I-Rex: An Interactive Debugger for SQL

Yihao Hu
yihao.hu@duke.edu
Duke University
Durham, NC, USA

Zhiming Leong
zhiming.leong@duke.edu
Duke University
Durham, NC, USA

Alex Chao
ac590@duke.edu
Duke University
Durham, NC, USA

Zian Chen
zian.chen@duke.edu
Duke University
Durham, NC, USA

Sharan Sokhi
sharan.sokhi@duke.edu
Duke University
Durham, NC, USA

Zachary Zheng
zachary.zheng@duke.edu
Duke University
Durham, NC, USA

Kristin Stephens-Martinez
ksm@cs.duke.edu
Duke University
Durham, NC, USA

Sudeepa Roy
sudeepa@cs.duke.edu
Duke University
Durham, NC, USA

Jun Yang
junyang@cs.duke.edu
Duke University
Durham, NC, USA

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 general-purpose programming languages (GPLs). While there are many GPL debuggers, SQL debugging has received much less attention. In this paper, we present I-Rex, 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. I-Rex 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, I-Rex 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, I-Rex 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 I-Rex through performance experiments as well as a user study in an educational setting.

PVLDB Reference Format:
Yihao Hu, Zhiming Leong, Alex Chao, Zian Chen, Sharan Sokhi, Zachary Zheng, Kristin Stephens-Martinez, Sudeepa Roy, and Jun Yang. I-Rex: An Interactive Debugger for SQL. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

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 only grown over time. However, SQL is difficult to understand and debug. In debugging GPLs (e.g. C++ or Python) that are typically *procedural* (i.e. explicitly describing the 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, the 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 an iterative process: the user may examine execution multiple times, sometimes with minor modifications to the query. However, even small changes can lead to the optimizer choosing a very different plan; for example, the addition or removal of even a simple 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, the size of the buffer memory, the choice of the hash function, and variations in the speed of parallel threads at run-time 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, this may result in inefficient handling of complex queries. Furthermore, correlated subqueries, a frequently used SQL construct, render this workaround ineffective for many queries. In Section 2, we give concrete examples that illustrate this challenge and demonstrate how I-Rex helps debug these types of queries.

In this work with I-Rex, 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 completely reproducible manner that is faithful to how it is written and must be easy for programmers to understand.
(2) The debugger should offer features analogous to those in GPL debuggers so that they are easy to learn and adopt.
(3) Unlike GPL debuggers, which must execute underlying programs in full to reach points of interest, this debugger should

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

support efficient implementation of powerful features that allow the user to jump directly to points of interest.

(4) The debugger must scale to large databases 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 first glance, (1) necessitates executing the SQL query "literally" using a completely unoptimized plan, which runs counter to (4). Our key insight is that, at any given point in time, the user can examine 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), requires novel optimization — SQL's built-in `OFFSET` and `LIMIT` constructs fail to deliver acceptable performance for interactive debugging.

Addressing the above challenges through the development of I-Rex, we make the following contributions:

- I-Rex introduces a novel debugging paradigm for SQL that draws many parallels to GPL debugging. Each query block is viewed as a function, and correlated subqueries are considered functions with arguments. We define the *canonical execution* of a SQL query, which is reproducible and faithful to query syntax.
- I-Rex 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, I-Rex supports 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 efficiency on large databases.
- I-Rex 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 I-Rex easy to adopt and deploy.
- Our performance evaluation using the TPC-H benchmark shows the scalability of I-Rex 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 I-Rex with a large-scale user study of over 100 students in an introductory-level database course. Its findings indicate that I-Rex significantly improves students' efficiency in debugging SQL queries.

2 EXAMPLE USE OF I-REX FOR DEBUGGING

Since I-Rex conceptually executes SQL queries as 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 I-Rex to debug an incorrect query; we introduce the interface, concepts, and features of I-Rex. More formal and detailed discussions will be presented in later sections.

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.

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: 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}?*

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 the original subquery for all possible* `drinker` *values in a single effort. Then, the result will be 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 from the original query. Users without in-depth knowledge of query optimization will likely be confused.*

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

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

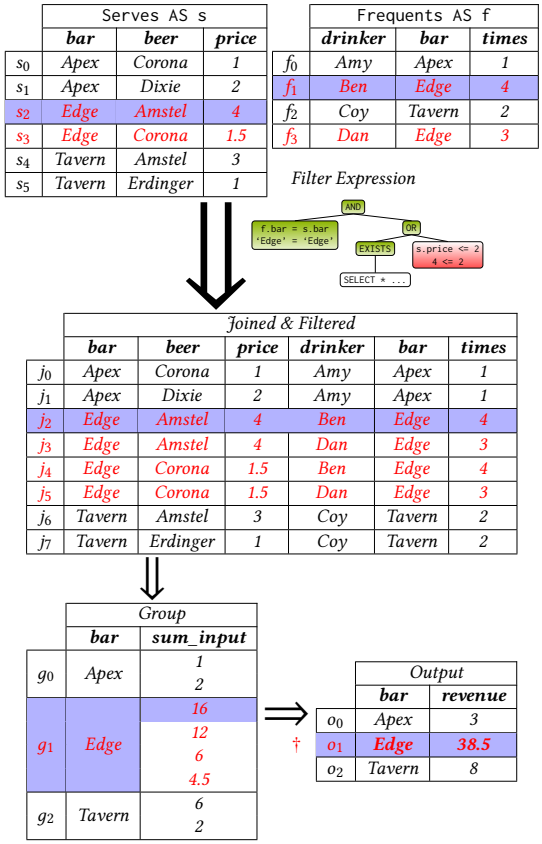


Figure 2: Debugging context for outer query block, Example 2.2.

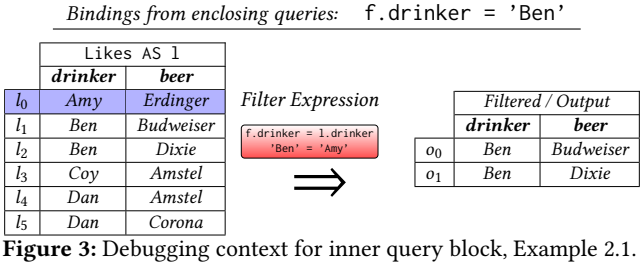


Figure 3: Debugging context for inner query block, Example 2.1.

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 I-Rex to debug the query, starting with this observation.*

I-Rex 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). I-Rex shows the execution of this block in stages, from top to bottom. At the very top, I-Rex 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, I-Rex 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. 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, I-Rex 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 output table shows the final result of the query block.*

For the convenience of subsequent discussion, we show a symbolic identifier (e.g., s_0, f_2, j_6...) for each row. We do assign internal row identifiers, whose purposes will be explained later in Section 3, but they are not explicitly displayed by the UI.

Given the unexpectedly high revenue of Edge*, the user naturally wants to examine how that output row was computed. I-Rex supports tracing backward from output to input using a more general mechanism called* pinning*, denoted here by the red † next to o_1 : ⟨Edge, 38.5⟩. A pinned output row intuitively narrows the 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 colored 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 further grouped into g_1 in the group table before finally producing o_1.*

As soon as the user pins o_1, I-Rex identifies the relevant input combos and "positions" execution at the first input combo in lexicographical order — this is how the input combo ⟨s_2, f_1⟩ was activated in the first place in Figure 2. Starting with this input combo, the user can step through other input combos relevant to o_1, or manually choose any combo to investigate. Throughout the process, I-Rex automatically updates highlighting in 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. I-Rex allows the user to "drill down" into the execution of Q_{inner} in this context. Recall that Q_{inner} is a correlated subquery. To a programmer, the analogy of evaluating Q_{inner} is a function call with a 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}, I-Rex 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 parameter `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 supports that Ben likes Amstel. In fact, this subquery does not even look for Amstel. Therefore, to fix the query, we should let the outer query "call" Q_{inner} with an additional parameter* `s.beer`*, and let Q_{inner} additionally test whether* `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, I-Rex will produce a very consistent experience for the user: the execution of the new query will

Figure 4: A paginated table in I-Rex.

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 larger. Even with moderately sized databases, joins can easily produce 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, it is impractical to cache entire results at any location (database server, middleware, or user browser) or send them over the network.

Meanwhile, users generally cannot examine many rows simultaneously. Therefore, I-Rex UI supports a pagination mechanism to display each table (input, intermediate result, or final output); Figure 4 shows a screenshot. 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, I-Rex adjusts the visible portions of all displayed tables accordingly, so that each reflects the execution point defined by the current input combo. Similarly, the user can visit any pages of input tables to select a new input combo for tracing; I-Rex would automatically adjust downstream table displays to show corresponding intermediate result rows.

Hence, efficient pagination is key to scalable SQL debugging. With efficient pagination, I-Rex effectively allows the user to "teleport" across points of interest without incurring the execution cost in between, making I-Rex 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 and associated debugging information. Furthermore, I-Rex's requirement of making execution reproducible imposes specific ordering of intermediate results that complicates optimization. We discuss our solution in Section 4.

3 DEBUGGING PARADIGM

This section describes the debugging paradigm of I-Rex. I-Rex supports a rich set of SQL query features, such as `SELECT` queries with inner cross joins, set/bag operations (e.g., `UNION`, `INTERSECT`, `EXCEPT`), subqueries (including correlated ones), outer joins and joins expressed in general `JOIN` syntax (except `LATERAL`), and `WITH`. However, it 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. I-Rex 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. 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 are still considered duplicates by SQL 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 case that the table has no key, we use the database's internal row id (e.g., PostgreSQL's `ctid`); such ids are unique among duplicate rows.

For a derived table, we define its IID according to how the table 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 can compute 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, I-Rex 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 I-Rex are "logical" instead of physical.

3.1.2 Query Blocks and Debugging Contexts. Given a query, I-Rex 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. I-Rex 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 helper function with parameters, while a non-correlated subquery is comparable to a helper function without parameters. For instance, for Q_{inner} in Example 2.1, the caller is responsible for passing the values of external column references (e.g., `f.drinker = 'Ben'` in Figure 3) as parameters.
- 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 (i.e., an execution of a query block), I-Rex creates a

debugging context as needed, analogous to an "activation frame" in GPLs. As an example, Q in Example 2.1 calls Q_{inner} multiple times, with potentially different `f.drinker` values as arguments, resulting in different executions. This debugging context holds information specific to the particular execution of the query block, such as the values of external column references (i.e., parameter values).

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. For a complex construct such as `SELECT`, we further decompose its execution into *stages*, where each stage can be seen as an operator with its own canonical execution procedure and result IID synthesis function. We only discuss `SELECT` with inner cross joins below; Appendix A.1 discusses SQL set and block operations; Appendix A.2 discusses general join expressions including outerjoins.

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 with parameter 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. Considering Example 2.2 and Figure 2, the IID for `Serves` is its primary key (`bar`, `beer`), and the IID for `Frequents` is its primary key (`drinker`, `bar`). Therefore, the IID for the derived joined & filtered table has the format ((`s.bar`, `s.beer`), (`f.drinker`, `f.bar`)).[1] In particular, j_2's IID would be (s_2, f_1), where we abuse notation slightly and use s_2 and f_1 to denote their IIDs (`Edge`, `Amstel`) and (`Ben`, `Edge`) respectively.

If the block contains grouping or aggregation, a *grouping* stage will be next. Its canonical execution is a stable sort of input rows according to the list of `GROUP BY` expressions[2] 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 input row IID. This order puts member rows of a group together, allowing the UI to detect and display group boundaries (Figure 2). The *stable* sort ensures that the IID ordering is consistent with the row production order. For instance, in Figure 2, the IID of the first member row of g_1 is (`Edge`, j_2), where `Edge` also identifies the g_1 group.

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 are 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. For instance, o_1 in Figure 2 would be `Edge`, same as g_1's. Otherwise, no aggregation is involved, and we simply produce one result row for each input row, using the same IID for the result. In either case, the IID order is consistent with the result row production order.

Several special cases associated with the final stage are worth noting. If any `HAVING` or `SELECT` expression contains subqueries, they would be handled the same way as in the join & filter stage. If `SELECT` is followed by `DISTINCT`, the canonical execution will further perform a sort of all result rows using all columns in some order (optimized to be maximally consistent with the input IID order) and output only distinct rows; the result IID will be synthesized as a vector whose components are all the columns in the order chosen. If `ORDER BY` is present,[3] the canonical execution will further perform a stable sort of all result rows according to the `ORDER BY` expressions, and the result IID will be synthesized as a vector whose components start with the `ORDER BY` expressions and end with the input IID.

3.2 Debugging Operations

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

3.2.1 Input Combo Space and Execution Positioning. Recall that a debugging context refers to a specific execution of a query block. To represent the entire execution of a debugging context, we define its *input combo space* as an ordered set of *input combos* (*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*, I-Rex automatically advances the active input combo to its successor (or predecessor if stepping in reverse order) in the input combo space. For example, (ignore the pin and) suppose the active input combo in Figure 2 were $\langle s_0, f_3 \rangle$; stepping would advance it to $\langle s_1, f_0 \rangle$, followed by $\langle s_1, f_1 \rangle$, consistent with the processing order of the canonical execution. 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; I-Rex will then update the active input combo accordingly. Even more advanced execution positioning can be achieved by pinning, which will be described shortly.

[1] In implementation, I-Rex performs straightforward optimizations to compact the IID by removing known redundancies. For example, since all result rows have `f.bar = s.bar`, we can simplify the IID as (`s.bar`, `s.beer`, `f.drinker`) without affecting uniqueness or ordering. For brevity, in subsequent discussions, please assume that such optimizations will be applied implicitly at the implementation level.

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

[3] Per SQL standard, every `ORDER BY` expression must correspond to an output column. For SQL dialects that do not have this restriction, complex `ORDER BY` may necessitate a separate stage to help debugging. We will not elaborate here.

Finally, to handle outer joins and the special case of empty input tables, we extend our definition of the input combo space with a special value $\perp$; see Appendix A.2 for details.

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. Intuitively, the input combo contributes to the computation of derivative rows; formally, the input combo participates in the how-provenance [38] of the derivative row. As discussed in Section 2, once the active input combo is set, I-Rex automatically refreshes all visualizations of expression trees, such that they reflect evaluation over the active input combo or its derivative rows. I-Rex also automatically refreshes all result table displays, 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 a 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.

For example, in Figure 2, the active input combo $\langle s_2, f_1 \rangle$ has one derivative row per result table, namely j_2, the first row in g_1, and o_1. In Figure 3, there is no derivative row for the active input combo $\langle l_0 \rangle$. By design, I-Rex 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. I-Rex 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 while debugging.

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.

Considering Figure 2, the pinned space for the pinned o_1 is $\{\langle s_2, f_1 \rangle, \langle s_2, f_3 \rangle, \langle s_3, f_1 \rangle, \langle s_3, f_3 \rangle\}$ because all these input combos contribute to the total revenue calculation for o_1. If the user additionally pins f_1 in `Frequents`; this pinned row will have a pinned space of $\{s_0, \ldots, s_5\} \times \{f_1\}$. Thus, the overall pinned space for the debugging context will be $\{\langle s_2, f_1 \rangle, \langle s_3, f_1 \rangle\}$, meaning the user only wants to investigate how Dan contributes to Edge's total revenue.

Pinning has multiple uses. First, it augments I-Rex's tracing capability: 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, I-Rex restricts stepping to the pinned space, effectively setting a watchpoint that pauses execution only at points relevant to the pinned rows. In Figure 2, with o_1 pinned, execution automatically stops at the 4 relevant input combos $\langle s_2, f_1 \rangle$, $\langle s_2, f_3 \rangle$, $\langle s_3, f_1 \rangle$, $\langle s_3, f_3 \rangle$, skipping irrelevant portions of execution before, after, and in between.

3.2.4 Drilling Down and Pulling Up. As illustrated in Example 2.2, I-Rex 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, I-Rex 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; I-Rex 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, I-Rex essentially maintains a "call stack" of debugging contexts. To let the user *pull up* from a subquery debugging context, I-Rex 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.

4 SYSTEM AND OPTIMIZATIONS

In this section, we describe the implementation and optimizations of the I-Rex system. Given a query Q being debugged, a straightforward approach would be to carry out the canonical execution of Q and collect all debugging information in one go, but this approach is not scalable, as the following example shows.

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

```
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
```

If we were to compute and display the ten entire tables (eight input, one joined & filtered, and one output) for the debugging context, their contents alone would take 1,825*MB. Lineage data needed to relate input combos and derivative rows would take another* 88*MB, not to mention information for the filter expression tree. Shipping close to* 2*GB of data (or more) from the database server to the I-Rex client introduces unacceptable overhead. Furthermore, this level of memory usage is demanding, as most browser tabs only use less than* 1*GB of memory. The situation becomes more unattainable when debugging on bigger databases and/or complex queries with multiple blocks.*

To address this challenge, I-Rex uses three high-level ideas. 1) Rather than showing the entire canonical execution of a query Q being debugged, 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 support from the database.

As described in Section 2, I-Rex realizes idea (1) above using a paginated display for each table (based or derived). For each debugging context, the active input combo, introduced in Section 3.2, marks the current point of execution and controls which pages to display by default: pages containing the input combo for input tables, and the page containing the derivative row for each subsequent stage. Together, these pages define the "window" of execution as seen by the user. In the following, we first show in Section 4.1 how I-Rex optimizes pagination using ideas (2) and (3) above. Then, Section 4.2 discusses how I-Rex optimizes tracing and pinning. Section 4.3 describes the overall system and other details.

4.1 Optimizing Pagination

Given a base or derived table to display, the potential savings of pagination are easy to see: by focusing on one page at a time, we only need to compute, transmit, and render content on this page alone. The baseline solution to pagination supported by SQL uses its `OFFSET` and `LIMIT` features. However, `OFFSET` and `LIMIT` alone often do not lead to faster queries (which we will experimentally verify in Section 5). The optimizer typically has insufficient knowledge to skip directly to the `OFFSET`-th result row, so the query often executes from the very beginning to `OFFSET+LIMIT`, creating enormous waste. Furthermore, for queries that enforce a specific result order (which is the norm in I-Rex as it aims to provide consistent and reproducible orderings for all results), simply determining the order would often involve computing and sorting all result rows, which ends up saving no computation cost.

To overcome this limitation, we observe that if we know which input rows contribute to the particular result page, we can use such information to prefilter the input rows and reduce execution cost. For example, in the joined & filtered table of Figure 2, suppose a user only needs to retrieve the page containing rows j_2 through j_5. It turns out that we only need $\{s_1, s_2\}$ from `Serves` and $\{f_1, f_3\}$ from `Frequents` to compute this page. In general, explicitly enumerating a set of such input rows is not scalable, but we can instead compute a compact summary of this set, at the expense of potentially introducing some false positives (but never any false negatives).

I-Rex has three types of summary-based filters, described further below, to cover the range of design trade-offs: 1) IID-based filters, 2) "sargable" [62] filters, and 3) Bloom filters [12]. All three filters require precomputing a summary of what input rows contribute to the content of each page. We will show later in Section 4.3 how I-Rex does so as part of the initialization step of the debugging context. That step produces a *milestone table* for each input table of the debugging context and for the result table of each stage. For each page of table contents, a milestone row contains the summaries of input rows that contribute to the page. The milestone tables are cached by the I-Rex client. Subsequently, when the user requests a particular page of a table, I-Rex consults the corresponding milestone table to generate a *page-fetch query* that incorporates filtering using the summaries pertaining to the page requested.

	min_iid	s.bar	s.beer	f.drinker	f.bar
0	((Apex, Corona), (Amy, Apex))	[Apex, Edge]	[Amstel, Dixie]	[Amy, Ben]	[Apex, Edge]
1	((Edge, Amstel), (Dan, Edge))	[Edge, Edge]	[Amstel, Corona]	[Ben, Dan]	[Edge, Edge]
2	((Tavern, Amstel), (Coy, Tavern))	[Tavern, Tavern]	[Amstel, Erdinger]	[Coy, Coy]	[Tavern, Tavern]

Table 1: Milestone table for the joined & filtered table in Figure 2, with 3 pages and page size of 3 (rows). The IIDs have the format ((`s.bar`, `s.beer`), (`f.drinker`, `f.bar`)), and three minimum IIDs are those of the rows j_0, j_3, and j_6 respectively. The Bloom filter column is omitted.

4.1.1 IID-Based Filtering. Recall that I-Rex sorts every table by its IID to ensure consistency and reproducibility. Hence, pages of a table partition its rows into consecutive, non-overlapping IID ranges. To enable IID-based filtering, we precompute, in the milestone table, the minimum IID among rows on each page. For example, Table 1 shows the milestone table for the joined & filtered table in Figure 2, with the minimum IID for each page captured by the `min_iid` column. Using this information, I-Rex can add a tight range condition on IID to the `WHERE` clause of a page-fetch query.

Continuing with the same example, the following query fetches the contents of the second page of the joined & filtered table, while also synthesizing the IID for each result row:

```
SELECT ((s.bar,s.beer),(f.drinker,f.bar)) AS _iid, -- synthesize IID
  *
FROM Serves s, Frequents f
WHERE (...) -- original WHERE conditions
-- IID-based filtering:
AND ((s.bar,s.beer),(f.drinker,f.bar))>=(('Edge','Amstel'),('Dan','Edge'))
AND ((s.bar,s.beer),(f.drinker,f.bar))<(('Tavern','Amstel'),('Coy','Tavern'))
ORDER BY 1; -- order by IID
```

Note that the lower IID bound is the minimum IID of the second page as recorded in Table 1, and the (open) upper IID bound is the minimum IID of the next page.

IID-based filtering serves two purposes. First, it rejects any result row not on the requested page. This feature is indispensable because the other types of filtering implemented may introduce false positives and admit result rows outside the requested page; therefore, I-Rex always activates IID-based filtering to ensure correctness. Second, IID-based filtering can enable more efficient execution. However, its potential is limited: from a range bound on a multi-component IID, we can safely infer a range bound only on the leading component, but not on subsequent components. For example, in the page-fetch query above, the query optimizer may infer that `s.bar` (and hence `f.bar` by transitivity) falls within [`Edge`, `Tavern`] and use an index on `s.bar` (or `f.bar`), but nothing is known about `s.beer` or `f.drinker`. This limitation motivates other types of filtering below.

4.1.2 Sargable Filtering. To enable efficient page-fetch queries, we aggressively look for opportunities to inject safe, *sargable* [62] predicates that enable index plans. To this end, for each column A in an input table, where an index already exists on A, I-Rex computes a concise summary of the A values, in the form of a single range [min, max], over all input rows that contribute to each page of the result table. For example, Table 1 shows the milestone table for the joined & filtered table in Figure 2, assuming that `s.bar`, `s.beer`, `f.drinker`, `f.bar` are the four indexed columns in `Serves` and `Frequents`. With this information, I-Rex injects a range condition for each of these columns in the `WHERE` clause of a page-fetch query.

Using the same example, the page-fetch query for the second page of the joined & filtered table can now be augmented as follows:

```
SELECT ... FROM Serves s, Frequents f
WHERE (...) -- original WHERE conditions
-- sargable filtering:
AND s.bar BETWEEN 'Edge' AND 'Edge'
AND s.beer BETWEEN 'Amstel' AND 'Corona'
AND f.drinker BETWEEN 'Ben' AND 'Dan'
AND f.bar BETWEEN 'Edge' AND 'Edge'
AND ... -- IID-based filtering
ORDER BY 1; -- order by IID
```

Since the sargable filters are always on indexed columns, they enable the optimizer to consider index plans that access only the relevant parts of the input tables. Even when index plans are not the most optimal, the inexpensive filter conditions still reduce the number of rows involved in downstream processing (e.g., join) and hence overall execution cost.

Remarks. Note that an intelligent optimizer might be able to infer the same bounds on `s.bar` and `f.bar` from the IID-based filtering condition; however, sargable filtering makes them explicit and more easily recognizable by the optimizer. Furthermore, the bounds on `s.beer` and `f.drinker` cannot be inferred.

On the other hand, pushing down filters too aggressively may have a negative effect when the filters are not selective, which can happen if the page size is not small and the result row ordering does not correlate with the filter column value. The optimizer may underestimate the output cardinality of the filter and choose a secondary index scan that is less efficient than a table scan. Therefore, we only inject a sargable filter for column A if its range on the requested page covers a small percentage of the entire domain of A. I-Rex uses 30% as a cutoff, which works well empirically.

Finally, instead of using one range to summarize input column values for a page, we can use multiple ranges to reduce false positives, at the expense of higher precomputation and storage costs for milestones. This trade-off is worth investigating as future work. However, even if individual input column values are free of false positives, we must still keep the IID-based filtering condition, because duplicate column values and joins can still admit result rows outside the requested page.

4.1.3 Bloom Filtering. Correlated subqueries in `WHERE` can still bottleneck query execution, especially if the external columns they reference are not indexed and therefore not covered by sargable filtering. One approach to avoid evaluating expensive correlated subqueries is memoization: regarding each correlated subquery q as a function, we can cache all value settings of the external columns that q is invoked with, along with the corresponding results returned by q. However, this approach is not scalable as the number of possible value settings can be high (even with sophisticated decorrelation techniques [63] to restrict such settings) and results can be large, requiring large cache spaces. Furthermore, if this cache is deployed on the client, the client must embed the cache contents explicitly into the page-fetch query, which makes resulting SQL complicated and harder to optimize.

To balance space efficiency and performance, we take an approximate approach at a coarser grain: instead of capturing the behavior of each subquery precisely, we consider the entire `WHERE` condition (including any constituent subqueries), and use a Bloom filter [12] to track, for each page, the set of "relevant" input column values for which the entire `WHERE` evaluates to true. We choose relevant columns to be all input table column externally referenced by any subquery, plus any input table column involved in an atomic condition together with some subquery. For instance, in our running example (Q from Example 2.1), the only relevant input column is `f.drinker`. As another example, if the `WHERE` condition is A `IN` $(q(B))$, where q is a correlated subquery with external column reference B, the set of relevant input columns would be $\{A, B\}$. As shown in Section 4.3, a single SQL query can precompute such Bloom filters conveniently using a user-defined aggregate function, together with the rest of the milestone table.

Continuing with our running example, the page-fetch query for the second page of the joined & filtered table is now updated to:

```
SELECT ... FROM Serves s, Frequents f
WHERE (...) -- original WHERE conditions
AND ... -- sargable filtering
AND ... -- IID-based filtering
--- Bloom filtering:
AND BLOOM_CHECK('10101010', f.drinker)
ORDER BY 1; -- order by IID
```

Here, `BLOOM_CHECK(`F`,`e`)` is a user-defined function that checks whether an entry e is in the Bloom filter F. Given the probabilistic design of Bloom filters, `BLOOM_CHECK` may return false positives but not false negatives. `'10101010'` is the Bloom filter computed for the second page, encoding all `f.drinker` values over input combos that contribute to this page. Failing `BLOOM_CHECK` allows query execution to save time by bypassing the evaluation of the expensive subquery. In this example, row f_2, with `f.drinker` of `Coy`, fails the subquery condition, so the computed Bloom filter will not have the contribution of `Coy`. Hence, when evaluating the above page-fetch query, even though `Coy` still passes the sargable filter, it will likely fail `BLOOM_CHECK`, allowing the rest of `WHERE` including the subquery to be skipped. On the other hand, because of false positives, passing `BLOOM_CHECK` does not mean `WHERE` must evaluate to true; the original `WHERE` condition still must be included.

Remarks. The proposition below (proof in Appendix B) implies that there is considerable freedom in choosing the set of relevant input columns to track for a Bloom filter. To one extreme, we can simply feed the IIDs of the input rows into the Bloom filter, such that it tracks which input combos contribute to the requested page. However, the number of distinct entries can be huge, especially if the query involves aggregation, raising the false positive rate. Instead, we have decided to track input column values that influence the outcome of evaluating conditions involving subqueries, because there may be far fewer distinct values that need to be tracked. This heuristic has worked well in practice for I-Rex.

Proposition 4.2. *Suppose functions* `BLOOM_GEN` $: \mathcal{P}(tuples) \rightarrow bitstrings$ *and* `BLOOM_CHECK` $: bitstrings \times tuples \rightarrow \{true, false\}$ *satisfy* $e \in \mathcal{V} \Rightarrow$ `BLOOM_CHECK(BLOOM_GEN(`$\mathcal{V}$`),`e`)` *for any tuple* e *and any set* $\mathcal{V}$ *of tuples of the same sort. Let* $\{R_i\}$ *denote a list of (potentially aliased) input tables and* Θ *a condition over* $\{R_i\}$*. For any subset* $\mathcal{A}$ *of columns in* $\{R_i\}$*, the following two queries are equivalent:*

Q_1: `SELECT * FROM` $\ldots, R_i, \ldots$ `WHERE` Θ;

Q_2: `SELECT * FROM` $\ldots, R_i, \ldots$ `WHERE` Θ `AND BLOOM_CHECK(`F`,` $\langle\mathcal{A}\rangle$`)`;
where F = `BLOOM_GEN(SELECT` $\langle\mathcal{A}\rangle$ `FROM` $\ldots, R_i, \ldots$ `WHERE` Θ`)`.

Finally, note that Bloom filter conditions are not sargable and cannot be used to avoid full table scans. Evaluating them introduces overhead too. Therefore, I-Rex uses Bloom filtering only for "short-circuiting" evaluation of expensive WHERE clauses with subqueries. Also, if a Bloom filter returns too many false positives, the overhead of BLOOM_CHECK may outweigh the savings achieved by skipping subquery evaluation. Hence, we estimate the false positive rate of each constructed Bloom filter using its size (bits), number of hash functions, and the number of insertions. We inject the BLOOM_CHECK condition only if this rate is less than 50%.

4.2 Optimizing Tracing and Pinning

With optimized page fetches, users can freely move around a table with a paginated display. However, a debugging context displays multiple tables, and I-Rex must coordinate the pages displayed across tables to show the end-to-end derivation from the current input combo to output, so that users can forward- or backward-trace (using pinning). We now discuss how to optimize these operations.

4.2.1 Forward Tracing. For forward tracing, we already have access to the input combo as well as their IIDs and row contents. Conceptually, to forward-trace through a particular stage, it suffices to determine the IID (say t) of the derivative row produced by this stage, if one exists. Then, given the target IID t, I-Rex searches the milestone table of the stage's result table for the page whose IID range contains t, and fetches this page. If the fetched page indeed contains t, we have successfully forward-traced through the stage; otherwise, we know that the input combo yields no result rows.

Determining the target IID in the first place is usually straightforward, thanks to the logical nature of our IIDs. For example, given the input combo $\langle s_2, f_1 \rangle$ in Figure 2, the target IID for the join & filter stage is simply the concatenation of the IIDs of s_2 and f_1. In cases where we cannot easily determine the target IID, I-Rex can compute it using a query. For example, to forward-trace through the group stage, we need to compute the GROUP BY expression value, the leading component of the result row IID. In this running example, the GROUP BY expression is simply `s.bar`, so we could have read its value Edge directly from row s_2. Otherwise, in general cases where the GROUP BY expression is complex (e.g., involving subqueries), I-Rex would generate a query to compute its value, e.g.:

```
SELECT s.bar -- arbitrary GROUP BY expression
FROM Serves s, Frequents f -- same as the original query block
WHERE (s.bar, s.beer) = ('Edge', 'Amstel') AND (f.bar, f.beer) = ('Ben',
       'Edge'); -- use IID values to specify input combo
```

The cost of such queries is negligible because of the highly specific WHERE condition.

4.2.2 Pinning for Backward Tracing and Watchpointing. Without loss of generality, assume that a single derivative row is pinned.[4] Our goal is to determine the first (lexicographically) input combo in the pinned subspace. If only one input combo contributes to the pinned derivative row, we can simply infer the former from the pinned row's IID (the same applies when backward-tracing from a stage to its preceding stage). For example, in Figure 2, if a user pins the first member row of g_1 in the group table, the IID of this row, (Edge, (s_2, f_1)), will reveal the input combo $\langle s_2, f_1 \rangle$.

[4]Since I-Rex enforces that there is only one derivative row per stage, it suffices to consider the derivative row in the earliest stage.

If multiple input combos contribute to one pinned derivative row, the situation is more complicated. Such cases arise when backward-tracing from a pinned row in a post-grouping stage, e.g., the final stage of Figure 2. Here, we first infer the group, identified by its GROUP BY expression value (say g), from the IID of the pinned row. There are two cases. First, if there are no additional pins on the input rows, we simply need to determine the first input combo contributing to group g. To this end, I-Rex searches the milestone table of the group table for the last page whose `min_iid` is less than $(g, -\infty)$, and fetches that page. If the fetched page contains any member row of g, the IID of the first such row will reveal the desired input combo. Otherwise, it can be shown that the next page's `min_iid` must give the desired input combo. Hence, the cost of recovering the input combo is only one page fetch in the worst case.

Example 4.3. Considering Figure 2 again. Assuming a page size of 3 (rows) for all tables and o_1 is pinned, I-Rex can quickly determine that the first page in the group table (which contains all tuples from g_0 and one tuple from g_1) is the last page whose `min_iid` is smaller than $\langle$Edge, $-\infty\rangle$. It thus generates the following query to fetch the first page of the group table:

```
SELECT (s.bar) -- group IID
       ((s.bar, s.beer), (f.drinker, f.bar)) -- input combo IID
       s.price * f.times -- sum_input
FROM Serves s, Frequents f
WHERE (...) -- original WHERE
AND ... -- sargable filtering
-- group IID and input combo IID filtering
AND ((s.bar), ((s.bar, s.beer), (f.drinker, f.bar))) >= (('Apex'), (('Apex',
       'Corona'), ('Amy', 'Apex')))
AND ((s.bar), ((s.bar, s.beer), (f.drinker, f.bar))) < (('Edge'), (('Edge',
       'Amstel'), ('Dan', 'Edge')))
ORDER BY 1, 2; -- first order by group, then by input combo
```

After obtaining the following result, it can be easily identified that the last tuple in the page carries the first input combo IIDs that contribute to o_1:

group IID	input combo IID	sum_input
(Apex)	((Apex, Corona), (Amy, Apex)))	1
(Apex)	((Apex, Dixie), (Amy, Apex)))	2
(Edge)	((Edge, Amstel), (Ben, Edge)))	16

Second, when there are additional pins on the input rows, I-Rex instead generates a query to determine the first input combo in the subspace further constrained by the additional pins. This query inherits the FROM and WHERE clauses from the original query, but further includes in WHERE conditions to restrict input rows by the pins and ensure that GROUP BY expression evaluates to g; it then computes the minimum input combo in SELECT.

Example 4.4. Continuing from Example 4.3. When s_2 is further pinned in addition to o_1, to compute the first input combo in the subspace of the group table, I-Rex generates the following query:

```
SELECT (s.bar) -- group IID
       ((s.bar, s.beer), (f.drinker, f.bar)) -- input combo IID
FROM Serves s, Frequents f
WHERE (...) -- original WHERE
AND (s.bar, s.beer) = ('Edge', 'Amstel')
ORDER BY 1, 2 -- first order by group, then by input combo
LIMIT 1; -- only need first input combo
```

The above query locates $\langle s_2, f_1 \rangle$ as the first input combo, and I-Rex can now decide the page to fetch in the group table by a simple binary search using the group IID and input combo IID.

Recall that to support watchpointing, I-Rex automatically colors rows in each table relevant to the pinned subspace. The case where the pinned subspace contains only one input combo is straightforward: relevant rows are simply those in the input combo and those that are its derivatives. Otherwise, I-Rex extends the page-fetch query to return an additional column that indicates whether each row is relevant to the pinned subspace. The SELECT expression for this column is a Boolean expression testing whether the input rows conform to the pins and the GROUP BY expression evaluates to the same value as the pinned group.

4.3 System Implementation Details

Architecture. I-Rex has a client-server setup, with a Web frontend as the client and a middleware server between the client and the database server. In a common use case in education, many student users may run debugging sessions against one large, shared, read-only database. To ensure scalability and easy deployment, we adhere to a strictly stateless design, where neither the middleware nor the database server stores any session-specific information:

- When a user starts a debugging session, I-Rex middleware analyzes the query and decomposes it into a graph of query blocks in an internal representation. This representation specifies how each block is further broken down into stages, and contains metadata such as how to resolve external column references for correlated subqueries. The client caches this representation.
- When the user enters a debugging context (i.e., "calling" a query block), the client sends the cached representation of this block back to the middleware along with any parameter settings for the call. The middleware's *debugging context initializer* computes the milestone tables for all input tables of the debugging context and all result tables of its constituent stages, by querying the database (discussed below). The debugging context initializer also generates SQL query templates needed to support various debugging operations discussed earlier in this section. The client caches these milestone tables and SQL templates.
- When user carries out various operations in the debugging context, the client consults the cached milestone tables to instantiate required SQL queries according to the cached templates. It also caches the results of page-fetch queries, so repeated accesses to the same page, which can happen frequently during debugging, do not need recomputation. Cached pages can be evicted to make space for new ones.

Overall, note that debugging sessions leave no state on the middleware or the database. The middleware handles all complex SQL query analysis and rewrites, and the client simply needs to fill in the values of certain query parameters. The only cached data whose size depends on the database are the milestone tables. However, since there is only one milestone row per display page, and all columns are compact summaries of constant size, the milestone tables are orders of magnitude smaller than the corresponding tables. To make them even more scalable, milestones could be made hierarchical and refined on demand, but as shown later in Section 5, the current single-level design already works well.

Additional Details on Query Rewriting. As discussed earlier, all information needed for debugging can be computed by rewriting the original SQL query in some fashion. I-Rex performs most query rewriting in its debugging context initializer. We have already shown how to generate queries for fetching pages (Section 4.1) and supporting other debugging operations (Section 4.2). As for computing milestones for each table in the debugging context, a single SQL query suffices, as illustrated by the following, which computes Table 1 (including the Bloom filters) for our running example:

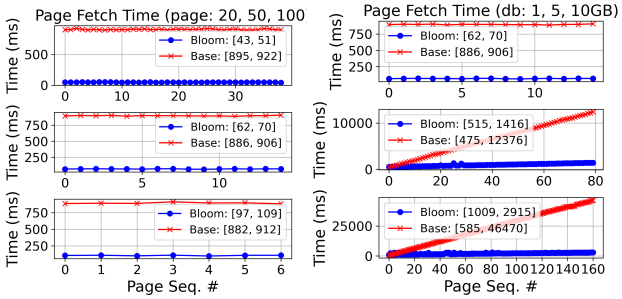
```
WITH tmp(seq, iid, s_bar, s_beer, f_drinker, f_bar) AS (
  SELECT ROW_NUMBER() OVER (ORDER BY s.bar,s.beer,f.drinker,f.bar) - 1, --
         result row sequence number when sorted by IID
    ((s.bar, s.beer), (f.drinker, f.bar)), -- IID
    s.bar, s.beer, f.drinker, f.bar -- relevant for sargable & Bloom filtering
  FROM Serves s, Frequents f WHERE (...) -- same as original query
)
SELECT MIN_IID(iid), -- per-page minimum IID
  -- per-page range bounds for sargable fitering:
  ARRAY[MIN(s_bar), MAX(s_bar)], ARRAY[MIN(s_beer), MAX(s_beer)],
  ARRAY[MIN(f_drinker), MAX(f_drinker)], ARRAY[MIN(f_bar), MAX(f_bar)],
  -- per-page Bloom filter:
  BLOOM_GEN(p_partkey)
FROM tmp GROUP BY seq / page_size ORDER BY seq / page_size;
-- page_size is the number of rows per page
```

After generating the SQL queries as discussed above and earlier, I-Rex further applies several rewrite optimizations. First, if a scalar subquery contains no external column references, the debugging context initializer will simply precompute its result and replace the uses of this subqueries by its result. Second, for any sargable condition injected into a query, we consider pushing it down further into a subquery. Such cases often arise when, for example, we identify a range bound on some indexed column, and this column (or another column equated to it by WHERE) is referenced by a subquery as an external column; here, we inject the range bound into the subquery as well. Third, if the query's FROM contains a subquery or table defined by WITH, we check whether the injected conditions imply an equality or range condition the IID of the input table. If yes, we consult the milestones of the input table to construct sargable filters to further inject into the subquery defining the input table. We call this last optimization *recursive pushdown*. Finally, pushdown through outer joins, which is especially tricky, is discussed in Appendix B.

5 PERFORMANCE EXPERIMENTS

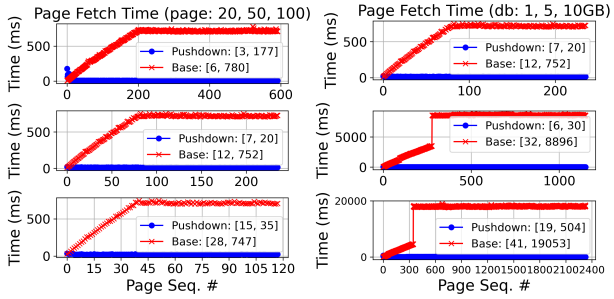
We conduct experiments to evaluate the performance and scalability of I-Rex. We focus on evaluating page-fetch queries (Section 4.1) and milestone computation (Section 4.3), as they are the most expensive queries that I-Rex uses; other operations (Section 4.2) either use page-fetch queries or relatively cheap queries with highly selective conditions. The baseline for fetching pages is to use SQL OFFSET and LIMIT. We enable various pagination optimizations in I-Rex to evaluate their respective benefits. IID-based filtering is always enabled (for correctness). Section 5.1 enables Bloom filtering but not other optimizations, while Section 5.2 enables sargable filtering but not others. Finally, Section 5.3 enables all optimizations, and also evaluates the overhead of milestone computation.

We use the TPC-H benchmark [10] and generate three database instances of sizes 1GB (benchmark default), 5GB, and 10GB. In addition to the default indexes on the primary keys, we also create a reasonable set of secondary indexes (details in Appendix C) that simulate typical usage. For Sections 5.1 and 5.2, we show one query for each, where the respective optimization is applicable and can be



(a) *Varying page size;* 1*GB database.* **(b)** *Varying database size;* 50*-row pages.*

Figure 5: Bloom filtering vs. baseline: time to fetch each page. $[\cdot, \cdot]$ in legend shows min/max page fetch times.



(a) *Varying page size;* 1*GB database.* **(b)** *Varying database size;* 50*-row pages.*

Figure 6: Sargable filtering vs. baseline: time to fetch each page. $[\cdot, \cdot]$ in legend shows min/max page fetch times.

best evaluated; for the general evaluation in Section 5.3, we show results for all 22 benchmark queries.

All experiments were done locally on a 64-bit Ubuntu 22.04 LTS server with Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, 64GB RAM, and 256GB disk space. We used PostgreSQL [59] (version 16) and only changed the following configurations: we turned off parallelism for simplier and fair interpretability of results; we set `work_mem` to 128MB and `shared_buffers` to 8GB.

5.1 Effectiveness of Bloom Filtering

This experimental setup uses a variant of TPC-H Q2:

```
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 from outer query
    AND ...);
```

As discussed in Section 4.1, I-Rex precomputes a Bloom filter for the `p_partkey` values for input rows that contribute to each result page, and when fetching a page, uses the Bloom filter to short-circuit the evaluation of the rest of `WHERE` containing an expensive correlated subquery. However, the bloom filter is only used for membership checking if the false positive rate is less than 50%.

By default, we set the number of Bloom filter bits to $m = 1024$; we conservatively estimate the number of unique `p_partkey` in each page (denoted by n) as the page size; accordingly, we set the number of Bloom filter hash functions to $\frac{m}{n} \times \ln 2$.

We conduct two sets of experiments. First, we fix the database size to 1GB and vary the page size. Second, we fix the page size to 50 and vary the database size. We compare page-fetch queries with Bloom filtering with the baseline using `OFFSET` and `LIMIT`. We collect the execution times reported by the `EXPLAIN ANALYZE` command for all queries and show them in Figure 5.

Overall, the results show that Bloom filtering is very effective for Q2. When the database size is relatively small (Figure 5a), the baseline consistently takes around 900ms to reproduce a page, because its execution cost is dominated by the computation of the entire result and sorting them first; in contrast, Bloom filtering takes 43 to 109ms, with larger pages requiring more time. Under larger database sizes (Figure 5b), baseline switches to a different plan whose cost grows linearly with the starting position of the page fetch, eventually matching the cost of executing the entire query when fetching pages near the tail; Bloom filtering performance remains scalable and depends much less on the fetch position, saving as much as 11s (8.7× speedup) and 43s (15.9× speedup) for 5GB and 10GB databases, respectively.

5.2 Effectiveness of Sargable Filtering

This setup uses the joined & filtered table for benchmark Q7:

```
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 ...; -- same as original query
```

We create sargable range filters for all index columns in the input tables based on the precomputed milestones, but only if the ranges cover no more than 30% of the domain, as discussed in Section 4.1. We run two sets of experiments similar to those for Bloom filtering, with results shown in Figure 6. Overall, we observe that the baseline using `OFFSET` and `LIMIT` performs progressively worse when it fetches pages later in the result table, until its running time plateaus when the plans switch to essentially computing and sorting the entire result; in contrast, sargable filtering performs well across all pages regardless of their position, and its advantage over the baseline widens dramatically toward later pages. With a 1GB database (Figure 6a), baseline can take up to 780ms to fetch a page, while sargable filtering takes no more than 180ms (4.3× speedup). Fixing the page size at 50 and using larger databases (Figure 6b), sargable filtering's benefit is even greater. For the 10GB database, the baseline takes about 19s to fetch a page positioned at 1/6-th of the entire result or later, while sargable filtering takes about 0.5s (38× speedup) in the worst case.

5.3 General Evaluation

For general evaluation, we apply all I-Rex optimizations to all TPC-H queries, and compare the execution times of the resulting page-fetch queries with those of the baseline approach. IID-based and sargable filtering can be applied to all queries, while Bloom filtering is only applicable to Q4, Q16, Q18, and Q20–22. Because of limited space, we only show in Table 2 the results for the joined & filtered table of Q18, which is the most expensive query in our experiments; remaining results are presented in Appendix C. The results generally echo the findings from the experiments on Bloom/sargable filtering, and confirm the benefit of combining optimizations. Specifically in Table 2, I-Rex performs no worse (and sometimes

	1GB (ms)		5GB (ms)		10GB (ms)	
Page	I-Rex	Baseline	I-Rex	Baseline	I-Rex	Baseline
head	3,919.89	8,801.75	21,717.035	75,903.361	52,355.55	158,664.23
middle	3,919.76	12,270.41	24,999.351	91,594.34	47,736.515	181,742.56
tail	4,423.78	15,739.07	21,717.478	107,285.319	49,884.912	204,820.89

Table 2: I-Rex vs. baseline: time to fetch a page in the joined & filtered table of Q18; 50-row pages and varying database size. Here, head refers to the first query while tail refers to the second to the last (as the last page sometimes is not full).

	Milestone Query		Original Query	
Q18	Exec. time (ms)	Output (MB)	Exec. time (ms)	Output (MB)
joined & filtered	22,619.063	52	16,847.586	1272
group	18,208.509	9	18,368.903	70
output	19,340.111	9	20,751.758	75

Table 3: Milestone vs. original queries: execution time and output size; 50-row pages and 1GB database.

better) than the baseline for pages at the beginning of the results, and significantly better for later pages, with 3× to 5× speedup.

Finally, we evaluate the overhead of debugging context initialization, which is dominated by the cost of running milestone queries (such as the one in Section 4.3). We compare their costs with those of simply computing the entire results of the corresponding original query blocks. Because of limited space, we only show in Table 3 the results for the three most expensive milestone/original query pairs, with a 1GB database and 50-row pages; the remaining results are in Appendix C. All three pairs come from Q18's stages. Since milestone queries conceptually summarize the output of the original queries, we do not expect them to run faster than the latter. As Table 3 shows, their performance is in fact comparable to the original queries. Recall from earlier experiments that baseline page-fetch queries often cost as much as the original queries, so this observation implies that I-Rex can benefit from milestone-enabled optimizations for many page-fetch queries by paying only a one-time overhead equivalent to a single baseline page-fetch query. Finally, we note that the execution times reported are taken from `EXPLAIN ANALYZE`, which only measures the time spent in executing the query but not transmitting its result. We intentionally chose to exclude the latter time, because the outputs from the original queries can be too overwhelmingly large to transmit. For example, printing the entirety of Q18's joined & filtered table took more than an hour for a `psql` client running on the database server console. In contrast, as Table 3 shows, milestone queries return much smaller results, which are feasible to transmit to and handle by the client. The full results in Appendix C lead to the same conclusions.

6 USER STUDY

We conducted a user study in an undergraduate database course to evaluate the overall effectiveness of I-Rex on two aspects: 1) whether I-Rex helps users catch more logical bugs in SQL queries, and 2) whether I-Rex reduces the time to find such bugs.

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

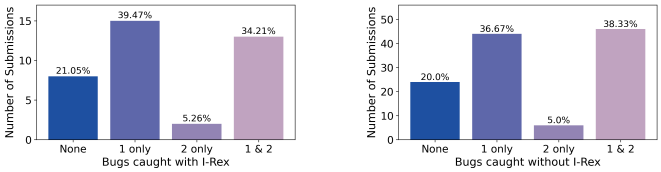


Figure 7: Bugs caught (out of two) for P1, using I-Rex vs. not.

the lack of control would make results difficult to interpret. The demanding nature of the tasks (as evidenced by the long debugging times measured, to be reported later) also makes quality control difficult for asynchronous and remote completion.

Preparation and Setup. The study was conducted during two 75-minute discussion sessions in consecutive weeks. In the first session, students were given a tutorial on I-Rex and informed about the format of the survey-style quiz, which contained two SQL debugging problems P1 and P2. After the first discussion, I-Rex is made public for students to get familiar with. In the second session, students completed the quiz synchronously in a proctored environment, where they were asked not to discuss with classmates. For each problem in the quiz, students were provided a problem statement, an incorrect query, its incorrect output, and what the correct output should be; the details are in Appendix D. The database instance in the user study, on which both the incorrect and expected outputs were based, had been installed and worked on by students on their laptops early in the course. We also prepared a public `pgAdmin` Web interface for students to use during the study in case any had trouble accessing their own instances, thereby mitigating potential impact of setup issues on the measured debugging times.

To create treatment and control groups, students received the two problems in a random order. For the first problem they received, they were free to use any tool of their choice (e.g., Gradescope autograder, `psql` console, `pgAdmin` Web interface) except I-Rex. For the second problem, I-Rex was made available along with other tools; students had the option to use or not use I-Rex, and were asked to indicate it in the survey. For each problem, students were to describe the bugs they found in a free-response text box. P1 had two bugs and Q2 had three, though the students were not told how many. While students self-paced, they were recommended to spend 15-20 minutes on each problem. They were not allowed to move on to the second problem until they finished the first.

Results and Analysis. We collected the time it took students to solve each problem, whether they chose to use I-Rex, and the bugs they found. Of 237 students, 140 completed both problems and provided legitimate answers. Therefore, we based our analysis on these 140 responses. For the first problem, which must be completed without I-Rex, 73 students received P1, and 67 received P2. For the second problem, where using I-Rex was an option, 73 students received P2, and 36 of them chose not to use I-Rex; 67 students received P1, and 29 of them chose not to use I-Rex. Therefore, for P1, we have 38 submissions using I-Rex, and 102 not using I-Rex; for P2, we have 37 submissions using I-Rex, and 103 not.

We manually reviewed each response and checked whether the student identified the bugs correctly. Figure 7 shows the fraction of the submissions that correctly identified each possible subset of the bugs for P1, with (left) or without (right) help from I-Rex; Figure 8 does the same for P2. While the total submission numbers differ

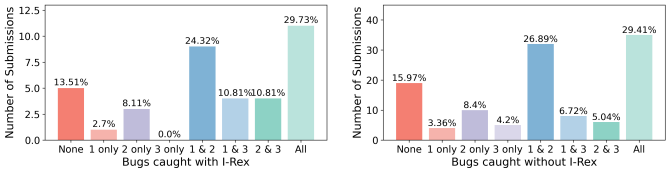


Figure 8: Bugs caught (out of three) for P2, using I-Rex vs. not.

Problem	# responses	Avg. time (s)	Avg. bugs found
P1 w/ I-Rex	38	771.52	1.13 / 2
P1 w/o I-Rex	102	1,235.66	1.18 / 2
P2 w/ I-Rex	37	1,147.04	1.91 / 3
P2 w/o I-Rex	103	1,657.53	1.85 / 3

Table 4: Average debugging time and bugs caught per problem.

between using/not using I-Rex, the distributions (fractions for possible subsets) are similar. Indeed, use of I-Rex had no discernable impact on the average number of bugs identified, as shown later in the last column of Table 4. While I-Rex did not seem to help students find more bugs, the comparison of debugging times, summarized in Table 4, reveals a major advantage of I-Rex. For P1, use of I-Rex reduces the average debugging time from 21 minutes to 13, a 38% reduction; for P2, I-Rex reduces the time from 28 minutes to 19, again by more than 30%. Combined with the earlier observation that the distributions of bugs correctly found are similar, we conclude that the I-Rex significantly improves students' efficiency in finding bugs without compromising accuracy. Had time been more constrained, use of I-Rex would result in more bugs found.

Remark. A counterargument to our conclusion might be that students might rush to get out of class or have just "warmed up" their debugging skills, so they generally spent less time on the second problem received. However, even if we restrict the scope of time comparison to the second problem received only, we can still see that I-Rex offers improvements. Of the 102 submissions of "Q1 w/o I-Rex", 29 of them had I-Rex as an option but did not use it, and their average debugging time was 1,029 seconds, compared with 771.52 seconds with I-Rex. Of the 103 submissions of "Q2 w/o I-Rex", 36 of them had I-Rex as an option but did not use it, and their average was 1,310 seconds, compared with 1,147.04 seconds with I-Rex.

7 RELATED WORK

Earlier versions of I-Rex, previously demonstrated [43, 53], precomputed and stored all debugging information in the client, which did not scale. The version described in this paper has refined existing debugging features and introduced new ones, and most importantly, added support for scalability, which requires a redesign of the system and development of new optimization techniques.

Finding Logical Errors in Queries. Work toward finding logical errors in queries can be classified into two categories. The first category [23, 24, 26–28, 33, 36, 42, 54, 60, 68] assumes knowledge of a correct reference query and uses it to help identify errors in an incorrect query. XData [24] checks the correctness of a query by running the query on self-generated testing datasets. Cosette [26–28], SQL-Solver [33], and QED [68] test query equivalence using constraint solvers and theorem provers. RATest [54] and c-instances [36] aim at constructing small and illustrative database instances to show the differences between two queries. [23] develops a grading system that canonicalizes queries with rewrite rules and then decides query similarity using edit distance between the resulting logical plans. SQLRepair [60] and QR-Hint [42] focus on fixing the wrong query by proposing syntactical edits. I-Rex differs from this line of work in that it assumes no knowledge of the reference query.. Dropping this assumption fundamentally changes the problem and makes the solution applicable in more settings.

The second category of work, including I-Rex, helps users debug a query without a given reference query, using a myriad of approaches. Qex [67] generates input relations and parameter values for unit-testing parameterized SQL queries. SQLLint [13–15] looks for patterns in the query indicative of common semantic errors and alerts users to them. Interactive query builders and query visualizers [1–5, 22, 40, 47, 52, 55] use diagrams to gain more intuitive understanding of query logic, hence helping with debugging. Frameworks for data exploration [5, 7, 32, 50] use signals such as query history and user feedback on query results to suggest queries, which may serve as an approach to debugging. A line of work known as algorithmic debugging [18, 19] guides users through a series of questions on whether intermediate query steps produce intended results, to locate buggy steps. Also useful to debugging is explaining to users what queries do in natural language, which has been researched over the years [16, 35, 46, 49, 64, 69]. While the systems discussed above provide complementary approaches to debugging, none supports debugging by tracing through query execution, as I-Rex and most GPL debuggers do.

Two systems in this second category, DESQL [41] and Habitat [31, 39], are the closest to I-Rex in terms of approach, as they also recognize the need to debug queries according to how they are written. DESQL [41] is a debugger for SQL in Spark, which decomposes the query into subqueries and helps users examine subqueries' output. While DESQL focuses on scalability like I-Rex, its optimizations primarily concern Spark instead of traditional database systems. Also, unlike I-Rex, DESQL does not support correlated subqueries or row-level tracing, and debugging commences only after the query fully executes. Habitat [31, 39] allows users to mark SQL subexpressions and inspect intermediate results side by side, connecting related result rows. It also lets users filter these results in order to focus on a subset of interest. However, Habitat differs from I-Rex in important ways. First, focusing on scalability, I-Rex avoids computing or showing full results of queries and subqueries, while Habitat executes its queries in full and shows their results in bulk. Although Habitat's focus filters can restrict these queries, they need to be defined manually and explicitly; in contrast, I-Rex's pinning and automatic pagination are intuitive and demand less user effort and expertise, and I-Rex has a suite of rewrite optimizations designed to improve query efficiency. Second, I-Rex and Habitat have different conceptual designs: I-Rex defines its canonical query execution to be row-oriented and reproducible (including row ordering), while Habitat presents a set-oriented view of SQL execution and allows more free-form exploration among query subexpressions. For example, consider debugging a correlated subquery. I-Rex lets the user "step into" its execution with specific settings of its external column references supplied by the "caller" (its enclosing query block), which is intuitive for GPL users. On the other hand, Habitat shows an intermediate table showing

the binding values and the corresponding subquery evaluation results side by side, analogous to the result of a decorrelated subquery. Users with different background and SQL expertise levels may prefer one design over the other; we believe I-Rex offers an attractive alternative.

Other Areas of Related Work. Several areas of research are related to some of the ideas and techniques used by I-Rex. First, data provenance [11, 17, 25, 38, 44] provides a natural way to trace query execution, explaining why and how a particular output is or is not produced. Many previous works [6, 8, 9, 29, 30, 37, 45, 48, 51, 61] have focused on making provenance capture a built-in feature in database systems. In particular, [56] proposes techniques for capturing provenance by rewriting SQL queries, just as I-Rex uses query rewriting to help support tracing and other debugging operations. However, [56] captures full provenance information for all query results; in contrast, for scalability and interactivity, I-Rex avoids materializing all provenance expressions, or even the entirety of one large provenance expression (e.g., for an aggregate result row), opting to compute pieces of provenance on demand instead.

Second, efficient pagination of query results has been studied in query processing and optimization literature. Recent advances have been made on direct access to query answers [20, 21, 34, 66], but their techniques require specialized indexes and apply only to conjunctive queries, less general than what I-Rex supports. Work on data skipping and predicate pushdown [57, 58, 65, 70] also requires significant changes to database system internals, but I-Rex chooses to leverage existing database systems for efficient pagination. The debugging use case and the stateless server design of I-Rex also motivate unique optimizations, such as client-cached milestones.

8 CONCLUSION AND FUTURE WORK

In this paper we have presented I-Rex, a novel interactive debugger for SQL that is easy to deploy on existing database systems and scales to massive databases. Conceptually, I-Rex executes the query being debugged in a canonical fashion that faithfully follows how the query is written, and lets users examine the execution using a rich set of features including but not limited to those found in GPL debuggers. I-Rex supports efficient exploration of any arbitrary point of execution without fully executing the underlying query. It does so by fully leverarging the database system: it formulates the selective computation of the debugging information around the point of interest as SQL queries, and employs materialization and query rewrite strategies to ensure their execution efficiency. Our performance experiments and user study demonstrate the efficiency and effectiveness of I-Rex.

There are multiple directions for future work. First, we can extend I-Rex to cover the SQL constructs that we do not already support, as mentioned at the beginning of Section 3. Second, it is worth investigating milestone designs beyond single ranges and Bloom filters used in Section 4.1. Third, there is no reason to restrict pins to input and derivative rows; we can support user-defined conditions as well. Finally, we plan to extend I-Rex so that it can isolate and help debug runtime SQL errors (such as division by zero), for which database systems usally provide uninformative feedback.

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] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[13] 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.

[14] 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

[15] 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

[16] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[17] 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.

[18] 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.

[19] 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.

[20] 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

[21] 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

[22] 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.

[23] 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

[24] 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

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

[26] 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

[27] 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

[28] 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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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

[34] 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)*, Graham Cormode and Michael Shekelyan (Eds.), Vol. 290. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:20. https://doi.org/10.4230/LIPICS.ICDT.2024.4

[35] Sebastian Gehrmann, Falcon Dai, Henry Elder, and Alexander Rush. 2018. End-to-End Content and Plan Selection for Data-to-Text Generation. In *Proceedings of the 11th International Conference on Natural Language Generation*. Association for Computational Linguistics, Tilburg University, The Netherlands, 46–56. https://doi.org/10.18653/v1/W18-6505

[36] 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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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

[42] 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

[43] 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.

[44] 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.

[45] 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.

[46] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[47] 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.

[48] 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.

[49] Georgia Koutrika, Alkis Simitsis, and Yannis E Ioannidis. 2010. Explaining structured queries in natural language. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 333–344.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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

[55] 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.

[56] Tobias Müller, Benjamin Dietrich, and Torsten Grust. 2018. You Say 'What', I Hear 'Where' and 'Why'? (Mis-)Interpreting SQL to Derive Fine-Grained Provenance. *Proc. VLDB Endow.* 11, 11 (2018), 1536–1549. https://doi.org/10.14778/3236187.3236204

[57] 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

[58] 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

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

[60] 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

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

[62] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[63] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. 1996. Complex Query Decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, Stanley Y. W. Su (Ed.). IEEE Computer Society, 450–458. https://doi.org/10.1109/ICDE.1996.492194

[64] Chang Shu, Yusen Zhang, Xiangyu Dong, Peng Shi, Tao Yu, and Rui Zhang. 2021. Logic-Consistency Text Generation from Semantic Parses. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. Association for Computational Linguistics, Online, 4414–4426. https://doi.org/10.18653/v1/2021.findings-acl.388

[65] 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

[66] 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

[67] 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)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, 425–446. https://doi.org/10.1007/978-3-642-17511-4_24

[68] 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

[69] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. 2018. SQL-to-Text Generation with Graph-to-Sequence Model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 931–936. https://doi.org/10.18653/v1/D18-1112

[70] 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 ADDITIONAL DETAILS ON DEBUGGING PARADIGM

A.1 Set/Bag Operation Blocks

A SQL set/bag operation block has the form

|| Q_R \UNION|\INTERSECT|\EXCEPT [ALL] Q_S

where Q_R and Q_S are subqueries. Let R and S denote the tables returned by Q_R and Q_S, respectively. The canonical execution procedure of this block consists of two stages: a sort stage that sorts both R and S by their row contents and a final output stage that merges them to produce the result table. To determine the order for the sort stage, we pick a particular ordering of all columns, preferring to reuse R's order (dictated by its IID) as much as possible. Further, to ensure a stable sort in the case of duplicates, we append each input table's IID to the column ordering as needed when sorting the table. The synthesized sort result IIDs have the same format. For instance, suppose that the IID for $R(A_1, A_2, A_3)$ is (A_3, A_1), which implies that R is free of duplicates. In this case, R is already sorted by $(A_3, A_1, A_2, \text{IID}(R))$, and we would sort $S(B_1, B_2, B_3)$ accordingly by $(B_3, B_1, B_2, \text{IID}(S))$, assuming that S may contain duplicates. As another example, suppose that both input tables come from unsorted base tables with duplicates, and these tables' internal row ids serve as the IIDs. In this case, we would sort both tables by their columns, in order, followed by their respective IIDs.

The final output stage merges the two sorted tables. The output row IID always contains the value of columns in the same order as the sort stage IID, followed by a Boolean flag indicating the source input table (0 for R and 1 for S) and a sequence number (0-based) indicating its position among duplicates in the source input table. Depending on the set/bag operation involved, the stage's behavior and the last two components of the IID are defined differently. In the following, let t denote the content of a row in either R or S, and $R[t]$ and $S[r]$ the lists of duplicate rows in R and S (respectively) with this content.

- `INTERSECT`: We consider only the case when $R[t] \neq \emptyset$, and output only the first row of $R[t]$, with the last two components of the IID set to $(0, 0)$.
- `EXCEPT`: We consider only the case when $R[t] \neq \emptyset$. Only if $S[t] = \emptyset$, we output the first row of $R[t]$, with the last two components of the IID set to $(0, 0)$.
- `UNION`: If $R[t] \neq \emptyset$, we output the first row of $R[t]$, with the last two components of the IID set to $(0, 0)$. otherwise, we output the

first row of $S[t]$, with the last two components of the IID set to $(1, 0)$.

- `INTERSECT ALL`: Let $m = \min\{|R[t]|, |S[t]|\}$. We output the first m rows of $R[t]$, with the last two components of the IID set to $(0, 0), \ldots, (0, m-1)$.
- `EXCEPT ALL`: Let $m = |R[t]| - |S[t]|$. Only if $m > 0$, we output the first m rows of $R[t]$, with the last two components of the IID set to $(0, 0), \ldots, (0, m-1)$.
- `UNION ALL`: Let $m = |R[t]| + |S[t]|$. We output all rows of $R[t]$ followed by all rows of $S[t]$, with the last two components of the IID set to $(0, 0), \ldots, (0, |R[t]| - 1)$ and then $(1, 0), \ldots, (1, |S[t] - 1|)$.

The input ("combo" would be somewhat a misnomer here) space for the debugging context is the concatenation of the rows of R followed by those of S. Hence, from the two input tables, only one row can be highlighted as currently active, and no input pinning is allowed because it is not useful in this context. Although the definitions above associate each output row with a particular input row, the I-Rex interface hides this detail, so users will still observe the SQL semantics. To be more specific, given an input row, we define its derivative "row" (if one exists) in the final output table as the group of all duplicate rows, which can only be pinned/unpinned together. The interface also provides an explanation for the number of duplicates or the absence of output rows for the given input row.

A.2 Join Expressions

I-Rex treats explicit `JOIN` expressions in `FROM` essentially as subqueries. Hence, each such expression gives rise to a separate debugging context with two input tables (base or derived). Inner joins can be handled in the same way as a more general `SELECT` block (Section 3.1). We focus on outer joins here. Let R and S denote the two input tables. We augment the input combo space by adding a special IID $\perp$ to each input table whose side of the output can be padded with `NULL`s. Intuitively, $\perp$ stands for "no row" from that input table, and behaves in the I-Rex interface as a special last row of the table. For example, for S in R `LEFT JOIN` S, we add $\perp$ to indicate "no row from S." If a row $r \in R$ joins with no row from S, the left outerjoin result should contain a row for r padded with `NULL`s in S columns; its IID would be $(r, \perp)$.

When ordering by IIDs, $\perp$ should be considered last in value. Internally, I-Rex represents $\perp$ in SQL using an IID whose components are all `NULL`s, assuming input rows do not have `NULL`s for all their IID components.[5] When generating queries involving comparison with IIDs possibly containing `NULL`s, however, we need to replace the `ROW(...)` comparisons with special code because SQL comparisons involving `NULL`s always yield the `UNKNOWN` truth value. For example, instead of generating `ROW(A,B)>=ROW(1,2)` when B can be `NULL`, we would generate

```
|| A>1 OR (A=1 AND (B IS NULL OR B>=2))
```

Selection of the active input combo and stepping work the same way as in `SELECT` (Section 3.1), except when the input combo involves a $\perp$, I-Rex show an explanation of why a `NULL`-padded result row is or is not produced, instead of the filter expression tree.

[5] In the extremely rare case when this assumption is violated, we choose an unused value from each domain instead.

B ADDITIONAL DETAILS ON DEBUGGING PARADIGM

B.1 Proof of Proposition 4.2

Proof. First note that when $\Theta \Leftrightarrow \perp$ (i.e., Θ is equivalent to logical false), both queries are obviously equivalent as they return empty results on all possible database instances regardless of the evaluation of `BLOOM_CHECK`. Therefore, we now proceed to prove that both queries are equivalent when at least Q_1 returns a non-empty result (i.e., Θ is not equivalent to $\perp$).

Now assuming Q_1 and Q_2 are not equivalent, then there must exist input tables $\{R'_i\}$ where Q_1, Q_2 return results $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively, and a tuple $e' \in \mathcal{V}_1$ but $e' \notin \mathcal{V}_2$ (it cannot happen vice versa as $\Theta \Leftarrow \Theta$ AND `BLOOM_CHECK`$(F, \langle \mathcal{A} \rangle)$), thus there must exist a subset $\mathcal{A}'$ from e' such that `BLOOM_CHECK`$(F, \langle e'[\mathcal{A}'] \rangle)$ is evaluated to false. Observing that F is obtained from `BLOOM_CHECK`$(\mathcal{V}_1[\mathcal{A}'])$ and $e' \in \mathcal{V}_1$, this implies `BLOOM_CHECK`$(F, \langle e'[\mathcal{A}'] \rangle)$ must be true as Bloom filter returns no false negative. Therefore, e' must also be in $\mathcal{V}_2$ and $\{R'_i\}$ does not exist. As a result, Q_1 and Q_2 must be equivalent. □

B.2 Pushdown through Outerjoins

Consider a page-fetch query that I-Rex generates for a full outer join debugging context, which has the form:

```
|| SELECT ROW(R.K, S.K) AS _iid
|| FROM R FULL JOIN S ON R.A = S.B
|| WHERE Θ(R, S);
```

Here, Θ is condition based on the IID range of the requested page, which constrains R and S. Efficient execution of this query requires pushing down filters inferred from Θ to R and S. Unfortunately, such pushdowns are not always safe through outer joins. For example, suppose that the requested page is the last one, and its IID range implies the filter on R to be `R.K IS NULL OR R.K>=100`. Further suppose that some $s \in S$ joins with a single row $r \in R$ with `R.K<100`. The above page-fetch query should not return any output for s, since $\langle r, s \rangle$ does not belong to the requested page and hence fails the final `WHERE`. However, if we push the filter on R to below the outer join, the outer join will return a row containing s and R columns padded with `NULL`s, which passes the final `WHERE`.

To avoid the above issue and still enable pushdown, I-Rex first computes the inner join between R and S with pushdown. Then, only if the number of returned rows falls below the desired number on the requested page, which can be determined from the milestone table and should happen rarely, we issue additional queries to find rows in the outer join result but not in the inner.

C ADDITIONAL EXPERIMENTAL RESULTS

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 5, Table 6 and Table 7 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 5: Experiment results for Page Size 50

		1GB		5GB		10GB	
Query	Page	Opt.	Base	Opt.	Base	Opt.	Base
Q1	head	0.092	0.157	0.095	0.106	0.258	0.089
	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
Q2	head	44.592	318.82	46.69	2580.597	45.526	5048.784
	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
Q3	head	2.298	8.315	14.022	50.756	23.466	154.061
	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
Q4	head	1.004	0.842	2.822	0.862	15.327	4.289
	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
Q5	head	18.294	16.98	35.305	22.441	367.671	230.179
	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
Q6	head	0.745	0.534	2.076	0.534	2.895	0.538
	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
Q7	head	41.808	28.7	61.38	52.018	946.975	55.532
	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
Q8	head	28.152	12.265	28.002	22.943	247.418	24.732
	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
Q9	head	2.227	389.589	25.745	6031.313	27.844	29081.397
	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
Q10	head	3.528	1.349	8.099	8.962	12.316	3.35
	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
Q11	head	52.714	5.543	296.591	12.83	615.864	21.86
	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
Q12	head	1.907	1.645	3.757	4.948	3.159	1.836
	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
Q13	head	0.379	0.257	0.37	0.642	0.82	0.656
	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
Q14	head	6.925	3.679	42.57	4.942	121.01	11.66
	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
Q15	head	0.662	0.51	2.037	0.479	4.343	0.478
	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
Q16	head	3.164	7.008	17.155	10.224	24.908	20.641
	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
Q17	head	5.152	3.786	12.614	3.626	13.25	3.637
	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
Q18	head	3963.758	8643.507	24134.374	74575.1	48786.827	169499.561
	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
Q19	head	80.62	95.444	275.504	516.311	425.97	836.663
	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
Q20	head	0.692	0.083	0.751	0.078	1.054	0.891
	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
Q21	head	41.894	830.955	85.693	20382.458	678.648	51923.664
	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
Q22	head	4.299	68.512	4.51	319.805	2.876	659.813
	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)
Q1	1	13029.523	15.579	2457.667	761.653
	5	66150.01	77.927	39046.702	3809.747
	10	132810.372	155.834	66195.529	7618.516
Q2	1	280.11	0.003	501.518	0.129
	5	1438.351	0.015	3650.785	0.647
	10	3197.387	0.031	7330.617	1.31
Q3	1	1402.237	0.108	2391.46	9.358
	5	7609.511	0.528	29926.468	45.735
	10	20829.922	1.072	119386.492	92.932
Q4	1	393.68	0.191	602.307	6.367
	5	1956.036	0.95	2871.491	31.649
	10	19492.825	1.895	37588.076	63.165
Q5	1	2794.965	0.468	3934.344	30.979
	5	13919.163	2.337	31427.944	154.836
	10	35787.956	4.662	88961.208	308.884
Q6	1	1077.996	0.898	1028.798	43.884
	5	5606.841	4.476	5449.391	218.804
	10	15937.32	8.962	83693.457	438.125
Q7	1	750.001	0.076	1388.376	0.947
	5	8949.756	0.369	11575.322	4.602
	10	18841.027	0.745	40373.876	9.306
Q8	1	6068.222	5.835	4796.095	65.639
	5	30360.543	29.027	23391.072	326.55
	10	61229.624	58.142	93859.353	654.091
Q9	1	2824.782	2.134	3857.867	23.998
	5	20769.045	10.495	25856.049	118.061
	10	41547.519	21.006	64925.319	236.312
Q10	1	2235.767	1.449	2775.5	77.861
	5	11587.553	7.269	14053.989	390.698
	10	24067.6	14.524	38715.702	780.631
Q11	1	137.414	0.121	125.681	5.928
	5	689.607	0.576	617.165	28.146
	10	1311.196	1.149	15862.307	56.193
Q12	1	1737.807	0.394	2334.203	10.507
	5	6617.319	1.974	22186.895	52.627
	10	14404.903	3.94	25798.262	105.057
Q13	1	5899.366	3.596	2138.399	74.4
	5	28939.75	17.98	12261.105	372.001
	10	58316.145	35.96	24704.791	744.001
Q14	1	380.802	0.27	476.924	18.571
	5	2001.128	1.347	7033.305	92.793
	10	3954.975	2.696	35885.574	185.705
Q15	1	1014.581	1.083	707.173	52.914
	5	5256.718	5.423	11050.796	265.099
	10	10870.044	10.848	40151.84	530.346
Q16	1	551.868	0.37	523.076	16.455
	5	2798.609	1.839	4488.929	81.74
	10	5636.501	3.689	6598.465	163.946
Q17	1	6134.681	0.72	6181.627	11.2
	5	27933.944	3.6	34723.602	56.0
	10	57508.86	7.2	64700.14	112.0
Q18	1	21986.933	26.108	16799.919	1272.737
	5	119339.262	130.43	131820.25	6358.46
	10	252651.048	260.7	273911.052	12709.096
Q19	1	162.155	0.022	404.626	1.501
	5	905.482	0.109	6894.219	7.47
	10	1505.552	0.211	60837.304	14.506
Q20	1	3110.616	3.289	2668.742	1056.214
	5	15414.311	16.443	45633.142	5279.964
	10	35581.996	32.845	91427.229	10557.545
Q21	1	866.568	0.026	1455.808	1.332
	5	13742.687	0.125	25023.161	6.538
	10	34559.892	0.256	95153.337	13.417
Q22	1	162.503	0.023	165.083	0.405
	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 6: Experiment results for Page Size 100

D ADDITIONAL DETAILS ON USER STUDY

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)

D.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.

D.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.

		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 7: Experiment results for Page Size 200