BQL: eBPF Observability with SQL Convenience

Franco Solleza Morgan Borjigin-Wang Faizah Naqvi* Justus Adam

Ziyun Song Akshay Narayan Malte Schwarzkopf Andrew Crotty †

Nesime Tatbul★◇

Brown University † Northwestern University ★ Intel ◇ MIT

Abstract

eBPF allows writing powerful and safe operating system extensions, but writing these extensions is difficult and requires expert knowledge. This complicates real-world eBPF use, e.g., to query the kernel for observability purposes.

BQL is a new system that makes writing observability-oriented eBPF extensions accessible to users without deep kernel or eBPF expertise. Key to BQL is a division of labor between observability engineers, eBPF experts, and BQL itself. Observability engineers write queries against a stable abstraction of reusable views defined by eBPF experts. Both queries and view definitions use a high-level, declarative SQL dialect, and BQL generates a schema from kernel code to define views over. Finally, BQL hides eBPF's complexity through *verifier safety*: a valid BQL query over views never results in an eBPF program that fails the verifier.

Evaluation with complex eBPF telemetry queries from the BCC collection shows that BQL can fully express 42 of its 57 queries, and partially supports the rest. The BQL queries are simpler and have comparable overhead to the original expert-written, hand-optimized implementations. BQL's optimizations are critical to achieving this performance.

1 Introduction

eBPF is a technology for extending operating system kernels with custom functionality. An especially popular type of eBPF extension targets *observability*, or gathering telemetry about the kernel's processing, usually with respect to a particular target application. For example, 25 out of the 54 projects listed on the eBPF homepage [2] target observability use cases [3, 5, 6, 9, 11, 13, 22, 23, 27, 29, 30, 36–38, 41–43, 45, 47–49, 51, 54, 57, 58], and many measurement frameworks and off-the-shelf tools now use this "superpower" [19].

Although eBPF enables the creation of powerful observability tools, writing application-specific or custom eBPF programs is difficult and requires significant expertise. Since eBPF executes in the kernel's context, eBPF developers must be experts in the kernel's implementation details to write useful programs. eBPF programs also need to pass the kernel's restrictive static verifier, which ensures that the program is safe. As a result, *observability engineers*, who simply wish to observe a particular application's behavior, must today rely on *eBPF experts* to write observability programs for them. eBPF experts, in turn, expend significant engineering effort to simultaneously perform four tasks: (*i*) decide where to instrument the kernel; (*ii*) determine what data to extract; (*iii*) specify how to filter this data for future queries; and (*iv*) implement all of this functionality efficiently.

BQL is a new system for observability that simplifies the lives of both observability engineers and eBPF experts. Using BQL, both observability engineers and eBPF experts write declarative observability queries in a dialect of SQL, and BQL executes these queries across the kernel (in eBPF) and userspace. Key to BQL's success is a novel division of labor between eBPF experts, observability engineers, and BQL itself: eBPF experts create reusable *views* over kernel information that supports arbitrary queries on top, observability engineers can write flexible queries on these views to satisfy their specific needs, and BQL provides an auto-generated schema that organizes kernel information to allow eBPF experts to write SQL views over kernel information.

Making BQL work required us to overcome three major challenges. First, kernel data sources and events come in many shapes and forms: eBPF supports half a dozen classes of hook points that each have access to structs (and nested structs) and helper functions that provide even more data. BQL addresses this challenge by transforming the information into a *kernel schema*. To do so, BQL parses the kernel code and events and data structures and organizes them as hooks and and data sources that views can query. This schema is specific to the running kernel version and adaptable for common eBPF use cases, such as casting between different socket types. Second, observability engineers must write their queries against a stable abstraction that hides low-level kernel minutiae, provides meaning to kernel sources and events, and never exposes eBPF verifier errors. BQL's views solve this problem. In BQL, eBPF experts define reusable

*Paper approved as a Master's report for this author. See §10 for more details.

views that organize the schema's hooks and data sources into semantically-meaningful groupings (e.g., "system calls" or "block I/O requests"). BQL guarantees that queries using these views are *verifier-safe* and will always run. Third, since eBPF programs interpose on the kernel's critical path processing, any logic injected via eBPF into the kernel must be efficient to avoid slowing down the application. BQL addresses this with its query optimization and code generation. BQL applies both classic query optimizations and novel ones specific to the eBPF setting, which requires splitting query processing into a kernel part (in eBPF) and a userspace part. BQL also merges generated programs where possible.

We built a prototype of BQL that supports querying Linux tracepoints, `fentry`/`fexit` hooks, perf events, and `uprobes` using a streaming, SQL-like abstraction. Our prototype generates eBPF code that runs in the kernel, installs and manages the resulting eBPF programs, and generates and deploys the corresponding userspace code that processes the exported data. The result is a convenient end-to-end experience for observability engineers. Indeed, we frequently turn to BQL for its convenience ourselves: e.g., we used it to run quick, exploratory eBPF observability queries to debug performance issues with BQL and its baselines for this paper.

In summary, this paper makes the following contributions:

(1) A data model that captures kernel state and behavior in a SQL-like schema. This schema minimizes the effort needed for eBPF experts to write reusable views that provide a stable abstraction to queries.
(2) Techniques to guarantee *verifier safety* for queries, which insulate observability engineers from eBPF verifier errors.
(3) An eBPF-aware query optimization approach that splits a query's execution between userspace and eBPF, and a code generation approach that generates efficient eBPF and userspace programs.
(4) BQL, an end-to-end solution for querying kernel observability data via a convenient, SQL-like interface.

Evaluation shows that BQL can express a wide variety of typical observability queries, that it matches the performance of the expert-written, hand-optimized queries, and that BQL's optimization techniques are essential to its competitive performance. BQL also achieves lower probe effect than eBPF-based queries in osquery [44], a prior system for querying kernel observability data that relies on hand-coded eBPF programs. We will open-source BQL.

2 eBPF for Observability

Many *observability engineers* wish to understand how their applications interact with the kernel. They are familiar with their own application and have the context required to design custom, application-specific queries over kernel data. They understand that eBPF can provide answers, but they often have little or no eBPF expertise. By contrast, *eBPF experts* are far less numerous, but possess the background and skills necessary to write complex eBPF programs. However, they lack the application knowledge of observability engineers.

This state of affairs satisfies nobody: eBPF experts expend significant engineering effort developing general-purpose tools, while observability engineers must choose among existing tools that fail to directly address their specific question. If observability engineers could author their own eBPF queries, they would be able to answer their own questions.

2.1 Writing eBPF programs

Today's workflow for writing an eBPF program proceeds roughly as follows. First, eBPF experts identify *hooks*, which are known code locations to which eBPF programs can attach. When execution reaches a hook, the kernel runs the eBPF programs attached to that hook. These eBPF programs use a variety of helper functions to access kernel objects and read kernel data available in their respective context. Programs can also communicate with each other and with userspace by storing state in restricted, kernel-provided data structures known as *eBPF maps*. The eBPF expert must reason about how to read the correct state and store it in eBPF maps, so that other programs can retrieve it. Finally, the eBPF expert must write a userspace program that periodically reads from these maps to return results.

At each step, both eBPF and kernel expertise are needed, and the programming interface is essentially a C program.

2.2 eBPF-based Observability

Several existing approaches attempt to remedy this situation.

Collections of programs. eBPF experts developed the BCC collection of eBPF programs [36], comprising 57 ready-to-use tools, each expressing a particular observability query. This approach takes significant engineering effort but yields inflexible queries. For example, one such program, `syscount`, which counts the number of invocations of each system call, is over 600 lines of C code and has six command-line options. Yet, the program is too inflexible for even simple variations of the query, such as instead calculating the average system call latency or the duration between two different system calls. Adding these features would require an eBPF expert to make significant modifications to the program's code.

Domain specific languages (DSLs) like BPFTrace [37], Ply [38], and Kernelscript [12] provide a simpler syntax for writing eBPF programs (e.g., special syntax for declaring maps) and prevent some verifier errors (e.g., DSL programs cannot access invalid memory). DSLs might allow engineers to write custom programs that express their exact query, but using these DSLs still requires eBPF expertise because their users must: (*i*) navigate the kernel's implementation details; (*ii*) manually optimize their programs; and (*iii*) reason about

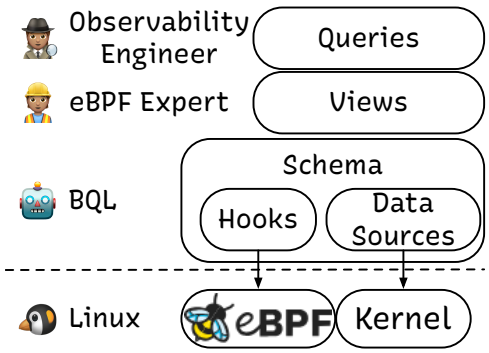


Figure 1: BQL provides three abstractions that build upon each other. BQL organizes kernel information into a schema, eBPF experts create reusable views on top of the schema to present kernel information in a meaningful way, and observability engineers write BQL queries on top of these views.

and handle verifier errors. Instead, eBPF experts commonly use these DSLs as tools to offer "one-liner" incantations, which offer only marginally better usability to observability engineers than the off-the-shelf eBPF programs they replace.

Query language interfaces like osquery [44] expose kernel data as tables and columns, and allow querying those tables via SQL. But osquery's query and execution model does not extend into eBPF. Instead, osquery relies on eBPF experts to manually define tables in its schema and write bespoke eBPF programs that populate these tables. In addition, osquery's static table schema prevents some optimizations to the eBPF program, as the eBPF program must return the full data for each row, including columns never used by any queries.

3 BQL Overview

BQL is a system to run observability queries on Linux kernel events using a dialect of SQL. Observability engineers and eBPF experts use BQL to accomplish their observability goals. Given a query, BQL generates eBPF and userspace programs that implement it and deploys those programs to run the query. BQL executes queries using an event-driven model that materializes records whenever a target event triggers.

Goals. BQL embodies three design goals. First, observability engineers should be able to write custom queries without managing eBPF or kernel details. Second, eBPF experts should be able to easily expose kernel data to observability engineers. In BQL's context, we refer to these eBPF experts as *view developers*. Third, queries should cause a low probe effect on applications running on the system.

Key abstractions. To achieve these goals, BQL provides three layered abstractions that separate implementation concerns between BQL, view developers, and observability engineers (Figure 1). At the lowest level, BQL provides a kernel

```
CREATE VIEW system_calls AS
SELECT A.task.parent.pid AS ppid,
    A.task.pid as pid,
    A.context.id AS syscall_number,
    B.timestamp - A.timestamp AS duration,
FROM "sys_enter" AS A
CAUSAL JOIN "sys_exit" AS B
ON A.task.tid = B.task.tid
```

(a) A BQL view representing system calls.

```
SELECT AVG(A.duration) AS total,
    A.syscall_number,
FROM system_calls AS A
GROUP BY A.syscall_number
WHERE A.pid = <pid>
```

(b) A BQL query on the `system_calls` view.

Figure 2: View developers and observability engineers write SQL to create views and write queries.

schema that organizes kernel events and data sources to fit in a relational abstraction. View developers then decide which schema information to expose in their views. Finally, observability engineers write queries using these views to answer their application-specific questions.

Example view and query. Figure 2 shows an example of a view (`system_calls` in Figure 2a) and a query using that view (Figure 2b). The view developer uses BQL's schema to identify the events and fields they want to expose in the view. This view exposes, for each system call, its duration, syscall number, the invoking thread's ID. The query (Figure 2b) then calculates the average latency of system calls for a given PID without needing to know the kernel's implementation details. Importantly, the same view can serve many queries. For example, if the observability engineer instead wanted to determine the total time spent on networking system calls, they would write a query that filters by syscall number and sums the durations.

3.1 BQL Abstractions and their Guarantees

While BQL borrows concepts from classic database literature like schemas, views, and query optimization, BQL adapts them to the eBPF and Linux kernel context. This requires solving challenges at each abstraction level and providing certain guarantees in each case—§4 describes how BQL provides these guarantees.

Schema. BQL's schema exposes all kernel events and state such that views can access them. To make writing views easier, BQL enforces that they only access events and types from the schema.

Views. If BQL accepts a view developer's view, it guarantees that queries over these views will execute and read the view's data. BQL accepts a view only if it can generate a corresponding eBPF program that passes the kernel's verifier.

Queries. BQL guarantees that a observability engineer's query that only reads from views is *verifier-safe* and will execute. Thus, views insulate observability engineers from kernel details since the observability engineer only needs to know about the view rather than the underlying kernel.

3.2 Optimization and Execution

BQL's decision to use a declarative language enables automated optimizations to reduce query probe effect. For example, BQL alters the split of processing between eBPF and userspace, and chooses particular eBPF map types to communicate between eBPF and userspace. §5 explains the details.

4 BQL Design

We now describe the design of BQL's abstractions—schema, views, and queries—and how BQL realizes their guarantees.

4.1 BQL's Schema

BQL parses the kernel's source code and generates a schema specific to the running kernel that lists the set of available eBPF hooks and the kernel types accessible to those hooks. The schema includes *BQL data sources* and *BQL hooks*. BQL views (§4.2) reference these schema elements.

BQL Hooks. A BQL hook corresponds to a kernel location where BQL can attach an eBPF program, such as the entry and exit points of a kernel function or an instrumented tracepoint in the kernel source. In Figure 2a, the view's `sys_enter` BQL hook corresponds to the entry point of the kernel function `syscalls/sys_enter`. BQL associates each hook with a set of BQL data sources accessible from that hook. Each data source is an instance of a kernel data type that the hook can access, via either its hook-specific context or eBPF helper functions. BQL exposes both this context and also other data sources from hook-appropriate eBPF helper functions (e.g., `bpf_ktime_get_ns()` or `bpf_get_current_pid_tgid()`).

BQL supports four classes of observability-related eBPF hooks: (*i*) kernel function entry and exit points (i.e., Linux `fentry` and `fexit`); (*ii*) manually instrumented kernel tracepoints; (*iii*) hardware performance counters (i.e., via the Linux `perf_trace` subsystem); and (*iv*) userspace function entry and exit points (i.e., `uprobe`, `uretprobe`).

BQL Data Sources. A BQL data source corresponds to a kernel type that contains fields, which can be primitive types (e.g., integers or strings) or nested data sources. For example, in Figure 2a, the view accesses both the PID of the process making the system call (`A.task.pid`) and its parent process (`A.task.parent.pid`). The kernel stores this data in an instance of `struct task_struct`, which BQL's schema represents as the `task` data source (`A.task`). The `task` data source exposes its fields as primitives or nested data sources. For example, it exposes the `task.pid` as a `u64` and the `task.parent` as a nested `task` data source.

Generating BQL's Kernel Schema. BQL generates a kernel schema specific to the running kernel version. To locate all hooks in the Linux kernel, BQL parses Linux's tracepoint documentation (`/sys/kernel/tracing/events`) and kernel function signatures to find `fentry` and `fexit` eBPF events. Each tracepoint and function entry/exit corresponds to one BQL `hook`. The function signatures and tracepoints also provide the definitions for each hook's context. To find kernel types and generate BQL data sources, BQL parses Linux's `vmlinux.h` header file, an auto-generated header file that contains all type definitions in the running kernel.

Userspace function entry/exit events (e.g., Linux `uprobes`) require a different approach because they hook into arbitrary programs whose source code is unavailable to BQL. To define these hooks, an eBPF expert identifies a function name and specifies how to extract its arguments as a data source.

For hardware performance counters, BQL generates a minimal schema containing only data sources accessible through eligible eBPF helper functions.

4.2 eBPF Experts' Views

View developers use BQL's schema and their knowledge of the kernel's implementation to create *views* over kernel data. While view developers write views using SQL, many eBPF programs perform operations outside the scope of native SQL. In particular, eBPF programs may cast kernel pointers from one type to another or accept kernel hook points as parameters. In addition, BQL needs to guarantee that queries that only access views are verifier-safe.

Reinterpreting Data Sources. eBPF programs frequently re-interpret a pointer from one type to another based on context. For example, in the Linux network stack's TCP implementation, it is common to cast `struct sock` pointers (representing a generic socket) to `struct tcp_sock` pointers (a TCP socket). BQL provides `REINTERPRET` statements for this purpose, which globally allow views to reinterpret one data source as another. For example, the statement:

```
REINTERPRET sock TO tcp_sock AS tcp;
```

allows view definitions to reinterpret a `sock` data source as a `tcp_sock` data source when they access a field named `tcp`. The view can then access fields of the `tcp_sock` data source, which map to the kernel's `struct tcp_sock` struct. View developers can define arbitrary reinterpretations between data sources, though incorrect reinterpretation definitions can cause verifier errors or garbage output.

Hook-Parameterized Views. Observability engineers often need to read the same information across a variety of

events. For example, the `funclatency` BCC program generates a latency histogram for any kernel function the observability engineer specifies. While it is possible to generate a custom view for each possible kernel hook, doing so is impractical. BQL supports this operation with *hook-parameterized* views. To create a parameterized view, a view developer specifies the hook class(es) the view works for (e.g., kernel function entry points). Since this view must work for *all* hooks in the specified class, BQL limits the data sources it can access to those available across all hooks in that class.

Verifier Safety. BQL needs to guarantee that queries on views are verifier-safe. When creating a view, BQL tests whether its generated eBPF program for that view passes the verifier. To perform this test, BQL generates the minimal eBPF program required to execute the view—i.e., one that reads the specified data from the kernel and passes it to userspace—and checks that this program passes the verifier. If it does, BQL can fall back to executing any query over this view by running all operations other than this minimal eBPF program in userspace.

For parameterized views, BQL performs a variant of this test. In Linux, eBPF programs that attach to userspace (i.e., uprobe) or hardware performance counters (i.e., `perf` events) can omit the specific hook for the kernel to verify them at, so BQL simply loads eBPF programs for these views without a hook to test them against the verifier, then attaches the programs to hooks later when needed. Other hook classes, such as kernel function entry and exit, require the eBPF program to specify a hook. BQL tests these views using an arbitrary hook (`fentry/do_exit`). If the parameterized view passes the verifier for this hook, it will also pass the verifier for any hook of the same type because BQL guarantees that parameterized views cannot access hook-specific information.

Although BQL has the option to fall back to the "minimal eBPF" query execution strategy, this is not desirable since it causes high probe effect (§7.3). By default, BQL applies query optimizations (§5) that reduce this probe effect. These optimized eBPF programs could in theory fail the verifier, but we never encountered such a case when developing or using BQL. Rather, since views use BQL's data sources, there are only two reasons why one might fail the verifier: (*i*) exceeding stack size or instruction count limits, or (*ii*) incorrectly reinterpreting a data source. If a view exceeds the verifier's stack size limit, the eBPF expert might fix it by exposing smaller fields (e.g., if a string value is too large). Similarly, if a view exceeds the instruction limit, the eBPF expert could break it into two views.

4.3 Observability Engineers' Queries

BQL queries use common SQL operators, including `SELECT`, `WHERE`, `UNION ALL`, and `GROUP BY`. BQL also supports common aggregation operators such as `SUM` and `COUNT`. In addition, BQL devises customized SQL concepts to support common observability queries and maintain its guarantees.

Capturing Relationships Between Events. Queries and views can express relationships between events using BQL's `CAUSAL JOIN` and `MATCH JOIN` operators. The example in Figure 2a includes a causal join between the `sys_enter` and `sys_exit` hooks. The `CAUSAL JOIN` operator between two views or hooks performs a one-to-one match of data from an event on the initiating side with data from a *subsequent* event on the receiving side. [1] BQL's `MATCH JOIN` generalizes this operator to perform a one-to-one match between two hooks that could occur in an arbitrary order.

While these operators are verifier-safe, query and view authors remain responsible for their semantics; they must ensure that these events actually have a causal (or one-to-one) relationship and will receive meaningless output otherwise.

Parameterizing Views. To query parameterized views, observability engineers specify the BQL hook(s) they want to query as an argument to the view. BQL returns an error if this hook that is not a member of the classes of permitted hooks that the parameterized view accepts.

Queries vs. Views. Queries must target views rather than querying BQL's schema directly. This is because direct queries over BQL hooks can fail the verifier. For example, such a direct query might select fields from so many data sources that its eBPF program exceeds the stack size limit. Since BQL guarantees that queries are verifier-safe, BQL rejects queries that directly access hooks and their data sources.

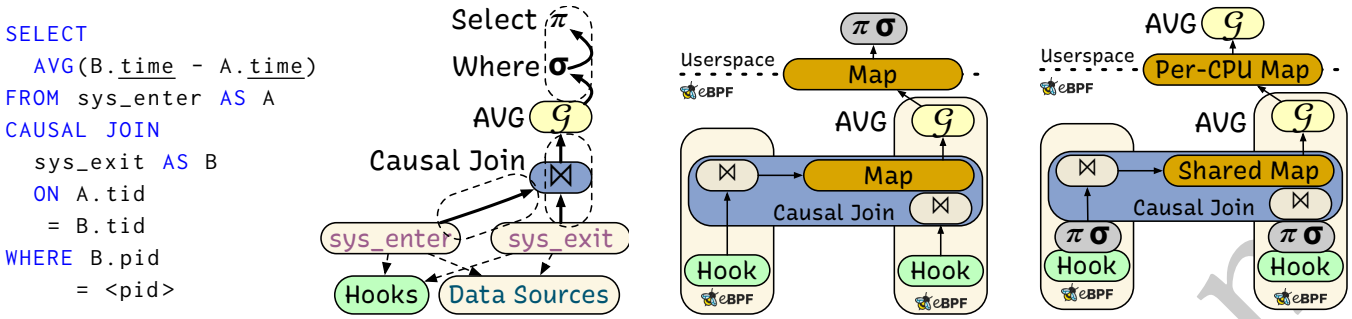
4.4 Summary

Taken together, BQL's schema, views, and verifier-safe queries simplify both the eBPF experts' and observability engineers' tasks. eBPF experts can write SQL views to expose kernel data without worrying about implementation details, and observability engineers can write custom verifier-safe queries using these views.

5 Executing BQL Queries

To execute a query, BQL must split its work between eBPF and userspace, generate the necessary programs, and coordinate their execution. Since eBPF programs run in the critical path of the kernel's processing, BQL must carefully choose how to partition work between eBPF and userspace so that

[1] BQL's causal join operator takes inspiration from prior work on "happens-before" joins by Mace et al. [34]. While "happens-before" joins have a constant discriminator, BQL's `CAUSAL JOIN` must allow users to specify the discriminator that relates causal events.



(a) Query source **(b)** Query operator graph **(c)** eBPF-userspace split. **(d)** Query optimization.

Figure 3: BQL represents a query (a) as a graph of operators (b; dotted lines show the query graph's pipelines). BQL splits (c) the query graph between eBPF and userspace at the lowest pipeline breaking operator. Finally, BQL's query optimization pushes (d) data reduction operations (e.g., filters) below these pipeline breakers so they execute in eBPF.

the query's eBPF programs avoid imposing a high probe effect on applications. Three core factors affect this probe effect: (*i*) the amount of work the eBPF programs perform; (*ii*) the amount of data they send to other eBPF programs; and (*iii*) the amount of data they send to userspace. These factors complicate code generation. For example, consider BQL's "minimal eBPF" query plan that ensures verifier safety if its optimized plan fails; while such a plan places little query logic in eBPF, it sends a large volume of data to userspace, and thus incurs a high probe effect since transferring data to userspace is expensive (§7.3).

We now describe how BQL chooses the kernel-userspace query split, and how BQL implements optimizations that result in eBPF programs with low probe effect.

5.1 eBPF-Userspace Query Split

BQL first transforms a query into a logical plan, which is a directed acyclic graph of operators. Any views that the query uses are sub-graphs of the plan. Figure 3a reproduces and simplifies the query in Figure 2b, and Figure 3b shows BQL's plan for the query. BQL must decide where to split the query's graph of operators between eBPF and userspace components. To make this decision, BQL considers whether sequences of operators in the query's plan can form a pipeline (i.e., they can immediately perform their processing after the preceding operator in the graph). For example, any operator following a filter operator (σ) can immediately process the operator's output data. Other operators are pipeline breakers that materialize intermediate state, meaning that subsequent operators must block and cannot immediately make progress. For example, the `CAUSAL JOIN` operator breaks the pipeline of the sub-graph corresponding to the initiating event, since the operator materializes the event's state to send to the receiving event. However, the receiving event does not break its pipeline, since it only reads state from the operator. Meanwhile, `GROUP BY` breaks *all* its input pipelines because it accumulates aggregates, so a separate program needs to read the operator's state. In BQL, the operators that break all their input pipelines are two aggregation operators (`GROUP BY` and `AGGREGATE`), and `UNION ALL`.

In each branch of the query plan, BQL splits the plan at the lowest operator that breaks all its input pipelines, as Figure 3c shows. This results in one eBPF program per processing pipeline below each split. The splitting operators' input pipelines execute in the kernel as eBPF programs that update the operators' state stored in an eBPF map. Then, all other operators execute in a single userspace program, which reads the state in each program's terminating eBPF map and executes the remaining operators. The insight behind this split is that, since the query cannot make progress after a pipeline breaker until another kernel event occurs, reading the map value in userspace will occur off the critical path.

BQL must also choose the eBPF map type each pipeline breaker uses to reduce probe effect, since contention over shared maps can be expensive. For example, the `GROUP BY` and `AGGREGATE` operators use per-CPU eBPF hash maps for associative aggregations to avoid synchronization. Similarly, the `UNION ALL` operator uses eBPF's `PERF_EVENT_ARRAY` map, a per-CPU data structure that sends data to userspace. On the other hand, `CAUSAL JOIN` relies on ordering across events that could occur on different cores, so BQL uses shared maps for this operator.

5.2 Query Optimization

Merely splitting the query plan is insufficient for good performance, since queries may still unnecessarily send data between eBPF and userspace. BQL adapts three classic database optimizations aimed at data reduction: projection, predicate, and aggregation pushdown. Unlike in traditional database

systems where these optimizations are almost always beneficial, the eBPF-userspace split creates a new trade-off. Pushing operators below the eBPF split adds more processing to the kernel, but in turn reduces the amount of data the eBPF program writes into maps. Efficacy of each optimization depends on the query, although generally it is better for performance to minimize the data eBPF programs write to maps, even at the expense of running more logic in the kernel (evaluated in (§7.2)).

BQL implements projection pushdown by only materializing (i.e., assigning to variables and writing into maps) those variables that the query will ultimately read. Although an eBPF expert's view may expose many fields, BQL's generated eBPF program will only read the fields that the active query over that view reads. Similarly, BQL's predicate pushdown reduces processing for kernel events that fail any filters the query includes. For example, a query with a `WHERE pid = <pid>` filter imposes probe effect for *all* processes without predicate pushdown (cf. §7.3), but only has probe effect for the target PID with the optimization. Finally, aggregations also reduce data volume. Aggregation operators are also pipeline breakers, so BQL uses them to split the query plan between eBPF and userspace. However, aggregation's data-reducing nature usually makes it preferable to push them into eBPF even though doing so requires more in-kernel processing.

Figure 3d shows the final plan after performing these optimizations. The plan has two eBPF programs: the first program begins with the `sys_enter` hook and ends with storing data in the `CAUSAL JOIN` operator; and the second program begins with the `sys_exit` hook, reads data from the `CAUSAL JOIN` operator, and ends with a `GROUP BY` aggregation.

5.3 Code Generation

BQL must generate, compile, load, and run eBPF programs for each of the pipelines in the plan that execute in the kernel. For each eBPF program it generates, BQL first materializes (i.e., generates C code to read into variables) the fields used in any predicate operators that the pipeline includes. Then, BQL adds an implementation of the predicates themselves; an operator such as `WHERE pid = 123` would result in C code of the form `if (pid != 123) goto END;`.

After the predicate(s), BQL materializes all other fields that the projection operator identifies using the metadata for each data source stored in BQL's schema. For example, BQL generates calls to `BPF_CORE_READ` to access the data for a nested data source, such as a field in a hook's `ctx` struct. BQL then generates code for the other operators in the pipeline that use these materialized values.

`GROUP BY` and Aggregates. For aggregation operators, BQL generates code that accumulates aggregate values in a per-CPU map. For `GROUP BY`, BQL generates a `key` struct corresponding to the operator's predicate. Since these operators are pipeline breakers, eBPF programs exit after this code.

`UNION ALL`. Similar to aggregations, BQL generates code for the `UNION ALL` operator in the program of each input pipeline that writes every record in a perf event array and then exits the program.

`CAUSAL JOIN`. BQL generates code across two eBPF programs to implement the `CAUSAL JOIN` operator: the initiating event's eBPF program writes into a shared eBPF map with a key corresponding to the `ON` predicate and exits, and the receiving event's eBPF program reads from this map. If no matching value exists, the program drops the event and exits. Otherwise, the program executes the code for the remaining operators in the second event's pipeline.

`MATCH JOIN`. The `MATCH JOIN` operator also has two pipelines, but since the events could occur in an arbitrary order, both pipelines read from the shared map. If a pipeline's program fails to find the `key` in the map, then its event happened first, so it writes its data to that map and exits. Otherwise, it reads the map value and continues the pipeline.

BQL code generation merges eBPF programs on the same hook into a single program. This is necessary for two reasons. First, query plans can result in multiple eBPF programs attached to the same hook. eBPF's runtime specifies no execution order for these programs, which could cause `CAUSAL JOIN`s between two events on the same hook to return incorrect results. Second, each eBPF program incurs initialization overhead. Merging avoids both issues. If two programs from the same hook involve a `CAUSAL JOIN`, BQL places the receiving event's program first, so it reads data from the prior event on the hook. While it is possible for a merged eBPF program to exceed the verifier's stack or instruction count limits, we have never encountered this in practice. If it happened, BQL would safely fall back to its "minimal eBPF" execution strategy and run the `CAUSAL JOIN` in userspace.

6 Implementation

We implemented a prototype of BQL for Linux in 16k lines of Rust. The prototype generates C source code for its eBPF programs and compiles them with Clang before loading and attaching them into the kernel using libbpf[32]. Its userspace program reads from each eBPF map that corresponds to a split operator in the query plan and executes any subsequent operators in userspace using the Volcano [18] model. The SQL frontend is adapted from Apache DataFusion's sqlparser[4] library.

Schema extraction. When extracting the schema, BQL resolves all macros in the kernel source and then uses Hancock [61], a script that uses `ctags` to extract function signatures from C source.

Limitations. Our current prototype supports four Linux eBPF hook types: tracepoints, fentry/fexit, `perf_trace`, and `uprobes`. We found these to be sufficient to support a wide range of queries, but adding support for additional hook types (e.g., kprobes) is straightforward. Our prototype's query planning and code generation also lacks support for some query optimizations, such as pushing down predicates past aggregation operators. Similarly, it is possible in some cases to eliminate spurious `CAUSAL JOIN` operations (i.e., when the query uses data from only one of the two events). We plan to add these optimizations in future work.

7 Evaluation

Our evaluation seeks to answer five key questions:

(1) Can BQL queries express a broad range of eBPF-based observability programs, and does BQL make it easier to write these programs? (§7.1)
(2) How does BQL's probe effect compare to that of handwritten, expert-tuned eBPF programs? (§7.2)
(3) How do BQL's optimization techniques affect the probe effect imposed on (*i*) the monitored application (§7.3) and (*ii*) unrelated "bystander" applications (§7.3)?
(4) How does BQL's probe effect compare to that of osquery, an alternative SQL-based observability system that relies on handwritten eBPF programs? (§7.5)
(5) Can BQL express specialized observability queries, and what probe effect does it impose? (§7.4)

Hardware. We conduct the experiments on Cloudlab D430 machines with two 8-core Xeon E5-2630v3 CPUs, 64 GiB DDR4 memory, and a 200 GiB, 6 Gbps Flash SSD.

7.1 Expressivity

We evaluate BQL's expressivity by re-implementing 57 observability programs from the BPF Compiler Collection (BCC) using BQL. These queries cover system calls, scheduler events, block I/O operations, the network stack, and other parts of the kernel. Two queries monitor userspace applications using `uprobes`. We also re-implement MeshTrek, a complex eBPF program that instruments the latency of the Envoy HTTP proxy's processing phases.

BCC. For 42 of the 57 BCC queries, the BQL implementation is functionally equivalent to the original (Table 1). Specifically, for these queries, the BQL version returns the same data, and in cases where the BCC implementation supports modified operations with command-line arguments, the BQL version returns the same data if provided the appropriate `WHERE` clause or aggregation operator. For another six queries, the BQL implementation differs only in output formatting from the BCC one. For example, the `profile` program displays foldable stack traces. A seventh query, `ksnoop`, has two modes: `ksnoop info`, which prints kernel functions' signatures, and `ksnoop trace`, which traces kernel functions' argument and return values. BQL's schema stores the same information that `ksnoop`'s `info` queries, but our prototype lacks schema introspection. `ksnoop`'s `trace` mode provides functionality that BQL realizes as queries over views.

While BQL fully or partially (i.e., except for our prototype's implementation limitations) supports 49 of the BCC programs, the remaining eight contain optional features that would require extensions to BQL to support. Six of these queries require new BQL data types to implement; three work over stack traces and the other three read large strings in eBPF with custom logic. To support these, BQL would require data model extensions for stack traces and large strings, respectively. The final two BCC queries both perform memory accesses unsupported in BQL: `drsnoop -e` reads a kernel array at a static offset, while `klockstat -s` uses Linux's `container_of` macro to perform pointer arithmetic.

Naturally, since BQL is based on SQL, BQL queries are significantly shorter than BCC's original C programs. The average length of a BCC program is 499 lines of code, compared to 45 lines of query and view code in BQL. For example, the longest BCC program is `memleak`, which traces memory allocations over 1,605 lines of `C`. The implementation of this query in BQL uses one view that consists of 78 lines of SQL. BQL uses succinct queries over this view to support `memleak`'s command-line arguments, which filter by process ID or by allocation size or age, print addresses, instrument a particular object's allocation function, etc.

MeshTrek. To stress-test BQL's ability to express complex queries, we also re-implemented a query from a recent instrumentation tool, MeshTrek, targeting the Envoy [16] HTTP proxy. This query places 12 `uprobes` in Envoy to measure the latency overhead that its various processing stages impose on HTTP request processing. This requires correlating multiple events across the life-cycle of two Envoy objects, "connections" and "streams." Each `uprobe` in MeshTrek tracks an operation on either an Envoy connection or stream and correlates it to an HTTP request/response pair's common distributed tracing header value. MeshTrek's original implementation of this query in eBPF is 200 lines of C code, and our implementation in BQL has 62 lines.

7.2 Probe Effect on End-to-End Workloads

Queries in BQL should ideally cause a similar probe effect to handwritten and expert-optimized eBPF programs. We evaluate whether this is the case using two workloads: a RocksDB [52] application that stresses storage operations, and the OpenTelemetry demo [15] microservice application that stresses the network stack.

Category	Count	Features needed	Example program
Fully supported	42	Fully expressible in BQL.	`syscount` counts system calls, grouped by properties.
Partially supported (implementation limitations)	6	Output formatting.	`profile` generates folded stacks; supportable with extra userspace code around BQL queries.
	1	Schema introspection.	`ksnoop info` prints kernel function signatures.
Partially supported (BQL limitations)	3	Limiting depth of stack traces.	`futexctn` with the perf-max-stack-depth filter. Support possible, but requires BQL data model extension with "stacktrace" datatype.
	3	Read large strings.	`filelife` with `-F` argument (file path).
	1	Static array read at arbitrary offset.	`drsnoop` with `-e` argument: free pages
	1	`container_of` Linux macro.	`klockstat` with `-s` argument: accesses netlink locks.

Table 1: Of 57 BCC queries, BQL fully supports 42 queries. "Full support" means that BQL supports all variants (via command line options in BCC) and that the output is equivalent. BQL partially supports all of the remaining 15 queries. Seven could be fully supported except for limitations of the BQL prototype implementation (output formatting and lack of schema inspection), while eight programs encounter deeper BQL limitations. Of these, three programs require a "stacktrace" datatype that BQL's data model currently lacks, while specific options for the remaining five programs rely on unsupported eBPF features.

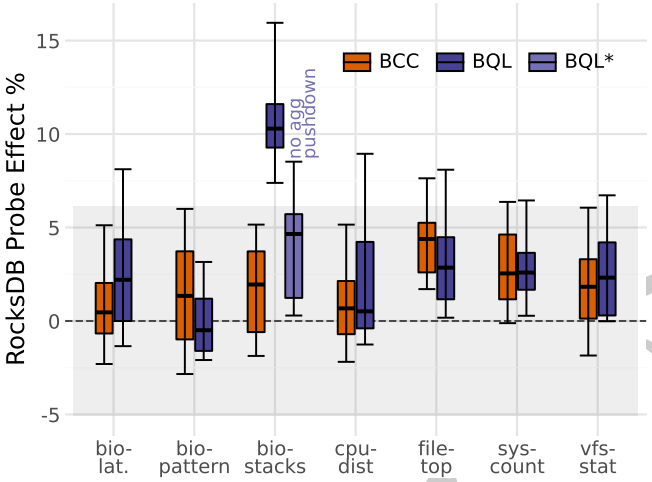


Figure 4: BQL queries impose a comparable probe effect on operation throughput in RocksDB as hand-tuned, expert-written BCC programs. Gray shading indicates the baseline performance range (p5—p95), and boxes and whiskers show (p5, p25, p50, p75, p95) for BCC and BQL across 25 iterations.

Baselines. We select queries according to each application's critical resource and measure the application's performance in three configurations across 25 iterations each: (*i*) *Baseline* runs the application without any eBPF programs running; (*ii*) *BCC* runs the application alongside a particular BCC query; and (*iii*) *BQL* runs the application alongside the BQL implementation of the same query.

For these experiments, a good result for BQL would show that BQL's query implementations impose a comparable probe effect to BCC's handwritten implementations.

RocksDB. We run seven queries over RocksDB that focus on storage operations. For example, the `biolatency` query gathers a histogram of block device operation latencies. We configure RocksDB with four client threads and 16 worker threads and run a closed-loop YCSB workload that performs 95% reads and 5% writes over 1M uniformly random keys with values ranging from 8 to 128 bytes. We disable RocksDB's write-ahead log on writes. We report the probe effect for the BCC and BQL query implementations on operation throughput relative to the median throughput achieved by the baseline (in this experiment, 370k operations/sec). The baseline itself experiences variable throughput, and we indicate the range of its performance with shading.

Figure 4 shows the results. BQL queries offer comparable probe effect to the BCC implementation and this probe effect is within the baseline configuration's range of variation, except in one case (`biostacks`). Digging deeper, we identified an implementation limitation in BQL's optimizer and code-generation that causes this high probe effect. This query traces and groups block device operations by the call path they took through the kernel. The BQL implementation uses a `UNION ALL` across three views produced with `CAUSAL JOIN` operators. Since BQL implements the aggregation pushdown operation, these `CAUSAL JOIN` operators on the same kernel events appear in multiple of BQL's eBPF programs, and our current BQL prototype is unable to merge programs in different pipelines. So, BQL executes the `biostacks` query using seven total programs over three eBPF hooks. We confirm that this is the source of the probe effect with an alternate configuration of BQL in which we manually disabled aggregation pushdowns (BQL * in Figure 4). With this configuration, BQL generates only one program for each hook. Although this runs counter to BQL's optimization heuristic of minimizing

data transfer between eBPF and userspace, in this case, disabling the optimization eliminates redundant work across eBPF programs; a future cost model could detect this.

OpenTelemetry. We use the OpenTelemetry microservices demonstration application [15], which comprises 15 services across 12 programming languages to evaluate BQL's efficacy for network-focused queries. We deploy this application across five machines on Cloudlab using Kubernetes. Neither BCC's nor BQL's queries imposed any measurable probe effect on the application's end-to-end latency. To stress BQL further, we adapted the application's partially open-loop Locust [24] load generator to connect directly with the "Cart" microservice over gRPC, and use a Locust configuration emulating 750 concurrent users. We measure the `GetCart` and `AddItem` endpoints, which both in turn issue requests to Valkey [1], an in-memory key-value store. We evaluate nine BCC queries, all of which focus on TCP operations, and measure probe effect on each endpoint's latency. We run the experiment with 25 iterations and calculate the probe effect of the BCC and BQL configurations relative to the median across iterations of the baseline's median latency (in our experiment, this median latency was 3.9ms for `AddItem` and 2.4ms for `GetCart`) in each iteration.

Figure 5 shows the results. Six of the nine queries observe infrequent TCP events such as connection establishment (e.g., `tcpconnect`, `tcpconnlat`, or `tcplife`), and so neither the BCC nor BQL query implementation have significant probe effect. `tcprtt` queries events that occur on every packet arrival, but calculates a histogram of packet latency in eBPF. This is a cheap operation, so the probe effect is withing the baseline's range of variation.

The remaining two queries, `tcptop` and `tcppktlat`, query events that occur on every socket write and every packet arrival, respectively. Since the Cart microservice uses small message sizes, these events occur approximately equally often, and correspondingly both queries incur similar probe effect with across the BCC and BQL implementations.

Takeaway. For 13 of the 16 queries we evaluated across RocksDB and OpenTelemetry, neither BQL nor BCC queries have a measurable probe effect relative to the baseline; only two TCP queries (`tcppktlat` and `tcptop`) show significant probe effect, and in these cases BQL and BCC achieve comparable performance. Indeed, when we inspected the eBPF programs BQL generated for these queries, we found that BQL's eBPF programs to be structurally similar (i.e., same number of map writes, etc.) to BCC's hand-optimized implementations. The only query that diverged from this norm was `biostacks`, which revealed a limitation in BQL's query optimizer, showing that the aggregation pushdown heuristic does not always yield the best performance. Future improvements to both BQL's optimizer and code generation will benefit all queries of this type at once, rather than needing to optimize each query's implementation separately as eBPF experts must do today.

7.3 Impact of BQL's Optimizations

We now evaluate the impact of BQL's optimizations on queries' probe effect. We consider three microbenchmarks that use the same RocksDB workload to measure the impact of aggregate pushdown, predicate pushdown, and split selection. To evaluate aggregate pushdown, we consider BCC's `filetop` query, which contains a `GROUP BY` aggregation operation (`filetop-aggregate`). For predicate pushdown, we modify the same `filetop` query with a selective predicate (`filetop-predicate`) that filters out file operations with size smaller than 100 KB. For split selection, we modify BCC's `syscount` query to calculate the mean latency of each syscall using a `CAUSAL JOIN` and a `GROUP BY` (`syscount-join`). We evaluate three configurations: (*i*) In "Minimal eBPF," BQL uses only the minimal eBPF programs needed to execute the query (i.e., those corresponding to views); (*ii*) in "Split Only", BQL only splits the query plan between eBPF and userspace without further plan optimizations, and (*iii*) in "Query Optimization", BQL applies operator pushdown operations.

The queries we evaluate hook into frequent kernel events. Without optimizations—i.e., in "Minimal eBPF" and "Split Only"—we observed that BQL's userspace program falls behind reading events from the eBPF maps. This results in dropping data, which triggers interrupts that further increase probe effect and might hide the impact of optimizations. To focus on the impact of the eBPF programs BQL generates, in this experiment, we replace BQL's userspace program with one that discards all data from eBPF maps. A good result for BQL would show that each of BQL's optimizations decreases its probe effect. Figure 6a shows the results.

Aggregate and Predicate Pushdown. BQL's query plan for `filetop` first performs a `UNION ALL` operation on the `vfs_read` and `vfs_write` events, then performs a `GROUP BY` (or, for `filetop-predicate`, a `WHERE`) on the result. In "Split Only," BQL splits the plan at the `UNION ALL` operator. In "Query Opt.," for `filetop`, BQL pushes the `GROUP BY` aggregation below the `UNION ALL` operator and splits the plan at the `GROUP BY`. For `filetop-predicate`, BQL similarly pushes the predicate filter below the `UNION ALL`. "Minimal eBPF" and "Split Only" perform similarly for `filetop-aggregate` and `filetop-predicate`. This is because both plans transfer the same data volume between eBPF and userspace. "Query Opt.," despite adding aggregation/predicate logic to the eBPF program, reduces this data volume and lowers probe effect.

Query Split Selection. For `syscount-join`, the "Min. eBPF" version exhibits high probe effect due to the data volume transferred between kernel and userspace. By contrast, the "Split Only" eBPF program places `FILTER`, `CAUSAL JOIN`, and

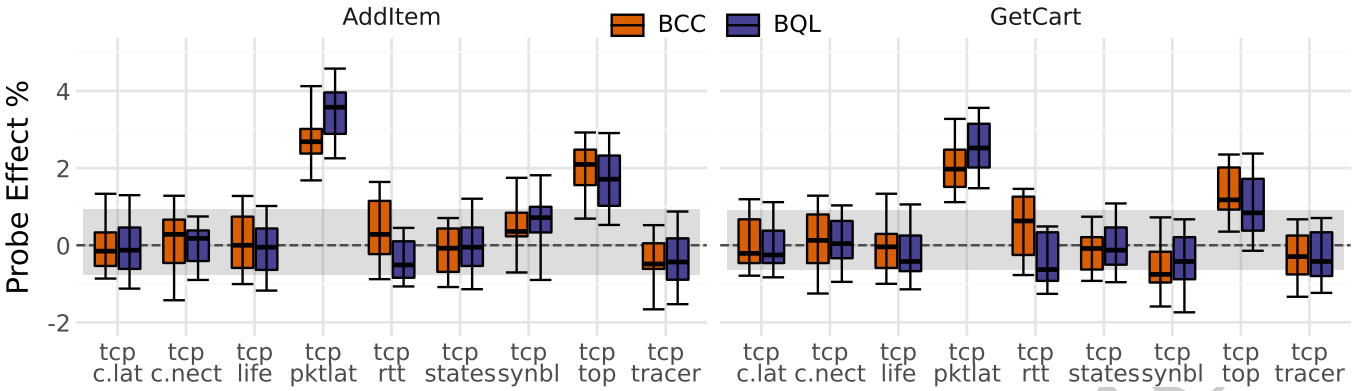
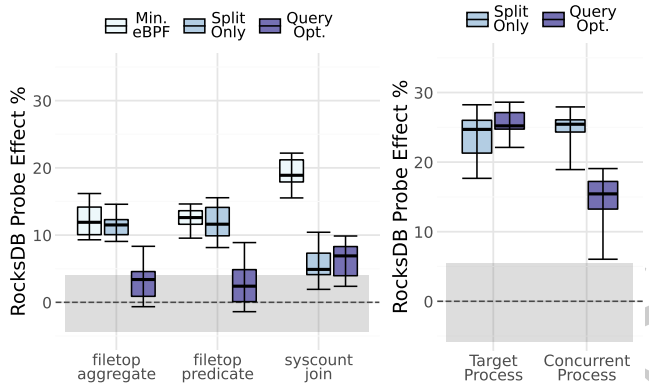


Figure 5: BQL has comparable probe effect on median latency compared to BCC's hand-tuned queries for two endpoints in OpenTelemetry's "Cart" microservice. The shaded region indicates the baseline's performance range, and boxes show each TCP-focused query's probe effect for BQL and BCC.



(a) Monitored application. **(b)** Across processes.

Figure 6: BQL's query split selection and query optimizations reduce probe effect by 5–12 percentage points. Both query split selection and query optimizations are necessary for good performance (a). Query optimizations also reduce probe effect on concurrent "bystander" processes because the eBPF program exits early for them (b).

`GROUP BY` in the kernel, which significantly reduces data volume and probe effect. This shows that BQL's default split heuristic is beneficial. Since "Split Only" already places all query operators in the kernel, pushdown optimizations in "Query Opt." cannot further reduce data volume and probe effect in this experiment.

Probe Effect on Concurrent Processes. eBPF programs can have probe effects beyond the observed application. To evaluate the impact of BQL' optimizations on "bystander" applications, we modified the `syscount` query to emit each syscall's enter and exit event for a RocksDB process (i.e., `WHERE pid = <x>`). We deploy a second RocksDB process that the query does not target and measure probe effect for both applications. A good result for BQL would show that "Query Opt." decreases the probe effect observed by a *concurrent* RocksDB process. Figure 6b shows that pushing the filter that selects only events for one process into the kernel indeed reduces probe effect for the concurrent bystander process. The target process's probe effect increases slightly as it must now evaluate this filter in eBPF.

BQL performs these optimizations automatically, without engineering effort. Importantly, since BQL performs these optimizations across the whole query plan, including any views, eBPF experts writing views avoid having to manually optimize their program towards the possible queries that an observability engineer might ask. Similarly, an observability engineer can query views knowing that BQL will automatically optimize their query and the view together.

7.4 MeshTrek

We now evaluate BQL performance on MeshTrek's query instrumenting the Envoy service proxy—a complex query that involves multiple userspace probes (`uprobes`). We use a microbenchmark consisting of two microservices that communicate using HTTP/1.1. One service sends requests to the other, and we deploy Envoy as a sidecar proxy (via the Istio service mesh) for both services. Each service returns responses after a fixed 2ms delay. We generate 900 requests/second of offered load using the wrk2 request generator.

A good result for BQL would show that even for this complex query, which includes 12 `CAUSAL JOIN` operators, the probe effect the BQL implementation causes remains comparable to the original hand-optimized version. Figure 7 shows the result. The BQL implementation causes slightly higher probe effect (7.74% vs. 7.52%) than the handwritten implementation. We inspected BQL's generated eBPF programs

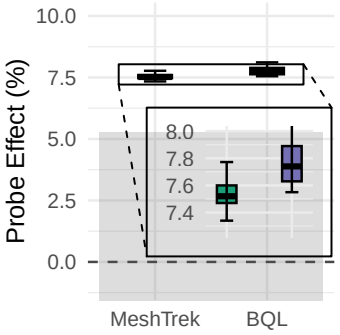


Figure 7: BQL runs MeshTrek's complex query with similar probe effect.

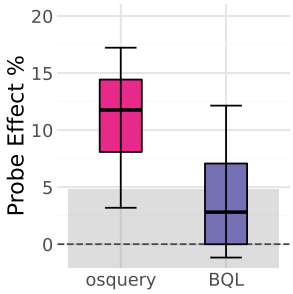


Figure 8: Query with BQL causes less probe effect than osquery.

and found that it uses more eBPF maps (one per `CAUSAL JOIN` operator) than the baseline, explaining the difference.

7.5 osquery

We now compare BQL to the closest prior work, osquery, in terms of probe effect imposed. osquery provides a wide range of tables to support observability queries, but focuses on OS state rather than events. Hence, osquery obtains most of its data from sources other than eBPF, such as `procfs` on Linux. We target osquery's `bpf_socket_events` table, which is based on a hand-written eBPF programs. This table projects 19 fields (and three osquery metadata fields) from eBPF programs attached to the enter and exit hooks for the `connect`, `bind`, `listen`, `accept`, and `accept4` syscalls. In this experiment, we run a query that calculates the average duration of socket-related system calls, grouping by the system call and process ID, similar to the `syscount-join` query in §7.3. We configure osquery to return results every five seconds. For BQL, we write views that select the same data from TCP events in the network stack, and run a query that reports the same data as the query we run in osquery, also at five second intervals.

We used a synthetic workload that establishes TCP connections in a closed loop on the loopback interface to stress both systems. osquery dropped data at the default load of 20k connections per second, so we reduced the offered load to 9k connections per second (BQL is able to handle the original 20k connections/second load). We run the experiment across 25 iterations and calculate the average connection establishment latency. We report the probe effect relative to the median of this value for the baseline across the 25 iterations. A good result would show that BQL's query imposes comparable or lower probe effect than osquery.

Figure 8 shows the results. osquery causes 5.3× higher median probe effect than BQL. This happens because osquery has a fixed table schema and must always extract all information necessary to populate its `bpf_socket_events` table from the kernel, while BQL can be more selective and extract only the information that a given query needs. Even though BQL's view contains the same fields as the osquery table, BQL's projection pushdown optimization removes eBPF code to extract fields that a given query never uses. This result shows that BQL's ability to perform optimizations across kernel and userspace reduces its queries' probe effect.

8 Related Work

Observability. Existing observability tools, including both those in the BCC collection [20] as well as other eBPF-based frameworks [7, 11, 46] rely on experts to manually write eBPF programs. Of these, osquery's [44] is the most similar to BQL, since it exposes a SQL interface. Two of the tables it provides gather their data from handwritten eBPF programs. We compared queries against one such table, containing a row per active socket, in §7.5, showing that osquery's inflexible handwritten eBPF programs cause high probe effect.

Bringing Databases to the OS. DBOS [55], OSDB [56], and BPF-DB [8] all use ideas from the database literature to interact with OS functionality. DBOS is an OS for compute clusters built on top of a database system. It represents OS state as tables and columns, so its implementation is tightly coupled to the schema it provides. In contrast, OSDB modifies the existing kernel's implementation to expose a relational schema. OSDB works by adding new syscalls that query a SQLite instance that OSDB adds to the kernel. Meanwhile, BPF-DB implements an in-kernel database that eBPF programs can use. BQL has complementary goals to DBOS and BPF-DB, and unlike OSDB, it works with an unmodified kernel; instead, BQL parses the kernel code to expose its hooks and their related data sources.

eBPF. Beyond observability, researchers have used eBPF to: re-implement existing kernel functionality more efficiently or flexibly [10, 25, 26, 28, 39, 60], or alternately offload custom application processing to the kernel [17, 50, 53, 62]. This wide variety of eBPF-based applications motivates continued research into improving eBPF programs' performance. Several approaches perform compile-time or run-time optimizations by e.g., merging eBPF tail calls [31], or stochastically exploring bytecode modifications [59]. Instead of focusing on eBPF programs directly, BQL's optimizations focus on the overall query's graph of operators. Ongoing work on improving eBPF programs' performance is thus complementary to BQL.

Declarative queries in other domains. The idea of introducing a declarative DSL to simplify extensibility is common across many domains, including networking [33, 35, 40], file systems [21], and system configuration [14]. BQL builds on this body of work but targets observability queries.

9 Conclusion

BQL makes writing eBPF observability queries easy. This benefits both observability engineers, who get a convenient and stable query interface they can easily adapt for their needs, and eBPF experts, who can support a wide range of queries with BQL's views instead of providing inscrutable off-the-shelf tools.

Evaluation shows that queries in BQL match the performance of hand-tuned baselines in all but one case; this case and our microbenchmarks raise questions about automatically optimizing eBPF programs, which we leave for future work. BQL shows that eBPF-based observability can be accessible to engineers without deep kernel or eBPF expertise through the careful design of stable abstractions.

We will open-source BQL.

10 *Master's Contributions*

F.N. contributed to the design and implementation of BQL's kernel schema. F.N. wrote a Linux kernel parser that organizes kernel objects and hookpoints and automatically generates BQL's schema. F.N. wrote BQL's views and queries that implement 35 out of the 57 BCC programs, identifying the necessary operators and abstractions to support these queries. Finally, F.N. helped design and implement BQL's expressivity evaluation with results found in Table 1.

References

[1] Valkey. https://valkey.io/.

[2] eBPF Applications. https://ebpf.io/applications/, 2026.

[3] Apache SkyWalking Authors. Apache SkyWalking: APM, Application Performance Monitoring System. https://skywalking.apache.org/, 2026.

[4] Apache Software Foundation. Apache datafusion sqlparser-rs: Extensible sql lexer and parser for rust. https://github.com/apache/datafusion-sqlparser-rs, 2024. Part of the Apache DataFusion project.

[5] Beyla Authors. Beyla: Zero-code automatic instrumentation with eBPF and OpenTelemetry. https://grafana.com/oss/beyla-ebpf/, 2026.

[6] BPFire Authors. BPFire: An Open Source firewall with eBPF applications addon. https://www.bpfire.net/, 2026.

[7] bpfman Maintainers. What is bpfman. https://bpfman.io/v0.5.5/#what-is-bpfman, 2024. Last accessed January 4, 2025.

[8] Matthew Butrovich, Samuel Arch, Wan Shen Lim, William Zhang, Jignesh M Patel, and Andrew Pavlo. BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications. 2025.

[9] Caretta Authors. Caretta: eBPF based Kubernetes service map. https://github.com/groundcover-com/caretta, 2026.

[10] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. eTran: Extensible Kernel Transport with eBPF. In *NSDI*, 2025.

[11] Cilium Authors. Cilium: eBPF-based Networking, Security, and Observability. https://cilium.io, 2026.

[12] Cong Wang. A Domain-Specific Programming Language for eBPF-Centric Development. https://github.com/multikernel/kernelscript, 2026.

[13] coroot Authors. coroot: Zero-instrumentation observability. https://coroot.com/, 2026.

[14] John DeTreville. Making system configuration more declarative. In *HotOS*, 2005.

[15] OpenTelemetry Developers. Opentelemetry demo. https://opentelemetry.io/docs/demo/, 2026. Last accessed March 31, 2026.

[16] Envoy Project Authors. Envoy Proxy. https://www.envoyproxy.io/.

[17] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached Using Safe In-Kernel Caching and Pre-Stack Processing. In *NSDI*, 2021.

[18] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[19] Brendan Gregg. Linux bpf superpowers. https://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html. Last accessed March 6, 2026.

[20] Brendan Gregg. Linux Extended BPF (eBPF) Tracing Tools. https://brendangregg.com/ebpf.html, 2016. Last accessed January 4, 2025.

[21] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Sqck: A declarative file system checker. In *OSDI*, pages 131–146, 2008.

[22] HUATUO Authors. HUATUO: eBPF for cloud native operating system observability. https://huatuo.tech/, 2026.

[23] Hubble Authors. Hubble: Network, Service & Security Observability for Kubernetes using eBPF. https://cilium.io/, 2026.

[24] Hugo Heyman and Jonatan Heyman and Joakim Hamrén and Carl Byström. Locust - A modern load testing framework. https://www.locust.io/.

[25] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *SOSP*, 2021.

[26] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *SOSP*, 2021.

[27] Kepler Authors. Kepler: Kubernetes-based Efficient Power Level Exporter. https://sustainable-computing.io, 2026.

[28] The kernel development community. Extensible scheduler class. https://www.kernel.org/doc/html/next/scheduler/sched-ext.html, 2026. Last accessed March 31, 2026.

[29] Kindling Authors. Kindling: eBPF-based Cloud Native Monitoring & Profiling Tool. http://kindling.harmonycloud.cn, 2026.

[30] kubectl trace Authors. kubectl trace: Schedule bpftrace programs on your Kubernetes cluster. https://github.com/iovisor/kubectl-trace, 2026.

[31] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. Verified programs can party: Optimizing kernel extensions via post-verification merging. In *EuroSys*, 2022.

[32] Libbpf-rs Maintainers. libbpf-rs: Safe rust wrappers for libbpf. https://github.com/libbpf/libbpf-rs, 2024.

[33] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, 2006.

[34] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–28, 2018.

[35] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[36] BCC Maintainers. Bcc bpf compiler collection. https://github.com/iovisor/bcc, 2024. Last accessed January 4, 2025.

[37] BPFTrace maintainers. bpftrace. https://github.com/bpftrace/bpftrace, 2024. Last accessed January 4, 2025.

[38] Ply Maintainers. Ply, a dynamic tracer for linux. https://github.com/bpftrace/bpftrace, 2024. Last accessed January 4, 2025.

[39] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing Linux with a Faster and Scalable iptables. *ACM SIGCOMM CCR*, 2019.

[40] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[41] netobserv Authors. netobserv: Flow based observability platform. https://github.com/netobserv/netobserv-ebpf-agent, 2026.

[42] Odigos Authors. Odigos: Zero-code distributed tracing via eBPF. https://odigos.io, 2026.

[43] OpenTelemetry eBPF Profiler Authors. OpenTelemetry eBPF Profiler: Whole-system, cross-language continuous profiler for Linux. https://github.com/open-telemetry/opentelemetry-ebpf-profiler, 2026.

[44] osquery Maintainers. Welcome to osquery. https://osquery.readthedocs.io/en/stable/, 2024. Last accessed January 4, 2025.

[45] Parca Authors. Parca: Continuous Profiling Platform. https://parca.dev, 2026.

[46] Pixie. How pixie uses ebpf. https://docs.px.dev/about-pixie/pixie-ebpf, 2024. Last accessed January 4, 2025.

[47] Pixie Authors. Pixie: Scriptable observability for Kubernetes. https://px.dev, 2026.

[48] pwru Authors. pwru: eBPF-based Linux kernel network packet tracer. https://github.com/cilium/pwru, 2026.

[49] Pyroscope Authors. Pyroscope: Continuous Profiling Platform. https://pyroscope.io/, 2026.

[50] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 780–794, 2022.

[51] Retina Authors. Retina: eBPF distributed networking observability tool for Kubernetes. https://retina.sh, 2026.

[52] RocksDB maintainers. RocksDB.

[53] Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. Automatic kernel offload using bpf. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 143–149, 2023.

[54] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *SIGCOMM*, 2023.

[55] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, et al. DBOS: a DBMS-oriented Operating System. 2021.

[56] Robert Soulé, George Neville-Neil, Stelios Kasouridis, Alex Yuan, Avi Silberschatz, and Peter Alvaro. OSDB: Exposing the Operating System's Inner Database.

[57] Tetragon Authors. Tetragon: eBPF-based Security Observability and Runtime Enforcement. https://tetragon.io/, 2026.

[58] wachy Authors. wachy: UI for interactive eBPF-based userspace performance debugging. https://rubrikinc.github.io/wachy/, 2026.

[59] Qiongwen Xu, Michael D Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. In *SIGCOMM*, 2021.

[60] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novakovic, Alex C. Snoeren, and Kimberly Keeton. PageFlex: Flexible and Efficient User-Space Delegation of Linux Paging policies with eBPF. In *USENIX ATC*, 2025.

[61] Zack Maril. Hancock. https://github.com/zmaril/hancock, 2024.

[62] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *OSDI*, 2022.