OurSys: An Interactive Debugger for SQL

ABSTRACT

SQL is declarative in nature and rich in its features. Writing semantically correct SQL queries and finding logical bugs in SQL are not easy, even for experienced programmers, who are often used to the mindset of working with a variety of general-purpose programming languages (GPLs). While there are many tools to assist with GPL debugging, SQL debugging has received much less attention. In this paper, we present OurSys, a SQL debugger that enables users to inspect the logical execution of SQL queries visually and interactively to identify and potentially fix logical bugs in the queries. OurSys draws analogies to the debugging paradigm of GPLs (e.g., stepping, watchpoints, etc.), making it easier for programmers to adopt. However, unlike debugging GPLs, which involves executing the underlying program in full to the point of interest, OurSys allows users to jump to arbitrary points of interest by leveraging the power of the database systems, through selective materialization and query rewrites. To simplify deployment, OurSys acts as a lightweight middleware on top of the database system; it imposes no overhead to prepare a database for debugging and maintains no state in the database systems during debugging sessions. We demonstrate the effectiveness of OurSys through performance experiments as well as a user study in an educational setting.

1 INTRODUCTION

Relational databases form the backbone of many data-intensive applications and scalable data analytics. Despite its age, SQL continues to retain its prevalence and importance due to its highly *declarative* nature (i.e., specifying what the answer should be rather than how to compute it) and its extensive set of features that have grown over time. On the other hand, SQL is difficult to understand and debug. In debugging GPLs (such as C++ or Python) that are typically *procedural* (i.e. explicitly describing the processing steps required to compute the answer), it is natural to trace the execution of programs to debug them. However, this method becomes much trickier for SQL.

As a first attempt, one may consider "tracing" a query's logical or physical plan, a tree whose leaves represent base tables and internal nodes represent relational operators. Through this approach, a user can examine the intermediate results produced by each of the plan nodes. Unfortunately, there are several problems with this approach. Firstly, the database optimizer often compiles a SQL query into a plan that bears no resemblance to the original query, making plan tracing unhelpful in finding and fixing logical bugs in the original query. Secondly, debugging is usually iterative: the user may examine execution multiple times, sometimes with minor modifications to the query. However, even small changes can lead to a very different plan (e.g., addition or removal of a condition in `WHERE` can enable or disable an index scan opportunity). Even if the physical plan remains the same, there is no guarantee that the execution and result order are reproducible. For example, size of buffer memory, choice of hash function, and variations in the speed of parallel threads at runtime can change the ordering of intermediate result rows. Such non-repeatable and seemingly inconsistent behaviors significantly complicate debugging.

Perhaps, one possible workaround would be to restrict the database optimizer to avoid optimization across syntactic blocks of a query, such that each subquery corresponds to some subtree in the plan, and the user can at least inspect the result of each subquery. However, correlated subqueries, a SQL construct frequently used in practice, render this workaround ineffectual for many queries. In Section 2, we give concrete examples illustrating this challenge and showing how OurSys aids in debugging these types of queries.

In this work, we focus on finding *logical errors* instead of fixing *performance issues*, and we aim to build an interactive SQL debugger with the following desiderata:

(1) The debugger should conceptually execute a SQL query in a manner that is completely reproducible, faithful to how it is written, and easy for average programmers to understand.
(2) The debugger should offer features analogous to those already in GPL debuggers, so they are easy to learn and adopt.
(3) Unlike GPL debuggers, which usually must execute underlying programs in full in order to reach points of interest, this debugger should leverage the power of SQL and database systems to support more powerful features and/or more efficient implementation of features.
(4) The debugger must scale to large database instances and gracefully handle prohibitively large intermediate results.
(5) The debugger should be simple to run (e.g., from a remote browser) and easy to deploy on top of a database, without modifying database system internals or requiring extensive preparation of the database for debugging.

At a first glance, (1) necessitates executing the SQL query "literally" using a completely unoptimized plan, which runs counter to (4). Our key insight here, however, is that at any given point in time, the user can be examining only a small "window" of the entire execution. It suffices to support fast access to any given "window" without incurring the full cost of the unoptimized plan. Supporting such accesses, along with (2) and (3) while respecting (5), require novel optimization ideas — simply relying on SQL's built-in `OFFSET` and `LIMIT` operators fails to deliver acceptable performance for interactive debugging.

Addressing the above challenges through the development of OurSys, we make the following contributions:

- OurSys introduces a novel debugging paradigm for SQL that draws many parallels to GPL debugging, where each query block is viewed as a function and correlated subqueries can be considered functions with arguments. We define the *canonical execution* of a SQL query, which is reproducible and faithful to query syntax; edits to a query lead to predictable changes in its execution.
- OurSys supports various debugging features analogous to GPL debugging, including stepping through execution, pausing to examine a particular point during execution (breakpoints), pausing automatically at points of interest (watchpoints), drilling down

into subqueries (stepping "into" a function), and row-level tracing (information flow analysis) in both forward (from input to output) and backward (from output to input) directions.

- While performing the canonical execution would have been extremely inefficient and impractical, OurSys supports effectively fast "teleporting" from one point of execution to another without paying the cost of execution in between. We do so by constructing queries that directly compute, for a given window, information needed for debugging, and by applying selective materialization and rewrite optimizations to ensure their efficiency on large database instances.
- OurSys has a web-based frontend and a middleware backend that runs on top of a database system. It imposes no overhead to prepare a database for debugging and maintains no state in the database during active debugging sessions. This architecture makes OurSys easy to adopt and deploy.
- Our performance evaluation using the TPC-H benchmark shows the scalability of OurSys on large databases. More specifically, it also demonstrates the advantage of our optimization techniques over standard database (PostgreSQL) support for retrieving windows of query result rows.
- We evaluate the efficacy of OurSys with a large-scale user study of over 100 students in an introductory-level database course. Its findings indicate that OurSys significantly improves students' efficiency in debugging SQL queries.

2 EXAMPLE USE OF OURSYS FOR DEBUGGING

Since OurSys conceptually executes SQL queries the way they are written, it is particularly suitable for novices who are learning how SQL queries work on the logical level. Furthermore, it serves as a powerful tool for novices and data professionals alike to find and fix logical bugs. This section provides a walk-through of how to use OurSys to debug an incorrect query; we introduce the interface, concepts, and features of OurSys. More formal and detailed discussions will be presented in later sections.

Example 2.1. Consider the toy database in Figure 1, which stores information about beers, bars serving them, and drinkers who like beers and frequent bars. We want to write a query for the following task: suppose every time a drinker frequents a bar, they buy one bottle of every beer they like or any beer priced $2 or lower; find the expected weekly revenue of each bar and rank them by revenue from high to low. A user may come up with the following (incorrect) query:

```
SELECT s.bar, SUM(f.times_a_week * s.price) AS revenue -- Q
FROM Serves s, Frequents f
WHERE f.bar = s.bar
AND (s.price <= 2 OR
    EXISTS (
        SELECT * FROM Likes l WHERE f.drinker = l.drinker -- Q_inner
    ))
GROUP BY s.bar;
```

The above query intends to first find drinkers and beers available for purchase using a join between Serves *and* Frequents*. It additionally applies the two (alternative) conditions for purchase: 1) the price is lower than $2 and 2) the beer is liked by the drinker. Then, the query groups the intermediate results by bar and calculates the sum of revenue. There is a bug in the* EXISTS *subquery Q_{inner}, but the question for now is: how would a user examine the result of Q_{inner}?*

bar	beer	price
Apex	Corona	1
Apex	Dixie	2
Edge	Amstel	4
Edge	Corona	1.5
Tavern	Amstel	3
Tavern	Erdinger	1

(a) Serves

drinker	bar	times
Amy	Apex	1
Ben	Edge	4
Coy	Tavern	2
Dan	Edge	3

(b) Frequents

drinker	beer
Amy	Erdinger
Ben	Budweiser
Ben	Dixie
Coy	Amstel
Dan	Amstel
Dan	Corona

(c) Likes

Figure 1: A toy database about beers, bars, and drinkers.

Note that Q_{inner} is correlated*, with the value for* f.drinker *coming from the outer (i.e. enclosing) block. As a result, there is no way to inspect this result independently. This situation cannot be handled by a query plan with relational operators, where the result of each subtree depends on this subtree alone.*

Indeed, most database optimizers will rewrite the above query for execution such that the subquery is decorrelated*. The decorrelated subquery would be a join involving both* Likes *and* Frequents *to compute, effectively, the original subquery for all possible* drinker *values in a single effort. Then, the result will be further combined with the rest of the outer query. The new plan now consists of only relational operators and can be computed/debugged in a bottom-up fashion, but unfortunately, it is vastly different to the original query. Any user without in-depth knowledge of query optimization will likely be very confused.*

Example 2.2. This incorrect query returns the following result for the database in Figure 1:

bar	revenue
Apex	*3*
Edge	*38.5*
Tavern	*8*

Based on the user's knowledge of the database instance, the revenue of bar Edge *seems to be higher than expected. We now walk through how to use OurSys to debug the query, starting with this observation.*

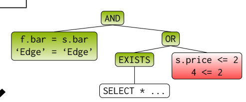
OurSys presents a panel of UI debugging elements for each block of the query. Figure 2 illustrates the UI for the outer query block Q (for simplicity, we do not show the actual interface here as it contains other details that may be distracting for this discussion). OurSys shows the execution of this block in stages, from top to bottom. At the very top, OurSys shows all input tables in FROM*. Note that one row from each input table is highlighted; this input combination ("combo") intuitively defines the current point of execution being examined. Then, OurSys presents the "joined & filtered" result, which is the intermediate output after the* WHERE *clause is applied. The intermediate result row produced by the current input combo is automatically highlighted for users. Between this result table and the input tables, a "filter expression" tree shows how the* WHERE *condition evaluates over the current input combo. The user can examine the value of each subexpression therein and see how the truth values (color-coded here) are combined by logical connectives. Following the joined & filtered result, OurSys shows the* GROUP BY *result. For each group, in addition to the* GROUP BY *value, each group member's contribution to the final* SUM *aggregate is also shown. Again, the group and the member that the current input combo contributes to are automatically highlighted. Finally, the final result of the query block is shown as the output table.*

For the convenience of subsequent discussion, we show a symbolic row identifier (e.g. s0, f2, j6...) for each row in all tables. We do in fact assign internal row identifiers, whose purposes will be explained later in Section 3, but they are not explicitly displayed by the UI.

Serves AS s	bar	beer	price
s_0	Apex	Corona	1
s_1	Apex	Dixie	2
s_2	Edge	Amstel	4
s_3	Edge	Corona	1.5
s_4	Tavern	Amstel	3
s_5	Tavern	Erdinger	1

Frequents AS f	drinker	bar	times
f_0	Amy	Apex	1
f_1	Ben	Edge	4
f_2	Coy	Tavern	2
f_3	Dan	Edge	3

Filter Expression



Joined & Filtered	bar	beer	price	drinker	bar	times
j_0	Apex	Corona	1	Amy	Apex	1
j_1	Apex	Dixie	2	Amy	Apex	1
j_2	Edge	Amstel	4	Ben	Edge	4
j_3	Edge	Amstel	4	Dan	Edge	3
j_4	Edge	Corona	1.5	Ben	Edge	4
j_5	Edge	Corona	1.5	Dan	Edge	3
j_6	Tavern	Amstel	3	Coy	Tavern	2
j_7	Tavern	Erdinger	1	Coy	Tavern	2

Group	bar	sum_input
g_0	Apex	1 2
g_1	Edge	16 12 6 4.5
g_2	Tavern	6 2

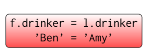
Output	bar	revenue
o_0	Apex	3
† o_1	Edge	38.5
o_2	Tavern	8

Figure 2: Debugging context for outer query block, Example 2.2.

Bindings from enclosing queries: `f.drinker = 'Ben'`

Likes AS l	drinker	beer
l_0	Amy	Erdinger
l_1	Ben	Budweiser
l_2	Ben	Dixie
l_3	Coy	Amstel
l_4	Dan	Amstel
l_5	Dan	Corona

Filter Expression



Filtered / Output	drinker	beer
o_0	Ben	Budweiser
o_1	Ben	Dixie

Figure 3: Debugging context for inner query block, Example 2.1.

Given the unexpectedly high revenue of Edge, *the user naturally wants to examine how that particular output row was computed. OURSYS supports tracing backwards from output to input using a more general mechanism calling* pinning, *denoted here by the red † next to $o_1 = \langle$Edge, 38.5$\rangle$. A pinned output row intuitively narrows the entire execution down to only parts that are "relevant" to it (which we define formally later in Section 3). As shown in Figure 2, relevant rows in upstream tables are automatically shown in red. Specifically, input rows s_2 and s_3 join with f_1 and f_3 in a nested-loop fashion to produce rows j_2 through j_5 in the joined & filtered table; they are then further grouped into g_1 in the group table before finally producing o_1 in the output.*

As soon the user pins o_1, OURSYS automatically identifies the relevant input combos and "positions" execution at the very first input combo in lexicographical order — this is how the input combo (s_2, f_1)

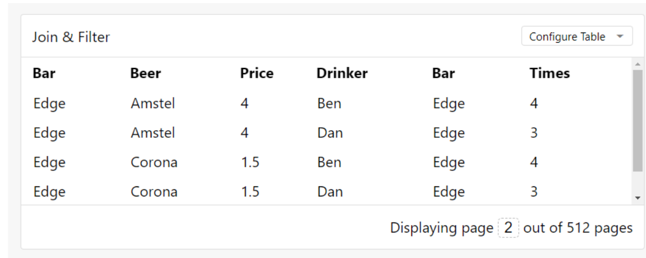


Figure 4: A paginated table in OURSYS.

was activated in the first place in Figure 2. Starting with this input combo, the user can then step through all other input combos relevant to o_1, or manually choose any specific combo to investigate. Throughout the entire process, OURSYS automatically updates the highlighting in all downstream tables as well as the state of any expression evaluation trees, in effect supporting forward tracing.

When examining the execution for (s_2, f_1) as shown in Example 2.2, the user notices that (s_2, f_1) contributes a value of 16 to the final sum. According to the filter expression tree, the price of Amstel is higher than \$2, but EXISTS(Q_{inner}) *returns true, meaning that Ben should like Amstel per query intention. OURSYS allows the user to "drill down" and explore the execution of Q_{inner} in this context. Recall that Q_{inner} is a correlated subquery. To an average programmer, the analogy is to think of evaluating Q_{inner} as a function call with an extra parameter setting for* f.drinker, *which takes its value from the current input combo. Hence, "drilling down" naturally corresponds to "stepping into" a function call in GPL debugging.*

Once the user drills down to Q_{inner}, OURSYS creates a new panel for debugging this subquery block, illustrated in Figure 3. The UI makes it clear that we are executing

```
SELECT * FROM Likes l WHERE f.drinker = l.drinker;
```

with our parameters (i.e. f.drinker*) set to Ben. In this case, a quick glance at the output table or the filter expression tree for this block should reveal the problem — nothing about this subquery supports that Ben likes Amstel. In fact, this subquery does not even know we are looking for Amstel. Therefore, to fix the query, we should let the outer query "call" Q_{inner} with an additional parameter (i.e.* s.beer*), and let Q_{inner} additionally test* s.beer = l.beer.

The user can modify the original query accordingly and restart the debugging session to verify that it fixes the problem. Because the new query is syntactically similar to the old, OURSYS will produce a very consistent experience for the user: the execution of the new query will be nearly identical to the old, with the exact same stages and exact same ordering of input combos and intermediate result rows.

A Note on Scalability. While Example 2.2 simplifies our discussion by assuming a small database instance, realistic databases are much bigger. Even with moderately sized databases, joins can easily produce very large intermediate results. As we will see in Section 5, for some TPC-H benchmark queries, even with a moderate scaling factor of 1, a single intermediate result table can easily take an hour to print out entirely on the database server console. Hence, caching entire results at any location (database server, middleware, or user browser) or shipping them is simply impractical.

At the same time, users are generally unable to examine many rows simultaneously anyway. Therefore, OURSYS UI supports a pagination mechanism to display each table (input, intermediate result, or final output); a screenshot is shown in Figure 4. The user only sees a page's worth of data at once in each table. Data outside the visible page can be computed and fetched on demand (and subsequently cached or evicted). As the user interacts with the UI, OurSys adjusts the visible portion of all displayed tables accordingly, such that each reflects the execution point defined by the current input combination (combo). Similarly, the user can visit any pages of input tables to select a new input combo for forward tracing; in that case, OurSys would automatically adjust the displays of downstream tables to show the corresponding intermediate result rows.

Hence, efficient pagination is key to scalable SQL debugging. With efficient pagination, OurSys effectively allows the user to "teleport" across points of interest without incurring the execution cost in between, which makes OurSys much more powerful than GPL debugging. Pagination also provides a more manageable and less overwhelming experience for users. While most database systems support efficient pagination of input tables, it is challenging to paginate intermediate results as well as associated information for debugging. Furthermore, OurSys's requirement of making execution reproducible imposes specific ordering of intermediate results that complicates optimization. We discuss how to tackle these challenges in Section 4.

3 DEBUGGING PARADIGM

This section describes the debugging paradigm of OurSys in detail. Several concrete instances of the models described are available in the full version [33]. OurSys supports a rich set of SQL query features, such as `SELECT` queries with inner cross joins, set/bag operations (e.g., `UNION`, `INTERSECT`, `EXCEPT`), and even outer joins and joins expressed in general `JOIN` syntax (except `LATERAL`). We also support both scalar and correlated subqueries. OurSys, however, currently does not support recursive `WITH`, `LATERAL` joins, `WINDOW` functions, and any built-in functions whose results cannot be reliably reproduced (e.g., `RAND`).

3.1 Data and Execution Model

3.1.1 Table Model and IIDs. OurSys allows users to interact with two types of tables: *base* tables and *derived* tables. Base tables are those that exist in the database, while derived tables are computed from the base tables during execution. To support reproducible execution (including ordering of intermediate result rows), we depart from the default unordered multiset semantics of SQL and instead model each table as an ordered list of rows, each associated with an *internal row id* (*IID*). A table's IIDs must be drawn from a totally ordered domain and uniquely identify the rows within the table. Moreover, the IIDs do not influence the semantics of SQL query operators and should not be considered as extra columns by these operators. For example, if two rows have identical values for all columns (ignoring IID), they should be considered as duplicates even though their IIDs differ.

For a base table with a primary key declaration, we simply define IID to be its primary key value. Otherwise, we choose a compact `UNIQUE` key if one is available. In the worst case, if the table has no keys we use the database system's internal row id (e.g., PostgreSQL's `ctid`); such ids are unique even among duplicate rows.

For a derived table, we define its IID according to how it is computed. For each SQL query operator, we define a *canonical execution procedure* and a *result IID synthesis function* (more details will be provided shortly). The IID synthesis function computes the IID for each result row on the fly during canonical execution, such that the result rows are always produced in the IID order.

In addition to maintaining a reproducible order, OurSys uses IIDs in supporting a variety of debugging features. Therefore, good IID designs positively impact performance, as we will see later in Section 4. While it is technically possible to simply make the IID of a row its sequence number in the result set, such numbers by themselves do not provide any information useful for tracing or possible query rewrite optimizations. As we will see below, most of the IIDs in OurSys are "logical" instead of physical.

3.1.2 Query Blocks and Debugging Contexts. Given a query, OurSys defines a *canonical execution procedure* that is always followed when debugging. A complex query can be viewed as a collection of syntactic blocks with dependencies among them. OurSys models the canonical execution of such queries in terms of function calls –

- The outermost query block defines a function that computes the query result when executed.
- A subquery block defines a function that can be called by the function corresponding to its enclosing query block. A correlated subquery is analogous to a function with arguments.
- Each table defined by `WITH` is a function that computes the table contents when the table is referenced.

Overall, the canonical execution starts by executing the function defined by the outermost query block, which then calls functions corresponding to its subqueries or tables defined by `WITH`, which may further call functions for their subqueries, etc.

As a function may be called multiple times, for each *invocation* of a function, OurSys creates a *debugging context* as needed, analogous to an "activation frame" in a GPL. This debugging context holds information specific to the particular execution of the query block, such as the values of external column references.

3.1.3 Canonical Execution for Each Query Block. Having discussed the overall canonical execution procedure, we now zoom in on the canonical execution of each query block. Due to space constraints, we will only discuss `SELECT` blocks with inner cross joins. Because of the complexity of `SELECT` (including grouping and aggregation), we further decompose its execution into *stages*, where each stage can be seen as a query operator with its own canonical execution procedure and result IID synthesis function.

The first stage in a `SELECT` block is *join & filter*. Its canonical execution is nested for-loops iterating through rows, one for each input table in `FROM`, in order. In the innermost loop, we test the `WHERE` condition (which may involve calling subqueries, potentially with argument values obtained from the loop variables). The result row IID is synthesized as a vector whose components are the IIDs of the joining input rows. Note that the lexicographic order of these vector IIDs is consistent with the row production order.

If the block contains any grouping or aggregation, a *grouping* stage will be next. Its canonical execution is a stable sort of its input rows according to the list of GROUP BY expressions[1] in order. Each result row starts with the GROUP BY expression values as columns, followed by additional columns needed to evaluate the remainder of the query (e.g. HAVING or SELECT expressions). The result row IID is synthesized as a vector whose components are the GROUP BY expressions in order, followed by the IID of the input row. Note that this order puts member rows of a group together, allowing the UI to detect group boundaries and display them specially (Figure 2). The *stable* sort ensures that the order of these vector IIDs is consistent with the row production order.

The final stage of the SELECT block produces the *final output* for the entire block. The canonical execution processes the input rows in order. If HAVING is present, input rows whose group does not pass the HAVING condition will be filtered out. Then, if the previous stage is a grouping stage, we produce one result row for each group of input rows, using the leading portion of the IID corresponding to GROUP BY expressions as the result IID.

3.2 Debugging Operations

We now describe what debugging operations a user can perform with OurSys, focusing on those requiring formalization and more in-depth discussion; others mentioned in earlier sections with straightforward semantics (such as visualization of expression tree evaluation) are omitted. Again, we will describe the operations mostly in the context of SELECT blocks with inner cross joins, although we have also generalized them to other SQL constructs.

3.2.1 Input Combination ("combo") Space and Execution Positioning. For each debugging context, we define an *input combo space* as an ordered set of *input combinations*, each representing a particular point in execution. The (conceptual) current point of execution is called the *active input combo* for the debugging context. For a SELECT debugging context with n input tables, the active input combo is the n input rows being examined inside the innermost loop by the canonical execution of the join & filter stage, and it is represented by an n-dimensional vector whose components are the IIDs of these input rows. For instance, the active input combo of the SELECT debugging context shown in Figure 2 is $\langle s_2, f_1\rangle$, drawn from the input combo space $\{s_0, \ldots, s_5\} \times \{f_0, \ldots, f_3\}$.

The user can position the execution of a debugging context at a particular point using either *stepping* or *teleporting*. With *stepping*, OurSys automatically advances the active input combo to its successor (or predecessor if stepping in reverse order) in the input combo space. With *teleporting*, given any input table, through the paginated display illustrated in Figure 4, the user can jump or scroll to any page and select a particular row as active; OurSys will then update the active input combo accordingly.

3.2.2 Forward Tracing. An active input combo in the debugging context can produce *derivative* rows in the downstream result tables produced by the stages. Informally, the input combo contributes to the computation of derivative rows. As discussed in Section 2, once the active input combo is set, OurSys automatically refreshes all visualizations of expression trees, such that they reflect evaluation over the active input combo or its derivative rows. OurSys also automatically refreshes all result tables in the debugging context, such that they show the pages containing and highlighting the derivative rows. This *forward tracing* feature allows the user to examine the effect of input rows on subsequent processing and potentially understand why the desired effect is not achieved.

Note that for a SELECT block, a given input combo can contribute to at most one result row per stage, so there is no ambiguity in which derivative rows to show downstream. By design, OurSys ensures this rule of at most one derivative per stage for other blocks (such as set/bag operations) as well.

3.2.3 Pinning: Tracing and Watchpointing. We first describe the semantics of *pinning* and then discuss its use for tracing and watchpointing. Consider all tables displayed for a debugging context. OurSys allows the user to *pin* up to one row from each of these tables. Formally, each pinned row defines a subset of the input combo space, called the row's *pinned (input combo) space*. Overall, the pinned space for the debugging context is its input combo space *intersected* with each of the pinned spaces defined by the pinned rows. Intuitively, the pinned space allows the user to narrow the execution down to the points of interest.

Consider a SELECT block joining n tables in FROM with input combo space $\mathbf{R} = \prod_{i=1}^{n} R_i$, where each R_i denotes the ordered list of IIDs for the i-th input table. A pinned row in the j-th input table with IID x defines a pinned space of $\prod_{i=1}^{j-1} R_i \times \{x\} \times \prod_{j=i+1}^{n} R_i$; i.e., the user is interested only in input combos with x participating. A pinned row in a result (intermediate or final) table defines a pinned space of $\{v \mid v \in \mathbf{R} \wedge x \text{ is a derivative row of } v\}$; i.e., the user is interested only in input combos that contribute to x.

Pinning has multiple uses. First, it augments OurSys's tracing capability: besides forward tracing from the active input combo, pinning effectively allows *backward tracing* from a pinned result row produced by any stage to the pinned space of input combos, and then from there, forward tracing to result rows further downstream. Second, combined with stepping, pinning provides a form of *watchpointing*. With a pinned space in effect for the debugging context, OurSys restricts stepping to the pinned space, effectively setting a watchpoint that pauses execution only at points relevant to the pinned rows.

3.2.4 Drilling Down and Pulling Up. As illustrated in Example 2.2, OurSys allows the user to *drill down* into a subquery, analogous to "stepping into" a function call. Besides drilling down into subqueries in WHERE, HAVING, and SELECT expressions, OurSys also allows drilling down into subqueries in FROM, which can be either directly nested therein or via a reference to some WITH definition. In these cases, the user can drill down directly through a particular row in a derived input table; OurSys will open the debugging context for the subquery responsible for producing that table and automatically pin that row in the subquery's final output table.

When debugging a complex query with many nested blocks, OurSys essentially maintains a "call stack" of debugging contexts. To let the user *pull up* from a subquery debugging context, OurSys simply returns the user to the previous debugging context on the stack, which belongs to the enclosing query block. The state of the subquery debugging context is still preserved until the user changes the active input combo in the enclosing block's debugging context, which forces "stepping out" from the last subquery function call.

[1] An aggregate query without GROUP BY can be regarded as having an empty list of GROUP BY expressions.

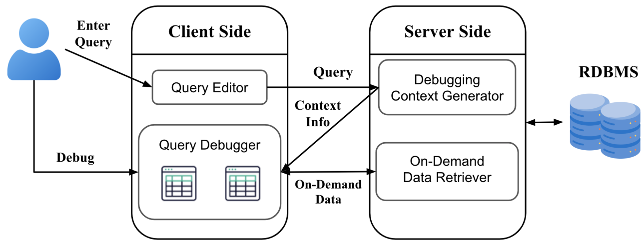


Figure 5: OurSys Architecture

4 SYSTEM AND OPTIMIZATIONS

In this section, we describe the system implementation and optimizations for OurSys. Despite the myriad of details, there are three important high-level ideas. 1) Rather than showing the entire canonical execution of a query Q being debugged, it suffices to let the user examine one small, relevant window of this execution at a time. 2) To obtain all debugging information needed for a particular execution window, instead of performing the canonical execution and instrumenting it, we can formulate SQL queries based on Q to compute such information directly and declaratively. 3) We can judiciously compute some summary data and then use them to further rewrite these queries to be more efficient, without requiring special indexing support in the database itself.

Figure 5 shows the architecture of OurSys, which follows a client-server setup for easy deployment and better maintainability. The client side is a web frontend responsible for rendering and displaying data. It contains two primary interfaces: a query editor and a debugger interface. The server side is an API server in charge of computing data to support the debugger on the client side. Its debugging context generator parses all query blocks and initializes all debugging contexts in a query, and the data retriever responds to subsequent data requests from the debugger. Both services query the user-specified database to compute their corresponding responses to the client. In addition, the API server does not cache or maintain any front-end state in the database, making OurSys a plug-and-play tool that significantly improves usability.

4.1 Optimization Challenges

While it is possible to fetch complete debugging information (e.g., table content, data lineage, etc.) for the entire canonical execution of all query blocks through the debugging context generator at the initialization of their debugging context, such an approach is not scalable.

Example 4.1. Consider a TPC-H [10] database instance generated with a scale factor of 1 (i.e., data size of all tables is 1GB), and the following instantiated inner query of Q8 in the TPC-H benchmark:

```
SELECT EXTRACT(year from o_orderdate) as o_year,
       l_extendedprice * (1 - l_discount) as volume,
       n2.n_name as nation
FROM part, supplier, lineitem, orders,
     customer, nation n1, nation n2, region
WHERE p_partkey = l_partkey AND s_suppkey = l_suppkey
AND l_orderkey = o_orderkey AND o_custkey = c_custkey
AND ... -- omit for simplicity
```

The minimum debugging information required to be poured into the client side is as follows:

There are ten stage tables fetched from the database: eight input tables, a join & filter table, and an output table. The data lineage

Item	Size
All table contents for all execution stages	1825 MB
Tuple-level data lineage	88 MB

contains two parts: the lineage between input tables and the join & filter table and the lineage between the join & filter table and the output table. While most browser tabs use less than 1GB of memory, the combined data here approaches 2GB. Allocating 2GB to a single browser tab is inefficient, especially when users examine queries in multiple tabs, each running as a separate process.

To summarize, the true challenge for OurSys is to accommodate large amounts of data with limited memory while ensuring scalability and interactive experience for query debugging. By focusing on what a user is capable of exploring at once, we state the intuition for an on-demand data retrieval solution:

- For each stage table, we implement a pagination technique and only retrieve the pages of tuples that the users focus on.
- For data lineage and expression evaluations, we retrieve them on a per-tuple level upon users' request, minimizing the computation on the server side to ensure an interactive experience.
- All data retrieval is done by rewriting part of or the entire original query, thus maintaining no state or materialized view in the database.

On-demand Retrieval Overview. On-demand data retrieval is done in two phases. In the first phase, for all stage tables in all query blocks, the debugging context generator creates metadata for quickly retrieving a particular page of a stage table and passes the metadata to the client side. While this can trigger a wait for inefficient queries as OurSys collects metadata on every stage table, the metadata helps guarantee interactive speed in subsequent debugging operations. In section 5, we show that collecting metadata is actually significantly more efficient than caching the entire table.

In the second phase, when a user starts to explore a particular page in any stage table, the client-side debugger makes requests to the server-side data retriever based on the metadata to fetch data only within the specified page. The computation overhead is relatively small since the page size (i.e., the number of tuples in a page) is usually much smaller than the entire table. If the user performs a debugging operation, the client-side debugger also relies on the metadata to fetch the corresponding data lineage that supports the operation, and such computation overhead is also insignificant on the server side, as debugging operations primarily act on a small number of tuples. As a result, OurSys avoids lengthy interruptions during a debugging session.

Baseline Approach for Page Retrieval. Before diving into the details of data retrieval, we realize that the current, most ubiquitous way of pagination is to rely on `OFFSET` and `LIMIT` to specify the page of interest. While this approach may work in fetching the leading pages of a query, it has significant drawbacks:

- The execution time grows rapidly as the `OFFSET` becomes larger. If a user is interested in the trailing pages of a table, using `OFFSET` and `LIMIT` essentially requires the database to execute the entire query every time the user jumps to a different page.
- If an explicit order is defined, using `OFFSET` and `LIMIT` has to execute the entire query to sort and determine the correct page,

which can take a long time and significantly worsen the debugging experience. Due to the necessity of stable orders on tables, OurSys cannot use `OFFSET` and `LIMIT` for pagination.

Cache Mechanism. Building on top of the on-demand data retrieval, OurSys caches computed pages and lineages on the client side to guarantee smooth browsing of frequently visited pages and avoid repetitive computation of the same content. It maintains an LRU-style cache whose max size is limited to 500 MB. Furthermore, OurSys caches the results of all independent scalar queries (given the minimal size of their results) and rewrites the query to replace the scalar query with the cached constant at the time of execution to further speed up the query.

4.2 Efficient Data Retrieval

OurSys collects statistical summaries for each page (done by the debugging context generator) and later uses the summaries as potential pushdown predicates to efficiently fetch a specific page through the on-demand data retriever. The metadata of a stage table contains two parts: (1) an ordered list of *milestones*, each of which contains statistical summaries of a page, and (2) a *table query* that produces the entire output of the table.

The composition of a milestone is as follows:

- The sequence number of the first tuple in the page to keep track of the page offset in the entire table.
- Minimum tuple IID to locate the first tuple on the page.
- Tuple count of the page for the client side to know the page size of the last page in case the total tuple count is not divisible by the user-specified page size (i.e. in situations where the last page has fewer tuples than the other pages).
- The minimum and maximum of all columns with an index in the database on the page which later provides the query optimizer with hints for potential optimization when fetching a page.
- A bloom filter on the external bindings (i.e., the columns referenced in a correlated subquery) only if correlated subqueries exist. The bloom filter is implemented as an extension to the database, and the purpose is to "short-circuit" the query on the input combos that do not evaluate to true on the subqueries, thus avoiding expensive subqueries.

To obtain milestones, the debugging context generator creates and executes a *milestone query* for all tables. Then, the context generator pairs the milestones with a corresponding table query and ships them to the client. The milestone query of a table is composed as follows:

(1) Given the table query, first replace the `SELECT` clause with a sequence number, a tuple IID, the number of tuples, and all indexed columns. Use the window function `ROW_NUMBER()` to create the sequence number and order the tuples by IID.
(2) Given the above query result, group the result by the page size, then project the minimum sequence number, minimum IID, and all the statistics summaries.

Example 4.2. Consider the following TPC-H query:

```
SELECT p_name, COUNT(DISTINCT o_orderkey) num_order
FROM part, lineitem
WHERE p_partkey = l_partkey
AND EXISTS (
    SELECT * FROM orders WHERE o_orderkey = l_orderkey)
GROUP BY p_name
ORDER BY p_name, num_order;
```

The following is an example of the join & filter table query, which evaluates the implicit join (in the `FROM`) and the conditions (in the `WHERE`):

```
-- table query
SELECT * FROM part, lineitem
WHERE p_partkey = l_partkey
AND EXISTS (SELECT * FROM orders WHERE o_orderkey = l_orderkey);
-- milestone query
WITH tmp(seq, iid, p_partkey, l_orderkey, l_linenumber) AS (
  -- modified table query
  SELECT ROW_NUMBER() OVER (ORDER BY p_partkey, l_orderkey, l_linenumber) - 1,
         ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)),
         p_partkey, l_orderkey, l_linenumber -- indexed columns
  FROM part, lineitem
  WHERE p_partkey = l_partkey
  AND EXISTS (SELECT * FROM orders WHERE o_orderkey = l_orderkey))
-- MIN_IID is a user-defined function to find the minimum IID
SELECT MIN(seq), MIN_IID(iid), COUNT(*) count,
       MIN(p_partkey), MAX(p_partkey), -- statistical summaries
       MIN(l_orderkey), MAX(l_orderkey),
       MIN(l_linenumber), MAX(l_linenumber),
       BLOOM_FILTER(l_orderkey)
FROM tmp
GROUP BY seq / 50 -- page size 50
ORDER BY seq / 50;
```

Suppose the milestone query returns the following results; headers are abbreviated:

seq	min IID	cnt	p_key	o_key	line #	bloom
0	((1), (1,3))	50	[1,9]	[1,4]	[3,6]	00100111
50	((9), (3,5))	20	[9,11]	[2,5]	[3,6]	10101010

When a user wants to retrieve a specific page, the client rewrites the table query to a *page query* by using the IIDs in the milestones to precisely define the page as well as appending extra pushdown conditions to the end of the table query based on the statistic summaries (i.e., min/max of the indexed columns). Once the rewrite is done, the client delivers the rewritten query to the on-demand data retriever, and the retriever executes the query and sends back the results. This process is also illustrated in Figure 5. Below is a sample query to fetch the first page of the join & filter table:

```
SELECT ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)), -- tuple IID
    *
FROM part, lineitem
WHERE p_partkey = l_partkey
AND EXISTS (
    SELECT * FROM orders
    WHERE o_orderkey = l_orderkey
    -- extra pushdown to subquery
    AND o_orderkey BETWEEN 1 AND 4
-- IID guarantees a page of 50
AND ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)) >= ((1), (1,3))
AND ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)) < ((9), (3,5))
-- indexed column pushdown
AND p_partkey BETWEEN 1 AND 9
AND l_orderkey BETWEEN 1 AND 4
AND l_linenumber BETWEEN 3 AND 6
AND BLOOM_TEST(00100111, l_orderkey) -- if false, no need to evaluate EXISTS
ORDER BY p_partkey, l_orderkey, l_linenumber;
```

Lineage Retrieval. When fetching a page, the page query also retrieves the lineage of each tuple in the page. In the previous example, there is no explicit `SELECT` expression that projects the lineage because the IID of the join & filter table is the lineage, as it is equivalent to the input IID combo. However, it is not clear how to retrieve the lineage of a group in the group table since a group consists of multiple joined tuples, and it is infeasible to aggregate all joined tuples as the lineage since the data size can be large. OurSys chooses the minimum joined tuple IID as the lineage, and we show how this choice helps with pinning later.

Remark. OurSys performs recursive predicate pushdown for subqueries. As shown in the previous example, it is natural to push down `l_orderkey` to the subquery. Furthermore, OurSys also pushes down predicates to derived input tables, i.e., OurSys only fetches the necessary input pages from the derived input table when computing downstream stage tables in the outer query. However, such a pushdown mechanism does not work when the range of the indexed column is too large or when the false positive rate of the bloom filter is too high. Therefore, OurSys sets a threshold for both range overlap (default to 0.3) and the false positive rate (default to 0.5). If the overlap or false positive rate is higher than their corresponding threshold, the corresponding condition will be removed to prevent ineffective predicate pushdown that can potentially slow down query execution.

4.3 Debugging Support

We now describe how OurSys supports in-context debugging operations (e.g., forward tracing, pinning, stepping).

4.3.1 Forward Tracing. Forward tracing enables users to select a particular input combo from each input page and observe how it propagates through the execution stages. Therefore, we safely assume that the client side has already fetched the input pages that contain the input combo before users can move the cursor of each input table to the input tuples of interest.

Knowing the input combo, OurSys first computes the downstream tuples (i.e., tuples in the join & filter, group, and output table) this input combo contributes to, and then it locates the pages that contain the downstream tuple. With the located pages, the client-side debugger can then request the page through the data retriever following the previous procedures. Thus, forward tracing is achieved in two phases. In the first phase, the client sends the selected input combo along with all downstream table queries to the data retriever, which then rewrites the table queries to compute the corresponding downstream tuples in the downstream stage tables and ships them back to the client. The query rewrite is done in the following order:

(1) The input IID combo is formulated as a condition and appended to the join & filter query. The data retriever executes the query. If the query returns an empty result, we stop here, and no other query needs to be executed as this input combo does not satisfy the `WHERE` clause/join condition.
(2) If the previous query returns non-empty results, the input combo must contribute to a group. Therefore, the input IID combo can be injected into the group table query. By executing the query, we obtain the group's IID.
(3) Inject the group's IID as a `WHERE` condition to the output table query. By executing the rewritten output query, we can compute the IID of the output tuple. The reason why we use the group's IID instead of the input IID combo is that the group formed by a single input combo might not pass the `HAVING` clause.

Consider the query in Example 4.2. Suppose the selected input combo is `((1), (1,3))`, and it contributes to the group ("`Nail`"). The on-demand data retriever rewrites the queries for the first phase as follows:

```
-- join & filter
SELECT ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)), -- tuple IID
       p_partkey, l_partkey, p_partkey = l_partkey, -- expression evluation
       EXISTS(SELECT * FROM orders WHERE o_orderkey = l_orderkey),
       p_partkey = l_partkey AND EXISTS(SELECT * FROM orders WHERE o_orderkey
              = l_orderkey)
FROM part, lineitem
WHERE p_partkey = l_partkey
AND EXISTS(SELECT * FROM orders WHERE o_orderkey = l_orderkey)
ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)) = ((1), (1,3));
-- group
SELECT ROW(p_name) -- tuple IID
FROM part, lineitem
WHERE ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)) = ((1), (1,3))
GROUP BY p_name;
-- output table
SELECT ROW(p_name) -- tuple IID
FROM part, lineitem
WHERE p_partkey = l_partkey
AND EXISTS(SELECT * FROM orders WHERE o_orderkey = l_orderkey)
AND ROW(p_name) = ROW('Nail')
GROUP BY p_name;
```

Note that here OurSys intentionally augments the join & filter query further to include the evaluation expression of all nodes on the predicate tree, so that this information can also be sent to the client side to render the predicate tree as shown in Figure 2.

In the second phase, the client uses the computed downstream tuples to locate the page (since tuples are sorted by IID, this can be quickly done via binary search) and requests those pages through the data retriever, which then efficiently fetches the pages based on the mechanism described previously.

4.3.2 Pinning. In the case of *SPJ* queries, pinning is essentially the same as forward tracing, as pinning any tuple in the join & filter table or output table traces back to only one input combo. Since the IIDs of these two tables are the input IID combos, the client side first fetches the corresponding input pages by decomposing the IID of the pinned tuple and then invokes the procedure of forward tracing.

For *SPJA* queries, pinning a tuple in the join & filter table is the same as forward tracing, as it traces back to only one input combo. Pinning in a group table or an output table automatically forward traces the first input combo in the pinned input combo space, and such information is returned by the page query as shown in Example 4.2. However, additional computation is required to compute if a tuple in the page is in the pin space, as the pin space for each table potentially contains more tuples. OurSys achieves such computation by inserting an extra Boolean `SELECT` expression in the page query as the flag to indicate if a tuple is in the pin space.

The inserted Boolean `SELECT` expressions differ for different stages, and OurSys works backward when determining the pin space for each table. Assuming a pin is placed in the output table, we can then use its IID to determine the corresponding group in the group table and create a corresponding equality condition as the flag `SELECT` expression. With the evaluation of the `GROUP BY` expressions, we can further create equality conditions as flags for the join & filter table to check if a single input combo contributes to the designated group. Finally, we check if an input tuple is in the pin space by verifying its contribution to the pin space of the join & filter table.

Consider the query in Example 4.2 again. The data retriever rewrites the page queries for upstream tables as follows, assuming a pin is placed on the output tuple (“`Nail`”, 30):

```
-- group
SELECT ROW(p_partkey), p_name,
       p_name = 'Nail' -- Boolean flag indicates pin space
FROM part p1, lineitem l1
WHERE ROW(p_name) >= 'Hammer'
AND ROW(p_name) < 'Screwer' -- define the page, 'Nail' is included
AND ... -- original WHERE predicates and pushdown conditions
GROUP BY p_name;

-- join & filter
SELECT ROW(p_partkey), *,
       p_name = 'Nail' -- Boolean flag indicates pin space
FROM part, lineitem
WHERE ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)) >= ((1), (1,3))
AND ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)) >= ((9), (3,5))
AND ... -- original WHERE predicates and pushdown conditions

-- input part, lineitem follows a similar style
SELECT ROW(p_partkey), *,
       EXISTS(
           SELECT * FROM part p2, lineitem l2
           WHERE p2.p_partkey = l2.l_partkey
           AND EXISTS (SELECT * FROM orders WHERE o_orderkey = l2.l_orderkey)
           AND p_name = 'Nail'
           AND ROW(p1.p_partkey) = ROW(p2.p_partkey) -- match IID
       )  -- Boolean flag indicates pin space
FROM part p1
WHERE ROW(p_partkey) >= ROW(1) AND ROW(p_partkey) < ROW(9);
```

As shown above, the extra `SELECT` expressions are inserted into the page queries of the corresponding tables. Therefore, it causes only a small extra computation overhead and does not affect the overall interactive experience of OurSys.

4.3.3 Stepping. Stepping can be performed in only two scenarios: (1) when there is no pin in any table, users essentially step forward/backward through the entire input combo space in the nest-loop order, (2) when there are pins in tables but the pinned input combo space has not been restricted to only one input combo (e.g., pinning a group).

Stepping through the entire input combo space can easily be achieved by fetching new pages followed by forward tracing because the next input combo is predictable, given the nest-loop order. On the contrary, stepping through a pinned input combo space needs special accommodation as the following input combo in the pinned input combo space cannot be predicted.

Therefore, the idea is to step in a pinned input combo space by using the pinned tuples as extra predicates to compute the following input combo given the current input combo. Since the IID of the join & filter table represents the input combo, the problem can be converted to find the next joined tuple in the pin space of the join & filter table, and the rewrites follow naturally.

Consider the query in Example 4.2 again. The data retriever rewrites the query of the join & filter table, assuming a pin is placed on the output tuple (“`Nail`”, 30):

```
SELECT ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber))
FROM part, lineitem
WHERE p_partkey = l_partkey
      AND EXISTS (SELECT * FROM orders WHERE o_orderkey = l_orderkey)
      AND ROW(ROW(p_partkey), ROW(l_orderkey, l_linenumber)) >
            %(current_input_combo_id)
      AND p_name = 'Nail' -- pin-induced predicate
ORDER BY p_partkey, l_orderkey, l_linenumber
LIMIT 1;
```

The above query computes the next input combo in the pinned input combo space, and the client can invoke the forward tracing procedure to move the input cursors to the next input combo. If the users usually take multiple steps in the input combo space, the client can choose to increase the `LIMIT` and cache the results.

5 PERFORMANCE EXPERIMENTS

We conducted experiments to evaluate the performance of our optimizations in Section 4. To compare our optimization with the baseline approach (i.e., `OFFSET` and `LIMIT`) on the efficiency of fetching pages, we ran three evaluations independently to examine how much performance improvement can be achieved

- via bloom filter by short-circuiting correlated subqueries.
- via predicate pushdown based on the statistical summaries.
- in general by combining all the optimizations (i.e., predicate pushdown, bloom filter, and cached independent scalar query).

Workload and Tasks. To evaluate the above items independently, we used TPC-H [10] as our benchmark database. We generated three database instances of sizes 1GB, 5GB, and 10GB. In addition to the primary indexes on the primary keys of each table, we also created a reasonable set of secondary indexes across multiple tables in the instances (details in Appendix A) to reflect a more realistic scenario. For the first two evaluations, we cherry-picked one query for each to show the effectiveness of the corresponding optimization. For the general evaluation, we applied all possible optimizations to all 22 queries and reported general results.

Test Environment. All experiments were done locally on a 64-bit Ubuntu 22.04 LTS server with Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, 64 GB RAM, and 256 GB disk space. We used PostgreSQL-16 [54] and only changed the following configurations: We turned off parallelism for a fair comparison, set the `work_mem` to 128MB, and set the `shared_buffers` to 8GB.

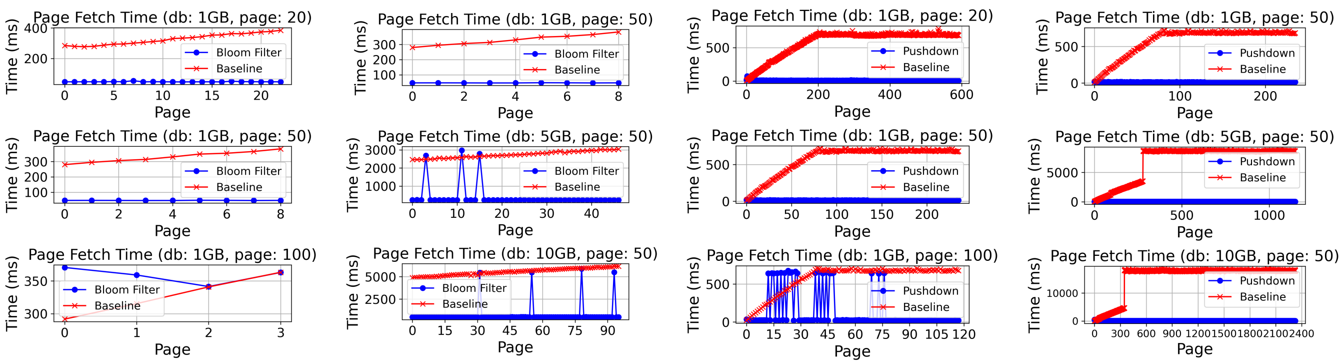
5.1 Bloom Filter Evaluation

We chose the join & filter table query from $Q2$ in the TPC-H benchmark for bloom filter evaluation:

```
SELECT ROW(p_partkey, s_suppkey, ps_partkey, ps_suppkey,
      n_nationkey, r_regionkey), *
FROM part, supplier, partsupp, nation, region
WHERE ... AND ps_supplycost = (
    SELECT MIN(ps_supplycost)
    FROM partsupp, supplier, nation, region
    WHERE p_partkey = ps_partkey -- p_partkey is external binding
    AND ...)
```

`p_partkey` is the only external binding we created a bloom filter for in the milestone query, which we applied in the page query to quickly fetch the page. By identifying if a specific `p_partkey` is in the page of interest, the page query is able to potentially avoid evaluating the expensive correlated subquery since the `WHERE` clause of $Q2$ is conjunctive. For the bloom filter, we set the number of bits (denoted by m) to 1024 by default, the estimated number of unique `p_partkey` in each page (denoted by n) was set to the page size, and the number of hash functions needed (denoted by k) was calculated by the formula: $k = \frac{m}{n} \times \ln 2$.

We ran two sets of experiments. First, we fixed the database size to 1GB and varied the page size. Second, we fixed the page size to 50 and varied the database size.



(a) Bloom Filter Test Set 1 **(b) Bloom Filter Test Set 2** **(c) Predicate Pushdown Test Set 1** **(d) Predicate Pushdown Test Set 2**

Figure 6: Experiments for Independent Evaluation of Bloom Filter and Predicate Pushdown

In both sets of experiments, we collected the execution time reported by the `EXPLAIN ANALYZE` command for all pages and observed its trend. We compared the bloom filter optimization with the baseline approach which use `OFFSET` and `LIMIT` to fetch a page. The results for both sets of experiments are shown in Figure 6a and Figure 6b, from which we derive the following conclusions:

- When bloom filter is effective, it outperforms the baseline approach. For page size of 50, the bloom filter keeps the execution time of the page fetch at around a tenth of the execution time of the baseline.
- Bloom filter is sensitive to page size as shown in Figure 6a. As the page becomes larger, it is likely more unique values are inserted into it, increasing the false positive rate. In the case where the bloom filter is ineffective, the DBMS pays extra cost to evaluate it with no benefit returned, causing the bloom filter to run more slowly with a page size of 100.
- Bloom filter is rather insensitive to the database size as shown in Figure 6b. While there are a few outliers, bloom filter performs fairly stably overall.

5.2 Predicate Pushdown Evaluation

We chose the join & filter table query from *Q7* in the TPC-H benchmark for evaluation of predicate pushdown:

```
SELECT ROW(s_suppkey, l_orderkey, l_linenumber, o_orderkey,
       c_custkey, n1.n_nationkey, n2.n_nationkey), *
FROM supplier, lineitem, orders,
     customer, nation n1, nation n2
WHERE ...
```

In addition to the primary keys of each table, we created pushdown conditions for any secondary indexed columns. Before executing the page query, we checked the pushdown range of the columns and removed all conditions whose ranges were more than 30% of the active domain of the corresponding columns, and this helped prevent us from "over-hinting" on the database and causing performance to degrade subsequently. We ran two sets of experiments similar to the evaluations for the bloom filter. The results are shown in Figure 6c and Figure 6d, based on which we can have the following conclusions:

- While predicate pushdown might perform similarly as the baseline for fetching the leading pages, its execution time does not grow linearly, resulting in a huge advantage over the baseline when fetching middle pages or trailing pages in a table. As shown in Figure 6c and Figure 6d, such an advantage can save about two orders of magnitude on the execution time as shown in Figure 6c when the database size is 10GB and the page size is 50. The execution times for the baseline and pushdown optimization are around 42000ms and 500ms for trailing pages, respectively.
- Based on Figure 6c, predicate pushdown is also sensitive to the page size. As larger pages contain larger ranges, it becomes less likely to trigger a fast index scan in the database.
- Based on Figure 6d, the predicate pushdown optimization is not sensitive to the database size, and it provides great performance even if the database size is fairly large.

5.3 General Evaluation

For the general evaluation, we applied the optimizations to all queries in the TPC-H benchmark. The predicate pushdown optimization can be applied to all queries, while bloom filter and scalar caching can be only applied to Q4, Q16, Q18, Q20-22 and Q2, Q15, Q11, Q22, respectively. For all tables in all the query blocks of each TPC-H query, we collected the following data:

- The execution time of the milestone query, and size of its output.
- The execution time of the table query itself, and size of its output.
- The execution time of page query and the baseline query (i.e., `OFFSET` and `LIMIT`) for the first page, middle page and the second last page of the table. We chose the second last page instead of the last page to guarantee the fetch of a full page for fair comparison.

5.3.1 Milestone Query. Due to the space constraints, Table 1 shows a partial comparison between top-3 longest-running milestone/table query pairs with 1GB database and page size of 50 (complete results in [33]). All of them are from Q18, which is the most expensive query in the benchmark.

	Milestone Query		Table Query	
Q18	Exec. Time (ms)	Output (MB)	Exec. Time (ms)	Output (MB)
join & filter	53581.53	52	17478.41	1272
group	19449.81	9	18503.99	70
output	20030.26	9	18415.74	75

Table 1: Comparison Between Milestone and Table Query

In summary, the milestone query usually runs longer than the table query as it performs extra computation on top of the table query. However, its output is usually much smaller than that of the table, making it easier to process for the client side.

	1GB		5GB		10GB	
Q18 join & filter	Opt.	Base	Opt.	Base	Opt.	Base
head page	24.21	3876.84	255.36	34380.39	10617.46	158664.23
middle page	25.32	6863.59	326.19	72862.8	9497.6	181742.56
tail page	19.42	9850.34	244.22	111345.2	9660.1	204820.89

Table 2: Page Fetch Time Comparison between OurSys and Baseline. Unit: millisecond

5.3.2 Page Fetching for All Queries. To show the scalability of OurSys we show the execution time comparison between the page query and baseline query for Q18 (the most expensive query in the benchmark) across all database instances in Table 2, with a fixed page size of 50. While the baseline can potentially take minutes to return a page on a large database, OurSys returns the results in at most 10 seconds. The full results for all queries are presented in the Appendix A. In summary, the conclusion follows from the independent evaluation of bloom filter and predicate pushdown: the combined optimization is generally sensitive to page size and insensitive to the database size. It performs no worse (sometimes better) than the baseline approach when fetching leading pages, and it always significantly outperforms the baseline approach when fetching a middle/trailing page.

Final Remark. The execution times reported in the experiment are taken from the `EXPLAIN ANALYZE` command, which only measures the time spent by the PostgreSQL process. Such execution times do not include the time between sending the command and receiving results, which accounts for network latency. We intentionally chose to exclude such time from our experimentation because the output can potentially be quite large and a test conducted took an hour for the psql client to fully receive the entire output of the join & filter table of Q18. This is yet another strong indication that executing the table query and caching the entire output is inadequate and entirely unsuitable for interactive debugging.

6 USER STUDY

We conducted a user study in an undergraduate database course to evaluate the overall effectiveness of OurSys on two aspects: (1) does OurSys help users catch more logical bugs? (2) does OurSys reduce the time to identify logical bugs?

Participants. We recruited 237 students from the course, who were SQL beginners and had just become familiar with SQL at the time of the user study. The participation was voluntary except for the incentive of extra exercise to practice debugging skills. We considered the possibility of recruiting participants from other sources (e.g. Amazon Mechanical Turk) but decided against it because it was hard to quantify and control participants' SQL familiarity to a similar level, potentially yielding inaccurate results on OurSys as SQL familiarity has a significant impact on the debugging time.

Preparation and Setup. The user study was conducted during two consecutive weekly 75-minute discussion sessions. In the first session, students were given a tutorial on OurSys and informed about the format of the survey-style quiz, which contained two SQL debugging questions. After the first discussion, OurSys is made public for students to get familiar with. In the second session, students completed the quiz synchronously in a proctored environment, and they were asked not to discuss the questions with classmates. For each question in the quiz, students were provided a problem statement, an incorrect query, its incorrect output, and the expected correct output. Q1 featured `UNION` and aggregation, containing two

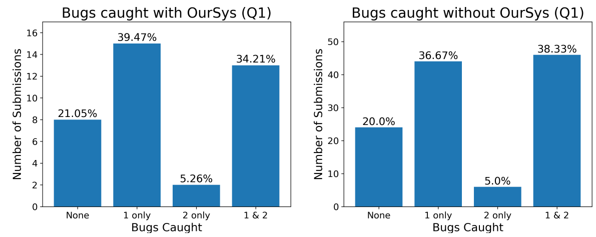


Figure 7: Bugs Caught Distribution for Q1

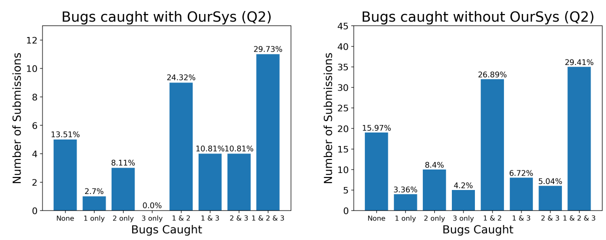


Figure 8: Bugs Caught Distribution for Q2

logical bugs. Q2 featured `OUTER JOIN`, `NULL`, `GROUP BY`, and `EXISTS`, containing three logical bugs. The testing database instance in the user study had been made public and installed by students early in the course. The incorrect and expected outputs were generated based on that instance. We also prepared a remote pgAdmin web interface for students during the debugging sessions to free them from setting up their own instances, eliminating potential noise in the measurement of debugging time.

Tasks. To create treatment and control groups, students received the two questions in a random order. For the first question they received, they were free to use any tool of their choice (e.g., Gradescope autograder, Postgres console, pgAdmin web interface) except for OurSys. For the second question, OurSys was made available along with all other tools, and students had the option to use/not use OurSys and were asked to indicate it in the survey. For each question, students were to identify the mistakes in a free response text box. Students self paced, but they were recommended to spend about 15-20 minutes on each question. Students were not allowed to move onto the second question until they answered the first.

Results and Analysis. We collected data on the time it took students to answer each question, whether they chose to use OurSys and the mistakes they noted. Of 237 students, 140 participated in the study with complete and valid responses to both questions. Therefore, we based our analysis on these 140 responses. In summary, 73 students received Q1 first, and 67 received Q2 first, and they completed the question without OurSys. Consequently, 73 and 67 students received Q2 and Q1, respectively, as the second question with the OurSys option. When OurSys was available, 36 (for Q2) and 29 (for Q1) students chose not to use OurSys. Therefore, for Q1, we have 38 submissions with OurSys and 102 without OurSys (29 of them had OurSys option); for Q2, we have 37 submissions with OurSys and 103 without (36 of them had OurSys option).

The authors manually reviewed each valid response and verified the correctness of student-identified logical bugs. Given two bugs in Q1 and three in Q2, the distributions of identified bugs are shown in Figure 7 and Figure 8 respectively. Each chart shows how many submissions caught the specified bugs for all possible combinations of caught bugs. While the numbers of submissions differ between using/not using OurSys, the distributions were similar. In fact, the average number of bugs identified are similar regardless of the usage of OurSys (shown in the last column of Table 3). Though this observation does not indicate OurSys helped students find more bugs, we notice a drastic drop in the debugging time as shown in Table 3, where using OurSys reduces the average debugging time for Q1 by 464.14 seconds and Q2 by 510.49 seconds. Based on the comparison between the bug combination distributions and the comparison between the debugging time, we conclude that using OurSys significantly improves debugging efficiency by helping users identify mistakes faster. The fact that all students had a similar familiarity with SQL further strengthens this argument.

Query	# Responses	Avg. Time (s)	Avg. Bugs
Q1 w/ OurSys	38	771.52	1.13 / 2
Q1 w/o OurSys	102	1235.66	1.18 / 2
Q2 w/ OurSys	37	1147.04	1.91 / 3
Q2 w/o OurSys	103	1657.53	1.85 / 3

Table 3: Average Time Taken and Bugs Caught per Question

7 RELATED WORK

Query Semantics Debugging. There are two main lines of work toward debugging query semantics. The first line helps users debug wrong queries against a correct or reference query to find the query syntax that cause semantic difference. Therefore, this line differs from OurSys fundamentally. XData [22] checks the correctness of a query by running the query on self-generated testing datasets. Cosette [24–26], SQLSolver [31] and QED [60] focuses on testing query equivalence using constraint solvers and theorem provers. RATest [50] and Cinstance [34] aim at constructing small and illustrative database instances to show the differences between two queries. [21] developed a grading system that canonical-izes queries with rewrite rules and then decides query similarity based on a tree-edit distance between logical plans. SQLRepair [55] and QR-Hint [40] fix the wrong query by proposing syntax edits.

The second line, more related to OurSys, focuses on debugging a query without a reference query. Qex [59] is a tool for generating input relations and parameter values for unit-testing parameterized SQL queries. SQLLint [12–14] detects and alerts users about suspected semantic errors in a query, but it does not help users step through the execution. Habitat [29, 37] is a query execution visualizer that allows users to highlight parts of a query and view their intermediate results, but it does not provide any explanation on how and why some results are produced. DESQL [39] is a debugger for SQL in Spark, which decomposes the query into subqueries and help users examine the subqueries output, but it does not provide explanation on how outputs are produced. While I-Rex [41, 49] built an interactive interface with several debugging operations to step through execution, it does not accommodate large-scale data.

The following areas also contribute ideas to query debugging:

Data Exploration. DataPlay [5] lets users directly manipulate a graphical query by changing quantifiers and dependencies among constraints, and provides real-time updated query answers/non-answers to show how the output reacts to changes. As users input various queries, AIDE [30] and SQL QueRIE [7] learns on users' query history and "predicts" queries that retrieve interesting data for users. Explique [46] extends users' queries by suggesting an extra set of possible selection predicates that potentially help users zoom into a specific area of the original answers. Toward the purpose of semantic debugging, Caballero et al. [16, 17] builds a computation tree which represents the execution flow of a query, guides users to annotate the output tuples and notifed users when unexpected results are produced.

Data Provenance. Data provenance [11, 15, 23, 36, 42] provides a natural way to trace query execution. It gives explanation to why and how a particular output is or is not produced. Many frameworks have been built to capture provenance in relational systems [6, 27, 45]. More recently, the Perm [35] and GProM [9] system extends PostgreSQL with support for provenance capture implemented using instrumentation and relational encoding, respectively. PUG [47] extends GProM with support for capturing and summarizing provenance for queries with negation. Smoke [56] implements provenance capture in a main-memory database system. In addition, many provenance-capturing frameworks [8, 28, 43] target big data platforms like Spark and MapReduce.

Query Visualization. Many interactive query builders ([1–5]) employ block diagrams to show the interactions among tables and their attributes. This line of work focuses on helping users construct queries from scratch, and users can manipulate the diagrams to build desired queries. Starburst [38] develops the Query Graph Model to help users understand query plans, but the generated graphs are usually disconnected from the query syntax; GraphSQL [20] and Visual SQL [44] follow the paradigm of entity-relationship representation to create query diagrams. QueryVis [48] creates unambiguous diagrams to capture comprehensive semantics of complex SQL queries, and has shown such diagrams effectively improve users' ability to understand query semantics. SQLVis [51] shares similar approach with QueryVis, relying on graphs to express the relationship among tables and attributes.

While the above approaches have demonstrated significant advantages for some specific tasks, they are not comprehensive enough for SQL debugging in a general and organized way. Browsing through intermediate/final query outputs enables users to observe and reflect directly on their queries, but it lacks context on how the output is produced. While data provenance seems to be a good complement, it relies on a query plan to identify how and why tuples are produced or missing during query execution and does not directly map plan operators to the original query syntax, making it difficult to reveal problematic SQL constructs, especially for SQL novices. Both approaches are incapable of handling large amounts of data as 1) computing provenance for large data is infeasible, and 2) query optimizers might generate different plans for the same query, causing arbitrary and unstable order of the output rows that is not cognitively possible for users to keep track of. Query visualization usually requires users to learn new representations, which potentially creates extra cognitive burden.

Efficient Query Evaluation. Besides the debugging paradigm, OurSys also accommodates large data. However, most of the related work focuses on optimizing the query evaluation without previous knowledge of the query, and this differs fundamentally from our setting. The line of work in direct access to query answer [18, 19, 32, 58] provides methods to access a random page in the query result, but it is restricted to conjunctive queries only. Therefore, it is not applicable in OurSys as OurSys needs to support large SQL fragments. Previous work on data skipping and predicate pushdown [52, 53, 57, 61] requires significant changes in the database systems. As a result, their proposed techniques are not applicable in OurSys.

8 CONCLUSION AND FUTURE WORK

As SQL continues to be a leading querying and programming language, tools for understanding SQL queries maintain their importance both for helping novices understand and learn relational queries and to aid experts and practitioners in debugging their own complex queries. Building upon the work discussed in the previous sections, we present OurSys, an interactive logical SQL debugger that allows users to examine the execution of SQL queries following the way they are written and debug them in a general and organized way. We define a debugging paradigm for SQL queries by making analogies to the debugging of general-purpose programming languages (GPLs). We develop multiple powerful debugging operations that support tuple-level examination of the execution. We designed and developed OurSys to accommodate debugging over large databases while maintaining an interactive experience. We conducted performance experiments and a user study to demonstrate the efficiency and effectiveness of OurSys. The development of OurSys inspires multiple directions for future work, including but not limited to: (1) design and develop more debugging operations to better support debugging over large databases, and (2) support the examination of runtime errors, to which database systems usually provide uninformative hints (e.g. “ERROR: division by zero”).

REFERENCES

[1] 2023. dbForge. https://www.devart.com/dbforge/mysql/querybuilder/.

[2] 2023. Microsoft Access. https://www.microsoft.com/en-us/microsoft-365/access.

[3] 2023. PgAdmin. https://www.pgadmin.org/.

[4] 2023. Rapid SQL. https://www.idera.com/rapid-sql-ide/.

[5] Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. 2012. Dataplay: interactive tweaking and example-driven correction of graphical database queries. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*. 207–218.

[6] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. 2006. Trio: A system for data, uncertainty, and lineage. In *VLDB*, Vol. 6. 1151–1154.

[7] Javad Akbarnejad, Gloria Chatzopoulou, Magdalini Eirinaki, Suju Koshy, Sarika Mittal, Duc On, Neoklis Polyzotis, and Jothi S Vindhiya Varman. 2010. SQL QueRIE recommendations. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1597–1600.

[8] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. 2011. Putting lipstick on pig: Enabling database-style workflow provenance. *arXiv preprint arXiv:1201.0231* (2011).

[9] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM-a swiss army knife for your provenance needs. *A Quarterly bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering* 41, 1 (2018).

[10] TPC Benchmark. [n.d.]. http://www.tpc.org/tpch.

[11] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-based why-not provenance with nedexplain. In *Extending database technology (EDBT)*.

[12] Stefan Brass and Christian Goldberg. 2004. Detecting Logical Errors in SQL Queries. In *Tagungsband zum 16. GI-Workshop Grundlagen von Datenbanken, Mohnheim, NRW, Deutschland, 1.-4. Juni 2004*, Mireille Samia and Stefan Conrad (Eds.). Universität Düsseldorf, 28–32.

[13] Stefan Brass and Christian Goldberg. 2005. Proving the Safety of SQL Queries. In *Fifth International Conference on Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia*. IEEE Computer Society, 197–204. https://doi.org/10.1109/QSIC.2005.50

[14] Stefan Brass and Christian Goldberg. 2006. Semantic errors in SQL queries: A quite complete list. *J. Syst. Softw.* 79, 5 (2006), 630–644. https://doi.org/10.1016/J.JSS.2005.06.028

[15] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*. Springer, 316–330.

[16] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2012. Algorithmic debugging of SQL views. In *Perspectives of Systems Informatics: 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers 8*. Springer, 77–85.

[17] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2012. Declarative debugging of wrong and missing answers for SQL views. In *Functional and Logic Programming: 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings 11*. Springer, 73–87.

[18] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. 2023. Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries. *ACM Trans. Database Syst.* 48, 1 (2023), 1:1–1:45. https://doi.org/10.1145/3578517

[19] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. 2020. Answering (Unions of) Conjunctive Queries using Random Access and Random-Order Enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, Dan Suciu, Yufei Tao, and Zhewei Wei (Eds.). ACM, 393–409. https://doi.org/10.1145/3375395.3387662

[20] Claudio Cerullo and Marco Porta. 2007. A system for database visual querying and query visualization: Complementing text and graphics to increase expressiveness. In *18th International Workshop on Database and Expert Systems Applications (DEXA 2007)*. IEEE, 109–113.

[21] Bikash Chandra, Ananyo Banerjee, Udbhas Hazra, Mathew Joseph, and S. Sudarshan. 2021. Edit Based Grading of SQL Queries. In *CODS-COMAD 2021: 8th ACM IKDD CODS and 26th COMAD, Virtual Event, Bangalore, India, January 2-4, 2021*, Jayant R. Haritsa, Shourya Roy, Manish Gupta, Sharad Mehrotra, Balaji Vasan Srinivasan, and Yogesh Simmhan (Eds.). ACM, 56–64. https://doi.org/10.1145/3430984.3431012

[22] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. 2015. Data generation for testing and grading SQL queries. *VLDB J.* 24, 6 (2015), 731–755. https://doi.org/10.1007/S00778-015-0395-0

[23] Adriane Chapman and HV Jagadish. 2009. Why not?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 523–534.

[24] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

[25] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[26] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 510–524. https://doi.org/10.1145/3062341.3062348

[27] Yingwei Cui, Jennifer Widom, and Janet L Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)* 25, 2 (2000), 179–227.

[28] Ralf Diestelkämper and Melanie Herschel. 2020. Tracing nested data with structural provenance for big data analytics.. In *EDBT*. 253–264.

[29] Benjamin Dietrich and Torsten Grust. 2015. A SQL debugger built from spare parts: Turning a SQL: 1999 database system into its own debugger. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 865–870.

[30] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An automatic query steering framework for interactive data exploration. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 517–528.

[31] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proc. ACM Manag. Data* 1, 4 (2023), 227:1–227:26. https://doi.org/10.1145/3626768

[32] Idan Eldar, Nofar Carmeli, and Benny Kimelfeld. 2024. Direct Access for Answers to Conjunctive Queries with Aggregation. In *27th International Conference on Database Theory, ICDT 2024, March 25-28, 2024, Paestum, Italy (LIPIcs, Vol. 290)*, Graham Cormode and Michael Shekelyan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:20. https://doi.org/10.4230/LIPICS.ICDT.2024.4

[33] Full version of this submission. [n.d.]. https://anonymous.4open.science/r/oursys_eval-2063/. ([n. d.]).

[34] Amir Gilad, Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2022. Understanding Queries by Conditional Instances. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 355–368. https://doi.org/10.1145/3514221.3517898

[35] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 174–185.

[36] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.

[37] Torsten Grust and Jan Rittinger. 2013. Observing sql queries in their natural habitat. *ACM Transactions on Database Systems (TODS)* 38, 1 (2013), 1–33.

[38] Laura M Haas, Johann Christoph Freytag, Guy M Lohman, and Hamid Pirahesh. 1989. Extensible query processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. 377–388.

[39] Sabaat Haroon, Chris Brown, and Muhammad Ali Gulzar. 2024. DeSQL: Interactive Debugging of SQL in Data-Intensive Scalable Computing. *Proc. ACM Softw. Eng.* 1, FSE (2024), 767–788. https://doi.org/10.1145/3643761

[40] Yihao Hu, Amir Gilad, Kristin Stephens-Martinez, Sudeepa Roy, and Jun Yang. 2024. Qr-Hint: Actionable Hints Towards Correcting Wrong SQL Queries. *Proc. ACM Manag. Data* 2, 3 (2024), 164. https://doi.org/10.1145/3654995

[41] Yihao Hu, Zhengjie Miao, Zhiming Leong, Haechan Lim, Zachary Zheng, Sudeepa Roy, Kristin Stephens-Martinez, and Jun Yang. 2022. I-Rex: An Interactive Relational Query Debugger for SQL. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*. 1180–1180.

[42] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F Naughton. 2008. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment* 1, 1 (2008), 736–747.

[43] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 9. NIH Public Access, 216.

[44] Hannu Jaakkola and Bernhard Thalheim. 2003. Visual SQL–high-quality ER-based query treatment. In *Conceptual Modeling for Novel Application Domains: ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, USA, October 13, 2003. Proceedings 22*. Springer, 129–139.

[45] Grigoris Karvounarakis, Todd J Green, Zachary G Ives, and Val Tannen. 2013. Collaborative data sharing via update exchange and provenance. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–42.

[46] Marie Le Guilly, Jean-Marc Petit, Vasile-Marian Scuturici, and Ihab F Ilyas. 2019. Explique: Interactive databases exploration with SQL. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2877–2880.

[47] Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2019. PUG: a framework and practical implementation for why and why-not provenance. *The VLDB Journal* 28, 1 (2019), 47–71.

[48] Aristotelis Leventidis, Jiahui Zhang, Cody Dunne, Wolfgang Gatterbauer, HV Jagadish, and Mirek Riedewald. 2020. QueryVis: Logic-based diagrams help users understand complicated SQL queries faster. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2303–2318.

[49] Zhengjie Miao, Tiangang Chen, Alexander Bendeck, Kevin Day, Sudeepa Roy, and Jun Yang. 2020. I-Rex: an interactive relational query explainer for SQL. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2997–3000.

[50] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 503–520. https://doi.org/10.1145/3299869.3319866

[51] Daphne Miedema and George Fletcher. 2021. SQLVis: Visual query representations for supporting SQL learners. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–9.

[52] Xing Niu, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. 2021. Provenance-based Data Skipping. *Proc. VLDB Endow.* 15, 3 (2021), 451–464. https://doi.org/10.14778/3494124.3494130

[53] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (2019), 252–265. https://doi.org/10.14778/3368289.3368292

[54] PostgreSQL. [n.d.]. https://www.postgresql.org/.

[55] Kai Presler-Marshall, Sarah Heckman, and Kathryn T. Stolee. 2021. SQLRepair: Identifying and Repairing Mistakes in Student-Authored SQL Queries. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 199–210. https://doi.org/10.1109/ICSE-SEET52601.2021.00030

[56] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained lineage at interactive speed. *arXiv preprint arXiv:1801.07237* (2018).

[57] Sivaprasad Sudhir, Wenbo Tao, Nikolay Pavlovich Laptev, Cyrille Habis, Michael J. Cafarella, and Samuel Madden. 2023. Pando: Enhanced Data Skipping with Logical Data Partitioning. *Proc. VLDB Endow.* 16, 9 (2023), 2316–2329. https://doi.org/10.14778/3598581.3598601

[58] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2021. Beyond Equi-joins: Ranking, Enumeration and Factorization. *Proc. VLDB Endow.* 14, 11 (2021), 2599–2612. https://doi.org/10.14778/3476249.3476306

[59] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL Query Explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 425–446. https://doi.org/10.1007/978-3-642-17511-4_24

[60] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proc. VLDB Endow.* 17, 11 (2024), 3602–3614. https://www.vldb.org/pvldb/vol17/p3602-wang.pdf

[61] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate Pushdown for Data Science Pipelines. *Proc. ACM Manag. Data* 1, 2 (2023), 136:1–136:28. https://doi.org/10.1145/3589281

A EXPERIMENT SUPPLEMENT

We present the remaining experiment results over TPC-H benchmark.

For each table in the TPC-H schema, we have the following indexes (all indexes are btree indexes in PostgreSQL):

- customer
  - primary index: c_custkey
  - secondary indexes: None
- lineitem
  - primary index: (l_orderkey, l_linenumber)
  - secondary indexes: l_partkey, l_suppkey, l_shipdate
- nation
  - primary index: n_nationkey
  - secondary indexes: n_name, n_regionkey
- orders
  - primary index: o_orderkey
  - secondary indexes: o_custkey, o_orderdate
- part
  - primary index: p_partkey
  - secondary indexes: None
- partsupp
  - primary index: (ps_partkey, ps_suppkey)
  - secondary indexes: ps_suppkey
- region
  - primary index: r_regionkey
  - secondary indexes: None
- supplier
  - primary index: s_suppkey
  - secondary indexes: s_name, s_phone

We ran experiments for three different page size: 50, 100 and 200 for all tables (stages) in all TPC-H queries. For each table, we prepared three queries: milestone query, page query and table query and thus collecting the following data over all testing instances (i.e., 1GB, 5GB and 10GB):

- The execution time and output size of the milstone query and the table query.
- The execution time of the page query and baseline query (by rewritting the table query with `OFFSET` and `LIMIT`) for retrieving the first page ("head"), middle page ("mid") and the second last ("tail") page.

Since each query potentially contains multiple query blocks and subsequently multiple tables, we present the statistics for the largest table (measured in MB) to compute for each query. For page size 50, 100 and 200, the experiment results are shown in Table 4, Table 5 and Table 6 respectively.

In summary, we make the following conclusions:

- The milestone queries usually run slower than the table queries, but with acceptable delays since they are run at the beginning of the debugging session without affecting debugging operations later. In some cases, the milestone queries run faster than the table queries. On the other hand, the output sizes of the milestone queries are always smaller than those of the table queires by a rough factor of the page size.
- The optimization for page query almost always outperforms the baseline, and differences between the execution time grows as the database size grows, especially for "mid" and "tail" pages. The optimizations are sensitive to page size but insensitive to the database size. There are only few cases where the optimizations "over-hint" the PosgreSQL optimizer and cause the execution time to be roughly the same as the baseline.

		1GB		5GB		10GB	
Query	Page	Opt.	Base	Opt.	Base	Opt.	Base
Q1	head	0.153	0.057	0.169	0.143	0.065	0.085
	mid	0.177	447.618	0.319	15298.829	0.107	30754.125
	tail	0.146	895.18	0.17	30597.515	0.05	61508.166
Q2	head	21.595	298.538	21.056	2438.586	22.218	5015.589
	mid	23.268	330.474	20.483	2581.97	22.005	5333.441
	tail	22.163	362.41	22.166	2725.353	21.301	5651.293
Q3	head	1.473	4.414	10.946	72.055	11.53	77.334
	mid	1.04	368.188	13.097	6661.215	12.327	59183.467
	tail	0.881	731.962	12.418	13250.375	16.014	118243.911
Q4	head	0.558	0.469	0.539	1.069	8.15	2.834
	mid	0.604	123.425	0.527	637.045	4.283	10776.948
	tail	0.572	246.381	1.418	1265.02	2.525	21551.063
Q5	head	15.559	10.106	13.385	18.279	211.732	146.347
	mid	10.107	1629.932	7.337	13900.534	98.028	41247.203
	tail	6.284	3249.758	7.046	27782.79	104.97	81004.647
Q6	head	0.507	0.252	0.845	0.661	1.347	0.659
	mid	0.377	289.018	0.965	1650.336	2.917	39943.432
	tail	0.279	576.43	1.685	3300.011	1.049	79886.205
Q7	head	21.327	18.308	33.371	29.567	471.423	64.334
	mid	16.771	362.32	28.868	4240.504	423.37	19761.341
	tail	19.863	704.532	20.394	8451.441	423.913	39458.348
Q8	head	13.442	8.619	13.993	19.104	131.611	22.151
	mid	10.776	1911.543	16.478	8284.359	98.392	25365.045
	tail	13.79	3814.467	14.709	16549.614	131.145	50707.939
Q9	head	1.416	381.569	15.264	6277.133	16.968	29119.466
	mid	1.8	1675.868	22.93	13832.928	12.232	45990.497
	tail	1.119	2950.868	15.102	21388.724	10.875	62861.529
Q10	head	1.219	1.874	4.67	4.74	7.683	2.989
	mid	1.036	1061.645	5.087	11725.928	7.036	15279.263
	tail	1.04	2069.06	4.742	23447.117	4.76	30555.537
Q11	head	30.55	7.21	197.92	6.68	372.715	10.575
	mid	28.616	40.924	184.97	283.856	16.976	511.313
	tail	26.376	74.638	169.896	561.033	342.103	1012.051
Q12	head	1.123	0.855	4.247	1.028	2.328	1.084
	mid	2.353	492.485	2.906	8910.229	1.056	9485.278
	tail	1.248	984.115	2.529	17819.43	0.82	18969.472
Q13	head	0.231	0.168	0.222	0.186	0.2	0.192
	mid	0.223	493.355	0.199	2686.416	0.24	5650.011
	tail	0.194	986.542	0.2	5372.647	0.207	11299.831
Q14	head	6.304	1.99	40.092	2.22	6.957	3.921
	mid	5.755	139.76	40.416	849.248	4.337	1894.417
	tail	5.629	277.53	38.635	1696.275	3.477	3784.913
Q15	head	1.138	0.297	0.405	0.814	1.956	0.267
	mid	1.306	212.734	0.358	1314.475	3.711	2494.516
	tail	0.732	425.17	1.258	2628.135	2.244	4988.766
Q16	head	2.615	2.438	13.866	10.562	26.847	21.011
	mid	2.632	210.645	10.262	1230.19	23.322	2471.066
	tail	2.395	418.852	9.427	2449.819	21.865	4921.12
Q17	head	5.2	1.96	7.37	1.955	5.8	24.278
	mid	2.414	1282.928	5.231	13873.386	5.965	31301.815
	tail	2.559	2563.897	5.067	27744.817	5.836	62579.351
Q18	head	3919.892	8801.745	21717.035	75903.361	52355.55	165931.984
	mid	3919.762	12270.408	24999.351	91594.34	47736.515	188145.545
	tail	4423.777	15739.07	21717.478	107285.319	49884.912	210359.105
Q19	head	63.219	50.042	73.492	69.801	100.864	71.948
	mid	60.963	97.058	71.073	422.959	102.82	680.822
	tail	66.037	144.074	75.354	771.506	85.536	1289.697
Q20	head	0.141	0.059	0.154	0.053	0.264	0.498
	mid	0.118	90.876	0.129	463.991	0.827	1136.924
	tail	0.124	181.692	0.792	927.93	0.844	2273.349
Q21	head	21.59	833.851	53.231	21584.818	419.195	52731.23
	mid	16.303	841.546	54.607	21390.896	357.162	52110.334
	tail	18.113	837.46	53.475	21254.656	518.892	51489.439
Q22	head	0.824	73.643	1.658	323.461	3.814	641.022
	mid	1.046	107.034	1.722	503.032	3.35	998.173
	tail	0.818	135.346	1.332	679.873	2.786	1355.323

(a) Optimization vs. Baseline for Page Query Execution Time

		Milestone Query		Table Query	
Query	DB size	Time (ms)	Output (MB)	Time (ms)	Output (MB)
Q1	1	12867.887	31.159	2427.891	761.653
	5	66757.254	155.853	34169.326	3809.747
	10	132567.89	311.667	63014.657	7618.516
Q2	1	279.172	0.006	517.768	0.129
	5	1311.686	0.031	3561.809	0.647
	10	3161.0	0.062	7699.492	1.31
Q3	1	1427.375	0.216	2445.316	9.358
	5	7443.149	1.056	43601.631	45.735
	10	20680.519	2.145	119363.427	92.932
Q4	1	386.745	0.382	584.459	6.367
	5	1955.258	1.899	2601.872	31.649
	10	19454.468	3.79	36842.007	63.165
Q5	1	2723.034	0.936	3953.535	30.979
	5	13759.176	4.675	30937.301	154.836
	10	35892.348	9.325	87252.014	308.884
Q6	1	1083.271	1.795	1019.504	43.884
	5	5589.597	8.951	5205.285	218.804
	10	15949.432	17.923	80263.135	438.125
Q7	1	778.878	0.152	1534.78	0.947
	5	8853.979	0.737	11474.507	4.602
	10	18658.72	1.489	40536.043	9.306
Q8	1	6048.157	11.67	4873.439	65.639
	5	30352.478	58.054	23120.867	326.55
	10	61170.506	116.284	94252.079	654.091
Q9	1	2776.322	4.267	3889.442	23.998
	5	20788.436	20.989	25215.48	118.061
	10	41372.085	42.012	65159.271	236.312
Q10	1	2255.98	2.897	2745.088	77.861
	5	11602.777	14.538	13945.075	390.698
	10	23639.116	29.047	38600.544	780.631
Q11	1	140.588	0.243	128.376	5.928
	5	689.197	1.152	694.078	28.146
	10	1318.619	2.299	15861.395	56.193
Q12	1	1719.099	0.788	2305.83	10.507
	5	6473.45	3.947	22033.092	52.627
	10	14571.62	7.879	26913.72	105.057
Q13	1	5854.933	7.192	2112.696	74.4
	5	28952.598	35.96	11886.587	372.001
	10	58447.122	71.92	24361.687	744.001
Q14	1	368.41	0.539	460.133	18.571
	5	2004.619	2.694	7120.311	92.793
	10	3980.634	5.392	35470.562	185.705
Q15	1	1013.049	2.165	693.849	52.914
	5	5222.714	10.845	10168.376	265.099
	10	10852.703	21.696	40647.668	530.346
Q16	1	565.51	0.741	523.866	16.455
	5	2909.98	3.678	5054.798	81.74
	10	5748.205	7.378	6851.197	163.946
Q17	1	6050.402	1.44	6291.193	11.2
	5	28083.481	7.2	33261.046	56.0
	10	56939.309	14.4	64608.342	112.0
Q18	1	22619.063	52.215	16847.586	1272.737
	5	123157.558	260.86	135585.817	6358.46
	10	256609.502	521.399	275347.146	12709.096
Q19	1	165.548	0.044	417.981	1.501
	5	905.326	0.217	8210.493	7.47
	10	1478.324	0.421	71677.367	14.506
Q20	1	3115.577	6.577	2605.561	1056.214
	5	15517.716	32.885	47390.359	5279.964
	10	35401.962	65.691	88059.276	10557.545
Q21	1	854.506	0.051	1473.684	1.332
	5	13891.706	0.25	26156.569	6.538
	10	34639.107	0.511	94727.192	13.417
Q22	1	162.444	0.046	162.46	0.405
	5	769.978	0.229	984.764	2.035
	10	1551.326	0.458	2012.412	4.069

(b) Milstone Query vs. Table Query

Table 4: Experiment results for Page Size 50

		1GB		5GB		10GB	
Query	Page	Opt.	Base	Opt.	Base	Opt.	Base
	head	0.092	0.157	0.095	0.106	0.258	0.089
Q1	mid	0.089	450.644	0.102	16320.778	0.405	30908.903
	tail	0.102	901.132	0.081	32641.45	0.233	61817.717
	head	44.592	318.82	46.69	2580.597	45.526	5048.784
Q2	mid	46.063	341.12	42.468	2707.977	46.031	5350.129
	tail	44.61	363.419	44.389	2835.357	43.822	5651.474
	head	2.298	8.315	14.022	50.756	23.466	154.061
Q3	mid	1.933	366.845	15.116	6688.845	20.304	59441.052
	tail	1.94	725.375	15.172	13326.933	20.182	118401.92
	head	1.004	0.842	2.822	0.862	15.327	4.289
Q4	mid	1.208	125.753	1.935	650.216	6.286	11112.106
	tail	1.016	250.664	1.38	1299.571	4.286	22219.924
	head	18.294	16.98	35.305	22.441	367.671	230.179
Q5	mid	17.585	1678.536	16.627	14065.395	146.61	40401.774
	tail	13.1	3285.056	17.244	28002.402	134.78	80573.369
	head	0.745	0.534	2.076	0.534	2.895	0.538
Q6	mid	0.676	296.977	0.606	1693.265	4.688	41093.476
	tail	0.509	565.074	1.23	3385.997	1.766	81893.843
	head	41.808	28.7	61.38	52.018	946.975	55.532
Q7	mid	27.809	369.064	55.629	4353.019	649.816	19842.907
	tail	27.667	695.718	35.851	8654.019	593.511	39630.282
	head	28.152	12.265	28.002	22.943	247.418	24.732
Q8	mid	20.745	1927.807	31.936	8373.202	221.967	25327.403
	tail	22.398	3843.349	29.754	16723.462	206.956	50630.074
	head	2.227	389.589	25.745	6031.313	27.844	29081.397
Q9	mid	1.988	1698.267	35.081	13767.171	25.463	46311.933
	tail	1.949	3006.946	24.147	21503.029	24.073	63542.47
	head	3.528	1.349	8.099	8.962	12.316	3.35
Q10	mid	1.588	1048.416	7.454	11860.303	10.354	15360.343
	tail	1.879	2095.483	7.331	23711.644	8.291	30717.336
	head	52.714	5.543	296.591	12.83	615.864	21.86
Q11	mid	55.176	41.634	317.3	294.6	724.889	508.906
	tail	58.274	77.724	295.98	576.369	559.893	995.951
	head	1.907	1.645	3.757	4.948	3.159	1.836
Q12	mid	3.929	499.537	2.384	9183.807	2.08	9426.057
	tail	1.964	997.429	1.923	18362.666	1.69	18850.279
	head	0.379	0.257	0.37	0.642	0.82	0.656
Q13	mid	0.366	477.302	0.337	2855.172	0.798	5597.041
	tail	0.324	954.347	0.304	5709.702	0.35	11193.425
	head	6.925	3.679	42.57	4.942	121.01	11.66
Q14	mid	6.068	145.425	41.569	855.593	126.782	1897.004
	tail	6.091	287.172	41.044	1706.244	117.149	3782.347
	head	0.662	0.51	2.037	0.479	4.343	0.478
Q15	mid	0.634	220.523	3.608	1313.553	3.689	2524.079
	tail	0.752	440.536	0.52	2626.628	1.913	5047.679
	head	3.164	7.008	17.155	10.224	24.908	20.641
Q16	mid	3.355	213.954	10.817	1248.899	20.854	2473.624
	tail	3.646	420.901	10.047	2487.575	20.262	4926.607
	head	5.152	3.786	12.614	3.626	13.25	3.637
Q17	mid	4.848	1306.111	9.252	12169.355	13.778	30319.048
	tail	4.816	2608.435	8.428	24335.084	28.102	60634.458
	head	3963.758	8643.507	24134.374	74575.1	48786.827	169499.561
Q18	mid	4061.897	12245.213	24618.811	90277.508	46661.664	190886.288
	tail	4091.53	15846.92	26659.445	105979.916	53063.774	212273.016
	head	80.62	95.444	275.504	516.311	425.97	836.663
Q19	mid	79.97	120.793	232.757	641.094	434.126	1076.905
	tail	78.181	143.358	271.998	765.877	433.759	1317.148
	head	0.692	0.083	0.751	0.078	1.054	0.891
Q20	mid	0.754	86.401	0.797	458.918	1.671	1013.644
	tail	0.642	172.72	0.536	917.758	1.112	2026.396
	head	41.894	830.955	85.693	20382.458	678.648	51923.664
Q21	mid	28.909	831.803	70.865	20134.344	1071.348	51429.637
	tail	30.357	832.652	61.558	20231.722	1051.124	51092.808
	head	4.299	68.512	4.51	319.805	2.876	659.813
Q22	mid	1.589	107.028	2.004	499.389	2.336	1007.544
	tail	1.604	136.537	1.529	677.494	1.997	1355.275

(a) Optimization vs. Baseline for Page Query Execution Time

		Milestone Query		Table Query	
Query	DB size	Time (ms)	Output (MB)	Time (ms)	Output (MB)
	1	13029.523	15.579	2457.667	761.653
Q1	5	66150.01	77.927	39046.702	3809.747
	10	132810.372	155.834	66195.529	7618.516
	1	280.11	0.003	501.518	0.129
Q2	5	1438.351	0.015	3650.785	0.647
	10	3197.387	0.031	7330.617	1.31
	1	1402.237	0.108	2391.46	9.358
Q3	5	7609.511	0.528	29926.468	45.735
	10	20829.922	1.072	119386.492	92.932
	1	393.68	0.191	602.307	6.367
Q4	5	1956.036	0.95	2871.491	31.649
	10	19492.825	1.895	37588.076	63.165
	1	2794.965	0.468	3934.344	30.979
Q5	5	13919.163	2.337	31427.944	154.836
	10	35787.956	4.662	88961.208	308.884
	1	1077.996	0.898	1028.798	43.884
Q6	5	5606.841	4.476	5449.391	218.804
	10	15937.32	8.962	83693.457	438.125
	1	750.001	0.076	1388.376	0.947
Q7	5	8949.756	0.369	11575.322	4.602
	10	18841.027	0.745	40373.876	9.306
	1	6068.222	5.835	4796.095	65.639
Q8	5	30360.543	29.027	23391.072	326.55
	10	61229.624	58.142	93859.353	654.091
	1	2824.782	2.134	3857.867	23.998
Q9	5	20769.045	10.495	25856.049	118.061
	10	41547.519	21.006	64925.319	236.312
	1	2235.767	1.449	2775.5	77.861
Q10	5	11587.553	7.269	14053.989	390.698
	10	24067.6	14.524	38715.702	780.631
	1	137.414	0.121	125.681	5.928
Q11	5	689.607	0.576	617.165	28.146
	10	1311.196	1.149	15862.307	56.193
	1	1737.807	0.394	2334.203	10.507
Q12	5	6617.319	1.974	22186.895	52.627
	10	14404.903	3.94	25798.262	105.057
	1	5899.366	3.596	2138.399	74.4
Q13	5	28939.75	17.98	12261.105	372.001
	10	58316.145	35.96	24704.791	744.001
	1	380.802	0.27	476.924	18.571
Q14	5	2001.128	1.347	7033.305	92.793
	10	3954.975	2.696	35885.574	185.705
	1	1014.581	1.083	707.173	52.914
Q15	5	5256.718	5.423	11050.796	265.099
	10	10870.044	10.848	40151.84	530.346
	1	551.868	0.37	523.076	16.455
Q16	5	2798.609	1.839	4488.929	81.74
	10	5636.501	3.689	6598.465	163.946
	1	6134.681	0.72	6181.627	11.2
Q17	5	27933.944	3.6	34723.602	56.0
	10	57508.86	7.2	64700.14	112.0
	1	21986.933	26.108	16799.919	1272.737
Q18	5	119339.262	130.43	131820.25	6358.46
	10	252651.048	260.7	273911.052	12709.096
	1	162.155	0.022	404.626	1.501
Q19	5	905.482	0.109	6894.219	7.47
	10	1505.552	0.211	60837.304	14.506
	1	3110.616	3.289	2668.742	1056.214
Q20	5	15414.311	16.443	45633.142	5279.964
	10	35581.996	32.845	91427.229	10557.545
	1	866.568	0.026	1455.808	1.332
Q21	5	13742.687	0.125	25023.161	6.538
	10	34559.892	0.256	95153.337	13.417
	1	162.503	0.023	165.083	0.405
Q22	5	765.807	0.114	886.335	2.035
	10	1533.474	0.229	2025.742	4.069

(b) Milstone Query vs. Table Query

Table 5: Experiment results for Page Size 100

		1GB		5GB		10GB	
Query	Page	Opt.	Base	Opt.	Base	Opt.	Base
Q1	head	0.146	0.389	0.433	0.387	0.49	0.163
	mid	0.16	454.435	0.464	16284.424	0.461	30276.205
	tail	0.133	908.481	0.372	32568.462	0.405	60552.247
Q2	head	55.255	321.231	91.769	2568.871	88.861	5104.092
	mid	52.181	355.251	87.877	2711.985	92.542	5427.883
	tail	53.35	389.271	91.139	2855.098	92.844	5751.674
Q3	head	3.94	14.391	35.196	132.234	39.181	226.782
	mid	3.909	365.739	26.025	6675.944	47.164	59440.776
	tail	3.611	717.087	26.169	13181.499	43.333	118022.429
Q4	head	2.242	1.713	2.14	1.733	13.369	7.061
	mid	2.269	123.996	3.002	650.459	11.325	11214.848
	tail	2.127	246.279	2.19	1299.185	8.845	22341.941
Q5	head	28.833	37.402	51.504	71.031	577.779	413.569
	mid	26.29	1746.625	36.167	14325.169	265.56	40522.6
	tail	22.047	3369.844	35.608	28095.786	237.461	80631.631
Q6	head	4.046	1.042	1.365	1.074	3.365	3.164
	mid	1.49	300.074	1.807	1670.156	4.891	40923.411
	tail	1.545	599.106	1.505	3339.238	1.842	81797.261
Q7	head	65.141	53.139	102.474	81.208	1511.564	80.04
	mid	52.02	390.365	73.626	4342.948	895.119	19402.032
	tail	49.058	722.125	65.171	8604.687	835.577	38724.024
Q8	head	54.929	20.102	66.214	28.021	415.944	56.301
	mid	44.767	1961.613	57.467	8389.363	312.842	25654.914
	tail	44.222	3903.124	51.861	16750.706	343.404	51253.527
Q9	head	3.807	391.343	41.007	6039.289	43.95	29193.922
	mid	3.982	1727.193	51.369	13735.98	41.83	46365.304
	tail	3.321	3056.58	47.51	21432.671	42.478	63536.687
Q10	head	3.34	2.084	15.334	14.991	25.553	4.648
	mid	3.258	1055.799	15.911	11791.208	18.078	15180.68
	tail	3.534	2109.514	9.998	23567.424	20.76	30356.711
Q11	head	107.842	10.874	562.752	18.136	1182.66	26.277
	mid	102.869	42.147	571.399	291.854	1228.931	514.186
	tail	110.701	73.42	540.832	565.572	1083.823	1002.094
Q12	head	4.596	3.41	4.939	7.39	5.62	3.739
	mid	8.235	486.03	3.992	9115.702	4.735	9420.754
	tail	3.682	968.651	3.468	18224.014	3.855	18837.768
Q13	head	1.654	0.473	0.867	0.495	0.578	0.729
	mid	1.587	456.661	0.765	2654.764	0.58	5625.206
	tail	1.36	912.849	0.947	6597.823	0.625	11249.684
Q14	head	7.708	9.054	45.371	15.041	122.05	15.072
	mid	7.238	147.356	46.699	885.645	120.864	1911.871
	tail	6.95	280.742	45.787	1734.791	123.123	3808.671
Q15	head	3.735	0.85	3.158	2.645	9.409	1.126
	mid	3.124	207.608	10.181	1307.789	5.79	2515.546
	tail	1.263	414.366	3.237	2612.933	3.955	5029.967
Q16	head	9.606	2.675	11.776	11.327	21.11	29.04
	mid	11.049	208.386	12.759	1266.214	22.488	2464.479
	tail	3.587	412.058	11.599	2521.1	21.577	4899.917
Q17	head	9.482	6.71	22.959	7.215	32.825	7.332
	mid	9.251	1268.832	22.988	16872.764	41.268	28139.693
	tail	8.654	2530.953	22.979	33738.314	29.277	56272.054
Q18	head	3903.961	8319.451	21528.099	81351.562	50643.69	171400.04
	mid	4013.044	11809.205	19208.862	96954.308	51412.839	193322.243
	tail	3931.8	15298.959	23889.779	112557.054	53669.283	215244.446
Q19	head	104.82	154.3	768.251	620.145	465.465	1135.042
	mid	110.143	147.477	765.594	694.52	496.925	1232.457
	tail	113.428	141.114	778.368	768.895	462.613	1329.872
Q20	head	0.461	0.33	1.302	0.12	0.74	1.838
	mid	0.409	87.733	1.228	557.974	2.738	1064.696
	tail	0.399	175.135	1.128	1115.828	1.079	2127.553
Q21	head	79.333	813.999	5911.734	20059.439	2018.207	51725.379
	mid	53.285	819.563	4663.958	19896.141	1971.71	51234.28
	tail	55.42	813.999	4649.823	19732.842	1761.852	50743.181
Q22	head	2.861	66.346	3.595	334.122	9.744	680.439
	mid	3.061	99.748	3.72	512.774	3.904	1033.266
	tail	2.684	133.149	3.3	691.427	3.547	1386.094

(a) Optimization vs. Baseline for Page Query Execution Time

		Milestone Query		Table Query	
Query	DB size	Time (ms)	Output (MB)	Time (ms)	Output (MB)
Q1	1	12918.259	7.79	2440.065	761.653
	5	66166.286	38.964	40957.384	3809.747
	10	132584.301	77.917	65152.122	7618.516
Q2	1	282.879	0.003	497.999	0.129
	5	1423.001	0.008	3694.901	0.647
	10	3186.098	0.016	7371.271	1.31
Q3	1	1394.346	0.054	2403.439	9.358
	5	7529.177	0.264	37672.174	45.735
	10	20757.397	0.536	119575.923	92.932
Q4	1	384.925	0.096	592.285	6.367
	5	1941.103	0.475	3502.926	31.649
	10	19626.549	0.948	37255.875	63.165
Q5	1	3029.376	0.234	3930.43	30.979
	5	13764.754	1.169	30815.472	154.836
	10	35644.992	2.332	88633.534	308.884
Q6	1	1165.378	0.449	1062.076	43.884
	5	5583.563	2.238	5043.746	218.804
	10	16028.371	4.481	83720.999	438.125
Q7	1	778.049	0.038	1504.937	0.947
	5	8787.166	0.184	11226.01	4.602
	10	18606.773	0.372	41071.627	9.306
Q8	1	6096.48	2.918	4880.251	65.639
	5	30248.758	14.514	22877.673	326.55
	10	60868.096	29.071	94180.357	654.091
Q9	1	2867.998	1.067	3785.434	23.998
	5	20706.537	5.247	25684.329	118.061
	10	41036.521	10.503	65536.758	236.312
Q10	1	2298.852	0.724	2741.803	77.861
	5	11682.06	3.635	13569.476	390.698
	10	24156.397	7.262	39305.001	780.631
Q11	1	141.382	0.061	123.338	5.928
	5	706.803	0.288	578.334	28.146
	10	1325.331	0.575	15713.276	56.193
Q12	1	1691.273	0.197	2261.535	10.507
	5	6628.787	0.987	22103.411	52.627
	10	14411.088	1.97	26482.419	105.057
Q13	1	5904.443	1.798	2023.404	74.4
	5	29358.318	8.59	12564.698	372.001
	10	58269.683	17.98	24649.779	744.001
Q14	1	364.521	0.135	461.061	18.571
	5	2023.895	0.674	18845.022	92.793
	10	3965.741	1.348	35490.329	185.705
Q15	1	981.171	0.541	673.001	52.914
	5	5233.873	2.712	20492.43	265.099
	10	10794.957	5.424	40226.156	530.346
Q16	1	517.513	0.185	509.302	16.455
	5	2663.285	0.92	3373.281	81.74
	10	5422.138	1.845	6661.765	163.946
Q17	1	5814.768	0.36	5818.887	11.2
	5	28270.684	1.8	34331.977	56.0
	10	57033.56	3.6	64742.184	112.0
Q18	1	20843.893	13.054	16149.079	1272.737
	5	117514.027	65.215	143138.813	6358.46
	10	247356.887	130.35	269322.91	12709.096
Q19	1	159.498	0.011	387.317	1.501
	5	937.639	0.054	10230.62	7.47
	10	1524.313	0.105	63414.724	14.506
Q20	1	3002.178	1.644	2583.319	1056.214
	5	15469.147	8.221	49269.443	5279.964
	10	36276.083	16.423	93364.505	10557.545
Q21	1	811.625	0.013	1411.304	1.332
	5	14187.38	0.063	25798.959	6.538
	10	35656.018	0.128	96969.24	13.417
Q22	1	155.141	0.012	158.698	0.405
	5	779.235	0.057	864.045	2.035
	10	1561.38	0.114	1999.535	4.069

(b) Milstone Query vs. Table Query

Table 6: Experiment results for Page Size 200

B USER STUDY SUPPLEMENT

The user study is conducted with an instance of beer database with the following schema (keys are underlined):

- Drinker (name, address)
- Bar (name, address)
- Beer (name, brewery)
- Frequents (drinker, bar, times_a_week)
- Serves (bar, beer, price)
- Likes (drinker, beer)

B.1 Debugging Quesiton 1

Question Statement. For each bar Ben visits, find price of the most expensive and cheapest drink at that bar. Format the output as (bar, price), no duplicates.

The wrong query presented to the students are as follows:

```
WITH t1 AS (
  SELECT bar, price
  FROM serves
  WHERE price = (
      SELECT MAX(S1.price)
      FROM serves S1
      WHERE S1.bar = bar
    )
  UNION ALL
  SELECT bar, price
  FROM serves
  WHERE price = (
      SELECT MIN(S1.price)
      FROM serves S1
      WHERE S1.bar = bar
    )
)
SELECT t1.bar, t1.price
FROM t1, frequents
WHERE t1.bar = frequents.bar
  AND frequents.drinker = 'Ben';
```

The above query has two mistakes:

- `UNION ALL` creates duplicates when the most expensive and cheapest drink share the same price.
- The `bar` in both scalar subqueries are referencing the wrong column. Without correct aliasing, both `bar` refer to the `bar` in `S1`, making the `WHERE` condition a tautology.

B.2 Debugging Quesiton 2

Question Statement. Suppose every time a drinker frequents a bar, he buys all his favorite beers at that bar. Find the expected weekly revenue of each bar and rank them by the revenue from high to low. The output should be in the format of (bar, revenue). If a bar is not frequented by any drinker, or it does not serve any beer, or none of its beer is liked by any drinker, output (bar, NULL).

The wrong query presented to the students are as follows:

```
SELECT S.bar,
  SUM(F.times_a_week) * SUM(S.price) AS revenue
FROM serves S,
  frequents F,
  likes L
WHERE S.bar = F.bar
  AND S.beer = L.beer
GROUP BY S.bar
ORDER BY revenue DESC;
```

The above query has three mistakes:

- The join predicate `F.drinker = L.drinker` is missing.
- The expression for sum is incorrect as it will blow up the result. The correct expression is `SUM(F.times_a_week * S.price)`.
- There will be no "`NULL`" tuple produced by the query, i.e., bars which do not serve any beer / serve no liked by anyone will not be included in the result.