Debugging the OmniTable Way
Andrew Quinn, UC Santa Cruz; Jason Flinn, Meta; Michael Cafarella, MIT; Baris Kasikci, University of Michigan

https://www.usenix.org/conference/osdi22/presentation/quinn
Debugging the OmniTable Way

Andrew Quinn  
UC Santa Cruz

Jason Flinn  
Meta

Michael Cafarella  
MIT

Baris Kasikci  
University of Michigan

Abstract
Debugging is time-consuming, accounting for roughly 50% of a developer’s time. To identify the cause of a failure, a developer usually tracks the state of their program as it executes on a failing input. Unfortunately, most debugging tools make it difficult for a developer to specify the program state that they wish to observe and computationally expensive to observe execution state. Moreover, existing work to improve our debugging tools often restrict the state that a developer can track by either exposing incomplete execution state or requiring manual instrumentation.

In this paper, we propose an OmniTable, an abstraction that captures all execution state as a large queryable data table. We build a query model around an OmniTable that supports SQL to simplify debugging without restricting the state that a developer can observe: we find that OmniTable debugging queries are more succinct than equivalent logic specified using existing tools. An OmniTable decouples debugging logic from the original execution, which SteamDrill, our prototype, uses to reduce the performance overhead of debugging. The system employs lazy materialization: it uses deterministic record/replay to store the execution associated with each OmniTable and resolves queries by inspecting replay executions. It employs a novel multi-replay strategy that partitions query resolution across multiple replays and a parallel resolution strategy that simultaneously observes state at multiple points-in-time. We find that SteamDrill queries are an order-of-magnitude faster than existing debugging tools.

1 Introduction
Developers spend the majority of their time debugging their software [26]. Usually, a developer debugs by iteratively executing their program and using debugging tools to observe its state during the failing execution.

A developer can often identify the root cause of a simple bug by making a few observations about their program’s execution state. However, to identify the root cause of a complex bug, such as a atomicity violation or performance degradation, the developer will need to make sophisticated observations. Conceptually, we can model the logic for such sophisticated observations as a debugging program, designed to make sense of the failing program. For example, when debugging, a developer may observe all of the values to which a variable is assigned during an execution. Their debugging program consists of a set data structure to store the values, logic after each assignment in the failing execution that adds the assigned value to the set, and a print statement to print the set when the execution terminates.

Unfortunately, many debugging tools, such as gdb, “printf”, debugging, and binary instrumentation, support debugging programs that have both high programming complexity and high performance overhead. Such tools support procedural debugging programs that observe state as a failing program executes. Procedural debugging programs have considerable programming complexity, especially for sophisticated tasks that track execution state over time (§6.2). High complexity can lead to bugs [15, 44] that prevent a developer from understanding the failing program. Additionally, such debugging programs impose high performance overhead since sophisticated debugging programs observe a lot of execution state which existing tools extract within the same execution context as the failing program. Consequently, procedural debugging programs can slow execution by between a factor of 2–1000 (§6.3), which can preclude the use of sophisticated debugging programs [11].

Alas, prior debugging work retains, or even exacerbates, high programming complexity or high performance overhead to improve the other. Some proposals lower the performance overhead of debugging by employing parallelism (e.g., Speck [30], SledgeHammer [35]) or low-level optimizations (e.g., optimistic hybrid analysis [9], efficient path profiling [3]). At best, such techniques require redesigning debugging programs, at worst, they require novel research contributions to accelerate even a single task (e.g., taint tracking [4]).

High-level debugging tools decrease programming complexity by allowing a developer to observe and summarize execution state using a high-level programming model (e.g.,
This paper proposes the OmniTable query model, a new debugging paradigm that reduces the programming complexity and performance overhead of debugging without restricting the execution state that a developer can observe. The new OmniTable abstraction empowers the model. An OmniTable reduces programming complexity by presenting all of an execution's state as a large queryable data object. An OmniTable reduces performance overhead by decoupling a debugging program's observations from the original programs' execution to enable automated optimizations of debugging programs.

The OmniTable query model enables debugging programs that can observe any execution state with low programming complexity by turning to relational logic. Concretely, an OmniTable is a database table representation of an execution that contains all architectural state (i.e., the value of all bytes of memory and all registers) before every instruction executed by the program. From a developer's perspective, an OmniTable is extracted as a program executes and can later be queried using an extended SQL language to observe the execution's state. The model bridges the gap between the architectural state in an OmniTable and common debugging abstractions (e.g., the functions executed, variables assigned, etc.) by re-purposing existing database primitives (e.g., high-level views) and creating new query operators (e.g., traversal functions).

Unfortunately, naively materializing an OmniTable would lead to considerable performance overhead, since it would require performing a core-dump before every instruction. Instead, our prototype, SteamDrill, employs lazy materialization. Lazy materialization defers the calculation of an OmniTable's state until a developer queries it. Rather than extract an OmniTable in its entirety during execution, SteamDrill uses deterministic record and replay to store the execution associated with the OmniTable. Deterministic record and replay enables SteamDrill to compress and store years of OmniTables on a commodity hard drive [10]. When a developer issues a query over an OmniTable, SteamDrill generates instrumentation which it injects into a new replay of the execution associated with the OmniTable to produce the execution state needed for the query.

SteamDrill reduces performance overhead by decoupling a debugging query's execution from the original program execution. SteamDrill uses a query planning approach that decomposes a debugging query into independent stages. SteamDrill implements a novel multi-replay query resolution strategy that executes each stage in a separate replay so that it can use data that is computationally inexpensive to observe (e.g., data about functions in an OmniTable) to reduce the compute cost of observing data that is computationally expensive to observe (e.g., data about each instruction in an OmniTable). In essence, multi-replay resolution uses the decoupling between an OmniTable query and the original execution to repeatedly observe OmniTable state at increasing detail. SteamDrill also uses decoupling to observe execution state from multiple points-in-time in parallel using thousands of machines [35, 47].

We built a SteamDrill prototype on top of Spark [47] and Arnold [10]. We evaluate the prototype using 5 detailed case studies of bugs reported in popular open-source applications (Memcached, redis, Apache, and SQLite). We identified 14 debugging programs that a developer would use to identify the root cause of each bug, including ad-hoc programs (e.g., “How many control-flow instructions did my function issue?”) and standard dynamic analyses (e.g., a memory leak detector). We implemented the debugging programs using OmniTable queries and gdb’s python bindings, which provide a high-level language over gdb features. We found that OmniTable queries require up to 11.67 times fewer lines (with a geometric mean of 3.74 times fewer lines), up to 5.73 times fewer terms (with a geometric mean of 1.70 times fewer terms), and up to 23.49 times less estimated development time (with a geometric mean of 2.75 times less estimated development time) than gdb scripts. We evaluated the performance of SteamDrill on 3 representative debugging queries and find that it is faster than gdb by a factor of 99 based upon geometric mean.

We make the following contributions:

- The OmniTable query model, which decouples a debugging program from a failing execution to reduce the programming complexity and performance overhead of debugging.
- SteamDrill, which optimizes OmniTable queries using query planning, cluster-scale parallelization, and a novel multi-replay query resolution approach.
- An evaluation of 5 case studies and 14 queries that shows that OmniTable queries are more succinct than gdb and SteamDrill has lower latency than state-of-the-art tools.

## 2 Motivation

In this section, we describe a motivational case study showing how the OmniTable query model simplifies debugging. In the case study, a developer uses an OmniTable to diagnose a performance problem in redis [36]. In the study, a developer deploys redis as an in-memory key-value LRU cache for a
slow back-end service. Over time, the average end-to-end latency of their deployment creeps upwards; the developer notices that the increase correlates with the back-end service processing a higher percentage of requests.

The bug is challenging to diagnose since the developer only starts with a high-level symptom and is unaware of which parts of the program are related to the error. To determine the root cause of the bug, the developer summarizes an execution’s behavior over time. The OmniTable allows the developer to observe all of the execution state of the program without requiring instrumentation; the query model’s support for SQL aggregations allows the developer to succinctly summarize large amounts of execution state. Moreover, the OmniTable enables repeated queries over the same buggy execution, instead of requiring the bug be reproduced for each query.

In contrast, summarizing execution state over time is challenging with existing tools (§7). To use a procedural debugging tool (e.g., gdb), the developer must identify numerous instrumentation points, track execution state over time in complex data-structures, and implement algorithms to group data and calculate statistics. Other debugging tools simplify execution summarization, but provide incomplete interfaces in the case study. The enterTime, callStack, and args columns are extracted upon function entry; the exitTime and rVal columns are extracted upon function exit (and are NULL for functions that never return); and the name and thread are extracted at both entry and exit and joined to match the entry and exit of each function. The rVal and args columns use the polymorphic type, Any, to encode different function signatures. For example, a developer specifies args["i"] to get the value of the argument i passed to a function.

The developer uses two derived views, Vars(ot) and Funcs(ot), which can be calculated over an OmniTable: ot. Figure 1 shows their schemas.

**Vars(ot)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>Long</td>
<td>Time of instruction</td>
</tr>
<tr>
<td>thread</td>
<td>Long</td>
<td>Thread that executed instruction</td>
</tr>
<tr>
<td>eip</td>
<td>Long</td>
<td>Program counter of instruction</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Variable name</td>
</tr>
<tr>
<td>value</td>
<td>Any</td>
<td>Assigned value</td>
</tr>
</tbody>
</table>

**Funcs(ot)**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enterTime</td>
<td>Long</td>
<td>Time of function entry</td>
</tr>
<tr>
<td>exitTime</td>
<td>Long</td>
<td>Time of function exit (or null)</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Function name</td>
</tr>
<tr>
<td>thread</td>
<td>Long</td>
<td>Thread that executed function</td>
</tr>
<tr>
<td>callStack</td>
<td>String</td>
<td>Call stack of function</td>
</tr>
<tr>
<td>args</td>
<td>Map[String-&gt;Any]</td>
<td>Argument values</td>
</tr>
<tr>
<td>rVal</td>
<td>Any</td>
<td>Return value of function execution</td>
</tr>
</tbody>
</table>

Figure 1: The schema of the Funcs(ot) and Vars(ot) views. Each line in each table describes a column in the view.

The time, enterTime and exitTime columns from Vars(ot) and Funcs(ot) expose an ordering of events contained in the views and provide a primary key that uniquely identifies each row in the views. Moreover, a developer can use the time, enterTime, and exitTime columns to correlate data across the Funcs(ot) and Vars(ot) views for the OmniTable. For example, a developer can determine the value of each in-scope variable at the entry to each function by joining Funcs(ot) and Vars(ot) on enterTime = time; the second query uses this feature (§2.2.2).
2.2 Queries

Next, we describe how the developer diagnoses the cause of the performance bug. First, they use deterministic record and replay to capture the OmniTable for an execution of redis during which the issue occurs. Then, they construct and execute the following five OmniTable queries.

2.2.1 First Query

The developer’s first query (Listing 1) uses a windowed aggregation to approximate the number of items that the deployment caches in redis (i.e., the working set size) during the performance degradation. The developer suspects that the working set size increases over time, which would lead to additional cache misses in redis. Each cache miss sends a request to the back-end service, so this hypothesis would explain the creeping latency of the deployment.

The developer begins by inspecting redis’s source code to identify the function, `lookupKey`, that finds an item in the cache. For each `lookupKey` execution, the developer creates a window containing the preceding 10,000 executions of `lookupKey` and counts the number of distinct keys passed to each function call in each window. The OmniTable query model succinctly represents this logic using SQL aggregates. SQL aggregates calculate a mathematical operation (e.g., `count`, `sum`) over a group of rows. An aggregate can operate over a window of requests, in which each group is an ordered list of rows that match an `Over` clause, as is the case in this query. Alternatively, an aggregate can operate over a group of rows that match a `Group By` clause, as is the case in the developer’s third query (§2.2.3).

In detail, the query uses the `Over` operator to create sliding windows, each of which contains 10,000 consecutive calls to `lookupKey`, by ordering `Funcs(ot)` by `enterTime` (Lines 2–3). The query filters non-lookupKey windows (Line 5). It counts the number of distinct keys passed to each function call in each window using the key argument (`args["key"]`) and the `count` and `distinct` operators.

Listing 1: The developer’s first query.

```
1     Select enterTime, count(distinct args["key"]) Over
2     Order By enterTime
3     Rows Between 10000 Preceding and Current Row
4     From Funcs(ot)
5     Where f.name="lookupKey"
```

Listing 2: The developer’s first query written for gdb’s Python bindings.

```
1     from gdb import Breakpoint, parse_and_eval
2     from collections import deque, defaultdict
3     class bp(Breakpoint):
4     indexed=defaultdict(int)
5     def stop(self):
6         keys.append(parse_and_eval("key")
7         indexed[keys[-1]] += 1
8         if len(keys) > 10000:
9             indexed[keys[0]] -= 1
10        if indexed[keys[0]] == 0:
11            del indexed[keys[0]]
12            keys.popleft()
13        print (len(indexed))
14        return False
15     bp("lookupKey")
```

Existing debugging tools either cannot support the query, impose high programming complexity, or impose high performance overhead. EndoScope [7] and Fay [12] could support the developer’s query, but imposes a high overhead since they tightly couple debugging logic’s execution with the original program execution. Most high-level debugging tools do not support windowed-aggregations and are either unable to compute the query (e.g., Pivot Tracing [24], Execution Mining [20]) or require a developer to write a custom operator to compute the query (e.g., G2 [17] requires expressing the window clause in terms of a vertex-based graph-traversal). Instrumentation-based debugging tools (e.g., Pivot Tracing [24]) would require the developer manually instrument the `lookupKey` function to produce the value of `key`.

An equivalent procedural debugging program is complex. The debugging program must navigate the performance-complexity tradeoff—creating a program with high overhead is straightforward, but creating one with low overhead requires complex logic to ensure consistency of two data structures. A mistake can lead to a misdiagnosis of the bug—our first version of the debugging program included such a mistake.

Listing 2 shows an implementation for gdb’s Python bindings, which provide a Python interface for `gdb` features such as breakpoints and backtraces. The developer creates a custom `Breakpoint` class, `bp` (Lines 6–15); by creating a `bp` with the argument `"lookupKey"` (Line 16), the developer instructs the gdb framework to call the developer-supplied `stop` function at each call to `lookupKey`. The developer tracks the sliding window of 10,000 requests by storing the value of the key argument into `keys`, a queue, and removing the first element if there are more than 10,000 elements in `keys` (Lines 7, 9, and 13). The developer could recompute the unique values in `keys` in `stop`, but that would add significant performance overhead since `lookupKey` is executed frequently. Instead, the developer uses a dictionary object, `indexed`, to track the number of times each `key` value appears in the `keys` window (Lines 5, 8, and 10–12). This logic is subtle and challenging to get right—for example, we initially used a `set` to track the unique elements in `keys` instead of using a dictionary to track the number of times each element appears in `keys`.

Our buggy implementation erroneously removes elements from `indexed` and produces misleading results.
The second query's output shows that the number of items in the cache implies that there is a memory leak. Since the deployment uses constant sized items, a declining number of items in the cache decreases throughout the execution. The developer first creates a view, DefinedMemory(ot) that contains the window of time during which each memory object is defined, i.e., allocated and not freed (Listing 4). The view joins each call to malloc with the subsequent call to free whose pointer argument, ptr, is equal to the return value from malloc (Lines 5–8). Since a pointer could be reallocated by malloc after being freed, the query only matches calls to free that occur after the call to malloc (m.exitTime < f.enterTime at Line 7). Additionally, it only matches each malloc with the next matching call to free, as ordered by exitTime and enterTime, respectively, by using NextJoin, a new operator provided by the OmniTable query model (Lines 5–6). The developer uses Left NextJoin, which produces output from the left relation even if there is no matching row in the right relation, so that memory which is never freed (i.e., leaked) has a NULL value for the end column.

The third query tracks the amount of data leaked by each allocation site (defined as the call stack of the allocation) over time. Allocation sites that produce bug-inducing leaks will have a gradual increase of leaked bytes throughout the execution. The developer’s query observes three separate types of execution events with different happens-before relationships, which is greatly simplified by the OmniTable query model.

2.2.4 Fourth Query

The output of the third query identifies a single leaking allocation site, leakSite. redis uses reference counters to manage allocations from leakSite. Each counter tracks the number of live references to each object; redis should garbage collect the object when the count reaches 0. So, the developer suspects a problem in the reference counting and writes a query to count the updates to the reference counters of leak-
ing objects (Listing 5). They identify leaked objects that were allocated at leakSite (Lines 2–3) and match each leaked object with corresponding executions of decreRefCount and increRefCount, the functions that modify reference counts (Lines 2, 4, and 5). The developer groups the rows by object and function name (Line 6) and determines the number of calls to increment and decrement the counter (Line 1).

Like the previous queries, existing debugging tools either cannot support the developer’s fourth query, impose high programming complexity, or impose high performance overhead.

2.2.5 Fifth Query

The fourth query’s output shows that the execution calls increRefCount and decreRefCount the same number of times on the leaked objects, indicating a problem in the implementation of increRefCount or decreRefCount. The developer chooses a few candidate objects and determines the call stack of the calls to increRefCount and decreRefCount for these objects. The final query\(^1\) shows that the leaked object’s reference counts are decremented by a lazy deallocation thread and by a logging thread and points to the root cause of the bug, a race condition in decreRefCount. In the fix for the original bug report, the developer redesigning the logging thread to copy objects instead of sharing them.

3 The OmniTable Query Model

We outline the features of the OmniTable query model that enable a developer to succinctly reason about the entire history of execution state. The central abstraction is an OmniTable, a database table containing all user-level architectural state of an execution immediately before every instruction in the execution. Concretely, an OmniTable contains a column for every byte of architectural state and a row immediately before each instruction. The model supports debugging queries over an OmniTable expressed using SQL-style `Select ... From ... Where` queries.

Alas, an OmniTable alone offers an inadequate debugging interface, since a developer would need to reference execution state in architectural terms. For example, a developer would need to determine the exact memory location of each variable whose value they wish to observe. So, the OmniTable model adopts and extends database concepts to enable debugging abstractions. It uses Generators, user-defined-table-functions that allow queries to reference non-execution state (e.g., debugging symbols). It adds new operators for debugging, such as traversal functions and new `Join` variants. Finally, the model uses derived views to label an OmniTable’s state according to familiar debugging abstractions such as the functions executed in an OmniTable or the variables in scope at each instruction in an OmniTable. A single row in a high-level view can expose execution state from multiple points-in-time during the execution (e.g., `Funcs(ot)`). Below, we elaborate on the model’s components.

3.1 Relations

The OmniTable query model supports two relational base tables, OmniTables and Generators. It supports columns with primitive types (e.g., `Long`, `String`), `Structs`, `Maps`, `Arrays`, and `Any`, a polymorphic type.

OmniTable. An OmniTable is a database table that includes all architectural execution state immediately before each instruction in the execution; Figure 4 shows an example. Before each instruction, the OmniTable contains the current thread, the value of all registers and memory addresses, and the top of the stack of the thread. To dereference an address, `addr`, a query specifies `Memory[addr]`. Additionally, each row includes a monotonically increasing logical time, which provides a total ordering of events in the OmniTable and uniquely identifies each row. In a multi-threaded program, the time field is a total ordering that is consistent with the partial ordering of the original execution. Together, the `thread` and `time` columns enable a developer to reason about concurrency.

Generators. Generators allow developers to bridge the semantic gap between traditional programming abstractions (e.g., functions, lines of code) and an OmniTable’s architectural state by referencing non-execution state (e.g., debugging symbols). For example, `Defs` identifies the functions defined in a binary; the following produces all such definitions for an executable, “a.out”: `Select * FromDefs("a.out")`. Generator input can depend on query data. For example, `Binaries` is useful for bootstrapping queries; it uses the deterministic record/replay log to identify the binaries mapped into the address space of an OmniTable. The following determines all functions defined in all binaries that are loaded in `ot`, an OmniTable, which we use to define the `FuncDefs` view: `Select * FromDefs(Select * From Binaries(ot))`. Developers create Generators by writing a program that produces relational output; we have built a Generator that determines all variables defined in all binaries mapped into an OmniTable, and one that creates stored procedures that produce the memory read and written by each instruction in an OmniTable.

3.2 Relational Operators

The model supports `join`, `group by`, `order by`, and `pivot`. It also introduces three `Join` variants for debugging.

StackJoin. SQL is unable to model a function stack, which would prevent the OmniTable query model from expressing critical debugging abstractions, such as `Funcs(ot)`, which is used in all of the queries in the redis case study (§2). Prior high-level debugging tools either remove support for such se-
functions, stored procedures, and standard aggregations (e.g., Count, Max, Min, etc.) over groups and windows. The model also supports pointer dereferences by converting them into expressions over the Memory column (e.g., a->b becomes Memory[a].b). We elaborate on traversal functions and stored procedures.

**Traversal Functions.** SQL makes it difficult to traverse the elements in a data structure since it does not support unbounded traversals. So, the OmniTable query model builds new primitives for these operations. Given a pointer-typed column and a field within the pointed-to type, the traverse (column, field) expression produces a row of output for each element in the transitive closure of the structure by starting at column and following field pointers until the value is NULL. For example, traverse(node, "next") traverses the next pointer of all elements in a structure, starting at node.

**Stored Procedures.** Debugging logic often varies by execution context (e.g., the memory location of function arguments varies by function). Stored procedures [43] store relational logic in a table and allow a query to decide query logic during query resolution. Developers can call stored procedures in their OmniTable queries with function syntax; for example, a developer could specify Var_Loc(esp) to use Var_Loc, a stored procedure that calculates the memory location of a variable given the value of the stack pointer.

### 3.4 Derived Views

The OmniTable query model allows developers to construct derived views for labeling execution state. The Define operator in Listing 4 shows how a developer constructs DefinedMemory(ot). Our implementation provides three high-level views, Funcs(ot) (§2.1), Vars(ot) (§2.1), and Insts(ot), a view that encodes information about each instruction in an OmniTable.

### 4 Design

In this section, we describe the design of SteamDrill, our system that supports the OmniTable query model. From a developer’s perspective, SteamDrill computes queries over OmniTables that are extracted during execution and stored in a database. However, materializing an entire OmniTable is infeasible due to high storage and compute costs: an OmniTable’s size is equal to the addressable memory size.

---

**Table 4:** An OmniTable for a short execution.

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>thread</td>
<td>stackTop</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>0x2000</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>0x2000</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>0x2000</td>
</tr>
</tbody>
</table>

**Listing 6:** An Example StackJoin.

```
Select *
From fenter(ot) as t StackJoin freturn(ot) as r
On e.time, r.time, e.thread=r.thread AND e.name=r.name
```

---

3.3 Column Operators

The OmniTable query model supports many column operators, including arithmetic and conditional operators, field expressions (a.b), subscript expressions (a[b]), traversal

---

The USENIX Association 16th USENIX Symposium on Operating Systems Design and Implementation 363
which its pointer was defined (Listing 4). The query uses the
materialization of each
during execution, SteamDrill uses these operators to limit the
execution state that the query needs from that
referenced in the query. The relational operators
relational operators (internal nodes) over data tables (leaves)
identify the instructions that operate on undefined memory.

During execution, SteamDrill uses these operators to limit the
execution state that the query needs from that
referenced in the query. The relational operators
relational operators (internal nodes) over data tables (leaves)
identify the instructions that operate on undefined memory.

SteamDrill introduces lazy materialization as a solution.
Rather than materializing an OmniTable during execution,
SteamDrill uses deterministic record and replay [10] to cap-
ture a log of non-deterministic inputs to the execution. The
system uses the log to generate OmniTable state on-demand
by instrumenting and re-executing the original execution as
necessary to resolve debugging queries. Delaying OmniTable
materialization allows SteamDrill to filter OmniTable data
before extracting state instead of afterwards.

In the rest of this section we describe how SteamDrill
resolves debugging queries. Listing 7 presents a simplified
use-after-free query as a running example. The query uses the
Insts(ot) view to identify the memory read and written by
each instruction in an OmniTable. It joins Insts(ot) with the
DefinedMemory(ot) view from the third redis query, to
match each memory access with the region of time during
which its pointer was defined (Listing 4). The query uses a
Left Join and identifies rows where start is NULL to
identify the instructions that operate on undefined memory.

SteamDrill’s design mirrors that of typical database man-
agement systems [2] (Figure 5). First, SteamDrill uses
conventional SQL parsing to decompose a query into a tree of
relational operators (internal nodes) over data tables (leaves)
(§4.1). The tree contains a separate leaf node for each
OmniTable referenced in the query. The relational operators
that consume data from each OmniTable in the tree identify
the execution state that the query needs from that OmniTable.
During execution, SteamDrill uses these operators to limit the
materialization of each OmniTable in the query by generating
instrumentation that it injects into replay executions (§4.3).

The order of OmniTable materialization has a large impact
on the amount of materialized data and the latency of
query resolution. Delaying the materialization of an other-
wise computationally expensive-to-materialize OmniTable
often allows SteamDrill to filter the computationally expen-
sive materialization using data from a less computationally
expensive-to-materialize OmniTable. For example, to reduce
the latency of the use-after-free query, the system first mate-
rializes the OmniTable state needed for the DefinedMemory
view and uses the materialized data to filter the materialization
of the Insts(ot) view.

Accordingly, SteamDrill uses multi-replay resolution—it
splits OmniTable materialization across multiple replay exec-
utions. It uses query planning to determine the OmniTable
materialization order and assigns OmniTable materialization
to replay executions (§4.2). SteamDrill implements
OmniTable-specific optimization strategies to decide the join
order and join algorithm for each join in the tree (§4.2.1)
which it uses to assign each operator to a stage. The system
uses a single replay execution to materializes all OmniTables
in the same stage. This approach minimizes the number of
replay executions (since each replay execution adds addi-
tional overhead) and enables SteamDrill to limit OmniTable
materialization.

### 4.1 Parsing

First, SteamDrill converts the query into a tree of relational
operators over data tables using a standard SQL parser [2];
the tree encodes the logic required to resolve the query in
terms of easy-to-optimize relational operations. The tree en-
codes each relational operator in the query as an internal node
(i.e., a projection (Π), selection (σ), or join operator) and each
OmniTable and each Generator in the query as a separate

```
1 Select eip, read, write
2 From Insts(ot) as i Left Join DefinedMemory(ot) as dm
3  On [i.read=dm.pointer Or i.write=dm.pointer]
4  And i.time>=dm.start And i.time<=dm.end
5 Where dm.start=NULL
```

Listing 7: A simplified undefined use query.
leaf node. SteamDrill recursively decomposes each view into the relational logic that generates them until the tree is comprised entirely of relational logic and base tables (§3.1). A directed edge from node n₁ to node n₂ in the tree identifies that the operator n₂ consumes the output of n₁.

Figure 6 is the relational tree produced by SteamDrill for Listing 7. SteamDrill contains the internal logic of the Insts(ot) and DefinedMemory(ot) views (shown as dotted rectangles). The tree contains the logic of the Insts(ot) view: a Join between an OmniTable and InstructionDefs, a Generator containing metadata about the instructions defined in binaries used by an OmniTable.

The tree includes the internal logic of DefinedMemory(ot) and, recursively, all of the derived views comprising DefinedMemory(ot). The tree contains the DefinedMemory(ot) logic (Listing 4): a Join between two Funcs(ot) views, one for executions of malloc (Funcs(ot) as m) and one for executions of free (Funcs(ot) as f). The tree contains the logic of each Funcs(ot) view (§3.2 and Listing 6): a StackJoin that combines fentry(ot) and fexit(ot), relations over the entry and exit to each function in the OmniTable. Finally, the tree contains the logic for each fentry(ot) and fexit(ot): a Join between an OmniTable and FuncDefs. Note, the tree includes the fentry(ot) of malloc and fexit(ot) of free even though the query does not use their output; during planning, SteamDrill determines that the query does not use the views and prunes them.

4.2 Planning

SteamDrill performs two tasks during planning. During logical planning, the system optimizes the relational tree using standard optimizations (e.g., predicate push-down) and determines the join order and join algorithm for each join in the tree using OmniTable-specific strategies (§4.2.1). The most crucial task in logical planning is determining the join order and algorithms for the query, since the join order and algorithms imply the partial order in which SteamDrill will materialize the OmniTable nodes contained in the query. SteamDrill supports two join algorithms: merge joins, which operate over two fully realized relations, and block-nested-loop joins (loop joins), which first calculate the left relation and use the left relation’s output to limit right relation materialization.

During physical planning, SteamDrill produces a staged execution plan, which uses the join order and algorithms assigned during logical planning to assign each operator in the tree to a stage. In particular, physical planning assigns the children of merge joins to the same stage (so SteamDrill materializes them using the same replay) and assigns the right child of a loop join to the stage after the loop join’s left child (so SteamDrill materializes them using different replays).
observe only the exit from malloc, entry to free, and load/store instructions. Moreover, the plan reduces latency by only producing data for load/stores to undefined memory as they are observed, rather than producing data for all loads/stores and performing a join to determine undefined uses afterwards.

First, SteamDrill uses traditional database optimizations (e.g., operator push-down) to push operators towards leaf nodes to (1) produce FuncDefs data for only malloc and free ($\sigma_1$ and $\sigma_2$), (2) produce InstructionDefs data only for loads and stores ($\sigma_3$) and (3) eliminate the fentry(ot) for malloc and fexit(ot) for free. The system uses loop joins for Loop Join$_1$ and Loop Join$_2$, which materialize $\sigma_1$ and $\sigma_2$ before OT$_1$ and OT$_1$ to limit OmniTable state to the exit of malloc and entry to free in OT$_1$ and OT$_2$, respectively. The system joins them using a Merge Join (Merge Join$_1$) to limit the number of replay executions that it uses. OT$_3$, created for the Insts(ot) view, is computationally expensive to materialize, so SteamDrill defers its materialization. SteamDrill uses a Merge Join (Merge Join$_2$) to join $\sigma_3$ and Merge Join$_1$, which requires a Cartesian product and violates the traditional rule that such approaches be avoided. Materializing Merge Join$_2$ and using a loop join (Loop Join$_3$) to join it with OT$_3$ allows SteamDrill to identify only the loads/stores to undefined memory (i.e., loads/stores that read/write an address at a time when it is not contained in DefinedMemory(ot)) as they are performed by the execution rather than in an expensive join afterwards. In some queries, using a loop join like Loop Join$_1$ enables SteamDrill to elide inspection of some instructions altogether (e.g., Listing 8).

### 4.2.2 Physical Planning

Next, SteamDrill converts the optimized relational tree into a staged execution plan by assigning each operator from the tree into a stage. Each stage corresponds to a new replay execution (§4.3). SteamDrill assigns operators to stages that follow the partial order of OmniTable materialization that is implied by the join order and algorithm, but uses as few stages as possible, since each stage will require the additional latency and overhead of a replay execution.

SteamDrill performs a depth-first traversal of the tree starting at the root node and maintains an integer id for the current stage, starting at 1. The system assigns leaf nodes (OmniTable, Generators) to the current stage and unary nodes (i.e., all non-join operators) to their child’s stage. The system assigns merge join operators to the largest stage among the join’s children. For loop join operators, SteamDrill first assigns stages to operators in the left (inexpensive) child, adds one to the current stage, assigns the loop join to the new stage and traverses the right (expensive) child.

Figure 8 shows the staged execution plan for Listing 7. SteamDrill assigns $\sigma_1$, $\sigma_2$, $\sigma_3$, $\Pi_1$, and $\Pi_3$ to the first stage. It assigns OT$_1$ and OT$_2$ to the second stage since Loop Join$_1$ and Loop Join$_2$ indicate that OT$_1$ and OT$_2$ should be materialized after $\sigma_1$ and $\sigma_2$, respectively. SteamDrill also assigns $\Pi_2$, Merge Join$_1$, and Merge Join$_2$ to the second stage since they inherit the largest stage of their children. The system assigns Loop Join$_3$, OT$_3$, and $\Pi_3$ to the third stage to follow the order required for Loop Join$_3$.

### 4.3 Execution

Finally, SteamDrill executes the staged execution plan. For each stage, the system generates instrumentation to materialize the state needed from each OmniTable, materializes each OmniTable, and calculates each operator in the stage.

#### 4.3.1 Instrumentation Generation

SteamDrill generates instrumentation that it will inject into a replay execution for the OmniTables in a stage by determining instrumentation operators for each OmniTable node in the stage. For each OmniTable node, the system gathers all stateless operators (e.g., projections ($\Pi$) and selections($\sigma$)) that only consume data from (1) the OmniTable node, (2) nodes resolved in previous stages, or (3) other nodes satisfying (1) and (2). For example, the instrumentation operators for OT$_1$ in Figure 8 includes $\Pi_1$ and Loop Join$_1$. Selecting stateless operations ensures that the resulting instrumentation will be parallelizable during materialization.

Then, SteamDrill creates a cursor object for each OmniTable node that combines all of the node’s instrumentation operations. Cursor objects contain a filter and an output clause; logically, a cursor inspects the execution instruction-by-instruction, producing the output whenever the filter is true. SteamDrill generates the filter clause of the cursor for each OmniTable in the stage by combining all selection ($\sigma$) and loop join instrumentation operators and generates the output clause using the output of the top-most projection ($\Pi$) instrumentation operator.
4.3.2 Materialization

Next, SteamDrill materializes OmniTable nodes by executing the cursor objects on top of a replay of the execution associated with the tables. It uses epoch parallelism [34, 35] to parallelize cursor evaluation. Epoch parallelism partitions a replay execution into time slices, called epochs. It assigns each epoch to a separate core in a compute cluster and uses checkpoints, generated during recording, so that each core executes each cursor over only its assigned epoch.

However, naive cursor evaluation (i.e., instrumenting every instruction) imposes a many orders of magnitude slowdown. So, SteamDrill analyzes the filter clause of each cursor to identify instructions at which the system can elide cursor evaluation to optimize performance. For example, SteamDrill identifies that the cursors in the second stage of Figure 8 only need to be evaluated at malloc and free and removes all other cursor evaluations. Our prototype identifies these optimizations by finding comparisons to the program counter.

Additionally, SteamDrill calculates operators in the stage that were not assigned as instrumentation operators for any OmniTable node (e.g., Merge Join₁ and Merge Join₄ in Figure 8). SteamDrill uses existing algorithms to calculate merge join and aggregation operators [1, 16]. Additionally, it executes the program associated with each Generator in the stage to calculate Generator operators.

5 Implementation

We implement our SteamDrill prototype on top of Spark [1] and Arnold [10]; below, we describe its key components.

Spark SQL. Our prototype introduces new relational operators and base tables for OmniTables and Generators. We added support for block-nested-loop joins, stored procedures, and polymorphic columns (§3) by serializing data to and from a JSON format. We added catalyst rules for our OmniTable-specific join order and algorithm preferences (§4.2.1). Each rule required 25 lines of code, so we expect that developers will be able to easily add custom rules as needed for their debugging workflows.

Instrumentation. Efficient cursor instrumentation plays a vital role in our prototype’s performance. Debugging tools often use dynamic instrumentation frameworks (e.g., PIN [23]), which are a scalability bottleneck when SteamDrill parallelizes the replay execution across many cores [34]. Our prototype performs static binary instrumentation. It disassembles the application binaries and rewrites the basic blocks contained in the application to call cursors, as required for the breakpoints determined from each cursor. The system single-steps execution for cursors that do not produce breakpoints.

Time Column. The time column is a critical element of the OmniTable query model, but, deriving the column by counting all instructions or basic blocks would be too expensive. We observe that instructions progress from low to high, except in the case of a backwards control-flow (e.g., branch, call, or return instructions that jump to a program location at a lower address). Thus, our prototype uses the number of backwards control-flow operations as the first element of the time column and breaks ties using the instruction pointer. Serendipitously, Intel provides deterministic performance counters for conditional branch and call instructions2, which allow our prototype to compute the number of backwards control-flow operations by counting the number of unconditional backwards branches during execution and adding the value of these performance counters.

6 Evaluation

In this section, we evaluate the OmniTable query model and SteamDrill by answering the following questions: “Does the OmniTable query model improve upon existing debugging interfaces?”, “Does SteamDrill accelerate debugging questions?”, and “How do SteamDrill design decisions impact query performance?”.

We perform 5 detailed case studies of how a developer could use an OmniTable and SteamDrill to solve real-world bugs from open-source servers (§6.1) from which we derive 14 debugging questions. We implement the debugging questions using OmniTable queries and gdb’s python bindings, which provide a python interface for traditional gdb features (e.g., breakpoints and backtraces). We compare the complexity of the 14 OmniTable queries and gdb scripts using metrics from the software engineering community (§6.2). We deploy SteamDrill on a CloudLab [37] cluster of 8 r320 machines (8-core Xeon E5-2450 2.1 GHz processor, 16 GB Ram, 10 Gbps NIC) to evaluate the performance for 3 representatives from the original 14 debugging questions (§6.3). We calculate the latency results below as the average over 10 trials and include 95% confidence intervals.

6.1 Case Studies

We performed 5 detailed case studies by identifying the debugging questions that a developer would ask when solving real-world bugs. We choose notoriously difficult bugs including livelock, intermittent performance problems, and atomicity violations (on average, the bugs in our study took 159 days from being opened to the commit that fixed the bug). We choose case studies from popular open-source applications: redis, Memcached, Apache, and Sqlite. The redis 4323 case study is described in §2; below, we describe case studies for debugging a livelock [28] and atomicity violation [27] in Memcached. We omit a description of a performance degradation in Apache [6] and a segmentation fault in Sqlite [41].

The case studies illustrate the benefits of the OmniTable query model along two key dimensions: first, the all-inclusive

2Note that most performance counters are not deterministic
We implemented the 14 debugging queries from our 5 case studies using OmniTable queries (OT) and gdb python scripts (gdb).

The OmniTable model, particularly SQL aggregations, provide a powerful tool for comparing the state of their program at many points-in-time to identify anomalous program state. The developer first isolates the module that contains the error. In particular, they determine if the bug arises when initially parsing requests or when processing them by using a count aggregate to count the number of times the function at the boundary between parsing and processing is called with each possible set of arguments. The query shows that the problem arises when processing requests.

The processing code maintains a boolean variable, valid, that tracks the validity of a global pointer used by the code. The developer’s second query, shown in Listing 9, identifies how often valid is set to true and false during each of the instruction within the processing logic. It uses a Pivot operator to produce a row for each instruction and show the number of times valid is set to true and false across all executions of the instruction. The second query identifies a few instructions at which status has an anomalous state. The anomalous instructions do not modify the status, so the developer concludes that another thread must modify the status and identifies a mistake in the processing logic’s use of a mutex.

6.2 Complexity

We implemented the 14 debugging queries from our 5 case studies using OmniTable queries and implemented equivalent logic using gdb python scripts. Qualitatively, we observe that OmniTable queries are less complex due SQL aggregations, the all-inclusive nature of an OmniTable, and the structured approach provided by high-level views: OmniTable queries usually involve an aggregation after joining a few high level views, whereas imperative debugging scripts regularly use multi-dimensional data-structures to track state, nested control-flow to implement aggregations, and complex
regular expressions to identify instrumentation points. We measure complexity of each OmniTable query and gdb script using three software engineering metrics: the number of lines of code, the number of terms in the abstract syntax tree (AST), and the Halstead complexity, which estimates the amount of time it would take to correctly produce the query or script using properties of the AST [18]. We included the definition of user-defined views (e.g., DefinedMemory(ot)) into the OmniTable queries that use them, so our results are an upper-bound on OmniTable query complexity.

Table 1 shows the results, indicating that OmniTable queries are less complex than gdb scripts. By geometric mean, OmniTable queries require 3.74 times fewer lines, 1.70 times fewer nodes, and 2.75 times less estimated time to develop than gdb scripts. There are only three queries that are more complex when expressed using the OmniTable model, the second and fifth redis 4323 queries, and the second Apache 60956 query. The two redis queries are small for both representations. The second Apache query suffers from the lack of kernel state in an OmniTable. The query identifies all blocking file descriptors, which requires substantial logic to track all function calls in the OmniTable model, but can be calculated in gdb using fcntl. Extending the OmniTable to include kernel state would reduce the complexity.

6.3 Query Latency

We evaluate the latency of OmniTable queries and gdb scripts for 3 representative queries from our case studies. We choose queries that use all of the high level views in our prototype (i.e., Funcs(ot), Vars(ot), and Insta(ot)) and offer a wide range of performance on current tools, from ~22 minutes to ~2 days. Figure 9 shows the latency of each debugging question evaluated using gdb, SteamDrill with a single core, and SteamDrill with 64 cores, with latency plotted on a log-scale. We executed Memcached 271 Q1 for 48 hours before killing the program and report its latency as 48 hours.

SteamDrill is significantly faster than gdb. SteamDrill query latency is between 2 and 290 times (with a geometric mean of 17) faster than gdb latency when using a single core, and between 6.9 and 1809 (with a geometric mean of 99) times faster than gdb latency when using 64 cores.

6.4 Optimizations

Next, we evaluate the impact of three optimizations on SteamDrill’s latency: parallelization, multi-replay resolution, and performance-counters.

Scalability. We evaluate the query latency of SteamDrill queries when using 1–64 cores; Figure 10 shows the speedup on a log-log scale. SteamDrill queries are 10.5 times faster using 64 cores than when run sequentially. Importantly, whereas prior parallelization efforts require the developer to substantially redesign their debugging code [30, 34, 35, 38, 46], the parallelized and sequential OmniTable queries are identical. The current scalability bottlenecks are caused by high initialization and serialization cost in Spark and the high cost of compiling cursors.

Multi-Replay Resolution. We evaluate the impact of multi-replay query resolution on the Memcached 271 Q2 query. We calculate the query latency when using two rounds of replay (the approach chosen by the SteamDrill planner) and when using a single round of replay on 64 cores. SteamDrill is 3.6 times faster when using multi-replay resolution.

Performance Counters. We evaluate the impact of using performance counters to accelerate the calculation of the time column in an OmniTable. We executed the 3 queries with and...
without using performance counters (when disabled, SteamDrill instruments all \texttt{jump, call, and return} instructions) on 64 cores. The performance counter optimization accelerates query latency by a factor of 1.6.

7 Related Work

The \textit{OmniTable} query model is the first debugging model that exposes all application state as a single entity and enables succinct observations via a high-level declarative language. Below we describe work related to high-level languages for debugging, using deterministic replay for debugging, and applying optimizations to accelerate debugging.

Existing systems support high-level debugging languages to reduce programming complexity; Table 2 illustrates the limitations of prior work compared to the \textit{OmniTable} model. Execution Mining [20], PQL [25], EBBA [5] and EndoScope [7] expose a time-stream model of execution, which complicates debugging since it is difficult to summarize data over time (e.g., these tools cannot express the \texttt{Funcs(ot)} view since it contains execution data from multiple points-in-time). Other systems limit visibility of execution state: Fay [12], PQL [25], EBBA [5], EndoScope [7], and PTQL [14] expose partial program state consisting of only the function calls or global variables values in an execution. Pivot Tracing [24], G2 [17], EBBA [5], and TQuel [40] require manual instrumentation to enable observations, which essentially amounts to supporting queries over software logs. Finally, many tools provide no, or very few, aggregates [14,20,25]; G2 [17] supports aggregates but requires that they be expressed in terms of a graph processing language.

Many \textit{OmniTable} queries compare correct execution behavior to incorrect execution behavior, similar to statistical debugging approaches [22]. There are two key differences (1) statistical bug isolation requires observing many correct and incorrect executions to come to a statistical verdict, whereas developers can often get a “sense” for correctness using an \textit{OmniTable} query with fewer examples and (2) statistical debugging approaches hard code the values that they compare (e.g., function argument values), whereas developers can customize \textit{OmniTable} queries to use program constructs best suited to their applications.

Many systems have noted that deterministic replay can be a great help when debugging software problems [8, 13, 19, 31, 42, 45]. Such systems enable a debugging program to explore an execution’s time-sequence in reverse, but retain a procedural interface.

Recently, JetStream [34] and Sledgehammer [35] use deterministic replay as a vehicle for parallelizing debugging, which our prototype uses to accelerate \textit{OmniTable} queries. However, these tools support procedural debugging models, similar to \texttt{gdb}, and consequently suffer from the programming complexity.

Existing tools do not decouple debugging logic’s execution from the original execution to optimize query latency. PARTICLE [14], Fay [12], Pivot Tracing [24] and PMSS [21] reduce the debugging performance overhead using traditional SQL optimizations (e.g., predicate push-down). However, these tools add instrumentation to the program and re-execute it to recreate the bug, which tightly couples the execution of debugging and the original execution and increases performance overhead. Additionally, by inspecting new executions, these systems are cannot perform all SteamDrill performance optimizations, particularly multi-replay query resolution.

8 Conclusion

In this paper, we propose the \textit{OmniTable} query model, a new debugging paradigm that reduces the programming complexity and performance overhead of debugging without restricting the execution state that a developer can observe. We show that the query model simplifies debugging questions compared to existing state-of-the-art tools by performing case studies of bugs reported in popular open-source software. Unfortunately, an \textit{OmniTable}, the key abstraction in the model, cannot be stored or calculated due to its extreme size. So, our prototype, SteamDrill, implements lazy materialization: it delays an \textit{OmniTable}'s calculation until a developer queries the table. It uses deterministic record and replay to store the execution associated with each \textit{OmniTable} and then generates instrumentation and traces a new replay execution to resolve each developer query on-demand. The system uses declarative optimizations, debugging optimizations, and a novel multi-replay strategy to accelerate debugging queries by an order of magnitude compared to state-of-the-art tools.

9 Acknowledgements

We would like to thank our shepherd, Ding Yuan, and the anonymous reviewers for their insightful comments. The work was supported by the National Science Foundation under grant DGE-1256260.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|}
\hline
\textbf{Tool} & \textbf{Model} & \textbf{Observations} & \textbf{Aggregates} \\
\hline
Execution Mining [20] & Stream & All & No \\
Fay [12] & Stream & Partial & Partial \\
Pivot Tracing [24] & Relational & Log-Based & Partial \\
G2 [17] & Graph & Log-Based & Manual \\
PQL [25] & Stream & Partial & No \\
PTQL [14] & Relational & Partial & No \\
EndoScope [7] & Stream & Partial & Yes \\
EBBA [5] & Stream & Log-Based & No \\
TQuel [40] & Relational & Log-Based & Partial \\
\hline
\textit{OmniTable} & Relational & Everything & Yes \\
\hline
\end{tabular}
\caption{Feature comparison of high-level debugging tools.}
\end{table}
References


[20] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution Mining. In Proceedings of
the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), March 2012.


