure 4 indicates the stages of SQL query processing in Spark SQL and DeSQL’s integration with it.

4.1 Query Decomposition

When a query returns unexpected results, a common debugging practice is to break down the query into a set of incremental subqueries and execute them one by one while monitoring their corresponding data. This process is intuitive, imitates breakpoint debugging (e.g., gdb), and helps understand the logical progression of data during query execution. However, finding such constituent subqueries is challenging as most subqueries are not evident, and the order in which they are executed is also unclear. SQL is a declarative language with a non-sequential execution pattern—it does not have statements that execute in sequential order. Instead, it has constituent parts (i.e., subqueries) that are executed to construct intermediate output for the next operation. Additionally, the user-written query does not control the sequence of execution of such subqueries, as a query optimizer decides the construction of the final execution plan. These two challenges make it infeasible to correctly decompose queries manually.
Algorithm 1: Query Decomposition Algorithm for DeSQL

**Input:** query — the SQL query to decompose  
**Output:** subqueries — list of decomposed subqueries  
\( \text{cut}(\text{tree, subtree}) \) — removes the subtree from the tree’s clone, and returns the reduced tree  
\( \text{astToCode}(\text{ast}) \) — translate a SQL query’s AST back to its corresponding SQL code  
\( \text{parseToAST}(\text{ast}) \) — parseToAST generates the abstract syntax tree of a given query. The function also identifies optional nodes in SQL grammar rules, treating them as pivot points to create constituent subqueries.

1: \( \text{parse_tree} \leftarrow \text{parseToAST}(\text{query}); \)  
2: \( \text{subqueries} \leftarrow []; \)  
3: \( \text{astToCode}(\text{ast}); \)  
4: \( \text{subQueryVisitor}(\text{parse_tree, subqueries}); \)  
5: Function \( \text{subQueryVisitor}(\text{tree, currentSubquery}) \)  
6: \( \text{forall} \; \text{node} \in \text{parse_tree} \; \text{do} \)  
7: \( \text{if} \; \text{node.isPivot} \; \text{then} \)  
8: \( \text{subqueryAST} \leftarrow \text{cut}(\text{parse_tree, node}); \)  
9: \( \text{subqueries}.\text{push}(\text{astToCode}(\text{subqueryAST})) \)

DeSQL intercepts the query execution plan when the query is being parsed into an Abstract Syntax Tree (AST) and applies its query decomposition technique to emit a series of constituent subqueries, as formally defined in Algorithm 1. We formally define a constituent subquery as one that adds, at most, one atomic relational, arithmetic, or dataflow operation to a previously found subquery. Our insight is to identify these subqueries by leveraging SQL grammar’s production rules with optional elements (e.g., `[where-clause]`), generating two distinct subqueries by considering both the inclusion and exclusion of the optional elements. We also apply this method to production rules with alternate sequences (e.g., `cond | cond OR cond`), where one alternate sequence is a prefix of another alternate sequence. Multiple unique subqueries can be generated by selecting one alternate at a time. This approach systematically finds the locations in the AST where constituent subqueries can be formed and marks those locations as pivot. Line 1 of the Algorithm 1 performs this step.

Table 3 describes a subset of operations and the transformations applied by Algorithm 1. Consider row one, for instance, that lists the SQL production rule: `<select> ::= SELECT <project> <from-clause> [WHERE-clause]`. Here, the node `[WHERE-clause]` is optional. DeSQL uses such optional nodes as pivot points for decomposition. Thus, whenever such a node is identified, two derived subqueries form: one that incorporates the optional node and another that excludes it. Furthermore, the pipe symbol (`|`) in the SQL grammar, especially when used in production rules with a shared prefix, serves as another pivotal point of decomposition for DeSQL. Row two lists an exemplary rule that showcases this behavior: `<condition-clause> ::= <condition> | <condition> AND <condition-clause>`.

DeSQL harnesses this grammar rule to produce multiple subqueries, effectively expanding the scope of the decomposition.

### 4.2 Taint Analysis

Once all the subqueries are generated, the next goal for DeSQL is to identify which physical-layer executor maps to each operation in the original query. However, there are two challenges that DeSQL must address. First, in Spark SQL, the physical-layer executors are RDD-based tasks that are scheduled to execute on individual worker nodes on a single partition of data. Second, as
Table 3. Query decomposition rewrites by DeSQL’s Algorithm 1.

<table>
<thead>
<tr>
<th>SQL Grammar</th>
<th>Example Query</th>
<th>Sub-query Rewrites</th>
<th>Example sub-queries</th>
</tr>
</thead>
</table>
| <select...<from clause> FROM Patrons WHERE name = "Dan" GROUP BY address HAVING COUNT(address) > 1 | • π′(table) | • π′(table1) | • SELECT * FROM Patrons WHERE name = "Dan"
| | • {π′(σ(c)(table) | c ∈ Conditions | • SELECT * FROM Patrons WHERE name = "Dan"
| | • {π′(σ(c)(table) | c ∈ Conditions ∧ g ∈ GroupingAttributes | • SELECT * FROM Patrons WHERE name = "Dan" GROUP BY address
| | • {π′(σ(c)(table) | c ∈ Conditions ∧ g ∈ GroupingAttributes ∧ h ∈ HavingConditions) | • SELECT * FROM Patrons WHERE name = "Dan" GROUP BY address HAVING COUNT(address) > 1 |
| | SELECT * FROM serves WHERE bar = "The Edge" AND beer IN (SELECT beer FROM likes) | • π′(table1) | • π′(table1) | • SELECT * FROM serves
| | • π′(table2) | • π′(table2) | • SELECT * FROM serves WHERE bar = "The Edge"
| | • π′(σ(c)(table1) | c ∈ Conditions | • SELECT * FROM serves WHERE bar = "The Edge"
| | • π′(σ(c)(table2)) | • π′(σ(c)(table2)) | • SELECT * FROM serves WHERE bar = "The Edge" AND beer IN (SELECT beer FROM likes)
| | SELECT FROM Patrons inner join frequents on name = patron | • π′(table1) | • π′(table1) | • SELECT * FROM Patrons
| | • π′(table2) | • π′(table2) | • SELECT * FROM frequents
| | • π′(σ(c)(table1)) | • π′(σ(c)(table1)) | • SELECT * FROM * from Patrons INNER JOIN frequents on name = patron
| | • π′(σ(c)(table2)) | • π′(σ(c)(table2)) | • SELECT * FROM * from Patrons INNER JOIN frequents on name = patron
| | SELECT FROM serves WHERE bar = "The Edge" AND price < 3.00 OR beer = 'Amstel' | • π′(T) | • π′(σ(g1)(T)) | • SELECT * FROM serves
| | • π′(σ(g2)(T)) | • π′(σ(g1 ∧ g2)(T)) | • SELECT * FROM serves WHERE bar = "The Edge"
| | • π′(σ(g1 ∧ g2)(T)) | • π′(σ(g1 ∧ g2 ∧ g3)(T)) | • SELECT * FROM serves WHERE bar = "The Edge" AND price < 3.00
| | | | • SELECT * FROM serves WHERE bar = "The Edge" AND price < 3.00 OR beer = 'Amstel' |

mentioned in Section 2, a user query transforms across different query optimizations like Predicate Pushdown, Column Pruning, and Join Reordering. Figure 5 shows the difference in the optimized plan after applying the PushDownPredicate optimization rule to filter entries with birth_year > 1980 before join operation. Hidden from users, these query transformations make it highly challenging to understand, or even guess, how the query progresses on the cloud and how the data looks at each stage. Currently, there is no way for a user to map a specific point in the logical plan of a user-submitted query to a specific task on a node such that they can monitor the intermediate results of the query. Finding such a mapping requires a deep understanding of each step of the query processor in the DISC system, every optimization strategy in its query optimization, and the final code generation phase that synthesizes the equivalent UDFs of the query.

Since the physical plan of a SQL query originates from operation(s) in the logical plan, our key idea is to perform fine-grained dynamic taint analysis of the query processor to trace data (i.e., nodes in the plan) flow through different optimization steps and processors. Using this approach, a final RDD-based task will contain the sources (nodes in the logical plan). To this end, we augment Spark’s query processing module by attaching a taint object, OpIndex, that contains a unique identifier initiated in the query parsing phase. Figure 5 depicts how a taint is attached to every node in the logical plan and propagates through each phase of the query processing.

We augment each node type in each type of plan (e.g., LogicalPlan, AnalyzedPlan, OptimizedPlan, SparkPlan, and CodeGen) with OpIndex i.e., class TreeNode {... var OpIndex: Int ...
Fig. 5. DeSQL generates a taint, OpIndex, for each node in the logical and propagates it through each step of query processing and optimization.

DeSQL debugs data monitoring.

4.3 Debugging Data Monitoring

The goal of this step is two-fold. First, during execution, it must annotate intermediate query data (referred to as debug data) with the appropriate constituent subquery identifier. Second, it must retrieve only the debug data requested for a constituent subquery. During the taint analysis phase, DeSQL injects the corresponding taint in the generated code (a user-defined function to MapPartitionRDD). The taint is later used to tag the intermediate data to match with the corresponding subquery.

DeSQL leverages Spark’s RDD lineage mechanism to address this challenge. We make the following observations. First, once a debug data is captured, it is completely immutable. Thus, no additional operations are made on it, and it can be recomputed as it adheres to Spark’s RDD properties. Second, by marking the intermediate data with an OpIndex, a simple and lightweight filter can quickly separate the debug data from the standard query output. Instead of creating a separate stream or storage of debug data, DeSQL merges the debug data with the existing output data. When a query is executed, the intercepted data is first duplicated and then appended with an OpIndex. This field is unset in regular output data. Finally, the debug data is re-injected back into the regular data stream. Note that DeSQL only records the lineage of transformations, leveraging Spark’s lazy materialization of RDDs. The duplicated rows are not physically created until an action is triggered on the RDD. Figure 6 shows how the data is intercepted and merged into the regular output stream and how the debug data bypasses the downstream query operations. Before collecting the final query results, the output stream splits into two data streams: debug and output.
DeSQL: Interactive Debugging of SQL in DISC

Fig. 6. DeSQL utilizes the regular output data stream to propagate constituent subqueries’ debugging data using FilterRDD \textit{i.e.,} stream.filter (_.isDebug) and stream.filter(!_.isDebug), but only the standard output is materialized whereas the other debug data is only materialized on demand.

Once the query execution finishes, a user may perform interactive debugging on the query. DeSQL presents a list of queries in a hierarchical view to allow the user to step through each data processing step in the query. If a user requests to see the data against a constituent subquery, DeSQL retrieves the RDD representing the debug stream, applies another filter to filter the data that contains the OpIndex of that subquery, and, finally, materializes the subquery results to show to the user. The use of RDD to represent debug data allows users to perform lightweight sampling and filter on the debug data on the cloud before gathering it.

Table 4. Data Regeneration Example

<table>
<thead>
<tr>
<th>All Possible Sub Queries</th>
<th>Data Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT bar FROM frequents</td>
<td>Available</td>
</tr>
<tr>
<td>SELECT bar FROM frequents WHERE patron = &quot;Ben&quot;</td>
<td>Unavailable</td>
</tr>
<tr>
<td>SELECT bar FROM frequents WHERE patron = &quot;Ben&quot; OR times &gt; 1</td>
<td>Available</td>
</tr>
</tbody>
</table>

Original Query: SELECT bar FROM frequents WHERE patron = "Ben" OR times > 1

4.4 Data Regeneration

Aggressive query optimization strategies often lead to misconceptions related to query execution, which further impedes a user’s ability to understand the root cause of issues. This is mainly because the user-written query looks widely different than the generated code that runs on the machine. Figure 5 shows one of the simplest query optimizations that pushes the filter closer to the data source for early data reduction and hence, faster query execution time. Such an optimization also causes challenges in providing a full logical view of all constituent subqueries. For example, the query in Table 4 is decomposed into: 1) SELECT bar FROM frequents; 2) SELECT bar FROM frequents WHERE patron = "Ben"; and 3) SELECT bar FROM frequents WHERE patron = "Ben" OR times > 1. Due to the optimization, the final physical plan does not contain any execution where the debug data for subquery SELECT bar FROM frequents WHERE patron = "Ben" could be intercepted.

Assuming that query optimizations are heavily verified against semantic equivalence, DeSQL regenerates intermediate data for subqueries for which no intermediate data exists in the original query execution. However, regenerating such data from scratch can be costly and requires rerunning partial queries. DeSQL aims for maximum data and computation reuse. Since it already has the data for most of the subqueries, DeSQL localizes the closest materialization point, \textit{i.e.,} a subquery with debugging data, and uses its data to regenerate the result for the subquery without the debug data. The closest materialization points represent the query with a minimum difference in operations and is a legal subquery of the current constituent subquery. In our example, \textit{patron = "Ben"} is the clause required to compute the missing data from the first subquery, which already contains the superset of the debug data. After concluding this step, DeSQL enables complete, step-through debugging of the user query where a user can watch the data at each step of the query execution.
### 4.5 Implementation and Limitation

DeSQL is offered as a custom distribution of Apache Spark 3.0.0. Internally, DeSQL comprises (1) a Scala-based external library that exposes the debugging controls to the query developer through the web-based user interface and (2) an add-on to Spark’s query processing engine. The overall design of DeSQL is mostly decoupled from Spark internal, which allows minimal modification as Spark’s internal code evolves in the future version. DeSQL can be invoked with a single-line modification in the SparkSQL application, as shown below.

```scala
+ val deSqlContext = new DeSqlContext()
+ val debuggerResults = deSqlContext.enable(query, sparkSession)
```

While DeSQL’s implementation supports standard SQL features, it may have limited applicability on non-standard SQL dialects and features. Thus, its utility may reduce when dealing with proprietary [34] or extended SQL dialects [34]. For instance, recursive SQL queries and correlated queries are valid SQL queries in many dialects [30]; however, such queries cannot be safely converted into directed acyclic graphs and, therefore, disallowed in DISC systems. DeSQL does not support debugging on the recursive and correlated queries. The general principles behind DeSQL are applicable to most DISC systems such as Apache Hive; however, a fully functional implementation of DeSQL for other DISC systems will require engineering efforts tailored for the specific query processor. The same limitation also holds in the case when a major redesign of the core query processor in Apache Spark will require trivial re-engineering of DeSQL’s hooks.

### 5 EXPERIMENTAL EVALUATIONS

We first evaluate DeSQL on its runtime cost, debugging efficiency, and scalability. Our evaluation plan spans four research questions:

- How much runtime overhead does DeSQL incur compared to baseline?
- How does DeSQL scale to larger datasets and an increasing number of workers?
- How efficient is DeSQL compared to a standard, naïve debugging approach?
- What is the cost of data regeneration in DeSQL?

#### Baselines.

We use two baselines, Vanilla Apache Spark and Naïve SQL Debugging, in our experiments. Since there is no existing technique to debug SQL queries interactively, we compare DeSQL’s
Fig. 7. These plots compare DeSQL’s running time overhead against the vanilla Spark for multiple SQL queries. The average DeSQL’s running time overhead is 10%. Scale factor x25 represents expanding the input tables 25 times.

debugging time against a naïve debugging approach where a user must re-run handcrafted sub-queries manually to inspect intermediate query data. While it is true that not every error requires running all subqueries, users are initially unaware of the specific subquery responsible for an error. If such information is available, both DeSQL and baseline will equally benefit from it.

**Benchmark Queries and Datasets.** We conduct experiments using 10 TPC-DS queries, an industry-standard benchmark for decision support systems [28]. It models retail product suppliers selling items via stores, catalogs, and the internet. TPC-DS queries have been consistently used as benchmarks in various DISC-based SQL platforms [11, 36, 40] and prior work on SQL query analysis [19]. TPC-DS comes with data generators. We use these generators to produce data tables with the scale factors of ×25, ×50, ×75, ×100, and ×200. Table 5 describes each subject query. These queries are of varying complexities, including both simpler and complex nested queries. The current implementation of DeSQL is equipped to handle a comprehensive array of query constructs and operators. This includes various types of JOIN, multi-layer nesting with IN and LIKE operators, and aggregation functions like COUNT and SUM with GROUP BY operators. For instance, Q10 retrieves data for customers from specific demographics. This query ingests four tables that collectively contain around 132.9M tuples and includes aggregation, joins, and nested queries. Due to query optimizations of Spark [2], DeSQL performs debug data regeneration for two of its constituent subqueries. The last column of this table indicates if data regeneration is needed for any of the constituent subqueries.

**Evaluation Environment.** We run all experiments on a 12-node cluster that contains 1 name node and 11 data nodes. Each node has at least 8 cores with a 3.10 GHz CPU, 48GB Memory, and 4TB disk space. Collectively, the cluster contains 104 cores, 53TB of storage, and 576GB of memory. We build DeSQL on Apache Spark version 3.0.0 and use vanilla Spark 3.0.0 for overhead comparisons. The associated datasets are stored on HDFS version 2.7 with a replication factor of 3. All experiments are run in a dedicated setting without any shared resources. We repeat each experiment three times and calculate the average of the reported metrics to eliminate any noise. Note that factors such as data distribution, task scheduling, and resource availability in Spark can lead to minor variations in query runtimes when executed multiple times.
Fig. 8. These plots compare DeSQL’s scalability against the vanilla Spark for multiple SQL queries to illustrate how both platforms perform in terms of scale-up and scale-out properties.

5.1 DeSQL’s Runtime Overhead

One of our design goals for DeSQL is that it must impose minimal overhead if debugging is deemed unnecessary. To measure the runtime overhead, we compare the job completion time of each subject query on (1) Apache Spark with DeSQL operating at full capacity and (2) vanilla Apache Spark. Figure 7 reports the results from each subject query. The y-axis measures the end-to-end job completion, and the x-axis represents the input dataset size. Generally, the difference between the job completion times from the two versions is marginal. For instance, in subject Query 5, DeSQL incurs an additional runtime of 2 seconds for dataset size $\times 25$ and 11 seconds for dataset size $\times 50$, resulting in less than 6% and 19% runtime overhead, respectively, over vanilla Apache Spark.

On a larger dataset size, the same Query 5 takes 15 seconds more to complete with DeSQL enabled, leading to 7% overhead. As the size of the dataset increases, the absolute overhead of DeSQL also increases slightly. Due to DeSQL’s lightweight data monitoring and fixed cost taint analysis, the relative overhead tends to stay the same (or sometimes decrease) for larger datasets. These overheads are slightly larger in queries with a higher number of constituent subqueries, as there are a higher number of locations in the user query where the intermediate must be intercepted and tagged. Even with over 12 subqueries in subject Query 10, the overhead is well within 9%. Overall, DeSQL incurs an average overhead of 10% across all ten subject queries, demonstrating its ability to provide interactive debugging solutions at a minimal cost. The primary contributor to this overhead is the data interception mechanism (DeSQL.Tap) that first duplicates the data row, attaches a tag, and then injects it back into the output data stream. By including debug data in the output data stream, the intermediate query data size increases, which can elongate shuffle times.

5.2 Scalability of DeSQL

We verify if DeSQL retains Apache Spark’s scale-up properties, i.e., changes in job time when the input data size increases, and scale-out properties, i.e., changes in job time when the number of worker nodes increases. Both experiments are crucial in validating that DeSQL does not impede Spark’s ability to scale to large datasets and extended cloud computing environments.

5.2.1 Scale Up Evaluations. Figure 7 shows the results from the scale-up experiment where DeSQL’s overhead cost is evaluated on increasing input dataset size. We scale the TPC-DS dataset by the factor of 25, 50, 75, 100, and 200. The default sizes for these datasets are relatively small and are
not suited for DISC systems. Thus, our scale-up experiment starts from the scaling factor of 25 i.e., the relevant tables are 25× larger in size than the original tables. For instance, in subject Query 9, DeSQL incurs an additional runtime of 5 seconds for dataset size ×25 and 15 seconds for dataset size ×50. This results in less than 14% and 25% overhead, respectively, over vanilla Apache Spark.

More importantly, subject queries running on DeSQL follow similar patterns of job completion times as vanilla Spark with a nearly constant overhead of 10%. For example, query Q7 running times in DeSQL and vanilla Spark are nearly identical. This query has three constituent subqueries, and it ingests a total of 20M rows. On the other hand, queries Q9 and Q8 have slightly larger overheads, between 19% and 29%, on larger dataset sizes. This is mainly because these queries require query regeneration in at least one of their constituent subqueries. Overall, across ten queries running on five different dataset sizes, DeSQL closely follows the scale-up behavior of vanilla Spark, validating its scalability in dealing with large amounts of data. The main cause behind such desirable behavior is that DeSQL’s data interception mechanism incurs a constant time overhead while processing a single data record, regardless of the type of operation.

5.2.2 Scale out Evaluations. We evaluate the scale-out properties of DeSQL by increasing the number of workers from 2 to 10 and measuring the job completion time. This experiment explicitly answers if the modification needed for DeSQL has influenced the ability to utilize additional nodes on the cluster. As the number of workers increases, the available resources and parallel processing capacity of the Spark cluster also increases. This enables Spark to distribute the workload more evenly among the workers, allowing for efficient parallel execution of queries. Figure 8 reports the results from this experiment, where the y-axis represents the end-to-end job completion time and the x-axis indicates the number of worker nodes. Overall, across all subject queries, DeSQL shows a similar job completion time pattern as vanilla Spark, indicating its minimal impact on resource utilization. In fact, the gap between the DeSQL’s performance and baseline narrows as more nodes are added to the Spark cluster, which is consistent across all subject queries. For example, in subject query Q3, DeSQL takes 92 seconds more than the baseline Spark on a 2-node cluster, which reduces to barely 10 seconds difference on a 10-node cluster. The results show that DeSQL retains Apache Spark’s resource utilization ability and, for a higher number of worker nodes, DeSQL’s performance starts converging with Spark’s processing performance.
5.3 Debugging Time Savings

We design experiments to measure DeSQL’s efficiency in performing debugging i.e., extracting the debugging data for each constituent subquery from their corresponding RDD abstractions. As mentioned earlier, there is no standard technology to facilitate interactive debugging in DISC-backed SQL platforms [11, 36, 40]. Our baseline, naïve debugging, is the current, most common practice of debugging an SQL query, which comprises breaking down the query manually and then running the resulting subqueries one after another.

In DeSQL, extracting debug data involves merging all debug data into a single RDD and then filtering the debug data based on a constituent subquery’s OpIndex. Figure 9 presents the results from these experiments. The y-axis represents the debugging time, and the x-axis represents the number of rows. The debugging time is the cumulative time DeSQL takes to collect the data for all constituent subqueries of a subject query. For example, in Q10, DeSQL takes a total of 191 seconds to find the complete debugging data for all subqueries, whereas naïve debugging takes 1391 seconds. DeSQL reduces the total debugging time for all subqueries by 86.2% compared to naïve debugging. We further categorize the cumulative debugging time into each subquery’s debugging time.

For almost every subquery, DeSQL’s debugging time is always lower than the naïve debugging time. One primary reason is that DeSQL intercepts the debug data during the original job execution, reducing the amount of redundant work needed to regenerate such data. Second, it uses RDD abstraction to store debug data for every subquery. Thus, all debug data is lazily computed and stored in distributed remote datanodes and only materialized when needed, leading to very low runtime overhead and quick debugging time. The cost of the first subquery is usually higher than the rest. This is due to a longer RDD lineage chain for debug data stream. When the first query is collected, the Spark traces its RDD lineage to the closest materialization point (similar to a checkpoint) of the original query’s DAG. It then computes the collective debug data stream from the checkpoint, and the collective data stream becomes the closest materialization point for the rest of the debugging subqueries.

Overall, DeSQL debugging time is 7.4× less than the naïve debugging time, on average, when finding the complete debugging data for each subject query. In some cases, DeSQL’s debugging time is lower with larger datasets than the debugging time with smaller datasets. This behavior is common in DISC systems due to reduce-phase skew arising from an imbalance in key distributions [22].

5.4 Cost of Data Regeneration

Due to query optimization, a constituent subquery’s debug data may not be observed during the original query execution. DeSQL regenerates such data from the latest materialization point i.e., the closest constituent subquery. The process of data regeneration takes longer compared to normal subqueries, as multiple operations must be applied to the closest subquery’s data to compute the relevant debug data. To understand the cost of regeneration further, we measure the difference in debugging time of individual constituent subqueries and compare it with subqueries that require data regeneration, as shown in Figure 10. Each block inside the bar represents the individual duration of all sub-queries within a single query. For instance, in Query 6, the last subquery takes significantly longer (13.8 seconds) than the average time of other subqueries (9.3 seconds). This is due to the presence of a `join` operator that requires data generation. Similarly, in Query 10, the second box from the bottom takes slightly longer, representing a subquery with `COUNT` that takes 14 seconds, which also requires data regeneration. Overall, subqueries that do not
require regeneration are $2.2 \times$ faster than subqueries that require regeneration. Note that Query 7’s debugging time is low owing to only three very low-complexity constituent subqueries.

6 USER STUDY
We augment our evaluation with a user study to measure the efficacy and usability of DeSQL in debugging faulty SQL queries. We aim to investigate the following research questions.

- Does DeSQL decrease SQL query debugging time compared to traditional methods?
- Without DeSQL, what debugging approaches do developers use for SQL queries?

6.1 Study Design
We opted for a mixed-method user study that includes a hands-on debugging task in which we measure a participant’s performance in debugging and a qualitative survey to learn the benefits and limitations of DeSQL. We recruited a total of 15 participants, including 9 graduate and 6 undergraduate students. The participants’ backgrounds varied in terms of their experience with Apache Spark and SQL. Almost all participants were proficient in SQL and had working knowledge of rational databases. Approximately 53.3% of the participants had no prior experience with Apache Spark, while 6.7% had worked with it for a few months, 33.3% for over a year, and 6.7% for 3-5 years. Each participant was tasked with debugging two distinct SQL queries. The participants were introduced to the database schema and the structure of the queries involved in the study during a tutorial session. They were given time to familiarize themselves with the database tables and the query descriptions. This ensured that participants were equipped with the necessary knowledge to proceed with the debugging tasks. The end-to-end study spanned 30 minutes per participant.

Table 6. Debugging Performance Comparison

<table>
<thead>
<tr>
<th>Participant</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>P9</th>
<th>P10</th>
<th>P11</th>
<th>P12</th>
<th>P13</th>
<th>P14</th>
<th>P15</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeSQL (min:sec)</td>
<td>03:12</td>
<td>02:15</td>
<td>03:05</td>
<td>04:45</td>
<td>04:16</td>
<td>03:18</td>
<td>05:16</td>
<td>02:36</td>
<td>03:03</td>
<td>04:18</td>
<td>01:06</td>
<td>03:16</td>
<td>01:09</td>
<td>02:16</td>
<td>03:04</td>
</tr>
</tbody>
</table>

6.2 Debugging Efficiency
We first measured the time it took for each participant to localize the root cause of the incorrect query output. Table 6 presents the time taken by each participant to debug the faulty SQL queries manually and using the DeSQL-assisted approach. The results indicate a substantial improvement in debugging efficiency when utilizing DeSQL. For example, participant (P5) who took the longest
with the DeSQL (5.3 minutes) was still faster than the majority of those using the vanilla method, underlining the benefits of DeSQL. On average, compared to manually debugging, participants were 74.1% faster at localizing the fault in the given queries when using DeSQL. Further, using a t-test to compare the average debugging time for participants in each setting, we found a significant difference in performance ($t = 7.0542, p < 0.00001$). These results demonstrate that DeSQL significantly reduces the time required for debugging SQL queries in real-world scenarios, making it a valuable tool for developers. The time with * represents participants (P1, P7, P9, and P13) who could not resolve the debugging issue within the given time frame. This was not the case with DeSQL, where every participant managed to address the issues. This further underscores the effectiveness of the DeSQL method.

### 6.3 User Behavior

Figure 11 provides a visual representation of the strategies and actions employed by participants while debugging SQL queries. These actions offer insights into the typical approaches developers take when encountering faulty SQL queries and serve as a valuable context for understanding the need for tools like DeSQL. One prominent observation is that a large portion of participants, accounting for 53.3% ($n = 8$), chose to “Check Data” as an initial action. This indicates that inspecting the underlying data sources is a common first step in the debugging process, highlighting the importance of data examination for identifying issues. Another prevalent action taken by 53.3% ($n = 8$) of participants is “Trial & Error”. This trial-and-error approach underscores the experimental nature of debugging and the importance of iterative adjustments to the query. Additionally, the “Re-check Query” and “Run Sub-Queries” behaviors were prevalent, each involving 40% ($n = 6$) of the total participants. These actions reflect a common approach where developers seek contextual information through query descriptions and execute specific sub-queries to isolate and diagnose problems within a complex SQL query.

Interestingly, a smaller number of participants, accounting for 13.3% ($n = 2$), applied “Decompose Query”. This action demonstrates an understanding of query complexity and the benefit of breaking down queries into manageable parts for analysis. It is worth noting that manual query decomposition can be a challenging and time-consuming task, which could explain why fewer participants opted for this approach. Furthermore, two (13.3%) attempted to “Find Slip Mistakes”. This action aligns with the concept of identifying typographical errors or minor mistakes that can impact query execution. Lastly, one participant (6.7%) opted to “Use limits”. This action reflects the strategy of limiting query results to pinpoint problematic areas, which can be an effective approach to isolating issues.

### 6.4 Participant Feedback and Insights

Table 7 presents a comprehensive summary of participants’ feedback regarding their experience with DeSQL while debugging SQL queries in Apache Spark. The participants were asked to rate their experiences across various aspects.

In terms of how well DeSQL’s interface aligned with participants’ workflows, 66.7% ($n = 10$) of respondents gave it the highest rating of 5, indicating that DeSQL seamlessly integrated into their debugging processes. An additional 33.3% ($n = 4$) rated it as a 4, further affirming the tool’s positive impact on workflow alignment. Importantly, none of the participants rated it lower than 4, indicating a high level of satisfaction and comfort with the tool’s interface. When comparing the
efficiency of using DeSQL to traditional methods of debugging SQL queries in Apache Spark, a substantial 53.3% ($n = 8$) of participants found DeSQL to be "Much More Efficient", while another 46.7% ($n = 7$) rated it as "More Efficient". This indicates a consensus among the participants that DeSQL enhances the efficiency of the debugging process compared to conventional approaches.

Regarding the most helpful features of DeSQL’s debugging interface, 66.7% ($n = 10$) of participants highlighted “Intermediate Data Monitoring” as the standout feature. This suggests that the ability to closely monitor and analyze intermediate data during the debugging process greatly aids in identifying and resolving SQL query issues. Additionally, 20% ($n = 3$) of participants found “Subquery Decomposition” to be beneficial, indicating that this feature assists in breaking down complex queries into more manageable components. A smaller portion (13.3%, $n = 2$) appreciated the “Step-through Debugging Experience”, which indicates the tool’s interactive step-by-step debugging functionality is valued by users.

Moreover, 93.3% ($n = 14$) of participants encountered no challenges while using DeSQL, indicating its overall user-friendliness. Only one participant cited “Navigation and Controls” as a minor concern, suggesting room for improvement. Also, 86.7% ($n = 13$) of users expressed a strong likelihood of choosing DeSQL over traditional debugging methods for future SQL query issues, while 13.3% ($n = 2$) showed moderate interest. Lastly, when self-assessing speed in debugging SQL queries, 100% of participants found DeSQL outperformed traditional methods.

<table>
<thead>
<tr>
<th>Feedback Category</th>
<th>Rating (Out of 5)</th>
<th>% of Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>How well did DeSQL’s interface align with your workflow when debugging SQL queries in Apache Spark?</td>
<td>5</td>
<td>66.7%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>33.3%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>Compared to traditional methods of debugging SQL queries on Apache Spark, how would you rate the efficiency of using DeSQL?</td>
<td>Much More Efficient</td>
<td>53.3%</td>
</tr>
<tr>
<td></td>
<td>More Efficient</td>
<td>46.7%</td>
</tr>
<tr>
<td></td>
<td>About the Same</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>Less Efficient</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>Much Less Efficient</td>
<td>0%</td>
</tr>
<tr>
<td>What feature of DeSQL’s debugging interface did you find most helpful in debugging Query?</td>
<td>Subquery Decomposition</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>Intermediate Data Monitoring</td>
<td>66.7%</td>
</tr>
<tr>
<td></td>
<td>Step-through Debugging Experience</td>
<td>13.3%</td>
</tr>
</tbody>
</table>

6.5 Threats to Validity

Our study, while yielding promising results, is subject to potential threats to validity that need to be considered. One potential threat is sampling bias. Since our participant pool primarily consisted of individuals with limited experience in Apache Spark, the results may be skewed toward those who are less familiar with traditional debugging tools. To mitigate this, we deliberately designed our user study to be user-experience independent of Apache Spark, focusing solely on SQL-related debugging tasks. This approach allowed participants to evaluate DeSQL based on their SQL debugging expertise rather than their familiarity with Spark. We acknowledge that not all SQL queries may be equally challenging for each user. To mitigate potential bias, we ensured that each participant worked with both the DeSQL and vanilla Spark debugging environments, with each query being assigned an equal number of times to both tools. This approach balanced the difficulty variation between the two tools, helping us obtain more objective feedback.

7 RELATED WORK

Explainability of SQL. When a query returns suspicious output, developers seek an explanation with pointers to the root cause. Prior research has extensively studied the problem of explainability,
or difficulties of explaining SQL query output in an understandable way to humans [12, 23, 24, 32, 38]. Miao et al. propose a technique to find the smallest counterexample to SQL queries, given an apriori known counterexample [24]. They explain inequivalence between the two queries by finding the minimal data set that causes them to produce different results. Scorpion [38] explores large data sets by identifying predicates that explain outliers in an aggregate query result. An overlapping theme in such automated approaches is that they often require a programmable output oracle, which is difficult to find, especially in exploratory analysis.

Roy et al. [26, 32] provide explanations for SQL query outputs by either removing tuples from the database or removing query components from the original query that have a significant impact on the answers. Such an exhaustive method is effective but likely to incur prohibitive slowdowns in DISC-backed SQL, where a given table can have millions of rows. X-Trace [12] enables tracing the execution of SQL queries by capturing and correlating metadata from multiple system layers. In contrast, I-REX [23] provides an interactive tracing interface for SQL queries that allows users to understand the evaluation of complex queries with correlated subqueries. DataPlay [1] presents an interactive query manipulation playground to perform trial-and-error debugging, which includes manipulating and reordering the query components. These three approaches are closely related to DeSQL, but are primarily designed for traditional relational database management systems. They do not address issues (e.g., scalability or data regeneration) arising from underlying DISC framework constructs such as query translation, optimization, and job scheduling. Habitat [15] allows users to mark SQL subexpressions and generates a new query based on those marks to observe their evaluation. However, it relies on executing queries on the target SQL database host repetitively from scratch, which is infeasible in a DISC settings.

Data Provenance. Data provenance (DP) seeks to identify the input data tuples that play a role in producing an output [5, 9, 10, 14, 27]. Traditional database management systems (DBMS) provenance approaches are either tightly integrated with the DB systems and thus are not operable in a new generation of SQL engines, or they require data and query manipulation, both of which can cause large overheads at scale. Even in recent DP approaches such as Smoke [31], the lineage capture logic is implemented into physical database operators with write-optimized lineage representations to assist lineage queries when future queries are known upfront. Unlike DeSQL that provides a transparent view into query execution, DP assists in a very specialized debugging focused on a narrow subset of output.

Data provenance in DISC systems [6, 20] mostly capture data lineage at the level of data transformations and store lineage tables in in-memory storage. Lineage tables captured at the physical layer are generally hard to port back to the original query in DISC-backed SQL, decreasing their value as a debugging tool. More fine-grained DP approaches [16, 18, 35, 39], inspired by dynamic taint analysis [7], capture data by attaching taint to original data. These approaches suffer from taint explosion problems—if one million rows are aggregated, the resulting taint will be a collection of one million taints. DeSQL’s use of taint is only restricted to query operators, which are several orders of magnitude less than the input data rows.

Interactive Debugging in DISC. Inspector Gadget (IG) [29] is the early work on enabling custom dataflow instrumentation for monitoring and debugging query data workflows in distributed environments, such as Apache Pig/Hadoop. Similarly, in the Apache Spark ecosystem, BigDebug [17] offers a set of interactive, real-time debugging primitives that allows users to selectively examine distributed intermediate data, pinpoint crash-inducing records, and determine the root causes of errors at the record level through a fine-grained data provenance capability. More recently, Texera [37] provides interactive and real-time feedback during long-running analytics tasks, addressing a common problem with batch processing in DISC systems like Spark SQL. Other related work [3, 8] also offers a mechanism to inspect DISC jobs. While such tools can augment DeSQL's
ability to collect and interact with physical layer data, the collected debug data must be translated to developer-friendly information that is consistent with the original query. This is one of the primary challenges that DeSQL addresses.

**Commercial tools of SQL Debugging.** Transact-SQL (T-SQL) [25] helps debug stored procedures, functions, and triggers, similar to traditional gdb debugging. However, it does not debug the relations and dataflow portion of a query. T-SQL is incompatible with DISC-backed SQL, as the stored procedures are translated into Java UDFs, making debugging on SQL stored procedures meaningless. A similar tool, KeepTool’s PL/SQL Debugger [21], enables debugging of PL/SQL code within the Oracle database environment, allowing developers to step through code line-by-line, set breakpoints, inspect variables, and perform other debugging operations. It also suffers from the same limitations as T-SQL. Both tools are now considered outdated and lack desired features to support interactive debugging [13].

8 CONCLUSION

Trial-and-error debugging is an intuitive and de-facto method of debugging SQL queries due to its broader utility. Despite its popularity, such debugging is highly error-prone, time-consuming, and costly, especially in DISC-backed SQL such as Spark SQL. With DeSQL, we make interactive, step-through debugging completely feasible and efficient on DISC systems. DeSQL’s value proposition is its automated way of decomposing queries into constituent subqueries and utilizing existing data pipelines of native Spark abstraction to deliver debugging data to the developer. DeSQL provides a complete and transparent view of query execution in 13% less time than the original job time, with a mere 10% runtime overhead. Our user study indicates that DeSQL is practical to facilitate SQL debugging tasks. We envision that DeSQL’s low-cost access to a query’s intermediate query data can enable a large variety of SQL explainability and provenance approaches, which are otherwise infeasible on DISC systems.

9 ACKNOWLEDGEMENT

This work is supported in part by the National Science Foundation under grant number 2106420 and MECRT, Indonesia. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

REFERENCES


