Optimizations for Performant Multiverse Databases

by

Jacqueline M. Bredenberg

S.B., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2020

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by .....
Robert Morris
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Optimizations for Performant Multiverse Databases

by

Jacqueline M. Bredenberg

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Modern web applications store data in backend databases, and access it through a variety of frontend queries. User permissions are implemented by checks on those queries, but a maliciously injected (or simply buggy) query can easily leak private data. Multiverse databases attempt to prevent these data leaks by creating a separate view of the database contents (or "universe") for each user, and enforcing in the backend that this universe contains only data that the user is allowed to query. These views are precomputed and materialized using a streaming dataflow system so that queries return promptly.

This design is difficult to make efficient. A simple approach makes copies of data and operators for each universe, but state size that increases proportionally to the number of users quickly becomes impractical. In this work, we developed optimizations for multiverse dataflow graphs, which aim to reuse the same computations (i.e. dataflow subgraphs) in many different universes while maintaining security invariants.

We evaluate these optimizations in the context of the HotCRP and Piazza web applications. The resulting graphs are about 2x more space-efficient and 3x more computation-efficient than the naïve ones. Graph size and processing time still scale linearly with the number of users, so our design may still not be efficient enough to be practical, but our optimizations make progress toward making multiverse databases a feasible solution to web application security.

Thesis Supervisor: Robert Morris
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I am immensely grateful to Malte Schwarzkopf for all phases of my work. Without Malte, it is unlikely that I would ever have written a Master's thesis. He inspired my interest in distributed systems and encouraged me to experiment with research in the field. He suggested multiple project ideas, eventually including the work presented here. Throughout my work, Malte has gotten me unstuck many times, and been consistently patient and supportive. I have been incredibly lucky to have him as my unofficial supervisor.

I am also thankful to Robert Morris for taking me under his supervision when Malte moved to Brown; to Jon Gjengset for providing detailed code feedback and Rust help; to Samyu Yagati for all her contributions to our policy language and planning ideas; and to the rest of the PDOS team for their discussions and work on Noria.

Contents

List of Figures

1 Introduction

From minor applications that we gave our emails to three years ago and then forgot about, to major web services like Facebook, websites' logic is often not bug-free. Data is often stored in a backend database and accessed by frontend queries. Current web services generally give these frontend queries full access to the data, meaning that a single poorly written query can leak data unrelated to its intended function. Facebook had one such leak in its photo sharing logic from 2013-2015 [2]: other applications granted access to certain photos could also view photos private to Facebook because a query checked only for the access token for photos and neglected to verify which application made the request. This is merely one prominent example among many. The goal of our work is to enable a secure and fast approach to web application queries.

Queries are prone to programmer errors and may leak private data. A better approach might maintain privacy with simple backend policies, guarding against buggy frontend queries. A system that automatically enforced privacy policies on all queries would ensure that no query could retrieve information that the querying user should not be able to see.

Multiverse databases are a new approach to improve security, developed in the context of the Noria dataflow system by the Parallel and Distributed Operating Systems (PDOS) group at MIT [8][5]. Each user has their own "universe" of data, containing only records they are allowed to view. All queries they submit to the system are evaluated against their universe. The initial multiverse database prototype copies the same data and operations into many different private universes, making it not performant enough to be practical.

In this work, we investigate a new approach to make multiverse databases more efficient and reduce their space footprint, primarily by reusing dataflow nodes across universes, while maintaining easy-to-reason-about correctness. We describe a simple initial multiverse graph planner, which lays out the dataflow for policies and queries over base tables, then present a series of optimizations. We evaluate our implementation of these optimizations, which reduces computation time by 67%-71% and state stored by 41%-66% overall.

The multiverse model has the potential to change the way that web applications approach security. However, no service will adopt a model that isn't performant enough to support their workloads. Our work demonstrates that the multiverse approach can continue to achieve its security goals while also becoming more practical for real systems. Websites could build on these optimizations when developing their own backends.

2 Background and Related Work

Over the years, there have been various approaches to support performant and secure logic for web application queries. For the sake of brevity, we focus on the ones directly relevant to our work.

2.1 Multiverse Databases

Traditionally, websites' backend databases do not address the problem of security. Frontend queries and application logic handle any privacy logic or restrictions, but there are many of them across the system and they are prone to human error, so a missing privilege check could result in sensitive data being leaked. Recently, the Noria team at MIT proposed to rectify this through multiverse databases [8]. A multiverse database presents a simple interface to define privacy policies all in one place. Policies may depend on computations over the database tables. The multiverse database then enforces the policies by anonymizing or removing data that a user should not be able to see, creating a universe of data tailored to that user, against which they can execute queries with no risk of accidentally revealing private records. The Multiverse DB vision paper illustrates this idea with Figure 2-1.

A multiverse database must either evaluate policies at the time of query execution, or precompute universe contents. The former repeats a lot of the same work and will slow down every query. The latter naively requires storing a large amount of data and recomputing every universe each time the underlying database receives an update. Fortunately, systems already exist that can make the latter feasible.

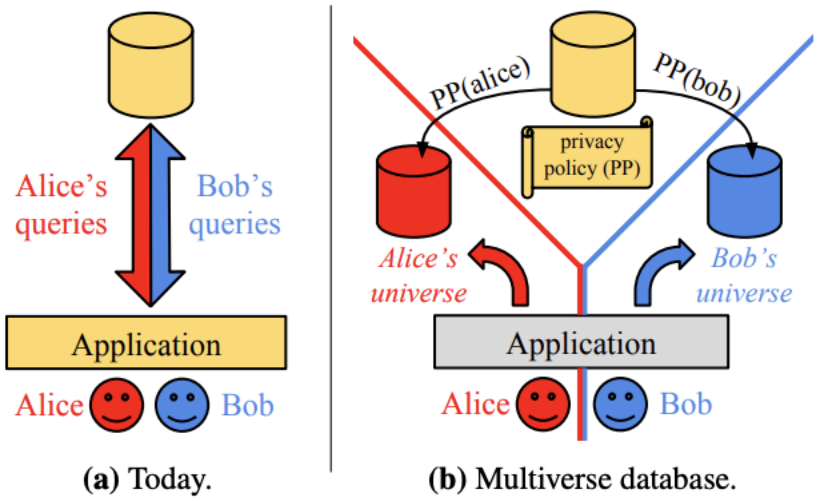


Figure 2-1: Multiverse databases as presented by Marzoev et al. [8].

2.2 Streaming Dataflow Computing

The current Multiverse DB prototype builds on streaming dataflow infrastructure from Noria [5], which it uses to precompute partial universe contents. Noria creates a stateful dataflow graph, where intermediate computations are stored and updated as new data is added to base tables.

Traditional databases repeatedly execute the same few queries every time users make requests, redoing mostly the same work many times. Queries can be much faster in Noria's model: instead of iterating over an entire table each time we wish to compute the sum of a column, we merely update the sum each time we add an entry to the table, and can access the stored value instantaneously. In complex queries with more nodes, updates to the data propagate through the operator nodes in the graph until they reach nodes where the query results are read. This model is especially powerful in web applications, where reads are usually much more common than writes, because we only need to update our stored results upon writes instead of executing a query for every read.

To clarify the idea of a dataflow graph, here is an example query from a benchmark based on the Lobste.rs website (https://lobste.rs), a news aggregator similar to HackerNews [4]. Lobsters is one of the benchmarks used by Noria [5].

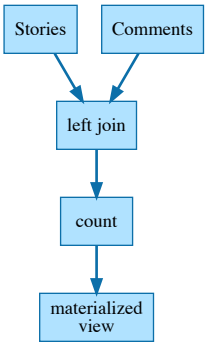


Figure 2-2: A dataflow graph for the example Lobsters query, with base tables for Stories and Comments.

Given the query:

```
SELECT stories.id, COUNT(comments.id) as comments
FROM stories
LEFT JOIN comments ON (stories.id = comments.story_id)
GROUP BY stories.id;
```

Noria generates a graph such as the one shown in Figure 2-2.

Whenever a user leaves a new comment on a story, the update propagates to the join and then the count so that the read result is up-to-date. If another query also requires joining the stories and comments tables, it can reuse the existing join node and add any additional operations as a child of the join. Sharing the join computation reduces the number of distinct nodes in the graph and the amount of work being done.

Noria itself builds on previous streaming dataflow systems, such as Naiad from 2013 [10]. Unlike previous work, Noria accepts online updates to the set of queries it supports. As new tables and types of queries are added, it dynamically adapts its dataflow graph, adding new computation nodes and reusing existing computation where possible.

2.3 Previous Approaches to Database Security

Several previous systems have attempted to enforce security policies in databases at runtime. For example, Qapla rewrites incoming queries to enforce policies before executing them, but this approach increases query complexity and slows the query down by 3-10x [9]. IBM's DB2 enables column masks to be applied, effectively anonymizing the data in the specified column [3]. The column mask approach does not support restrictions on access for entire rows or tables, however.

PostgreSQL allows a table owner to define row-level security policies on the table [6]. These policies are similar to the ones we attempt to enforce in multiverse databases. Each query must evaluate a boolean expression for every row at runtime before including that row in query execution, adding overhead to the query. This additional overhead to queries is a common theme among other approaches, and the multiverse approach attempts to avoid it by using streaming dataflow.

2.4 Dataflow for Multiverse Databases

When building a multiverse database, much computation is involved in constructing separate universes. Rather than perform these operations on the fly each time a universe is queried, the multiverse database design aims to keep query execution times fast by building on a streaming dataflow system. Noria, for example, will efficiently cache partial query results and universe computations. It will also propagate any updates to the underlying data so that only the relevant parts of the universe need to be recomputed.

Noria's flexibility in accepting online updates also provides support for multiverse databases. Each new user creates a new universe and therefore an update to the dataflow graph. Noria can dynamically adapt to new user universes without system downtime, whereas previous streaming dataflow systems would have needed to restart with a new graph.

2.5 Performance Limitations of Past Multiverse Work

While it improves security, the multiverse model is not yet performant enough to be a viable alternative to conventional database approaches. The current implementation creates a separate universe of data for each user and uses Noria to maintain intermediate computations as stateful dataflow. As a result, it stores too much data: separate copies of records and computations for each user. The multiverse approach will not be able to perform effectively until we build a way to reuse intermediate computations across different universes. At the same time, we must be careful to preserve the user-level separation of data so that we do not compromise security while reusing nodes.

2.6 Concurrent Research in Multiverse Databases

The PDOS team at MIT is researching ways to improve multiverse databases in various directions. The current prototype only supports security policies for reads; Alana Marzoev and Jonathan Guillotte-Blouin are developing a model for write policies [8]. Since Noria is only eventually consistent, special care needs to be taken with writes to avoid race conditions.

Samyu Yagati is working to solve the aforementioned performance problems by increasing sharing across universes with a new query planning approach [11]. In particular, she plans to rethink the traditional order of operators: whereas placing filters early in query execution is traditionally optimal to reduce the amount of data being processed, in a multiverse database those filters often specialize data to particular user universes and make it harder to share state, so it may be better to place them late. Our work draws on some of Samyu's ideas, but whereas she is developing a complex approach to graph planning, we use a simple planner and then develop generally-applicable optimizations to improve the graph.

3 Design

In section 3.1, we describe the policy language used to specify security constraints. Section 3.2 explains how we can construct a graph that correctly enforces these constraints. Section 3.3 describes the new dataflow operators designed for policy-related nodes, and how they differ from the existing Noria operators. The majority of our design work is in section 3.4, where we focus on optimizing the dataflow graph for faster performance and smaller state while maintaining the same behavior. Section 3.5 discusses an alternate approach to graph construction.

Our design's first priority is to be correct: a lightning-fast system that does not enforce the desired security policies would not meet our goals. Accordingly, we begin by constructing a dataflow graph that is slow but easy to reason about, and apply our optimizations incrementally to an existing dataflow. By arguing that each optimization preserves the output of the graph as a whole, we ensure that correctness is maintained.

This design choice restricts the space of optimizations that we can perform: some potential speed-ups may require constructing the graph differently[1] to begin with (see e.g. [11]). However, we believe that the focus on correctness is a reasonable trade-off. In section 3.6 we present some other ideas we considered for graph construction and cases where they break. These cases demonstrate the difficulty of building a correct initial graph, informing our decision to start simple and incrementally optimize our dataflow.

Throughout this section, we provide example images of how graph construction

[1]Although section 3.5 discusses an alternate graph construction, it has the same fundamental approach and merely uses different node combinations to achieve it.

and optimizations apply to a simplified version of the HotCRP benchmark (described in more detail in the Evaluation section).

3.1 Expressing Policies

Multiverse database managers express security constraints as a set of policies, each ranked with a priority. Each policy applies to a subset of users, and changes their permissions for entries in a given table matching a given SQL predicate. A policy can be an Allow, Deny, or Rewrite, respectively granting access, denying access, or anonymizing certain columns of the data. Policies are applied from lowest to highest priority, so that a higher-priority policy will override lower-priority ones – for example, if data is rewritten by a lower-priority policy and allowed by a higher-priority policy, the allow takes priority and affected users will see the data plain, but if the priorities are swapped they will see the rewritten version of the data.

The policy file also allows for defining global predicates and groups. A *global predicate* selects some data from a table and makes a view out of it, allowing it to be referenced within policy predicates. A *group* selects certain user IDs from the table of users, and policies can choose to apply only to users whose ID is in that group. Finally, the policy file allows specification of a default policy for all tables (either allow or deny), with maximally low priority.

As an example, in Figure 3-1 we present the simplified policy file for reviewer-related policies in HotCRP, a web application for conference review management [7]. Note that here "highest" priorities are ones with lower numbers, i.e. the 1st priority is more important than the 2nd priority, and that `$UID` gets replaced with the user ID value.

This policy language was largely developed for the Multiverse DB paper [8], but we continue to refine it in order to be expressive while also making it easy to plan. The combination of features described above allows a policy writer to encompass all the policies of HotCRP, Piazza, and other applications.

```
PolicySpecification(
  predicates: [
    ("REVIEWS_ASSIGNED", "SELECT ‘reviewId’, ‘paperId’, ‘contactId’ FROM PaperReview;"),
    ("MY_REVIEWS_ASSIGNED", "SELECT ‘paperId’ FROM PaperReview WHERE ‘contactId’ = $UID;"),
  ],

  groups: {},

  default: Some(
    Deny((
      priority: 999999,
      domain: Row,
      predicate: "ALL",
    ))
  ),

  policies: {
    "R0": Table((
      description: "R0: Reviewers can see reviews on papers they are assigned to review",
      table: "PaperReview",
      policies: [
        Allow((
         priority: 2,
         domain: Row,
         predicate: "‘paperId’ IN $MY_REVIEWS_ASSIGNED",
        )),
      ],
    )),

    "R1": Table((
      description: "R1: Reviewers cannot see reviewer information for any reviewer except self",
      table: "PaperReview",
      policies: [
        Rewrite((
         priority: 1,
         domain: Column,
         columns: Columns(["contactId"]),
         value: NULL,
          predicate: "ALL",
       )),

        Allow((
         priority: 0,
         domain: Row,
         predicate: "‘paperId’ IN $MY_REVIEWS_ASSIGNED AND ‘contactId’ = $UID;",
        )),
      ]
    ))
  }
)
```

Figure 3-1: A simplified version of reviewer-related policies in HotCRP. The policy specification defines a predicate `MY_REVIEWS_ASSIGNED` containing the paper IDs that each user will review, and allows the user to view paper reviews on papers with those IDs, but not the identities of other reviewers.

3.2 Initial Graph Construction

To ensure correctness, we begin by constructing a user-specific view of every table, and then make per-user copies of every query, reading only from the table views tailored to that user (see Figure 3-2). As long as the user-specific table views have correctly applied all security policies, the queries do not need to worry about leaking any data that the user should not be able to see.

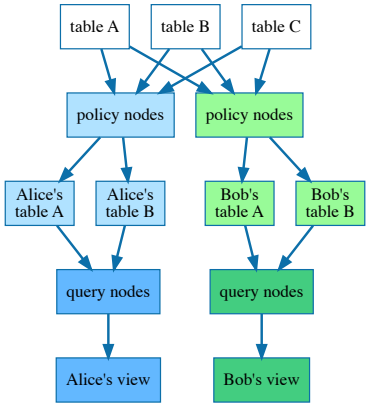
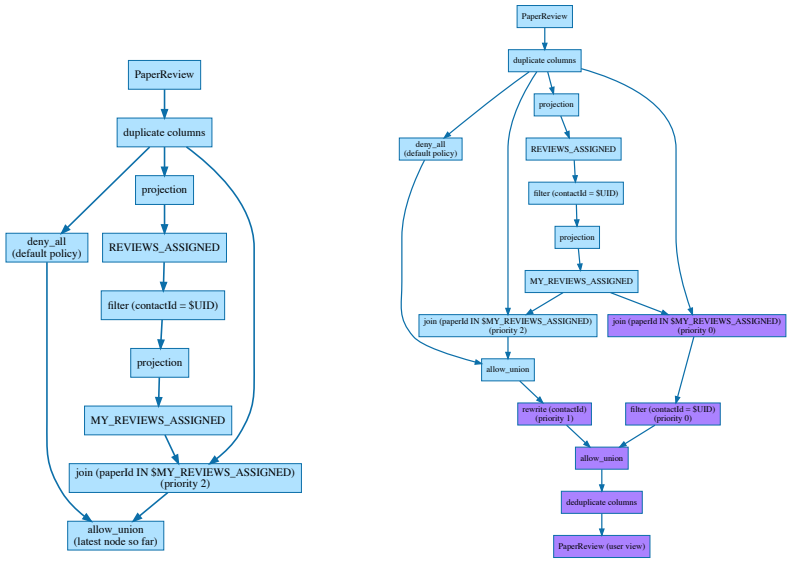


Figure 3-2: A graph with per-user copies of tables A and B, which are consulted by the query. Table C is used in evaluating policies but not queried. White nodes are base tables, blue nodes are for the user Alice, green nodes are for the user Bob, and the lighter nodes for each represent policy nodes whereas the darker ones are query nodes.

To construct such a view for a given table and user, we iterate over the policies that apply to that table, sorted from lowest to highest priority. We incrementally construct a chain of policy nodes, adding to the bottom each time we consider the next-lowest-priority policy, and keeping a pointer to the "latest" node (i.e. the last node in the policy chain so far). We update the latest node for each policy as follows:

- Deny: filter the latest node for rows that do NOT match the deny predicate. The latest node now points at this filter.
- Allow: filter the base table for rows that match the allow predicate, and union[2]

[2]Here we use a deduplicating union, which Section 3.3.2 will describe. In Section 3.5 we will see another approach to adding in allow policies that does not require deduplicating unions.



(a) Intermediate construction. (b) Full construction.

Figure 3-3: Constructing the view for the PaperReview table using policies R0 and R1 from Figure 3-1. The left figure shows the latest node after creating nodes for the predicates and policy R0, and the right adds to it with the purple nodes to create the complete graph. Note that the `IN` relation is implemented via a join.

these together with data already present in the chain's latest node. The latest node now points at this union.

- Rewrite: create two filters on the latest node, one for rows that do match the predicate and one for rows that do not match the predicate. Use a rewrite operator to anonymize the desired columns in the rows that do match the predicate. Union[3] the results together with the rows that do not match the predicate. The latest node now points at this union.

Thus at each step, the latest node incorporates all the policies with lower priority. Higher priority policies can overwrite the results of lower ones, because an allow brings in data directly from the base table bypassing any lower-priority denies and rewrites, and denies and rewrites remove information from the latest node.

One problem is that our policies often filter on data that may have been rewritten earlier in the chain of policy nodes. For example, suppose one policy anonymizes the

[3]Here our records are all disjoint, so we can use a regular union.

authors of papers, and another higher-priority policy denies anyone who is not an author or reviewer on a paper from seeing it. When we create policy nodes for the deny, we must filter on authorship, but all entries in the author column have already been rewritten to anonymous! To get around this, we duplicate every column in each table: one copy for determining whether the data matches filter conditions, and one copy to be potentially rewritten. Right before making the user-specific table view, we use a projection to keep only the potentially rewritten versions of the columns – see the `duplicate columns` and `deduplicate columns` nodes in Figure 3-3. This design could be made much more efficient by duplicating only columns that will actually be rewritten and then filtered on, but doing so is outside of the scope of the optimization work we discuss here.

The user-specific table view approach is not very efficient: it requires making per-user copies of every table and every query. Fortunately, later optimizations will mitigate this explosion in graph size as the system adds more users. In the meantime, we stick with this simple model as it is easy to reason about, and it makes it easy for multiple different queries by the same user against the same tables to reuse the policy computations for those tables.

Query planning is necessary for constructing the policy filters, the queries, and the predicates referenced by policies. For each of these cases, we call the same query planner logic, which converts a SQL select statement into a directed acyclic graph of dataflow nodes. Query planning has already been studied extensively, and our approach is not particularly novel, so we omit discussion of its correctness.

3.3 New Dataflow Operators

We use many of the same dataflow operators as Noria: aggregation, base table, extremum, filter, join (including inner, left, outer, and anti joins), projection, rewrite, and materialized view nodes. In addition, we present several new types of nodes that are useful in creating graphs for policies.

3.3.1 Deny Alls

The lowest priority policy will often be a universal deny, e.g. when the default policy is set to deny. In this situation, we need to create a latest node for the deny so that other policy nodes can chain downward from it as we continue layering on additional policies. (Note that this chain does not necessarily deny everything, since allow policies read from the base table and union into the chain.) Therefore, we create a Deny All node type, with the simple implementation of discarding all input rows. Since the node itself doesn't do anything, it can be optimized out later, but it is useful for constructing policy chains.

3.3.2 Deduplicating Unions

When we union together the results of multiple policies, both policies may allow viewing the same record, possibly with different rewrites. For example, the latest node in a policy chain may allow viewing a record with its author anonymized, and a new allow policy may allow for viewing the record in its entirety. It is incorrect to simply emit two versions of this record – instead we need to be able to merge the two into one record which contains all information that either side could see.

To solve this problem, we introduce the Deduplicating Union operator. It accepts any number of input sources, and outputs a single "deduplicated" version of each record that unifies information about it from all inputs. To determine when two records represent the same original record, we add an extra column called `dedup_index` to every table that labels records with the natural numbers, and project it back out when we are done constructing policy chains. Since this column does not exist in the real table, it will never be rewritten, so the Deduplicating Union can rely on it to determine whether two records are derived from the same table entry. Note also that the records in the user-specific table view (and each latest node during construction) are a possibly-rewritten subset of the records in the base table, without any aggregation or grouping operators, so the `dedup_index` column is always preserved.

The Deduplicating Union needs to remember what it has already seen for each

index, so it is necessarily a stateful operator. This creates an unfortunately large amount of state in policy construction, but combining it with Noria state-mitigation techniques such as up-querying may be able to make it substantially more efficient.

3.3.3 Gates

In some cases, a policy specifies that it only applies under certain conditions. We might have two different policy chains and need to switch between them depending on those conditions. A Gate operator acts as this switch: it has two possible input sources, and a third "control" input whose emptiness or non-emptiness determines which of the two inputs to read from.

We considered many situations where this abstract operator might apply, but in our current model it is only used for group policies. A group defines a view containing all the user IDs in the group. Then, a filter for the ID of the current user creates a control node which is nonempty if the user is in the group and empty otherwise. This allows us to construct policy components that only apply to users in a certain group, and use a gate to determine whether we should read from those components or from the policy chain not including the group-specific policy.

Gates also require state, since they must remember the contents of the other side of the gate in case their control input switches between empty and nonempty (e.g. a user is added or removed from a group). Group membership might change relatively infrequently in most practical applications, so up-querying in those situations would likely be a marked improvement.

3.4 Optimizations

Each optimization takes as input a list of graph nodes (and the edges between them) in topological order, and outputs another list in topological order. Our optimizations have the invariant that the records that reach the reader nodes of the graph are the same pre- and post-optimization. Thus it is safe to compose optimizations arbitrarily. Furthermore, since applying the optimizations does not require that the graph was

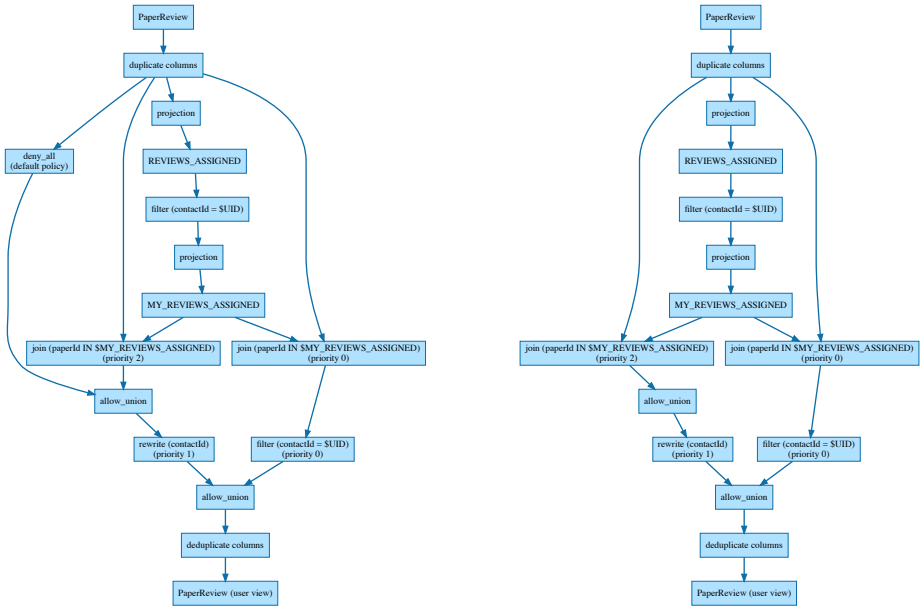
constructed in any particular way, this design is general-purpose enough to be useful even with a different approach to planning.

The graphs may contain nodes from many different user universes. Optimizations can combine some of these nodes to allow sharing state and computation across universes where possible.

Often, running optimizations in a certain relative order (or even running the same optimizations multiple times) allows them to take advantage of changes in the graph due to previous optimizations. Therefore, we will describe our optimizations in the order that they are first performed, and show diagrams of the impact that they have.

3.4.1 Dead Node Removal

There are several situations in which a naïve graph planning approach generates nodes that are never used and can safely be removed from the graph. In Figure 3-4, we show the construction of the PaperReview view before and after Dead Node Removal, which identifies that the Deny All node never emits any records and removes it.



(a) The original graph generated. (b) After Dead Node Removal.

Figure 3-4: Dead Node Removal eliminates a Deny All node.

In general, unused nodes include:

- **Deny Alls.** As described above, default deny policies will generate Deny All nodes, which could be completely removed from the graph without impacting its behavior. An optimization pass removes all edges to and from such nodes and omits them from the optimized list of nodes.

- **Orphaned Nodes.** An orphaned node is one that never receives records from upstream, e.g. because all of its parents are Deny All nodes. Children of these nodes which have no other parents will also never receive records. We include an optimization pass to remove orphaned nodes from the graph. By iterating over nodes in topological order, we ensure that we remove any orphaned parents before reaching their children, so that if those children are now left orphaned they will be detected by the same pass.

- **Childless Nodes.** A childless node is one whose outputs are never read or otherwise used by the graph. For example, if the planner generates a user-specific version of a table that is never queried, it is a childless node. If a node's only children are Deny All nodes, which will be removed by the above pass, it also becomes childless. To remove childless nodes, we iterate over nodes in reverse topological order so that if the parents of childless nodes become childless themselves, we will detect this in a single pass.

These optimizations remove dead nodes from the graph. In the case of childless nodes, they may also remove state from the graph that was being computed and never used. Deny Alls and orphaned nodes never perform computations or receive state in the first place, so their removal does not reduce state.

Besides the reduction in nodes, these optimizations enable planner design to focus more on correctness. There is no need to write more complicated code for edge cases where no node is needed; the planner writer can simply include inefficient constructs like the Deny All node or the user-specific version of an unused table, and trust that they will be removed later.

Some care must be taken when deleting nodes whose children expect multiple parents. For example, a Left Join whose parent is a Deny All may expect to still have two parents after any optimizations. In an early formulation of our design, removing one of its parents left it ambiguous whether the remaining parent was the right or left parent, which influenced correctness.

3.4.2 Identical Child Reuse

Often, multiple code paths will share several nodes with the same functionality. For example, two different policies (or queries) may begin by performing the same join. Of greater relevance to multiverse design, two different user universes may have identical components in their policy logic, neither of which makes reference to anything user-specific. In these cases, we wish to avoid duplicating state and computation by merging the two nodes into one.

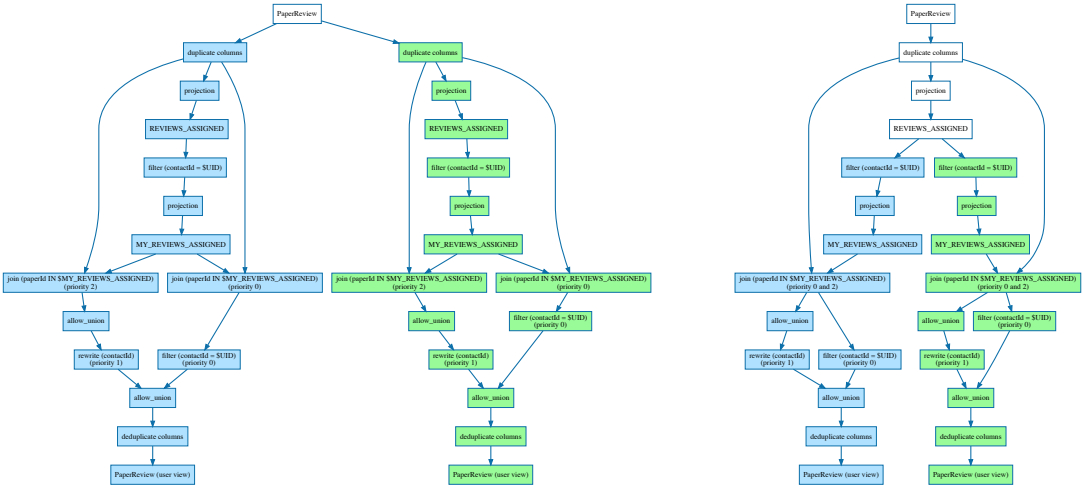
Figure 3-5 shows the pseudocode for this optimization. We iterate over all pairs of a node's children and check whether any two are equivalent[4]. If so, we manipulate pointers such that only the first is used in the graph, and remove the second from the list of nodes. By traversing the nodes in topological order, we ensure that higher nodes in an equivalent chain are merged before we consider lower nodes in that chain, allowing us to merge entire chains of nodes in a single pass.

We see the effects of this optimization in Figure 3-6. Note that we are reusing

[4] *Equivalent* nodes have the same parents and perform identical operations. They may have different child nodes from each other.

```
for node in topologically_ordered_nodes:
    for c1 in node.children:
        for c2 in node.children:
            if c1 != c2 and equivalent(c1, c2):
                remove ptrs from parents to c2
                change ptrs from c2's children to point at c1 instead
                mark c2 as removable from the topological order
filter topologically_ordered_nodes for unmarked nodes
```

Figure 3-5: Pseudocode for the Identical Child Reuse optimization.



(a) Before Identical Child Reuse. (b) After Identical Child Reuse.

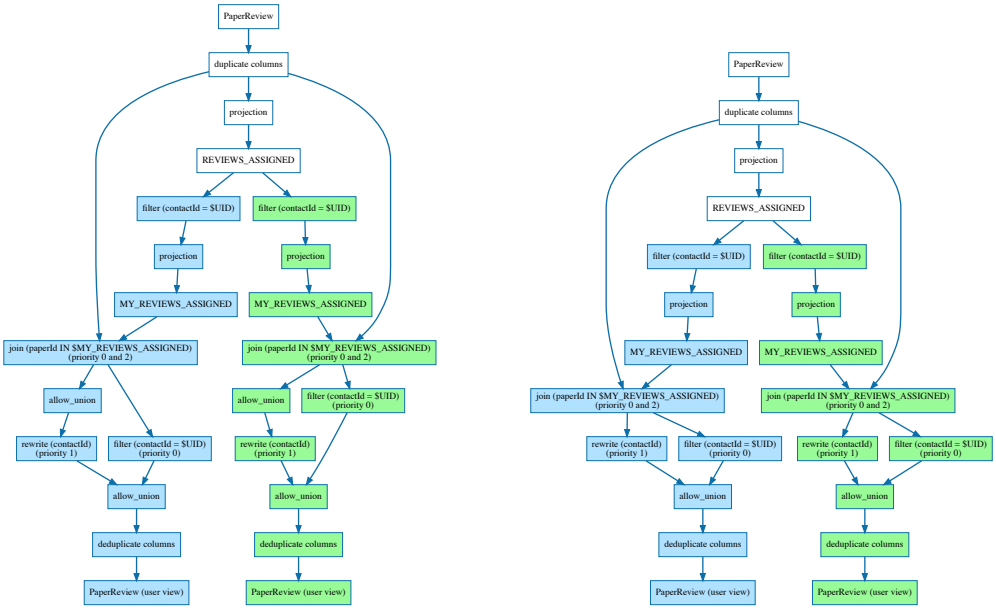
Figure 3-6: PaperReview view construction for two different users, blue and green, with Identical Child Reuse applied. Shared nodes are in white.

more than just the white nodes at the top: within each user universe, our optimization also reuses the identical joins below `MY_REVIEWS_ASSIGNED`.

3.4.3 Operator Simplification

This optimization detects when an operator can be combined or simplified in the context of the other nodes around it. It currently looks for the following cases:

- **Adjacent Filters.** Two filters are adjacent in the graph. They can be replaced with a single filter that checks for both of their conditions.
- **Adjacent Projections.** Two projections are adjacent in the graph. They can be replaced with a single projection that outputs the composition of the projection functions.
- **Trivial Unions.** A union or deduplicating union's parent is a dead node. If it has more than one parent remaining, this doesn't change anything. If it has only one parent left, the node can be removed and its parent connected directly to its children. An example is shown in Figure 3-7.



(a) Before Operator Simplification.

(b) After Operator Simplification.

Figure 3-7: Operator Simplification removes the `allow_union` node that used to have a Deny All parent.

- **Trivial Joins.** A join's parent is a dead node. In an inner join, or on the left side of a left join or antijoin, this makes the join dead too. In an outer join, or on the right side of a left join or anitjoin, the join becomes a projection where any columns coming from the dead node are always NULL. Since projections are stateless and faster to compute, this simplification is a significant gain.

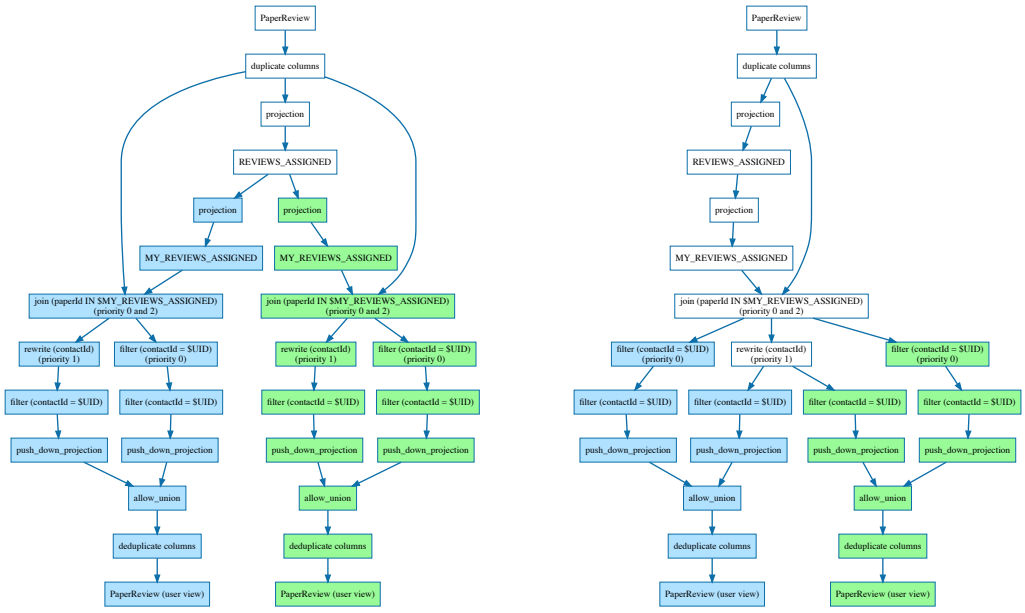
3.4.4 Filter Pushdown

In normal dataflow query plans, it is usually good to place filters near the beginning of the graph, because they reduce the amount of data that must pass through the rest of the graph. In multiverse query plans, dataflow often becomes user-specific by passing through a filter that depends on the user ID, so having these filters early in the graph necessitates having more user-specific nodes. More precisely, when our reuse optimization merges together policy chains from different users, it will have to stop merging when it hits a filter whose condition depends on the user ID because that filter will not be equivalent with its versions in different universes. Therefore,

more reuse optimization can occur if this user-specific filter is later in the graph.

We introduce the Filter Pushdown optimization to locate user-specific filters and reorder nodes so that these filters are later in the graph. We continue pushing the user-specific filters down until we reach a node which it may not be correct to push past – for example, a deduplicating union, since pushing the filter below it would filter all inputs to the union instead of just the one input. Since the filter conditions may refer to columns that are projected out later, we take special care to make sure that such columns are preserved as far down as the filters are relocated, and project them out only after the filter.

Notably, this optimization is not useful on its own. It moves nodes around but does not remove them, and by keeping projected columns around longer and adding a projection to remove them later, it increases the total amount of state and nodes in the graph. Its value is that it allows the reuse optimization to progress much further in merging equivalent operations.



(a) After Filter Pushdown without Reuse.

(b) After Filter Pushdown and Reuse.

Figure 3-8: Filter Pushdown by itself creates more nodes than the previous graph (Figure 3-7(b)), but it also enables more opportunities for reuse, including reusing the stateful join.

In Figure 3-8, we see that Filter Pushdown moves the `filter (contactId = $UID)` much lower in the graph, to just above the `allow_union`, and then follows it with a `push_down_projection` node to correct for any columns that we needed to keep around for the filter. Note that, confusingly, the `filter (contactId = $UID) (priority 0)` is not actually filtering on the same thing as the filter now placed below it, because they are referring to versions of the contactId column from different sides of the join. This figure has increased the amount of work done by the graph, but the second figure shows how we can gain significant savings by reapplying the reuse optimization. Most importantly, the join (one of the most state-expensive operators) now becomes shared.

It is important to keep in mind that here, pushdown entails the opposite of what it does in traditional database planning. Rather than pushing filters earlier, we are pushing them down later in the dataflow graph. In a non-multiverse model, this would be counterproductive, because putting filters early allows the database to reduce the amount of data it sends through other operators. In the multiverse model, that's still true, but we expect it to be outweighed by the savings of letting the database reuse operators across users.

After Filter Pushdown, it is sometimes the case that the same filter and associated projection have been pushed down multiple different paths to the same destination, resulting in multiple copies of them. Accordingly, we develop another optimization to push projections past filters, so that the copies of the filter are adjacent and the copies of the projection are adjacent. This allows Operator Simplification to merge these into a single filter and projection.

3.4.5 Summary of Optimization Pipeline

In total, our optimization pipeline currently makes 11 passes over the graph. Here is the top-level function, written in Rust:

```
pub fn optimize_all(nodes: Vec<MirNodeRef>) -> Vec<MirNodeRef> {
    let mut nodes = nodes;                          // mutable copy
```

```
    nodes = remove_deny_alls(nodes);                // 3.4.1
    nodes = remove_childless_nodes(nodes);          // 3.4.1
    nodes = remove_orphaned_nodes(nodes);           // 3.4.1
    nodes = reuse(nodes);                           // 3.4.2
    nodes = simplify_trivial_unions(nodes);         // 3.4.3
    nodes = simplify_trivial_joins(nodes);          // 3.4.3
    nodes = push_down_filters(nodes);               // 3.4.4
    nodes = reuse(nodes);                           // 3.4.2
    nodes = push_projections_past_filters(nodes);   // 3.4.4
    nodes = combine_adjacent_projections(nodes);    // 3.4.3
    nodes = combine_adjacent_filters(nodes);        // 3.4.3

    nodes                                           // return
}
```

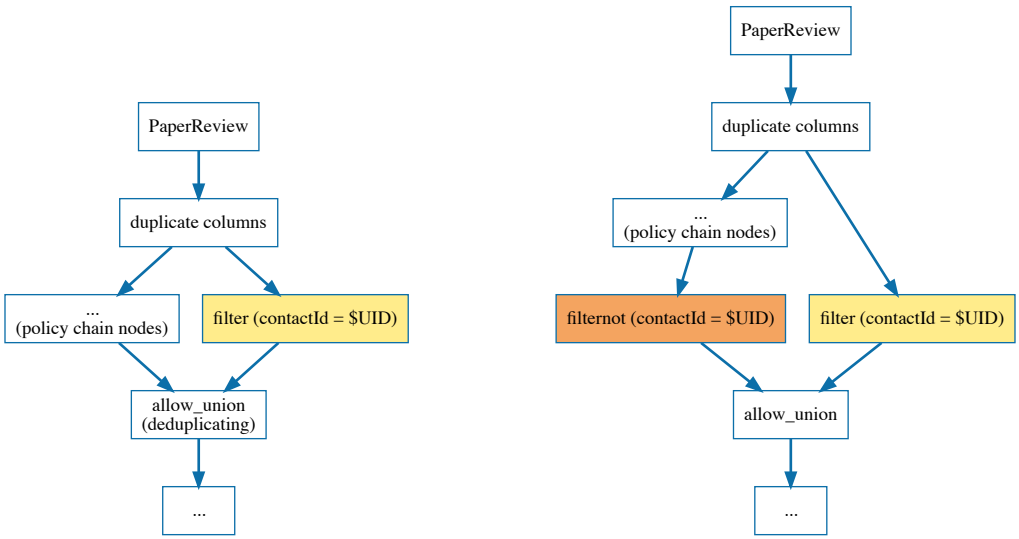
3.5 Avoiding Deduplicating Unions

In section 3.2, we described how our implementation builds policy chains, adding new nodes at the bottom of the chain for each successive policy. When the policy is an allow, we union together the newly allowed records from the base table and the latest node in the policy chain. Our stateless union operator, however, does not remember whether it has already seen a record, and so if a record is visible on both sides of the union, it will incorrectly emit duplicate copies. Furthermore, the two copies of the record may have had different rewrites applied to them, and we need to decide what version of the record should be visible.

Section 3.3.2 introduced deduplicating unions to solve the problem of multiple versions of a record from different policies. Unlike normal unions, deduplicating unions are stateful, so using them for every policy composition requires a lot of space. In this section we discuss how we might be able to avoid duplicate records without

using these stateful nodes, by modifying the policy chains to ensure that the two sides of a normal union are always disjoint.

An observant reader may notice that in practice, one version of the record (the one from the new Allow policy) is always unrewritten and therefore the version we want, and the other version could be discarded. If the Allow policy filters on a given condition, we could filter the latest node in the policy chain for records that do not meet that condition to discard those alternate versions. Then each record would appear on at most one side of the deduplicating union, and it would be safe to use a normal (stateless) union instead, as shown in Figure 3-9. We call this technique "complement filtering."



(a) With Deduplicating Union.

(b) With FilterNot and regular Union.

Figure 3-9: A policy chain adding an allow policy. We use ellipses to denote parts of the policy chain that come before and after, and show the filter condition of the new allow policy in yellow. The left graph shows how we would add to the chain with a deduplicating union. The right graph shows an alternative version that filters on the complement of the condition (shown in orange) and then uses a normal union.

At first glance, complement filtering seems like a straightforward improvement to graph construction: it replaces a deduplicating union (one large stateful node) with a filter and a union (two stateless nodes). Unfortunately, conditions can be

complex, and it is not always easy to filter on their complements. The filter node implementation computes the AND of its conditions, and taking the complement would require it to compute the OR instead. For example:

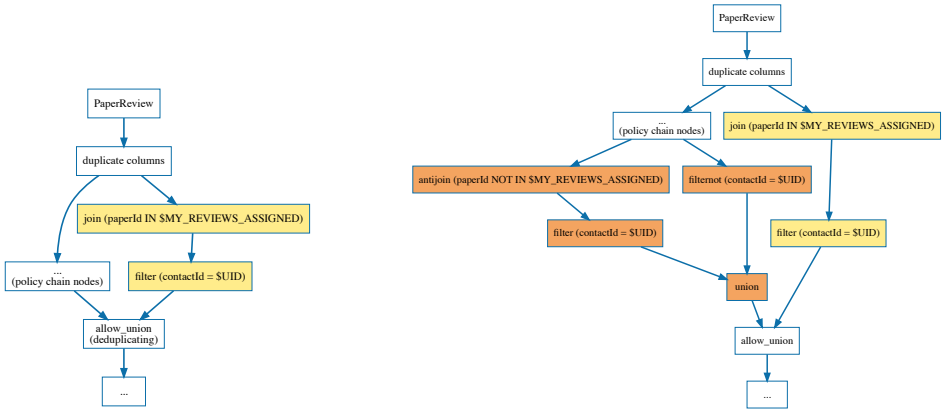
```
NOT(x = 5 AND y > 6) ==> (x != 5 OR y <= 6).
```

We resolve this by adding a new type of node, FilterNot, which passes through the complement of the records that a Filter node would pass through. This avoids needing to compute ORs for basic conditions (which could be done, but would require making our Filter implementation more complex).

We also have conditions that use the IN keyword, such as this one from policy R1 in Figure 3-1:

```
NOT(paperId IN $MY_REVIEWS_ASSIGNED AND contactId = $UID) ==>
(paperId NOT IN $MY_REVIEWS_ASSIGNED) OR (contactId != $UID).
```

We compute IN and NOT IN conditions with joins and antijoins, respectively, concatenated to the filter for the other conditions. The concatenation accomplishes AND, but computing the OR is trickier. We will need to union results together, and once again we must make sure we don't have duplicates of records. We can ensure that the two sides are disjoint by filtering on opposite conditions.



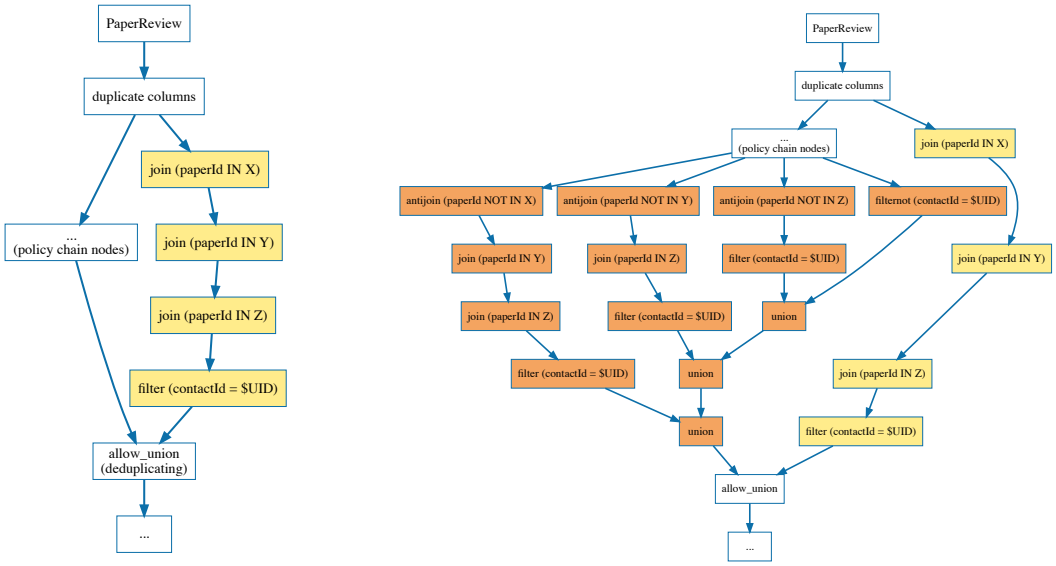
(a) With Deduplicating Union. (b) With complement filtering and regular Union.

Figure 3-10: A policy chain adding an allow policy that includes a single IN condition.

In Figure 3-10, we compute the antijoin, concatenate it with a filter for `contactId = $UID`, and then union this together with the FilterNot for `contactId = $UID` to make sure there are no duplicates.

Complement filtering with IN conditions is complex, and it's uncertain whether it's worth it: we need to compute an additional antijoin in exchange for removing the deduplicating union, so the total number of stateful nodes is the same. Which of these two stateful nodes is more efficient may depend on the policy and data as well as on the implementations of antijoins and deduplicating unions.

This is not the worst possible condition, however: in our applications we will see some predicates with multiple IN clauses, and in that case our technique of making sure the two sides of the union are distinct by filtering on opposite conditions can cause quadratically many stateful joins/antijoins in the number of IN / NOT IN clauses. Figure 3-11 shows how this can occur.



(a) With Deduplicating Union. (b) With complement filtering and regular Union.

Figure 3-11: A policy chain adding an allow policy that includes multiple IN conditions. Note the quadratic growth in the number of complement condition nodes versus original condition nodes.

We consider three options:

1. Just stick with the original deduplicating unions.

2. Assume that we will rarely have many INs combined in the same predicate, and proceed with complement filtering.

3. Stick with the original deduplicating unions whenever we have IN or NOT IN conditions, but whenever we only have normal filters, use complement filtering to eliminate the deduplicating union.

The right choice depends on the workload. We expect that in practice, the third option is likely a good choice because it reaps the strict benefits of removing stateful deduplicating unions in the simple case, but avoids introducing stateful joins.

3.6 Alternate Approaches

In our current planner, we construct user-specific views of each table, and then read from those for all queries. In earlier approaches, we considered interspersing policy and query logic so that all user-specific logic in either policies or queries would happen at the end, and all non-user-specific logic from both would happen at the beginning. Though tempting, constructing a graph in this way encounters several difficulties:

- Operators that include a group by (such as minimum, maximum, sum, count, top k, or distinct) will aggregate over many records, and so must happen after any filters on those records. Thus a large number of nodes will need to be placed after user-specific filters. It may be possible to mitigate this by doing a partial aggregation first, adding any columns to the aggregation's group-by that will be needed for filtering later, and then doing user-specific aggregations on the smaller dataset after the filter. It's unfortunately difficult to predict in each case whether this will improve the graph by reducing state, or worsen the graph by having an extra aggregation grouping by impractically many columns.

- Operators in the query may read columns that would have been rewritten by user-specific policy logic. For example, if a user wishes to query all posts written by a specific friend, but some of these posts would be anonymized by policy logic,

placing the non-user-specific query filter above the user-specific policy rewrite is incorrect: it will return all posts by that friend, and although some will be anonymized, the querier can infer their author from the fact that the posts matched the query.

Ultimately, the difficulty of reasoning about these and many other edge cases convinced us to prioritize a simple and correct core graph generator. We rely on our pushdown optimization (which pushes through the table view and down into query logic where possible) to enable much of the sharing gains that would be possible from planning things in a different order.

4 Implementation

4.1 Codebase

We developed a multiverse planner and optimizer in 8,022 lines of Rust source code plus 4,141 lines of test files. It turned out to be much easier to prototype our optimizations from scratch instead of working within the more complex Noria framework. We therefore implemented our own simple single-threaded dataflow system whose operators have identical semantics to Noria operators. However, this simplified implementation lacks support for partial materialization, multithreading, and sharding, focusing instead on the graph optimizer.

The author of this thesis contributed about 71% of the code by lines added, including all optimization work; Samyu Yagati contributed another 22% for her related work in planning; and Malte Schwarzkopf contributed the remaining 7% of code for policy specification parsing. We also relied on the nom-sql parser codebase that Malte developed for Noria.

4.2 User and Testing Interfaces

In our implementation, the user provides policy, schema, and query files for the database, and the planner constructs and optimizes a graph. Then, the user provides input data for each of the base tables. Each node propagates down positive and negative updates to its children. (A negative update revokes previously sent information that is no longer true; for example, a sum reports its new value as a positive update and its old value as a negative update, or a base table reports a removed

record as a negative update.) Then the user can ask the materialized views at the bottom for output values.

The unoptimized and optimized planners, as well as the planners with different approaches to deduplicating unions, are all subclasses of a generic planner interface. Tests are run against this planner interface, and report benchmark statistics (and optionally print graph visualizations) for each planner. Future approaches to graph planning could easily be added as additional subclasses of the interface.

4.3 Limitations

Our prototype does have several limitations. First, we are aware of a potential bug in Gate nodes where the parent-child relation between nodes is not symmetric, but empirically it has not yet caused any test failures, so we have prioritized developing optimizations rather than patching it. Second, we do not support `?`s in SQL queries, so the user must read all results from a materialized view instead of only those that match a certain key. We believe this does not have a significant impact on the amount of space or computation time used by our system.

Finally, whereas Noria emphasizes the ability to dynamically add queries to its dataflow graph, our prototype builds a graph once and then never adds to it. We expect it would not be difficult to add new tables or queries to our graph: it should be simple to construct the additional nodes and then re-run optimizations on the newly expanded graph. Adding new security policies would be more difficult, but could also probably be done without rebuilding the entire graph, by inserting the new policy nodes into the correct location (sorted by priority) in each policy chain.

5 Evaluation

We use several metrics for evaluation of our graphs: the number of nodes, number of records stored as state, number of total bytes of state, time to read output records, and time to process input records. In this section, we describe the relative merits of each, and how we expect optimizations to affect our measurements. We then introduce benchmarks based on two real-world applications, HotCRP and Piazza, and assess the impact of various optimizations on these benchmarks. We also investigate the impact of complement filtering on the same benchmarks. Finally, we evaluate the scalability of our approach, and conclude with some overall results.

5.1 Evaluation Setup

5.1.1 Hardware

We run all experiments on a 2016 MacBook Pro with 16 GB of memory and a 2.9 GHz processor.

5.1.2 Size: Node Count, Record Count, and Byte Count

We measure each of node count, record count, and byte count by having each node report on the size of its state (or in the case of node count, report 1 node).

We've discussed optimizations that reduce the node count of a graph, either by removing unused nodes or by merging multiple nodes together. It's not immediately obvious that this is a useful thing to measure: after all, removing these nodes often does not reduce the total amount of computation performed by the graph. However,

there is often significant overhead to communication between nodes, and locality allows us to process data more efficiently within a single node. For example, if we merge two filters together, the resultant filter node must still consider both of their filter conditions, but it is faster to process all the records by passing through them once rather than passing through them twice and transmitting them between the two nodes. In a larger system like Noria, we expect that sharding will increase the overhead of communication, making it even more impactful to reduce node count.

It's also important to measure the total space usage of the graph. There are two ways we might view this: number of records stored, or number of bytes stored. Both will be reduced when we reuse stateful nodes or when we simplify stateful nodes into non-stateful nodes. Both generally increase when we push filters later in the graph, since earlier filters remove much of the data before later nodes have to process it – an effect not captured by merely counting the nodes.

The byte count gives a more accurate estimate of the space footprint. It also reflects the negative impact of optimizations that make records larger, such as when Filter Pushdown avoids projecting out columns until later that are needed for the filter. Our byte measurement may be somewhat of an overestimate: strings stored as part of records in multiple nodes may actually share an underlying buffer that each node points to, but we count the full size of the string in each node.

The record count is more human-readable, allowing us to interpret the impact of optimizations more easily. Data is transmitted in record-sized chunks rather than a byte at a time, so if transmission time is dominated by the number of transmissions rather than their size, record count may also be a more relevant measure. Finally, some nodes (e.g. filters) perform operations that primarily interact with a single field of a record, and in those cases the number of records matters more than their sizes.

5.1.3 Record Processing Time and Read Time

When new records are added to base tables, we want to propagate them through the dataflow quickly. We expect that our optimizations will help us do so: for the same reasons described previously, condensing or reusing nodes will speed up processing,

and filtering on user-specific conditions later may slow it down. We anticipate that speedups in record processing will be similar to savings in our space measurements, but it is important to verify this effect.

Read time is critically important to web applications, and one of the biggest reasons why we use a stateful dataflow graph. However, we don't expect that our optimizations will have any impact on it: our materialized views should result in fast reads regardless of what's happening upstream. We therefore investigate this metric only enough to confirm that it is unchanged.

We measure times using Rust's `std::time::Instant`. The times measured for both record processing and reading should only be considered relative to each other. In order to make times large enough to be measurable, we process or read all records at once and measure the aggregate time. Since our simple implementation is not particularly optimizing for operator speed and does not use multithreading, we expect these times to be much slower than the corresponding times in Noria. It is nonetheless useful to measure record processing time as a proxy for the relative amounts of computation being performed in the graph pre- and post-optimization.

5.2 The HotCRP Application

HotCRP is a website developed by Eddie Kohler for "managing the conference review process" [7]. Authors submit their papers, and reviewers are assigned to review those papers. Before a paper is accepted, it is not visible to the general public; its authors and reviewers should be able to see it, and they should be anonymous to each other. Committee members can also see papers with the authors anonymized, and Chairs can see everything, except in the case of a conflict of interest.

These various factors in determining who can see what about a paper would be complex to incorporate into every query, but are easily expressed in our policy language. We construct a benchmark using these policies, the website's schema, and a query of the Paper and PaperReview tables.

Appendix A contains the policy specification for HotCRP. Figure 5-1 shows the

corresponding graphs with and without optimizations.

We use sample data scraped from the OpenReview.net website for our benchmark. The scraped data does not include reviewer identities, so we randomly generate those. We construct graphs with various levels of optimization and insert the sample data into the base tables.

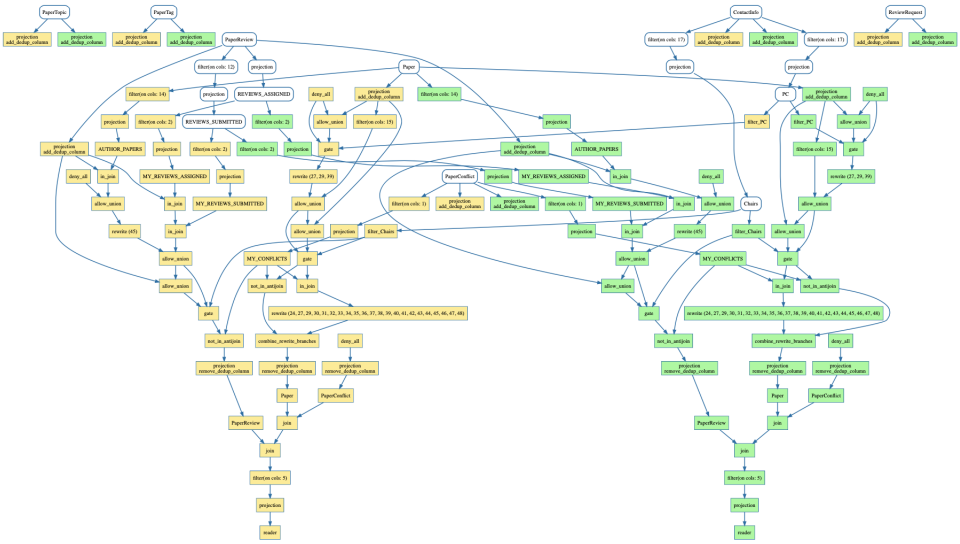
5.3 The Piazza Application

Many of our optimizations were developed while looking at a simplified HotCRP graph and considering how we might make it more efficient. We consider another application, Piazza, to see how well our optimizations generalize. We will see that some are not applicable to the Piazza dataflow graph, but we achieve significant gains nonetheless.

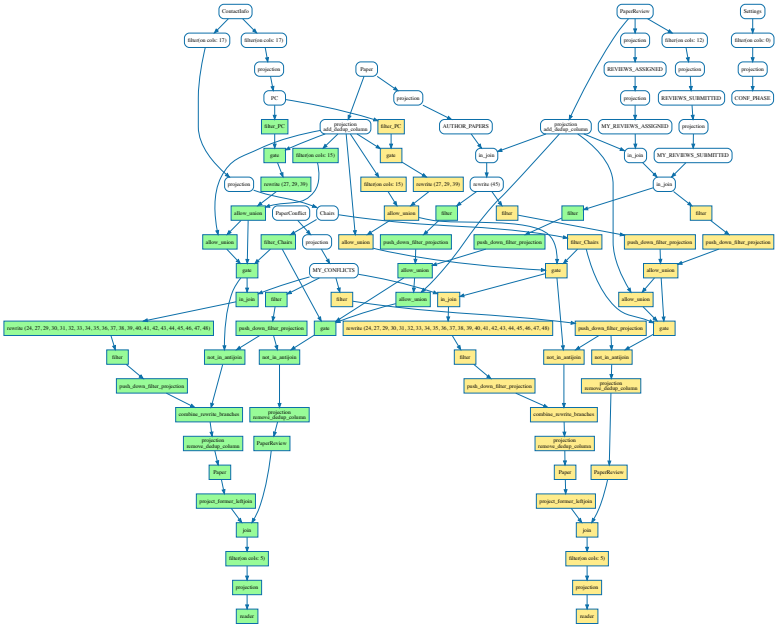
Piazza is an educational platform where various classes have a place to post questions and answers [1]. A user can create a post that's visible to a specific class. By default this is visible to everyone and the author is known, but users can optionally anonymize themselves and/or make their posts private (only visible to TAs and themselves). There are also restrictions on viewing who is a student / TA in a class; we put this data in a Roles table. Below we present a simplified version of Piazza, encoding the security policies that would describe the Posts and Roles tables.

Appendix A contains the policy specification for Piazza. Figure 5-2 shows the corresponding graphs with and without optimizations.

Since we do not have access to data from Piazza, we created a script to randomly generate content. Posts are generated as strings of random length between 0 and 20, and are authored by a random user in a random class.

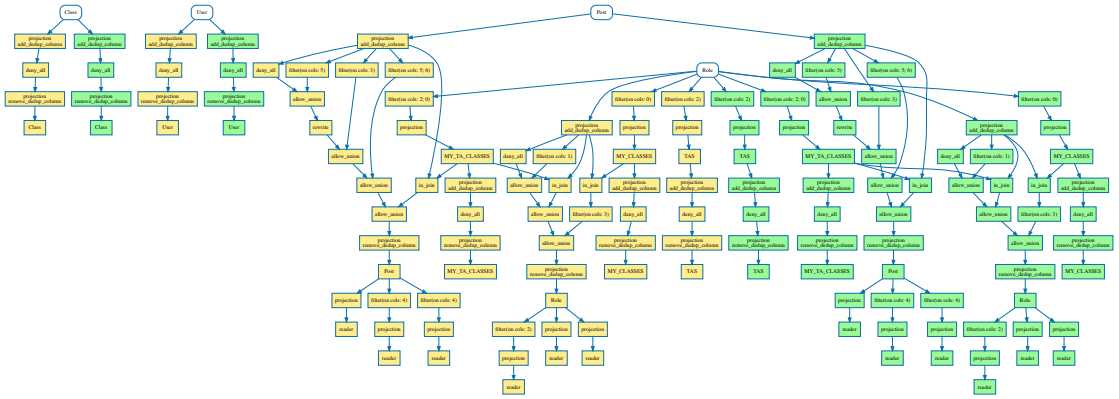


(a) HotCRP graph pre-optimization.

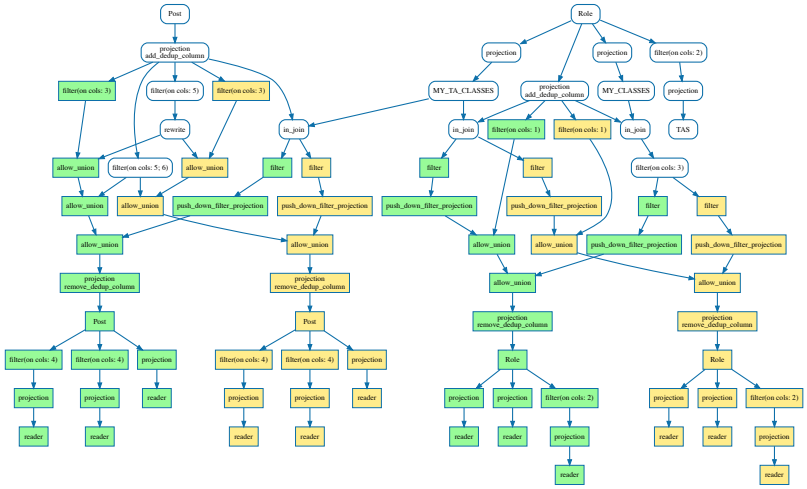


(b) HotCRP graph post-optimization.

Figure 5-1: Graphs generated for HotCRP with two users. These graphs are larger than the example graphs, but the main takeaway is that the graph optimizations reduce the size of the graph by roughly 2x.



(a) Piazza graph pre-optimization.



(b) Piazza graph post-optimization.

Figure 5-2: Graphs generated for Piazza with two users. These graphs are larger than the example graphs, but the main takeaway is that the graph optimizations reduce the size of the graph by roughly 2x.

Graph Optimizations	Nodes	Records	Bytes	Read Time	Processing Time
baseline values	2738	13318	1883519	0.26ms	927ms
+remove deny alls	-17.5%	0.0%	0.0%	2.9%	-0.9%
+remove childless	-20.3%	0.0%	0.0%	-3.8%	12.9%
+remove orphaned	-48.9%	0.0%	0.0%	-0.7%	-14.9%
+reuse	-6.2%	-8.6%	-5.7%	1.8%	-46.6%
+trivial unions	-2.4%	-0.5%	-0.3%	-0.6%	-4.2%
+trivial joins	0.0%	-0.1%	0.0%	-1.5%	-1.2%
+filter pushdown	19.0%	82.6%	50.6%	-0.5%	207.6%
+reuse	-26.8%	-59.3%	-55.0%	1.0%	-69.7%
+projection pushdown	0.0%	0.0%	0.0%	-0.6%	-2.3%
+combine projections	-2.7%	0.0%	0.0%	-0.4%	-1.0%
+combine filters	-2.8%	0.0%	0.0%	0.6%	4.8%
total pushdown	-17.6%	-25.7%	-32.3%	0.1%	-5.7%
total	-74.7%	-32.4%	-36.3%	-1.9%	-54.6%

Table 5.1: Performance of optimizations in the HotCRP application.

5.4 Impact of Each Optimization

5.4.1 HotCRP Measurements

We measure the impacts of each optimization on HotCRP using test data with 20 users and 20 papers. We repeat timing experiments five times and take the median. Empirically (over 24 different experiments) the median is within 7-15% of the minimum, but the max is sometimes more than 100% greater than the median, likely because of system fluctuations. Therefore we prefer the median to the average. We only run size benchmarks once; although sizes will change with different random data generations, we care only about the relative sizes after different optimizations, and these are not independent since they all use the same input data.

We now show the percentage improvements of each optimization over the previous set of optimizations (except the first row, which has baseline values instead of percentages). We also display the total improvement of all optimizations versus none. Finally, since the last five optimizations are all working together to make filter pushdown worthwhile, it is a bit unfair to show the numbers for filter pushdown alone; we aggregate those five in the "total pushdown" row.

Graph Optimizations	Nodes	Records	Bytes	Read Time	Processing Time
baseline values	1067	20007	545476	4.7ms	464ms
+remove deny alls	-10.3%	0.0%	0.0%	9.2%	-3.9%
+remove childless	-4.6%	0.0%	0.0%	-7.2%	0.0%
+remove orphaned	-21.5%	0.0%	0.0%	-5.9%	0.6%
+reuse	-20.2%	-28.1%	-22.6%	7.1%	-27.0%
+trivial unions	-2.9%	-2.4%	-1.2%	7.4%	7.8%
+trivial joins	0.0%	0.0%	0.0%	-6.7%	-9.1%
+filter pushdown	11.3%	26.5%	14.9%	-1.3%	139.2%
+reuse	-19.2%	-39.5%	-39.8%	-2.3%	-57.7%
+projection pushdown	0.0%	0.0%	0.0%	6.8%	1.7%
+combine projections	0.0%	0.0%	0.0%	-6.0%	4.9%
+combine filters	0.0%	0.0%	0.0%	7.2%	-7.4%
total pushdown	-10.1%	-23.5%	-30.9%	-3.8%	0.0%
total	-53.2%	-46.3%	-47.1%	-6.2%	-30.8%

Table 5.2: Performance of optimizations in the Piazza application.

5.4.2 Piazza Measurements

We measure the impacts of each optimization on Piazza using test data with 5 classes, 20 users, 2 TAs per class, 100 posts, and max post length 20 bytes. We again repeat timing experiments 5 times and take the median.

We display the same measurements as for HotCRP in Table 5.2.

5.4.3 Discussion

We observe that the reuse optimization does almost all of the heavy lifting in reducing state size and processing time, between its first pass and its second (post-filter-pushdown) pass. Several other optimizations (remove deny alls, remove childless, remove orphaned) reduce the number of nodes but do not remove any stateful nodes that were reached by any records, so they have no impact on state and no significant change in processing time. It is worth noting that many of the nodes removed are from tables that are never queried, particularly in HotCRP, which seems unlikely to be true in the real world.

In the Piazza application, there are several optimizations (trivial joins, projection pushdown, combine projections, combine filters) that do not apply anywhere in the

graph and therefore make no difference in the second chart. The trivial unions optimization makes a small but positive impact by cutting out one deduplicating union per user, plus one shared deduplicating union, that have Deny All nodes as parents.

In the HotCRP application, the removal of childless nodes appears to increase the processing time by 13%. Further runs suggest that this is a fluke in the data and processing time actually remains about the same. All optimizations apply somewhere in the graph (the only one that does not directly affect size is projection pushdown, which instead enables the combining of filters and projections after it).

Filter pushdown by itself increases the size of the graph (51% and 15%) and processing time (208% and 140%). In both cases, the increase is due to processing more records that would have been filtered out earlier – but why is the effect so much stronger on time than space? To answer this, observe that pushdown always stops before union or deduplicating union nodes, since it would not be correct to move a filter below those nodes. Since deduplicating unions are a considerable fraction of the stateful nodes in the graph, it follows that we are relatively more likely to push past stateless nodes than stateful ones. Processing more records in a stateless node will add to the time but not to the space. The reuse optimization following the filter pushdown appears to break even on time and net a 30% improvement on space.

As expected, none of the optimizations have a significant effect on read time, which remains roughly constant with some random noise.

5.5 Impact of Removing Deduplicating Unions

As discussed in section 3.5, there are three possible approaches:

1. **Baseline:** Always use deduplicating unions.
2. **AllDedupRemoval:** Always use complement filtering and regular unions.
3. **SimpleDedupRemoval:** Use deduplicating unions when there are IN / NOT IN conditions, and complement filtering plus unions when there are only simple conditions.

We show here how these approaches compare in the unoptimized graphs. Once again, these are medians over five samples. We use 20 users and 100 posts for Piazza, and 20 users and 20 papers for HotCRP.

HotCRP	Node Count	Byte Count	Processing Time
Baseline	2,738	1,883,519	905ms
AllDedupRemoval	3,078	1,310,345	864ms
SimpleDedupRemoval	2,758	1,882,059	871ms

Piazza	Node Count	Byte Count	Processing Time
Baseline	1,067	545,476	465ms
AllDedupRemoval	1,347	297,184	284ms
SimpleDedupRemoval	1,147	301,604	262ms

Table 5.3: Effects of different strategies for removing deduplicating unions on graph size and processing time.

We see that getting rid of unneeded deduplicating unions reduces byte count by up to 45% and processing time by up to 44% in Piazza. HotCRP does not benefit as much from the optimizations because it has mostly conditions with one or two joins, whereas Piazza has more simple filter conditions. AllDedupRemoval, the version that does convert deduplicating unions into extra joins and antijoins when necessary, nonetheless manages to reduce HotCRP byte count by 30%. This implies that the extra joins use less space than the deduplicating unions they replace.

Node count increases because of the extra joins, antijoins, and FilterNots, but in SimpleDedupRemoval (where we don't have the extra joins / antijoins) this is only a 7.4% increase.

AllDedupRemoval has a clearly greater impact in the HotCRP test case. SimpleDedupRemoval has better processing time in Piazza, so it may be preferable in that case. Neither of these use cases encounters more than two IN / NOT IN clauses in the same condition, so we do not run into the quadratically-growing nodes problem in AllDedupRemoval that SimpleDedupRemoval was designed to avoid; in other applications, this difference may be more relevant.

More research is needed to determine which of AllDedupRemoval and SimpleDedupRemoval is best for various use cases. Both are potentially good approaches.

5.6 Scalability

Finally, we investigate how the impacts of our optimizations scale with different numbers of posts and users. For the sake of brevity, we only consider Piazza. We measure how both space in bytes and processing time change as we scale up. The numbers reported are medians over five runs.

In the SimpleDedupRemoval and AllDedupRemoval frameworks, we run the non-pushdown optimizations twice over, because the first pass of operator simplification turns some antijoins into dead nodes which can be removed on the second pass.

We consider all combinations of optimization level and deduplicating union strategy. Optimizations levels can be any of (A) unoptimized, (B) using all optimizations except pushdown and the ones that run after it, or (C) using all optimizations. Preliminary results suggested that the pushdown optimizations may not scale well, which is why we perform these tests with and without them. Deduplicating union strategy can be any of (1) baseline, (2) AllDedupRemoval, (3) SimpleDedupRemoval. We abbreviate these options as 1A, ..., 3C in our tables for space's sake; see Table 5.4 for a key.

1A	Baseline, unoptimized
1B	Baseline, optimized (no pushdown)
1C	Baseline, all optimizations
2A	AllDedupRemoval, unoptimized
2B	AllDedupRemoval, optimized (no pushdown)
2C	AllDedupRemoval, all optimizations
3A	SimpleDedupRemoval, unoptimized
3B	SimpleDedupRemoval, optimized (no pushdown)
3C	SimpleDedupRemoval, all optimizations

Table 5.4: A summary of planner name abbreviations.

In Figures 5-3 and 5-4 we see that all of our planners have state sizes that scale linearly in the number of users and also linearly in the number of posts. The non-pushdown optimizations on the SimpleDedupRemoval and AllDedupRemoval graphs (3B and 2B) only remove stateless nodes, so the byte sizes are the same and we omit them from the figures. The full optimization suite saves about 60% of the state,

	1A	1B	1C	2A	2B	2C	3A	3B	3C
25 users, 100 posts	0.67	0.47	0.29	0.37	0.37	**0.19**	0.37	0.37	0.20
25 users, 200 posts	1.22	0.86	0.54	0.68	0.68	**0.36**	0.69	0.69	0.37
25 users, 400 posts	2.60	1.82	1.19	1.42	1.42	**0.79**	1.44	1.44	0.81
25 users, 800 posts	5.30	3.66	2.42	2.86	2.86	**1.62**	2.90	2.90	1.66
25 users, 100 posts	0.67	0.47	0.29	0.37	0.37	**0.19**	0.37	0.37	0.20
50 users, 100 posts	1.34	0.94	0.56	0.76	0.76	**0.38**	0.77	0.77	0.39
100 users, 100 posts	2.96	2.11	1.20	1.72	1.72	**0.81**	1.74	1.74	0.83
200 users, 100 posts	6.80	4.85	2.47	4.06	4.06	**1.68**	4.11	4.11	1.72

Table 5.5: Size scalability in Piazza: MB of stored state for various optimizations and deduplicating union combinations. The smallest state in each row is bolded.

	1A	1B	1C	2A	2B	2C	3A	3B	3C
25 users, 100 posts	157.7	243.0	205.9	267.9	443.7	347.4	291.6	**458.4**	376.2
25 users, 200 posts	176.8	241.4	219.0	274.6	442.2	347.8	289.5	**503.8**	379.3
25 users, 400 posts	152.5	229.9	185.8	293.9	**479.3**	387.4	298.9	459.5	382.9
25 users, 800 posts	116.8	171.7	161.1	272.0	**423.8**	336.0	291.6	416.0	358.5
25 users, 100 posts	157.7	243.0	205.9	267.9	443.7	347.4	291.6	**458.4**	376.2
50 users, 100 posts	67.6	106.7	78.6	105.8	199.0	106.8	126.9	**222.3**	128.2
100 users, 100 posts	35.3	55.8	28.9	58.9	98.1	37.0	59.6	**105.9**	37.8
200 users, 100 posts	15.2	24.3	7.6	22.8	39.9	8.8	24.5	**42.6**	8.7

Table 5.6: Processing throughput scalability in Piazza: posts processed per second for various optimizations and deduplicating union combinations, with highest bolded.

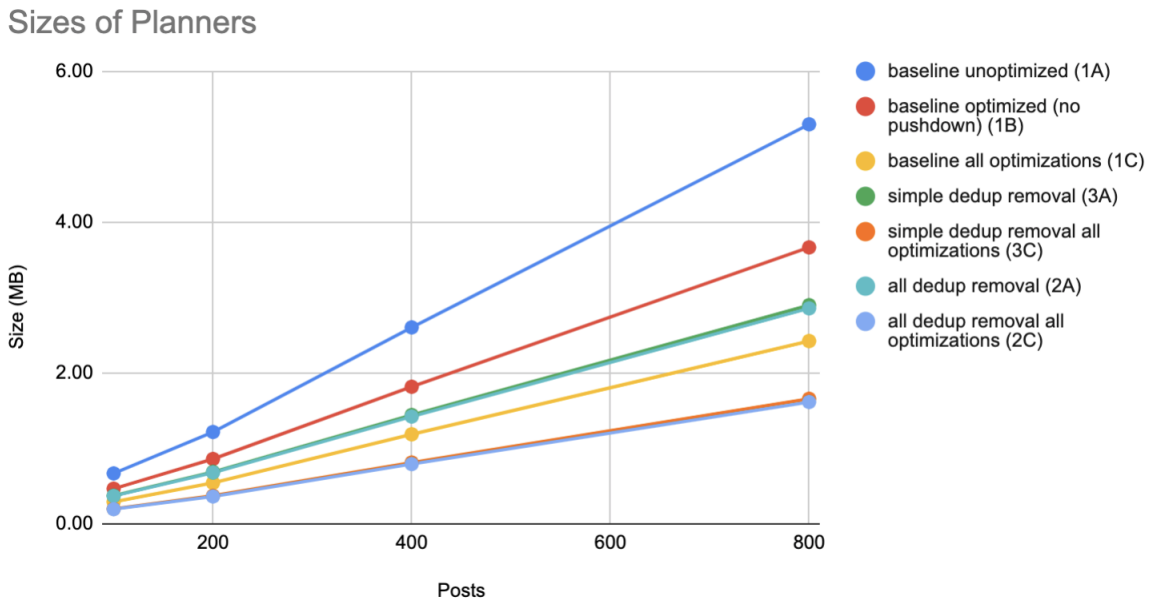


Figure 5-3: Planner size scaling with number of posts. Planners 2B and 3B have the same sizes as 2A and 3A respectively, so they are omitted.

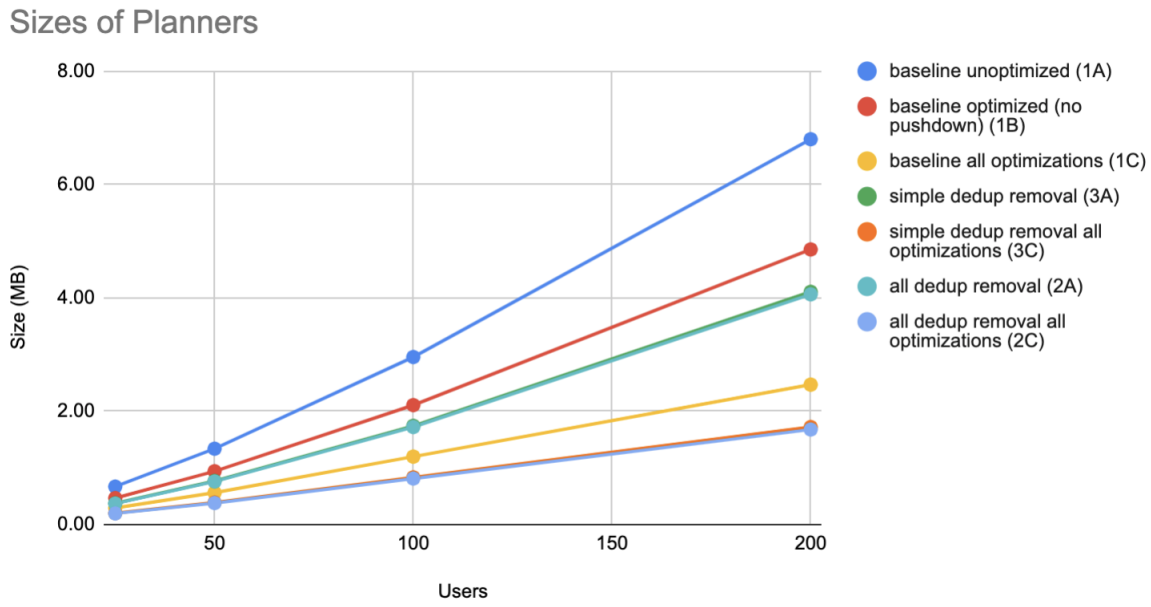


Figure 5-4: Planner size scaling with number of users. Planners 2B and 3B have the same sizes as 2A and 3A respectively, so they are omitted.

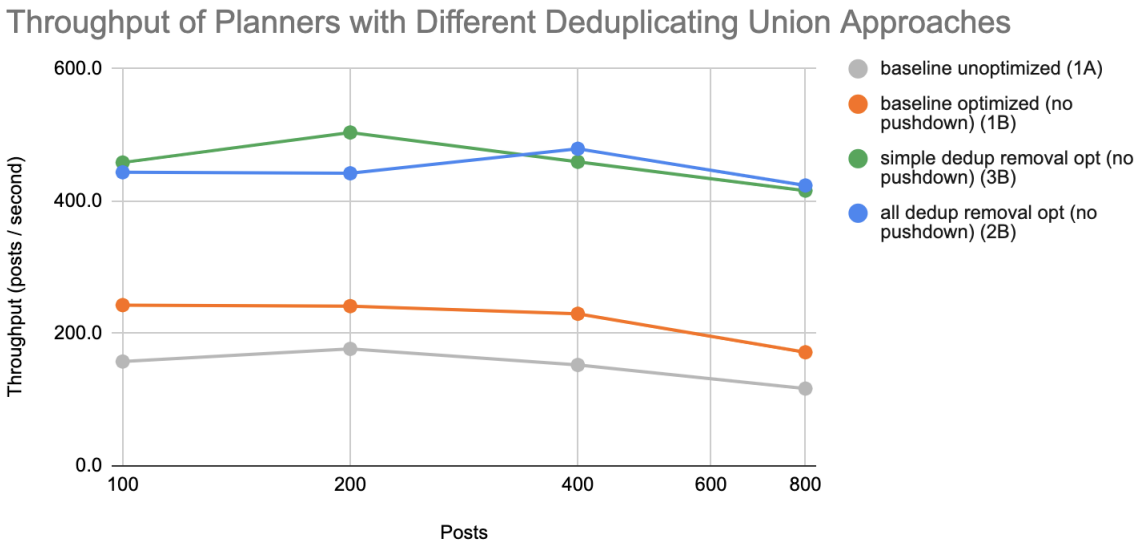


Figure 5-5: Planner throughput scaling with number of posts (log scale). We compare baseline, SimpleDedupRemoval, and AllDedupRemoval planners when each has non-pushdown optimizations applied, with the unoptimized baseline also present for contrast. Both DedupRemoval planners perform much better than baseline.

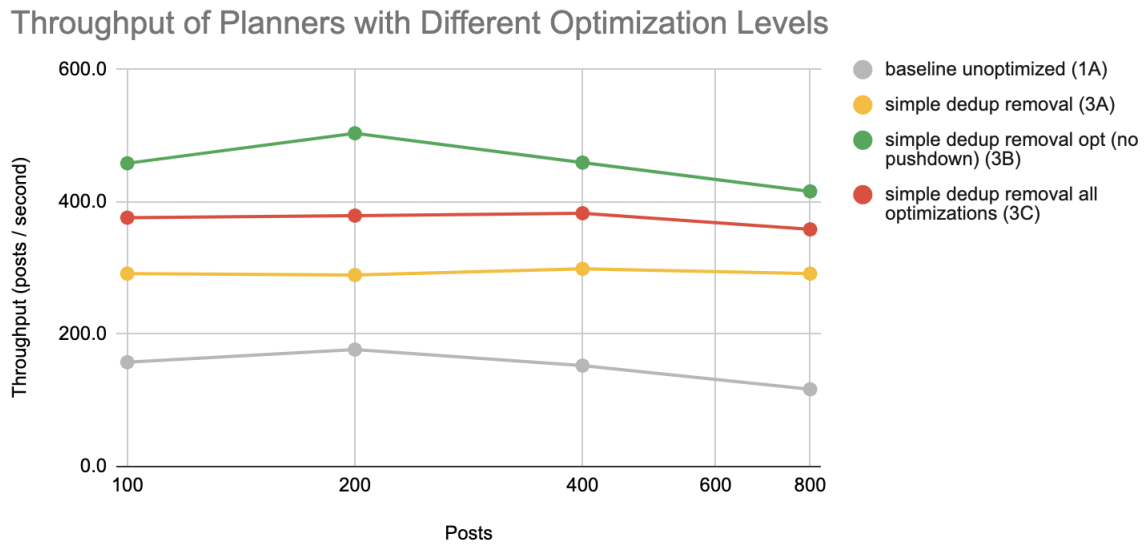


Figure 5-6: Planner throughput scaling with number of posts (log scale). We compare unoptimized, optimized without pushdown, and fully optimized planners within the SimpleDedupRemoval approach, with the unoptimized baseline also present for contrast. The non-pushdown optimizations perform best.

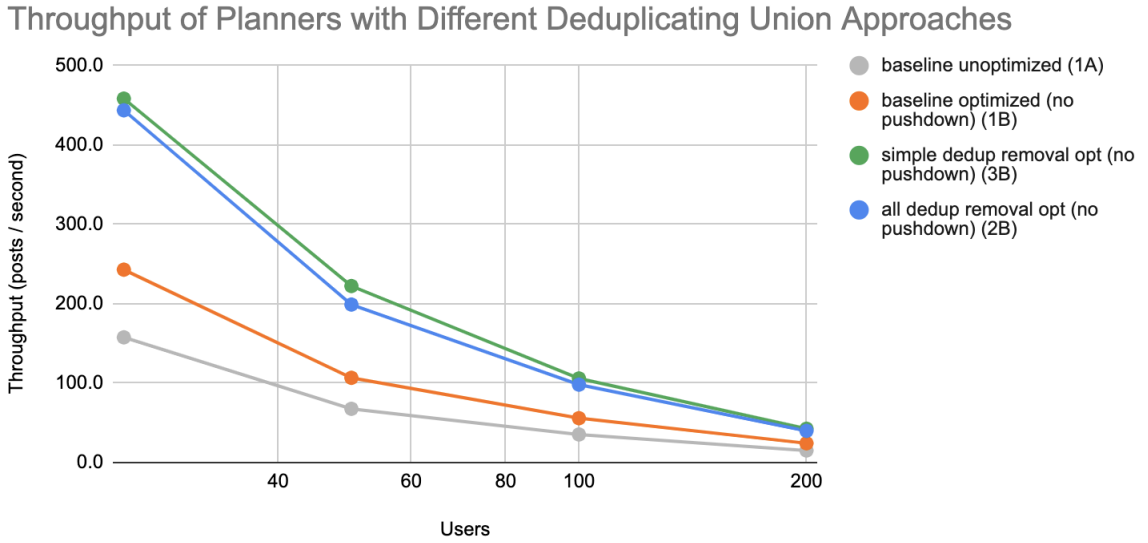


Figure 5-7: Planner throughput scaling with number of users (log scale). We compare baseline, SimpleDedupRemoval, and AllDedupRemoval planners when each has non-pushdown optimizations applied, with the unoptimized baseline also present for contrast. Both DedupRemoval planners perform much better than baseline.

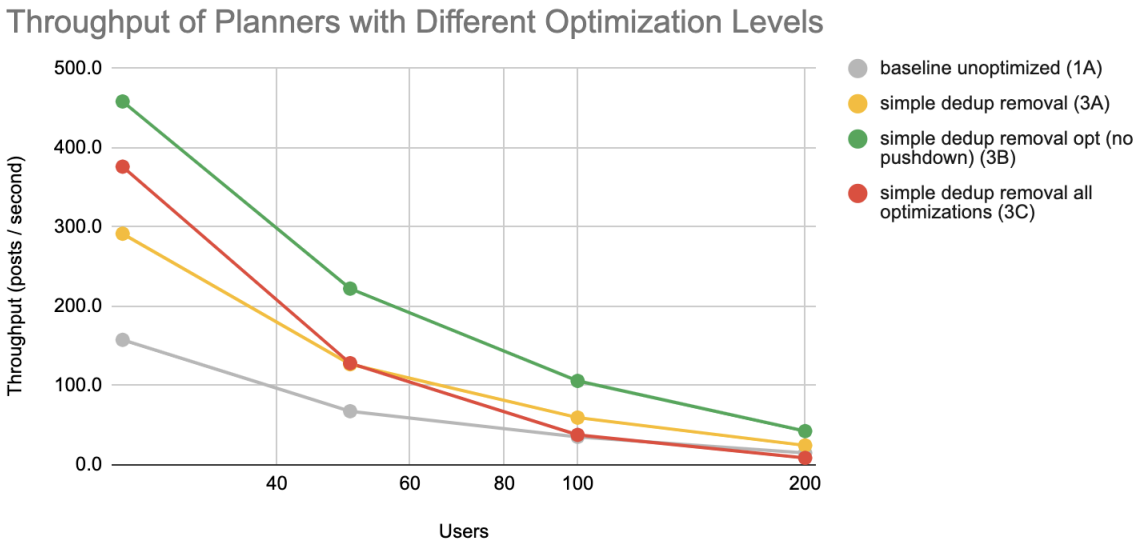


Figure 5-8: Planner throughput scaling with number of users (log scale). We compare unoptimized, optimized without pushdown, and fully optimized planners within the SimpleDedupRemoval approach, with the unoptimized baseline also present for contrast. The non-pushdown optimizations perform best; optimizations with pushdown scale poorly.

the non-pushdown optimizations save about 30%, and the SimpleDedupRemoval and AllDedupRemoval versions both save about 40% relative to the baseline planner. In total, the AllDedupRemoval with full optimizations (2C) saves about 70-75% of state size compared to the baseline (1A).

Next we measure the throughput of each graph in posts processed per second. In Figures 5-5 and 5-6 we see that throughput remains approximately constant as the number of posts scales. In Figures 5-7 and 5-8 we see that throughput drops as users are added, understandably: our single-threaded dataflow system processes fewer posts per second when performing the same computations in more universes.

SimpleDedupRemoval and AllDedupRemoval graphs have about 1.5-2x better throughput than their counterparts that always use deduplicating unions, and seem to scale about the same. The non-pushdown optimizations also consistently improve throughput by about 1.5x. Interestingly, the full optimization suite including filter pushdown does not scale well with the number of users: see the line for planner 3C in Figure 5-8. It starts out with 1.3x better throughput than the unoptimized 3A and

ends up with only 0.36x as much throughput. In Table 5.6 we can see that 1C and 2C have similar trends relative to 1A and 2A, starting off slightly better and ending with much worse throughput as the number of users grows.

Why might the filter pushdown optimization be scaling poorly in throughput but not in size? We hypothesize that the nodes that the user-specific filter is pushed past are now spending time processing data that corresponds to all the users instead of just one, and this creates greater overhead when there are more users. As we argued previously, most of these nodes are stateless, so this would increase processing time disproportionately more than state. This hypothesis is supported by the fact that the filter pushdown optimization results in a 140% increase in processing time in Piazza (see Table 5.2), much more than the 15% increase in byte count.

5.7 Overall Results

Having considered the impacts of individual optimizations, the efficiency of complement filtering, and the scalability of each approach, we now summarize the results of our approaches and discuss which is best in each scenario.

HotCRP Size (MB)	1	2	3
A	42.6	29.0	42.6
B	39.9	29.0	39.9
C	24.9	14.3	24.9

(a) Graph size in HotCRP.

Piazza Size (MB)	1	2	3
A	2.96	1.72	1.74
B	2.11	1.72	1.74
C	1.20	0.81	0.83

(b) Graph size in Piazza.

HotCRP Time (s)	1	2	3
A	20.4	19.5	20.5
B	8.5	6.6	9.3
C	9.2	5.9	7.7

(c) Processing time in HotCRP.

Piazza Time (s)	1	2	3
A	2.83	1.70	1.68
B	1.79	1.02	0.94
C	3.47	2.70	2.64

(d) Processing time in Piazza.

Figure 5-9: Heatmaps of the results for all combinations of optimizations and DedupRemoval approaches. The smallest sizes and times are greenest, and the worst ones are purplest.

Figure 5-9 shows heatmaps of sizes and times for the various combinations listed in Table 5.4. Since we have seen that scalability may be an issue for the pushdown

optimizations in (C), we use relatively large tests of 100 users and 100 posts/papers for both HotCRP and Piazza. (This results in much larger numbers for HotCRP than for Piazza, perhaps because the papers are much larger in bytes than the up-to-20-byte posts we generated for Piazza.)

We see that in HotCRP, pushdown optimizations seem to perform much better than they did in Piazza, even with 100 users. AllDedupRemoval improves both state size and time relative to the other two deduplication implementations. We therefore recommend approach 2C for this application, which reduces graph size by 66% and processing time by 71% relative to the baseline.

In Piazza, pushdown optimizations perform well in size but badly in time, as seen in the scalability tests. If we cared only about space, we could use approach 2C to improve graph size by 73% and processing time by 4% (with likely worse performance at larger scales) relative to the baseline. When placing a greater importance on time, approach 3B (SimpleDedupRemoval with non-pushdown optimizations) does best, reducing graph size by 41% and processing time by 67%.

We have seen that the best approach may depend on the application, on whether we prioritize time or size more, and on the amount of data we expect to handle. All of our optimizations have situations in which they are beneficial, and developing heuristics for when to apply them or not may allow us to do even better, which is out of the scope of this work.

6 Conclusion

We have described a series of topological graph optimizations that allow multiverse dataflow graphs to reuse and reduce nodes. We also explored approaches to avoiding the use of deduplicating unions in graph construction. After investigating the processing time, space usage, and scalability for each of these techniques, we conclude that it is likely best to use AllDedupRemoval for HotCRP and SimpleDedupRemoval for Piazza, and to use all optimizations except for the filter pushdown ones.

These optimizations improved the space usage of multiverse databases by an estimated 3x for HotCRP and 1.7x for Piazza, and reduced record processing time by 3.5x for HotCRP and 3x for Piazza. Since the optimizations were initially developed while considering the HotCRP use case, their performance in Piazza is good evidence that they may perform well in other applications as well. Our improvements make multiverse databases a more practical solution to the problem of security in web applications, although a truly practical solution would seek to avoid the linear growth in state size and drop in throughput that this approach experiences as the number of users increases.

7 Future Work

Although the evaluation of our prototype suggests significant potential speedups from optimizing multiverse dataflow graphs, it remains to be seen how much impact would be made on a real system. In the future, integrating these optimizations into Noria would allow us to test how they fare in a complex environment that includes sharding, multi-threading, and up-querying. Up-querying in particular would give us the opportunity to make our new operators more space-efficient, a direction of optimization that we have not focused on here.

It will also be enlightening to compare the results of our optimizations with other planning approaches. Samyu Yagati proposed a late-specialization model of multiverse planning [11]; when complete, we can compare results on various benchmarks, and determine whether optimizations are still useful when layered on top of her more elaborate planner. It's worthwhile to compare against the best manually constructed graphs for these situations, too, since humans are often better at figuring out efficient plans, and those plans may inform further optimizations.

Our optimizer currently only reuses nodes in cases where the state is the same for every user. In many situations, large groups of users would see the same state, but not all of them. (For example, all TAs for a class on Piazza can see the same posts, but not all users can.) In the future, we would like to investigate combining computations for everyone in such a group, and then using a gate at the end to decide whether a particular user sees the query result for that group or not. We anticipate that this will be complex but ultimately enable a significant amount of reuse.

The successes of both the AllDedupRemoval and SimpleDedupRemoval planners suggest that an intermediate approach may also be worth considering. SimpleD-

edupRemoval replaces deduplicating unions only when there are no INs or NOT INs to create joins or antijoins, but AllDedupRemoval sometimes does better by replacing deduplicating unions in every situation. Since the worst-case situations require quadratically many nodes to implement this replacement, it may be best to use an intermediate that replaces deduplicating unions only if there are at most some constant number of joins or antijoins involved.

Finally, as discussed in section 3.2, duplicating columns so that we never attempt to perform a policy filter on rewritten data is inefficient: it doubles the amount of state in policy nodes. A more intelligent heuristic for when column duplication is needed could therefore shave off up to half the state.

We developed many of our optimizations by looking at output graphs and envisioning how they might be improved, so we expect that other future optimization ideas may become apparent through application to more use cases.

A Policy Specifications

A.1 HotCRP Policy Specification

```
PolicySpecification(
  predicates: [
    ("AUTHOR_PAPERS", "SELECT ‘paperId‘ FROM Paper WHERE ‘leadContactId‘ = $UID;"),
    ("CONF_PHASE", "SELECT ‘value‘ FROM Settings WHERE ‘name‘ = ’phase’;"),
    ("REVIEWS_ASSIGNED", "SELECT ‘reviewId‘, ‘paperId‘, ‘contactId‘ FROM PaperReview;"),
    ("REVIEWS_SUBMITTED", "SELECT ‘reviewId‘, ‘paperId‘, ‘contactId‘ FROM PaperReview WHERE
        ‘reviewSubmitted‘ = 1;"),
    ("MY_REVIEWS_ASSIGNED", "SELECT ‘paperId‘ FROM $REVIEWS_ASSIGNED WHERE ‘contactId‘ = $UID;"),
    ("MY_REVIEWS_SUBMITTED", "SELECT ‘paperId‘ FROM $REVIEWS_SUBMITTED WHERE ‘contactId‘ = $UID;"),
    ("MY_CONFLICTS", "SELECT ‘paperId‘ FROM PaperConflict WHERE ‘contactId‘ = $UID;"),
  ],

  groups: {
    "PC": "SELECT ‘contactId‘ FROM ContactInfo WHERE ‘roles‘ = 1;",
    "Chairs": "SELECT ‘contactId‘ FROM ContactInfo WHERE ‘roles‘ = 2;",
  },

  default: Some(
    Deny((
      priority: 999999,
      domain: Row,
      predicate: "ALL",
    ))
  ),

  policies: {
    "A0": Table((
      description: "A0: Authors can see their own papers",
      table: "Paper",
      policies: [
        Allow((
          priority: 0,
```

```
      domain: Row,
      predicate: "‘leadContactId‘ = $UID",
    )),
  ],
)),

"A1": Table((
  description: "A1: Authors can see their paper’s anonymous reviews",
  table: "PaperReview",
  policies: [
    Rewrite((
      priority: 1,
      domain: Column,
      columns: Columns(["contactId"]),
      value: NULL,
      predicate: "ALL",
    )),
    Allow((
      priority: 2,
      domain: Row,
      predicate: "‘paperId‘ IN $AUTHOR_PAPERS",
    )),
  ],
)),

"R0": Table((
  description: "R0: PC members can see anonymized versions of all papers",
  group: Some("PC"),
  table: "Paper",
  policies: [
    Rewrite((
      priority: 1,
      domain: Column,
      columns: Columns(["leadContactId", "authorInformation", "collaborators"]),
      value: NULL,
      predicate: "ALL",
    )),
    Allow((
      priority: 2,
      domain: Row,
      predicate: "ALL",
    )),
  ],
)),
```

```
"R1": Table((
  description: "R1: Reviewers can see reviews on papers they are assigned to review once they submit",
  table: "PaperReview",
  policies: [
    Allow((
      priority: 0,
      domain: Row,
      predicate: "‘paperId‘ IN $MY_REVIEWS_ASSIGNED AND ‘paperId‘ IN $MY_REVIEWS_SUBMITTED",
    )),
  ],
)),

"R2a": Table((
  description: "R2: Reviewers can never info on conflicting papers, other than their existence",
  table: "PaperReview",
  policies: [
    Deny((
      priority: -100,
      domain: Row,
      predicate: "‘paperId‘ IN $MY_CONFLICTS",
    )),
  ],
)),

"R2b": Table((
  description: "R2: Reviewers can never info on conflicting papers, other than their existence",
  table: "Paper",
  policies: [
    Rewrite((
      priority: -101,
      domain: Column,
      columns: AllExcept(["paperId", "title", "abstract"]),
      value: NULL_OR_DEFAULT,
      predicate: "‘paperId‘ IN $MY_CONFLICTS",
    )),
  ],
)),

"C1a": Table((
  description: "C1: The PC chair can see everything, except for papers they’re conflicted with",
  table: "PaperReview",
  group: Some("Chairs"),
  policies: [
    Allow((
      priority: -1,
```

```
        domain: Row,
        predicate: "ALL",
      )),
    ],
  )),

  "C1b": Table((
    description: "C1: The PC chair can see everything, except for papers they're conflicted with",
    table: "Paper",
    group: Some("Chairs"),
    policies: [
      Allow((
        priority: -1,
        domain: Row,
        predicate: "ALL",
      )),
    ],
  )),
 }
)
```

A.2 Piazza Policy Specification

```
PolicySpecification(
  predicates: [
    ("TAS", "SELECT r_uid FROM Role WHERE Role.r_role=1;"),
    ("MY_TA_CLASSES", "SELECT `r_cid` FROM Role WHERE `r_role`=1 and `r_uid`=$UID;"),
    ("MY_CLASSES", "SELECT `r_cid` FROM Role WHERE `r_uid`=$UID;"),
  ],

  groups: {},

  default: Some(
    Deny((
      priority: 999999,
      domain: Row,
      predicate: "ALL",
    ))
  ),

  policies: {
    "P0": Table((
      description: "users are allowed to see public posts",
      table: "Post",
```

```
  policies: [
    Allow((
      priority: 3,
      domain: Row,
      predicate: "`p_private` = 0",
    )),
  ],
)),

"P1": Table((
  description: "author is rewritten unless the post is not anonymous",
  table: "Post",
  policies: [
    Rewrite((
      priority: 2,
      domain: Column,
      columns: Columns(["p_author"]),
      value: NULL,
      predicate: "ALL",
    )),
    Allow((
      priority: 1,
      domain: Row,
      predicate: "`p_private` = 0 AND `p_anonymous` = 0",
    )),
  ],
)),

"P2": Table((
  description: "users are allowed to see their private/anon posts they authored",
  table: "Post",
  policies: [
    Allow((
      priority: 1,
      domain: Row,
      predicate: "`p_author` = $UID",
    )),
  ],
)),

"P3": Table((
  description: "users are allowed to see private/anon posts from classes they TA",
  table: "Post",
  policies: [
    Allow((
```

```
          priority: 0,
          domain: Row,
          predicate: "`p_cid` in $MY_TA_CLASSES",
        )),
      ],
    )),

    "R0": Table((
      description: "users are allowed to see their enrollment information",
      table: "Role",
      policies: [
        Allow((
          priority: 1,
          domain: Row,
          predicate: "`r_uid` = $UID",
        )),
      ],
    )),

    "R1": Table((
      description: "users are allowed to see enrollment information of classes they TA",
      table: "Role",
      policies: [
        Allow((
          priority: 1,
          domain: Row,
          predicate: "`r_cid` in $MY_TA_CLASSES",
        )),
      ],
    )),

    "R2": Table((
      description: "users are allowed to see TAs of the classes they are enrolled in",
      table: "Role",
      policies: [
        Allow((
          priority: 1,
          domain: Row,
          predicate: "`r_role` = 1 and `r_cid` in $MY_CLASSES",
        )),
      ],
    )),
  }
)
```

Bibliography

[1] Piazza quick start guide. URL: https://piazza.com/pdfs/piazza_product_introduction.pdf (visited on 2020-05-04).

[2] Warwick Ashford. Facebook photo leak flaw raises security concerns. URL: https://www.computerweekly.com/news/2240242708/Facebook-photo-leakflaw-raises-security-concerns (visited on 2019-12-05).

[3] IBM Knowledge Center. Securing db2: Creating column masks. URL: https://www.ibm.com/support/knowledgecenter/en/SSEPEK_10.0.0/seca/src/tpc/db2z_createcolumnmask.html (visited on 2020-05-02).

[4] Lobsters Developers. Lobsters database schema (schema.rb). https://github .com /lobsters/lobsters/blob/93fe0fdd74028cf678134d6d112ae084d8fdd928/db/schema.rb.

[5] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. *USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, October 2018.

[6] Postgres Global Development Group. Postgresql 9.5.15 documentation: Row security policies. URL: https://www.postgresql.org/docs/9.5/ddl-rowsecurity.html (visited on 2020-05-02).

[7] Eddie Kohler. HotCRP. URL: https://hotcrp.com (visited on 2020-05-04).

[8] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. Towards multiverse databases. *Workshop on Hot Topics in Operating Systems (HotOS '19)*, May 2019.

[9] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, page 1463–1479, August 2017.

[10] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[11] Samyukta Yagati. Efficient privacy policies in multiverse databases. ACM SOSP Student Research Competition Entry, October 2019.