Multiverse Databases for Secure Web Applications

by

Lara Timbó Araújo

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

February 2018

Author . . . .
Department of Electrical Engineering and Computer Science
February 2, 2018

Certified by . . . .
Malte Schwarzkopf
Postdoc
Thesis Supervisor

Certified by . . . .
Frans Kaashoek
Professor
Thesis Supervisor

Accepted by . . . .
Chris Terman
Chairman, Department Committee on Graduate Theses

Multiverse Databases for Secure Web Applications

by

Lara Timbó Araújo

Submitted to the Department of Electrical Engineering and Computer Science
on February 2, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Most modern web applications authenticate users and enforce security policies in the application logic. Therefore, buggy applications can easily leak sensitive data. MultiverseDB addresses this problem in a new database architecture, where each user has her own private view of the database and declarative security policies restrict data-flow into a user's private universe.

To support multi-user universes, MultiverseDB builds on ideas of streaming data-flow systems and low-overhead materialized views. When a new user session starts, the system creates data-flow nodes to support the user queries and automatically inserts special nodes to enforce security policies at universe boundaries. MultiverseDB provides fast reads by storing the pre-computed results of user queries with policies already applied in incrementally maintained materialized views. To reduce space overheads created by these views and avoid redundant processing, MultiverseDB reuses views and allows system administrators to specify security groups for users subjected to the same security policies.

Thesis Supervisor: Malte Schwarzkopf
Title: Postdoc

Thesis Supervisor: Frans Kaashoek
Title: Professor

Acknowledgments

I would not have finished this thesis without the support and guidance of many people, whom I am deeply grateful to.

Malte has guided me through every step of the way in this thesis. From explaining weird Rust syntax to discussing new ideas for multiverse databases, he was always available and ready to help. His incredible attention to detail shaped the way I think about research and taught me more than I could have hoped for in a single year.

When things seemed a little bleak, Frans convinced me to stay and finish my MEng. Since then, he has supported and guided me through finishing a task that seemed impossible a year ago.

Every week during Xylem meetings, Eddie, Jon, Jonathan and Rob taught me that systems research is fun and exciting and that, with the right people, weekly morning meetings are also fun and exciting.

At last, I want to thank my parents and my brothers for always telling me that things will work out, that I am smart enough to make them work out and that, even if they don't work out, I am loved.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most modern web applications require fine-grained access control policies to restrict the data accessed by each user. For example, an application like the class Q&A platform Piazza[1] may only show posts from classes that a user is enrolled in; it may also restrict the visibility of a post to the class staff; and it may compute aggregate statistics over all posts, instead of only those visible to the user.

Web applications usually authenticate users in the application logic and share a single database backend connection across many user sessions. It thus falls to the frontend application logic to enforce *security policies*. This design compromises application security, since the entire database might be exposed if an attacker exploits a security breach in the application logic or compromises the frontend server.

Moreover, this design burdens developers with the responsibility to consistently enforce security policies in application queries. Mistakes in any query can lead to a compromise of user privacy by, for example, exposing private posts directly, or leaking their existence through aggregate statistics. Even if all queries are correct, changes to the security policies require developers to manually and painstakingly modify many queries spread across the application code.

MultiverseDB addresses both of these problems by (i) specifying security policies in a centralized place, and (ii) enforcing them at the database level, rather than in the application logic.

[1] `https://piazza.com`

1.1 Approach

Instead of all users sharing the same database backend, each user session in MultiverseDB creates a *"parallel universe"*, an individual view of the database which contains only data that the security policies allow that user to see. Within her universe, however, a user can issue *any* query without violating the application's security policies.

MultiverseDB is implemented on top of Xylem (see Section 1.2) and uses a data-flow graph of operators and materialized views to execute end-user queries and cache their results. As one of its inputs, the system takes a set of declarative and easily audited security policies that determine what data goes into user universes.

When a new user session starts, MultiverseDB modifies the underlying data-flow graph to build an individual version of the queries for that session. The system detects which policies are relevant to the queries and inserts special *security nodes* at universe boundaries. Security nodes enforce the relevant policies and restrict data flow into the views that can be read by the user.

The MultiverseDB approach is more secure than sharing a single database connection and applying security policies in application queries. A user universe reveals only data that the user is privileged to access, so a compromised or buggy application cannot expose data that should not be viewed by the current active session.

For example, MultiverseDB can prevent a password disclosure bug found in HotCRP [9], a conference management system. In HotCRP, a user is allowed to send a password reminder to another user's email address. This feature, when used in combination with HotCRP's email preview mode – which displays emails to the current session, instead of sending them – allowed a user to preview emails containing the passwords of other users [13], because the preview query did not apply the security policies (see Figure 1-1).

Figure 1-2 shows how MultiverseDB's architecture prevents this bug. In MultiverseDB, the buggy application is unable to leak other users' passwords, since a user's universe stores only her own password. To send the email containing another user's password, the current user session sends a request to a server-side email service. The email service also has its own database universe, but it stores all passwords (though it may not see other

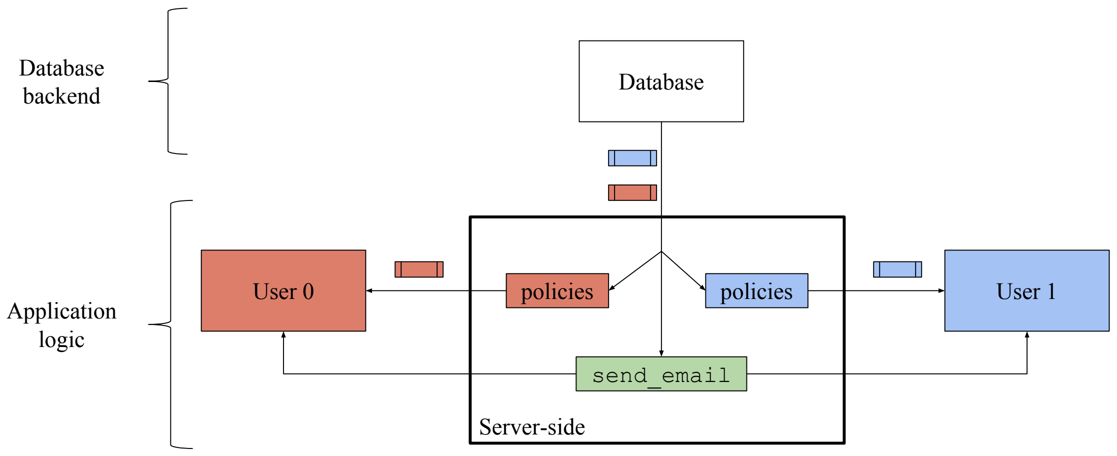


Figure 1-1: Traditional architecture: a buggy application can leak red data to the blue user or blue data into the red user through `send_email`*, which has access to both blue and red data.*

data irrelevant to emails). The service is then able to send the email without ever sharing password information with the buggy end-user application frontend.

1.2 Xylem: data-flow and low-overhead materialized views

Xylem is a database-like storage backend for web applications which combines ideas from incremental data-flow and relational materialized views.

In Xylem, applications pre-declare a *query schema* that includes base tables definitions and queries that specify views for reading. Given a query schema, Xylem implements it using a data-flow graph of operators and materialized views. Upon receiving a write, Xylem inserts it into the appropriate base table and feeds it into the data-flow graph. The internal nodes in the graph are relational operators, such as aggregations, filters or joins between nodes, and leaf nodes constitute materialized views. The writes propagate through the graph until they reach a leaf, where the precomputed query results can be read efficiently by the client application.

Xylem query schemas are dynamic and can change as the application evolves. When the query schema changes, Xylem dynamically updates the data-flow graph to implement the new schema. Xylem also supports live migrations i.e. it is able to service client requests during query schema migrations.

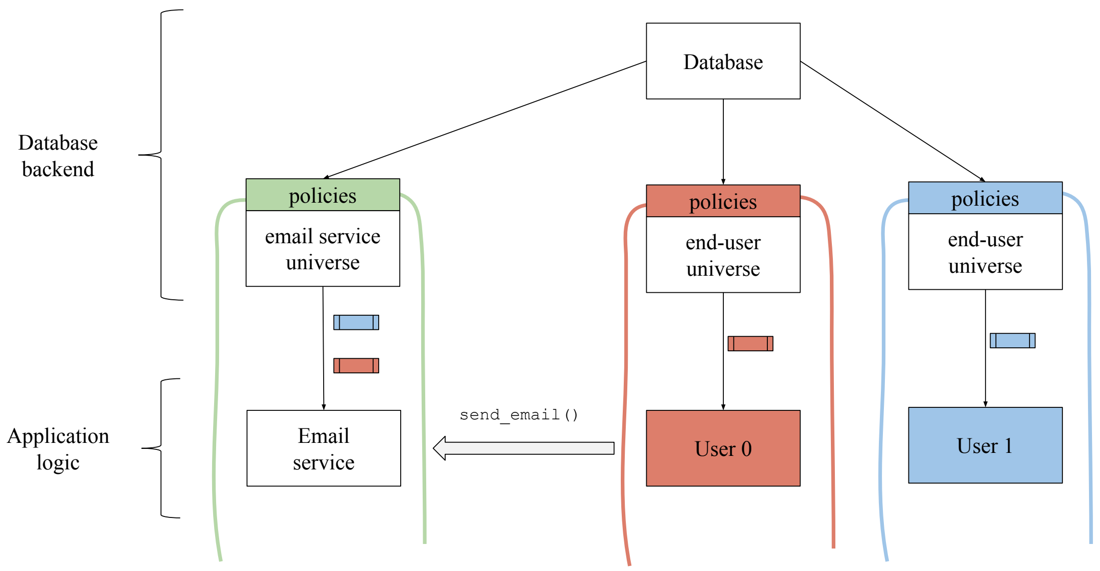


Figure 1-2: Multiverse architecture: the application logic accessing each user's universe only knows about the user's own data and cannot leak sensitive information.

Moreover, Xylem applies two techniques to reduce the storage overhead of materialized views: *view reuse* across query schema migrations and *partial materialization* of views.

Xylem's design is a good fit for the MultiverseDB approach to secure databases for three reasons:

1. In Xylem, query results are pre-computed based on data-flow of records. This easily supports per-user universes, since they can be instantiated as subgraphs of the data-flow. If policies are applied correctly on the data-flow, each user's view of the database is self-consistent: the results for all queries in the user's universe are computed based on the same records and reflect the same policies.

2. Xylem provides low-latency, live migrations between query schemas. It can efficiently modify the underlying data-flow graph to create a universe (i.e. a individual view of the database) for each user session at login time.

3. Xylem uses incrementally maintained materialized views and shifts the query processing effort from reads to writes. This provides an opportunity to enforce security policies on the write-side and avoid most of the read-side overhead imposed by previous approaches to database security (Chapter 2); an especially appealing benefit

for read-heavy web applications.

1.3 Challenges

As a secure storage system for web applications, MultiverseDB must be able to express complex security policies, enforce those policies correctly, and scale to thousands of users.

Complex applications require security policies that enforce row-level access control and column-level authorization. For example, HotCRP needs to deal with conflicts while assigning papers to reviewers, and must conceal the names of paper authors from reviewers during the review stage. To express these policies, MultiverseDB needs to have a flexible method of representing security policies that allows it to refer to other, seemingly unrelated database contents.

Moreover, MultiverseDB must properly enforce its security policies. The system must determine how to modify the underlying data flow graph to correctly restrict data flow into user queries. When policies are complex, MultiverseDB security nodes need to compute policies based on the content of an incoming record; relational expressions; and session-specific information such as the client IP address or user ID for the current session.

Finally, unlike Xylem, which supports a single set of materialized views for an application, MultiverseDB needs to maintain materialized views for hundreds or thousands of active users, who each see slightly different views of the database. If done naively, the memory overhead of these materialized views would be unbearably large. Indeed, this space overhead is the primary reason why similar approaches that create an individual view of the database for each user have been rejected several times [1, 2, 11].

1.4 Contributions

The contributions of this thesis are:

1. the MultiverseDB approach of enforcing centrally-specified, declarative security policies on a database by restricting data-flow propagation of updates;

2. techniques to enforce fine-grained access and information flow control policies in a streaming data-flow computation that maintains materialized views;
3. a validation that MultiverseDB's individual per-user views of the database can be implemented efficiently in terms of both space and performance;
4. a prototype implementation of MultiverseDB, an example class Q&A platform applications similar to Piazza with realistic security policies built for this prototype, and an evaluation of their performance.

The current MultiverseDB prototype has some limitations. MultiverseDB doesn't enforce access control on writes yet, so user sessions can write anything to any base table. The system also has some restrictions on which SQL predicates it can express because it supports only a subset of SQL operators. Moreover, due to limitations on Xylem, MultiverseDB currently does all its write processing in a single thread and materializes its leaf views, which results in lower write-throughput and higher space overhead.

1.5 Thesis outline

This thesis starts with a discussion of the related work (Chapter 2). Then, it explains the design of a multiverse data-flow system (Chapter 3) and covers the implementation of the MultiverseDB prototype (Chapter 4). Finally, it evaluates the performance of a class Q&A application, analyzes MultiverseDB's coverage of the security policies of a conference management system (Chapter 5) and concludes (Chapter 6).

Chapter 2

Related work

There have been various attempts to develop system and techniques to apply security policies to database-backed web applications.

Database views. Database views are commonly used to structure database contents for convenient querying and also, to limit what information end-users can access [5]. Materialized views [8] provide fast access to data by storing pre-computed query results and preventing view re-computation every time a view is used. MultiverseDB builds on these concepts and creates *user-specific* materialized views that cache query results with row- and column-level security policies already applied.

Client-side databases. The MultiverseDB application structure, with each user having her own version of the database, resembles that of Meteor [7] applications. Meteor applications use an in-memory client-side database and a publish-subscribe service to determine what data can be accessed by clients. The Meteor server specifies views with *publish functions* that act as security policies and limit the data sent to the client. When a client subscribes to a view, it creates an *observer* that forwards new records from server to client whenever the view changes. Each observer is associated with a publish function and, to avoid processing and network overheads, Meteor reuses observers when possible. However, Meteor is capable only of reusing observers for *identical* queries. Hence, a view with a publish function that contains user-specific values (e.g. the count of a user's private posts) creates an observer for each user.

Database access control. Most attempts to provide fine-grained access control on databases have focused on rewriting queries by inserting additional filter predicates that enforce the security policies [1, 2, 10]; or have attempted to validate queries by mapping them to a set of “authorized” views defined by the security policies [11]. Both approaches negatively impact read performance and are unsuitable for the read-heavy workload of most modern web applications.

Query rewriting can model complex security policies, but it transparently modifies the user query, which may cause the rewritten query to be much more expensive than the original query if the additional predicates contain complex sub-queries.

Query validation checks that a query can be executed using only “authorized” views and relies on inference mechanisms to map queries to views. However, if security policies require complex inference rules and a large number of authorized views, the overhead of query validation can be expensive. On top of that, because the set of inference rules does not capture all possible queries executed by the users, this approach can incorrectly reject an authorized query.

Information flow control. MultiverseDB’s security boundaries are conceptually similar to Resin’s *data flow boundaries* [13]. In both cases, boundaries enforce policies and restrict data flow from one part of the system to the other. However, while Resin trusts the application code and restricts data flow only when leaving the language runtime, MultiverseDB enforces policies at the database level and only allows the application logic to see data permissible to the current user session.

MultiverseDB and Jeeves [12] share the same goal of separating core program functionality from security policy definitions. However, while MultiverseDB enforces policies at the database level, Jeeves, a functional constraint language, implements them using symbolic evaluation and constraint solvers, where sensitive values are symbolic variables and security policies are constraints.

Similarly to MultiverseDB, UrFlow [3] uses SQL queries as security policies. However, MultiverseDB dynamically enforces policies in the data-flow graph, while UrFlow is integrated with the Ur/Web [4] compiler and uses symbolic evaluation and theorem-proving tools to statically check that applications queries do not violate the security policies.

Information flow control approaches usually require security policies to be written into each application. MultiverseDB, however, enforces security polices at the database level and allows the same database to be used by multiple applications that share the same policies.

Chapter 3

Design

MultiverseDB is a storage backend tailored to read-heavy web applications that enforces security policies while offering high read performance. It achieves this by pre-computing results for each user's active queries, with security policies already applied.

In MultiverseDB, each user has her own private view of the database that holds the pre-computed results for that user. Inside this "parallel universe", a user can read from any of her views without any possibility of violating the security policies.

Figure 3-1 shows a system overview of MultiverseDB. Given a global *query schema* and a *security configuration*, MultiverseDB builds a data-flow graph of materialized views and relational operators. The query schema is shared by all user sessions and when a new session starts, MultiverseDB builds a *user universe*, a reflection of the underlying global data-flow graph that implements the query schema for the new user. In MultiverseDB, all universes co-exist in the same data-flow graph and leaf nodes inside a user universe act as materialized end-user views where pre-computed query results can be read efficiently by a user session.

Upon receiving writes, MultiverseDB streams them through the graph until they reach a leaf. MultiverseDB evaluates security policies during write processing and automatically enforces its security policies inside the data-flow graph, restricting information flow into user views. This minimizes read-size overhead, since MultiverseDB computes policies once per write rather than at every client read.

MultiverseDB also reuses views to reduce the space overhead of materialized views and

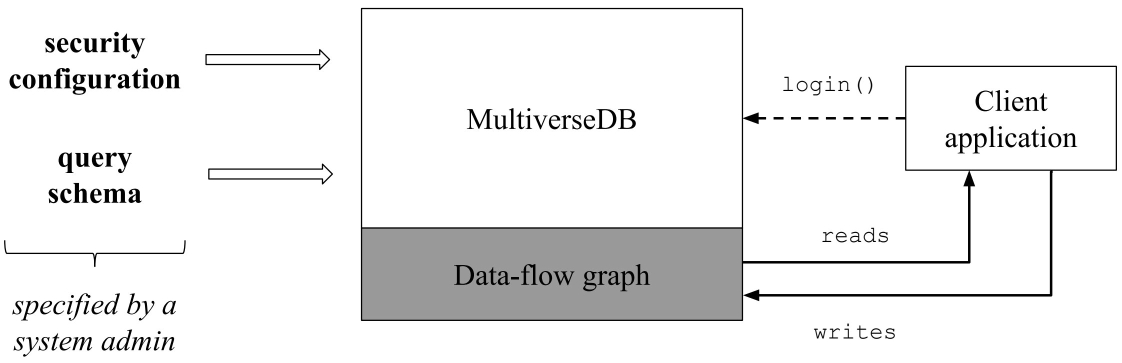


Figure 3-1: MultiverseDB system overview.

to prevent redundant processing.

This chapter discusses the MultiverseDB design in further detail and address the following questions:

1. How does a systems administrator declare security policies?
2. What is a MultiverseDB universe, and how is it represented in the data-flow graph?
3. How do multiple universes co-exist in the same data-flow graph and how does MultiverseDB enforce security and isolation between them?
4. How are universes created?
5. How does MultiverseDB reduce the overheads of maintaining materialized views?

3.1 Security configuration

A MultiverseDB security configuration consists of global declarative security policies and a set of security group definitions. The system administrator specifies the security policies and group definitions in a single input file, so that they are easily auditable.

3.1.1 Policy expression

MultiverseDB policies must be expressive enough to model a variety of access control patterns, such as row- and column-level authorization policies.

Each MultiverseDB policy is defined for a specific base table. In principle, MultiverseDB could also allow administrators to define policies over end-user views. However, this design could lead to data leakage or inconsistencies inside users' views for applications with a large number of queries if the database administrator is not careful. For example, assume a Piazza application that supports both the *post_count* and *posts* queries in Listing 3.1. Unless the same policies are applied to both queries, the count of posts for an author might be inconsistent with the posts the user sees, and a user might be able to infer information about posts that she is not allowed to view. To prevent this type of inconsistency, MultiverseDB defines security policies only over base tables.

```
/* base tables */
CREATE TABLE Post
(id int, cid int, author int, content text, private int,
   anonymous int)
/* queries */
post: SELECT * FROM Post;
post_count: SELECT author, COUNT(id) FROM Post GROUP BY author;
```

Listing 3.1: Example MultiverseDB query schema for the toy class Q&A application used throughout this chapter.

At a high level, MultiverseDB supports two types of policies: row-level policies and column-authorization policies.

Row-level security policies specify which rows in a base table a user is allowed to view, including derived, computed values that are based on these rows. A system administrator creates a row-level policy by specifying a base table and a SQL predicate as shown in Listing 3.2.

```
table: Posts,
predicate: WHERE private=1 AND author=UserContext.userid
```

Listing 3.2: MultiverseDB row-level policy that allows users to see their own private posts. The UserContext view stores ambient information about a universe, allowing the policy to reference the user ID.

Column-authorization policies determine a set of rows that must have one of its columns rewritten to a pre-determined value. Listing 3.3 shows a column-authorization policy that rewrites the *author* column for anonymous posts on Piazza. The column specified by the *rw_col* field is rewritten to *rw_value* if the row's *key* is present in the results of *rw_predicate*.

```
table: Post,
rw_value: "anonymous",
rw_col: author,
key: id,
rw_predicate: SELECT id FROM Post WHERE anonymous=1
```

Listing 3.3: A column-authorization policy that rewrites the author column if the post is anonymous.

SQL-like predicates are an intuitive way of expressing policies that allow some data to be accessed, since the results of the predicate represent the permissible data. Moreover, SQL predicates are expressive and enable security policies to refer to other views in the underlying data-flow graph. This is useful when a policy needs to make use of auxiliary information to determine if a row should be visible. Example of such auxiliary information are the contents of other tables or the user ID that each session injects into its user universe in a special *UserContext* view.

3.1.2 Security groups

While global security policies are enforced for all user sessions, security groups allow the system administrator to apply policies to a subset of users. For example, a reasonable security group for the Piazza application is the TAs for a class, who are more privileged than students, but less privileged than professors or the system administrators.

It would be impractical for an administrator to define a new security group whenever the classes in the database change, and MultiverseDB hence uses *group templates*. Group templates consist of a set of security policies and a membership view. A single group template defines multiple security groups which change dynamically as the membership view changes. Listing 3.4 shows how to express a group template for TAs.

uid	*gid*
Alice	6.824
Bob	6.824
Charlie	6.172

Table 3.1: Example membership view for the TAs group template

```
membership: SELECT e_uid as uid, e_cid as gid FROM Enrollment
    WHERE e_role = "ta",
policies: [{
   table: Post,
   predicate: WHERE private = 1 AND GroupContext.id = cid
}]
```

Listing 3.4: Definition of TA group template, and an associated policy that allows TAs for a class to see the private posts from that class

Group template policies have the same structure as global policies (see Section 3.1.1), but they are enforced only for members of each group. If needed, these policies can refer to a special *GroupContext* view that stores ambient information for each group, similar to UserContext for user universes.

A group template's membership view has two columns, *uid* and *gid*. Each row in the view represents a user in a group. For example, if Alice and Bob are both 6.824 TAs and Charlie is a 6.172 TA, Table 3.1 shows the contents of the membership view for the group template in Listing 3.4.

Whenever a new *gid* appears in the membership view, MultiverseDB creates a new *group universe* with the appropriate security policies for that *gid*.

Group universes are logically contained in multiple user universes. For example, Alice and Bob are both 6.824 TAs, so both their user universes must contain the 6.824 group universe. MultiverseDB realizes the dependencies between universes in a multiverse dataflow graph and enforces security policies at universe boundaries.

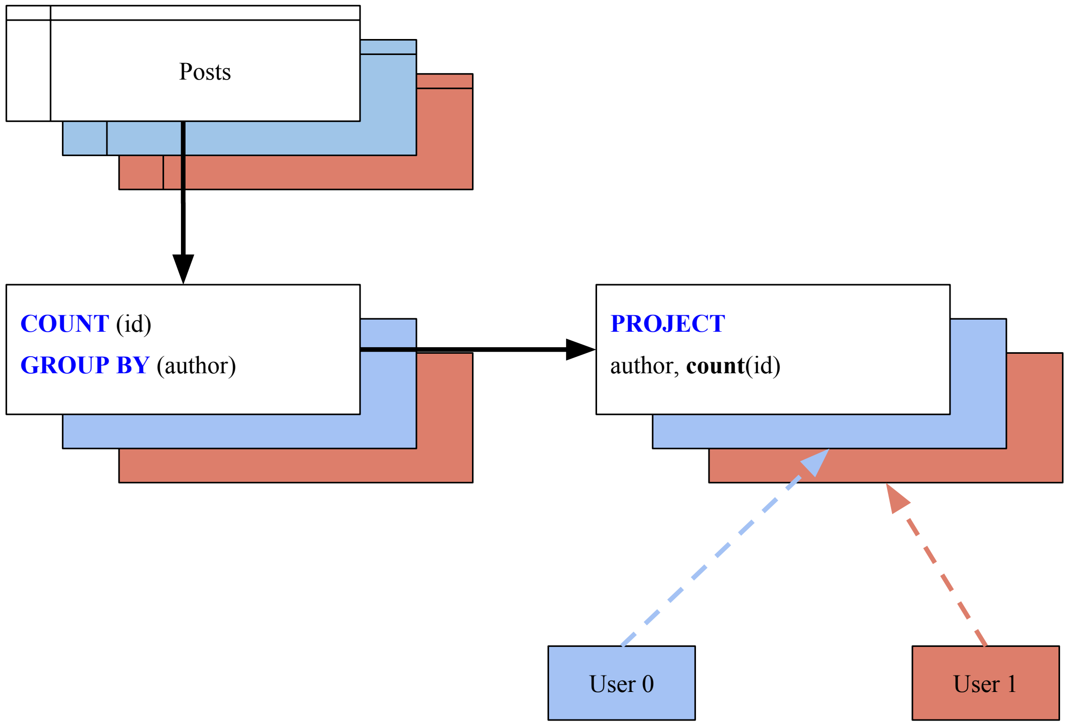


Figure 3-2: Schematic of the multiverse architecture: universes (in red and blue) mirror the original data-flow graph (in white). Security nodes (see Section 3.2.2) are omitted here.

3.2 Multiverse data-flow

A MultiverseDB *universe* is a reflection of the original data-flow graph constructed by the global query schema. Each universe contains per-universe materialized views that reflect its own version of each query in the schema, with potentially different results. Figure 3-2 shows a multiverse data-flow for the *post_count* query from Listing 3.1.

When a new user session begins, MultiverseDB reads its query schema and creates a new user universe by building a data-flow graph that updates materialized views for all queries in the universe.

A naive implementation would create an entire new data-flow graph for each universe, with only permissible data stored in the universe's base tables and security policies applied to writes before they enter the per-universe base tables. However, since universes support similar queries, they have many data-flow processing paths in common. The naive implementation would therefore result in redundant processing and significant data duplication due to overlapping materialized views in different universes.

Instead, MultiverseDB combines all universes into a single data-flow graph and creates dependencies between them. For example, a user universe depends on a group universe if that user belongs to the group. This allows MultiverseDB to share pre-computed results from one universe with multiple other universes.

3.2.1 Universe types

To reason about universe dependencies inside the same data-flow graph, MultiverseDB supports three types of universes: global, user and group universes.

The **global universe** is the core data-flow graph and has access to the entire database. It contains all base tables and unaltered materialized views. Because it is entirely unrestricted, only users with administrator permissions have direct read access to it.

User universes represent users' private view of the database. When a new user logs in and creates a new session, MultiverseDB spawns a user universe for the duration of that session. Only that session can query the views in the newly-spawned universe and those views contain only data that the session has access to, according to the security policies.

Group universes are the data-flow reflection of *security groups*. They are invisible to the outside world except through user universes. When multiple users are part of the same group, group universes allow for selective sharing of data and materialized views between multiple user sessions.

3.2.2 Universe boundaries and security nodes

MultiverseDB handles multiple co-existent universes in the same data-flow graph by enforcing security policies at the boundaries between universes. Figure 3-3 shows the relationship between global (white), user (red and blue) and group (green) universes.

When a write first arrives, MultiverseDB inserts it into the appropriate base table, which lives in the *global universe*. The write propagates through the nodes in the data-flow graph, until it reaches an intersection where a universe diverges from the global universe.

At this boundary between universes, MultiverseDB inserts a special *security node*. This

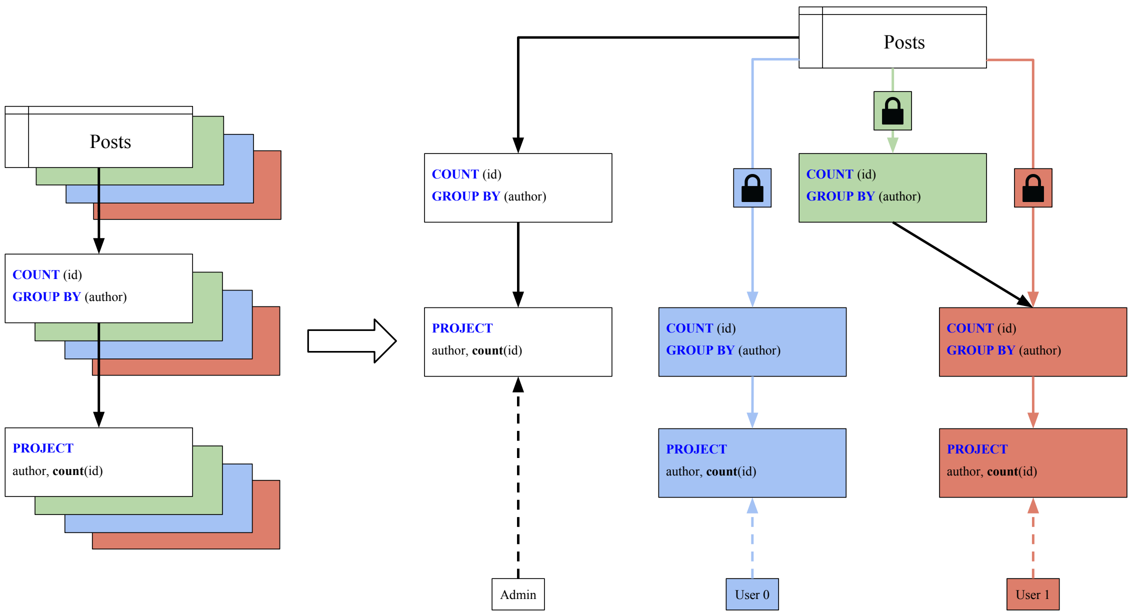


Figure 3-3: Security nodes enforce policies between global (white), group (green) and user (red and blue) universes in the data-flow below the Posts table.

node enforces the security policies defined by the system administrator and creates a *security boundary* that restricts data-flow between universes. Users who are part of a group have access to the group's query results and use them to compute the results for their private universe (e.g., the red user is part of the green group). Note that no policies need to be applied between group and user universes, since the group's policies are applied at the boundary between the global and the group universe.

Security nodes enforce policies based on three inputs: *(i)* the content of an incoming record; *(ii)* relational expressions; and *(iii)* ambient information stored in universes' context views. Since policies can be as complex as any application query, a single logical security node is in practice often composed of many physical data-flow operators.

3.2.3 Rewrite security nodes

Standard data-flow operators such as joins and filters are sufficient to implement policies that prevent an entire row from flowing into user views. However, in order to express column-authorization policies, MultiverseDB can selectively modify the value of a row's column. It does so using a special *rewrite* data-flow node.

A rewrite node has two parent nodes: a *source* and a *signaling* parent. The source parent forwards records that may or may not have their columns rewritten, while the signaling parent stores record identifiers that indicate which of these records should actually be rewritten.

When a rewrite node receives a record from its source parent, it queries its signaling parent using the record's key to decide whether or not to rewrite it. If instead, the rewrite node receives an update from its signaling parent, the node queries its source parent for the records affected by the update. Depending on the update, the rewrite node either rewrites the affected records or undoes any changes it had previously made to them.

3.3 Universe creation

All MultiverseDB universes co-exist in one data-flow graph. MultiverseDB uses Xylem's query schema migration mechanism to create new universes, changing the underlying data-flow graph to support the new universe's queries and adding security nodes at universe boundaries.

3.3.1 Group creation

MultiverseDB determines which group universes to create based on the existing membership views. The system creates membership views the first time a security group template is specified in the security configuration and, they continue to exist until the template definition changes.

When MultiverseDB creates a membership view, it appends a special *trigger node* below the last data-flow node of the membership view definition. The trigger node is responsible for creating a group universe whenever the application specifies a new group. Figure 3-4 shows the trigger node for the group template from Listing 3.4. If the application creates a new class and assigns TAs to it by writing to the *Enrollment* table, the enrollment record will propagate from the base table to the trigger node below the membership view.

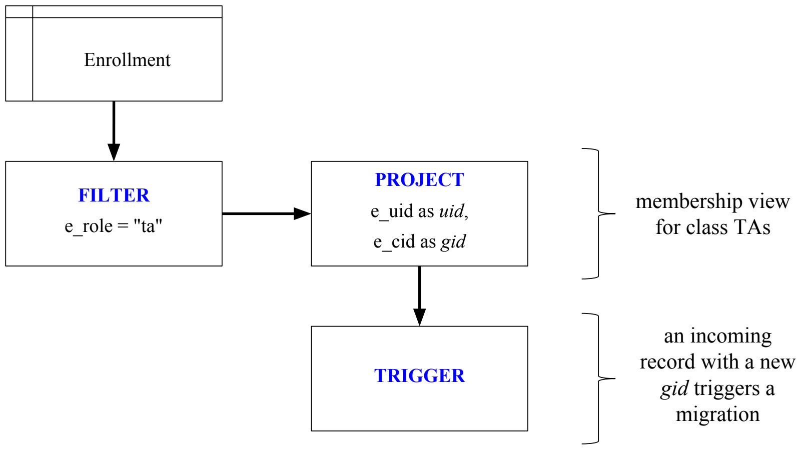

Enrollment
FILTER
e_role = "ta"
PROJECT
e_uid as *uid*,
e_cid as *gid*
membership view for class TAs
TRIGGER
an incoming record with a new *gid* triggers a migration

Figure 3-4: Data-flow for the membership view and trigger node of the TAs group template from Listing 3.4

Once the record reaches this special data-flow node, it triggers a migration on the data-flow graph to create a new group universe.

A MultiverseDB data-flow graph can therefore modify itself to accommodate a new group universe. A group universe migration extends the data-flow graph by translating security policies into security nodes and placing them at the boundary with the global universe – where the views in the group universe diverge from the original nodes in data-flow graph. Then, inside the group universe's security boundary, the migration creates the remaining data-flow operators necessary to implement the queries in the schema.

3.3.2 Session creation

When a new user session starts, MultiverseDB creates a new user universe for that session. Similarly to a group universe, the new user universe extends the data-flow graph, adding security and operator nodes.

However, user universes' migrations have additional complexities over group universes' migrations. For instance, a user universe can contain several group universes. If Alice is a TA for 6.824, her private user universe contains the 6.824 group universe, since she has the proper capabilities for that group (supplied by her entry in the membership view for the TAs group).

Algorithm 3.1 shows the process for creating a user universe. First, MultiverseDB finds which groups the user belongs to for each security group template. Second, it builds a version of the each query in the schema with the global policies applied. Third, once it has created the global version of the query, MultiverseDB finds the group's version of the query for each group the user belongs and coalesces the group views and global view into a single leaf view using a `union` data-flow operator. Finally, the client can then read from the new leaf view.

```
def create_user_universe(uid):
    leaves = [], groups = []
    # Start a user universe migration
    db.migration_start()
    # Find which groups in each template the user belongs to
    for (gt, membership) in db.group_templates:
        gid = membership.get(uid)
        groups.push((gt, gid))

    # Construct a user version of each query in the schema
    for query in db.query_schema:
        # Build a view for the query with the global policies applied
        global_policies = db.policies_for(query, uid)
        global_view = db.build_query(query, global_policies)

        # For each group the user is a member of get that group's
        # version of the query with group policies already applied
        group_views = [db.group_view(query, gt, gid) for (gt, gid) in groups]

        # Coalesce the group views and the user view into a single
        # leaf view that will be queried by the client
        leaf = db.coalesce_views(global_view, group_views)
        leaves.push(leaf)

    # Commit the migration and return the leaf views to the client
    db.migration_commit()
    return leaves
```

Algorithm 3.1: The process for creating a user universe.

3.4 Improvements over Xylem

MultiverseDB relies on mechanisms provided by Xylem to build a dynamic data-flow system, such as incrementally maintained materialized views and Xylem's query schema migrations. Moreover, MultiverseDB also extends Xylem's view reuse and partial materialization techniques to reduce the space overhead incurred by materialized views.

In order to benefit from these overhead-reducing techniques, MultiverseDB adapts them to the context of a multiverse database. MultiverseDB optimizes Xylem's migration mechanism to handle data-flow graphs with thousands of nodes in order to quickly spawn new universes (see Section 3.3) and extends Xylem's view reuse algorithm to realize reuse opportunities across different universes.

3.4.1 Migration complexity

Xylem performs schema migrations to enact changes in the data-flow graph. Upon receiving a new query schema, Xylem determines what changes it needs to make to the existing data-flow graph and detects shared common subexpressions between queries in the old and new schema.

After it finishes building the new data-flow graph, Xylem decides which nodes it can partially materialize and initializes newly created materializations. However, the new state of any new materialization needs to reflect all previous writes received by the system. To accomplish this, Xylem finds *replay paths* between the new nodes and their ancestor materializations and replays existing records from the ancestors to fill in the new materializations. Once all materializations are filled, Xylem completes the migration.

MultiverseDB migrations are conceptually similar to Xylem migrations, but with a key difference: I optimized MultiverseDB migrations for large data-flow graphs with thousands of operators, and changed Xylem's migration algorithms to achieve this.

Xylem targets applications with at most hundreds of queries and a data-flow graph of size $O(|\text{queries}|)$ nodes. By contrast, MultiverseDB must support thousands of user universes each with hundreds of queries. Not only does its graph have a size of $O(|\text{queries}| \times |\text{users}|)$, but it also changes frequently (whenever a new session starts).

To properly add nodes and realize replay paths, migrations need to establish dependencies between nodes and base tables. While Xylem gets away with analyzing the entire graph to find these dependencies (traversing it several times to do so), MultiverseDB can't afford such traversals without significantly affecting migration time, and therefore session creation latency.

Instead of analyzing the entire graph like Xylem, MultiverseDB lazily keeps track of the node dependencies. When a migration happens, MultiverseDB analyzes only the *new nodes* added by the migration and updates the dependencies accordingly. Because of this, MultiverseDB migrations need to consider only $O(|\text{new nodes}|)$, instead of Xylem's $O(|\text{nodes}|)$.

3.4.2 View reuse across universes

The key challenge in implementing MultiverseDB is the data-flow fan-out caused by thousands of different universes co-existing within the same data-flow graph. With many universes, the number of parallel data-flow processors is much smaller than the number of data-flow nodes in the graph, so a big fan-out decreases write throughput. Moreover, per-universe materialized views duplicate data across universes and can lead to severe space overhead. MultiverseDB addresses these problems by automatically reusing and sharing data-flow paths and views whenever possible.

View reuse in MultiverseDB is a simplified version of a multi-query optimization (MQO) problem. MQO tries to maximize reuse across a batch of queries, with the freedom to rewrite each query to suit the others. MultiverseDB's problem is the more restricted one of mutating new queries to maximize their opportunity to reuse *existing*, static expressions in the data-flow graph.

Xylem performs limited view reuse based on the Finkelstein algorithm for common subexpression detection of SQL queries [6]. However, the Finkelstein algorithm was developed for the more complex version of MQO, not MultiverseDB's simplified version. This means that, in order to avoid potential exponential solving time, the Finkelstein algorithm rejects view reuse opportunities by computing only coarse-grained query signatures. Because of that, MultiverseDB doesn't implement view reuse using the Finkelstein algorithm. Instead, MultiverseDB uses its own algorithm for the simplified version of MQO that allows it to identify reuse opportunities even across different universes.

By limiting the scope of the problem, MultiverseDB is able to achieve more view reuse with little to no performance impact. When a new query arrives, MultiverseDB generates a intermediate representation (IR) graph of the query and tries to merge it with the stored IR

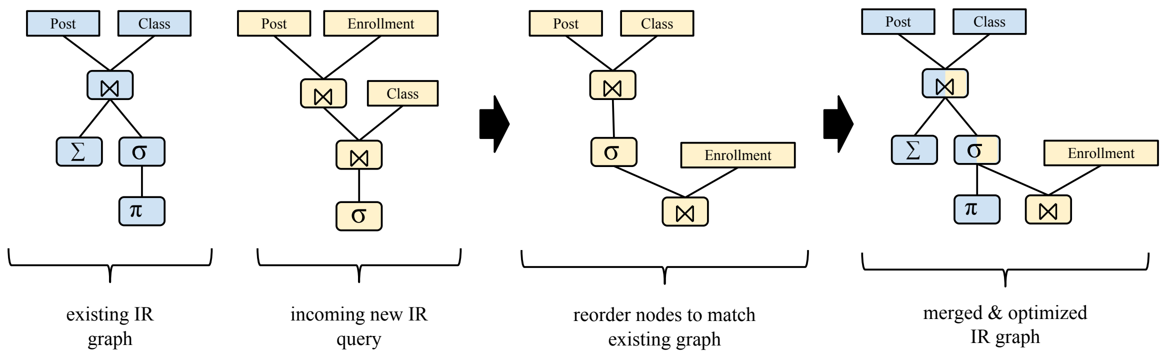


Figure 3-5: MultiverseDB traverses the new IR graph and reorders nodes to match those of the existing IR graph and merges the matched nodes.

graphs of existing queries.

MultiverseDB traverses the new IR graph in topological order and tries to match its operators to operators in the existing IR graphs. MultiverseDB also reorders joins in the new IR graph to match the order in the existing IR graphs. Once it has compared the new IR graph to all the existing IR graphs, MultiverseDB reuses the matched operators from the existing IR graphs and extends the data-flow graph with only the unmatched operators. Figure 3-5 shows the algorithm in action.

As each IR graph contains at most all existing data-flow nodes, the upper-bound runtime of this algorithm is $O(|\text{queries}| \times |\text{nodes}|)$. However, most queries are balanced and have few matching operators, meaning most IR graphs are quickly discarded. In practice, comparing a pair of IR graphs rarely takes $O(|\text{nodes}|)$ time.

While the queries specified in the query schema are policy-agnostic, their IR representation isn't and MultiverseDB stores a IR representation of each query for each universe. Hence, the general new reuse algorithm can be used for *all* cases of view reuse in MultiverseDB: within the same universe (between different queries), across different universes of the same type (e.g. user 0 and user 1) and across universes of different types (global/-groups, global/users or users/groups).

When it creates a new universe, MultiverseDB tries to merge the queries in the new universe with another universe of the same type. If there is no such universe (e.g. when the first user logs in), MultiverseDB tries to merge the newly created universe with the global universe and with one universe of each security group template.

3.5 Graph construction

View reuse allows MultiverseDB to reduce storage overheads and improve throughput. Hence, MultiverseDB attempts to construct the underlying data-flow graph such that it maximizes opportunities for view reuse.

3.5.1 Parallel policy computation

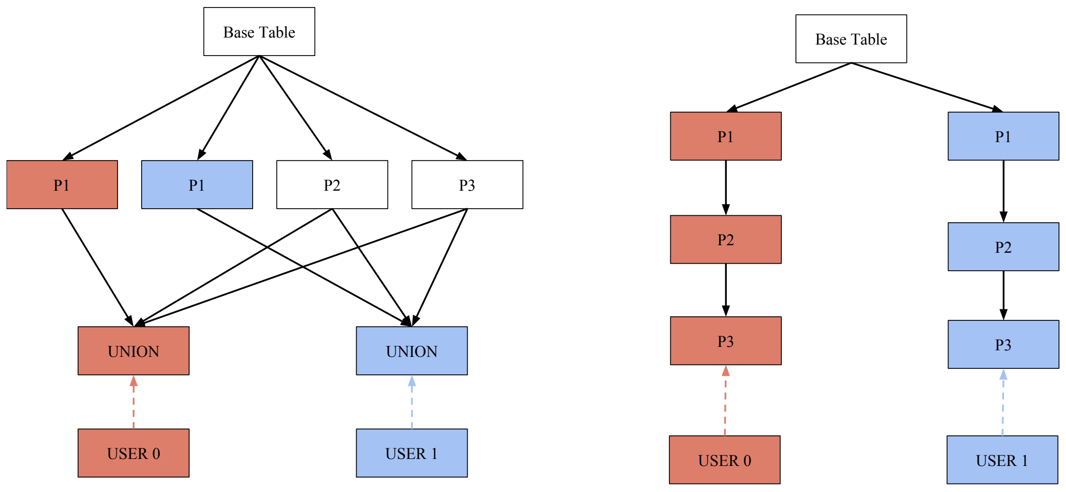
A single table might be a target of many security policies. By default, MultiverseDB computes all security policies in parallel and combines the results with a `UNION` operator. This means that as long as a row satisfies any one of the policies, it will be visible to the user.

The alternative approach, to compute policies in sequence, has some noticeable downsides. First, it impacts the time that takes for a write to be visible, since that write needs to pass through a long chain of policy nodes. Second, it reduces the opportunities for view reuse. MultiverseDB can only reuse *prefixes* of paths in the data-flow graph so, the order in which policies are chained affects which nodes can be reused, and chaining generally reduces reuse opportunities.

Figure 3-6 show a scenario with a security configuration with three policies (*P1*, *P2* and *P3*). While *P2* and *P3* can be reused between user universes, *P1* can't (e.g. because it uses user-specific information, like the session IP address).

When policies are computed in parallel, universes can always share *P2* and *P3*. If the policies were computed sequentially however, the order in which policies are applied affects reuse opportunities. With many policies, figuring out a good ordering of policies is complicated and even the optimal ordering might not yield good reuse opportunities.

However, parallel policy computation has the downside that if policies are not disjoint, the user will see duplicate records. This is an easier problem to solve than the ones of serial computation: for example, applying a `DISTINCT` operator at the end of each query allows correct reconciliation of policies over views with unique keys (a common occurrence).



(a) Parallel policies (b) Sequential policies

Figure 3-6: With parallel policies, it is always possible to reuse P2 and P3. With sequential policies, the order affects reuse opportunities.

3.5.2 Operator order

The order in which operators appear in the graph affects the opportunities for view reuse: the closer a node is to the roots of the data-flow graph, the more likely it is to be reused. Hence, MultiverseDB establishes a static operator order in the global universe, as far as permitted by the query. Joins come first, followed by aggregations and filters. This allows the system to frequently reuse stateful operators (joins and aggregations) and reduce space overheads. This order works well for reusing nodes across different queries. However, MultiverseDB must also reuse nodes from the *same* query across different universes.

Different versions of the same query differ only in their security nodes. Because everything above the security boundary is common to all versions of the query, MultiverseDB pushes the security boundary as far down as possible, being careful to prevent data leakage and incorrect query results while doing so.

Algorithm 3.2 shows the process of building a query and its security boundary. First, MultiverseDB places joins above the security boundary, in order to maximize reuse of potentially large materializations. Second, it pulls filters above the security boundary if there are no aggregations or it reorders filters that filter by the aggregation's group by column. Third, MultiverseDB places the security boundary before it loses information necessary to enforce the security policies. Finally, it creates the aggregations, the remaining filters and the projection nodes.

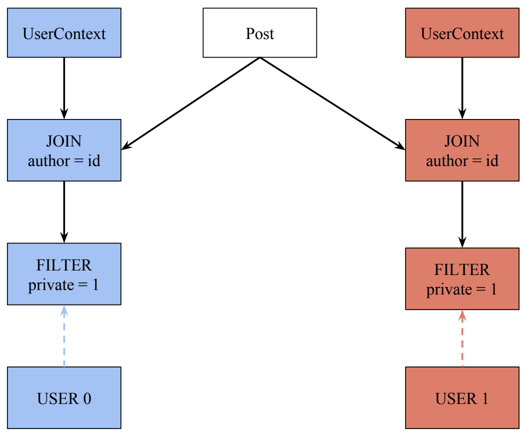
```python
def build_query(query, policies):
    # Join are always above security boundary
    make_joins(query.joins)

    if query.aggrs.is_empty():
        # If there are no aggregations, place filters above security boundary
        make_filters_after_aggr(query.filters)
    else:
        # Might need to reorder filters if they filter by GROUP BY column
        make_filters_before_aggr(query.aggrs, query.filters)

    # Place security boundary before any aggregations, since at that point,
    # we lose information we might need to enforce policies.
    make_security_boundary(policies)

    make_aggr(query.aggrs)

    # Create remaining filters that weren't reordered before the aggregation
    if not query.aggrs.is_empty():
        make_filter_after_aggr(query.aggrs, query.filters)

    make_projections(query.projects)

# All universes reuse the policy until the point it uses a Context node.
# Put joins last to maximize reuse and filters before to reduce the size
# of future stateful operators.
def make_security_boundary(policies):
    make_filters(policies.filters)
    make_joins(policies.joins)
    make_rewrite(policies.column_authorization)
```

Algorithm 3.2: The process of building a query with a security boundary.

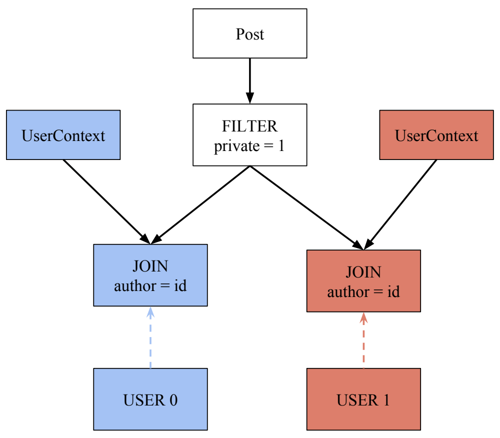
The choice of implementing security nodes as a combination of multiple operators allows MultiverseDB to partially reuse them. Much of the computation in security nodes is identical, except when they reference a universe's private context view.

In order to maximize reuse, MultiverseDB places any operator that uses the context view at the end of the computation. This is a powerful reuse strategy, since it means the security node's prefix can be reused by *all* the universes that implement this policy.

For example, assume a Piazza-like application that implements the policy in Listing 3.2. Figure 3-7 shows the possible ordering of nodes. If MultiverseDB places the filter *after* the join as in Figure 3-7a, the data-flow graph will end up with thousands of filters, one for each universe. By using the ordering in Figure 3-7b, the same filter node can be



(a) Filter after join



(b) Filter before join

Figure 3-7: MultiverseDB reorder security nodes before joins against Context views to maximize reuse.

reused by all universes that implement the policy, and MultiverseDB achieves higher write throughput since the filter computation is done only once.

Chapter 4

Implementation

I implemented a MultiverseDB prototype on top of the existing implementation of Xylem[1] in 2k additional lines of Rust code.

MultiverseDB expands Xylem with a security policy parsing module and modifies the existing view reuse, graph construction and migration modules. MultiverseDB also adds two new API calls to the Xylem controller: `set_security_configuration` and `create_universe`. I also implemented a class Q&A application backed by MultiverseDB. Table 4.1 shows the code distribution over the modules.

MultiverseDB's security configuration is specified in JSON and the system leverages Xylem's SQL parsing mechanism to translate security policy predicates. A database administrator uses `set_security_configuration` to configure the system with the appropriate security policies.

When the system administrator specifies a new security configuration, MultiverseDB starts a migration to adapt the data-flow graph with the new security group templates and creates their membership views. After that, future universes will uphold the new security configuration, but old universes need to be destroyed and reconstructed to conform to the new configuration.

A similar sequence of events occurs when the set of groups that a user is part of changes. When a client session creates a user universe, MultiverseDB establishes which groups the user belongs to at the beginning of the migration. Xylem currently doesn't support dynam-

[1]https://github.com/mit-pdos/distributary

Module	*LOC*
Graph construction	647
Migration	376
View reuse	286
Policy parsing	285
API	67
Class Q&A app	440
Total	2101

Table 4.1: Approximate distribution of additional lines of code per MultiverseDB module

ically adding or removing ancestors to existing nodes in the graph, so the policy changes are only visible after the user universe is reconstructed, i.e. once the current session has ended.

User universes are short-lived: MultiverseDB creates them when a user logs in and destroys them when they log out. On the other hand, group universes are long-lived and exist even if there are no active user universes that belong to the group. MultiverseDB only destroys group universes when a system administrator changes the security configuration or when a *gid* is removed from the membership table.

Client applications and trigger nodes use `create_universe` to spawn new user and groups universes, respectively. MultiverseDB doesn't itself authenticate users, nor does it currently authorize writes, so the application still needs to verify users' identities and must determine which users have permissions to write to which tables.

MultiverseDB currently executes all its write processing in a single thread due to limitations in the Xylem design. In order to support multiple write processing threads, Xylem needs to send $O(|nodes|)$ coordination messages and wait for acknowledgments whenever new nodes are added to the graph. This is prohibitively expensive for the large graph sizes that MultiverseDB supports. However, this isn't fundamental, the migration algorithm could message only the affected threads or it could piggyback the coordination messages on incomings writes and not wait for acknowledgments from the threads.

Chapter 5

Evaluation

This chapter evaluates the performance of my MultiverseDB prototype and answers the following questions.

1. Is MultiverseDB space-efficient and does it minimize the overhead of materialized views?

2. Does MultiverseDB provide the high read performance necessary for read heavy web applications?

3. Can MultiverseDB quickly adapt its data-flow graph to satisfy requests from new user sessions?

4. Can MultiverseDB express complex security policies required by modern web applications?

In a class Q&A application similar to Piazza with a realistic set of security policies and thousands of active sessions, MultiverseDB minimizes space overheads; quickly creates new user sessions; and outperforms the read throughput of previous approaches to database security.

MultiverseDB is also sufficiently expressive to support almost all of the policies enforced by JConf, a conference management system backend implemented in Jeeves [12].

5.1 Class forum application

I measured the performance of MultiverseDB using the *post_count* query described in Listing 3.1 and two security configurations, one SIMPLE and one COMPLEX.

In the SIMPLE configuration, users are allowed to see all public posts and their own private posts. This configuration has two global policies and doesn't require group policies.

In the COMPLEX configuration, users can see their own private posts, but can see only public posts for classes they are enrolled in. Moreover, TAs are allowed to see private posts for classes they TA.

The database is always populated with 1M posts (80% public) uniformly distributed over 1k classes.

5.2 SIMPLE security configuration

Using the SIMPLE security configuration, I measured the impact of the improvements in Xylem's migration mechanism and the effects of view reuse and partial materialization on memory usage and session creation latency.

5.2.1 Xylem improvements

Unlike Xylem, MultiverseDB is designed to support data-flow graphs with a large number of nodes, and implements optimizations to reduce the migration time complexity from $O(|\text{nodes}|)$ to $O(|\text{new nodes}|)$ (see Section 3.4.1). Figure 5-1 shows the impact of the optimizations made to achieve this.

Without optimizations, session creation latency increases with the number of active sessions, since every new session increases the number of nodes in the graph. With optimizations, however, MultiverseDB can support thousands of sessions while maintaining a constant session creation latency.

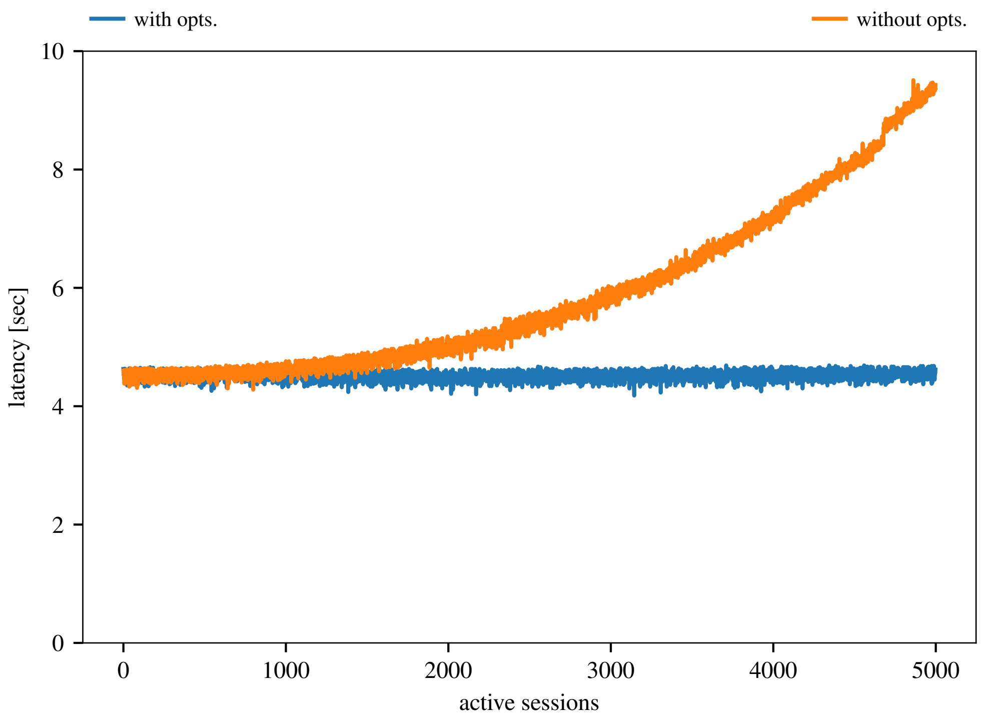


Figure 5-1: Without optimizations to migration time complexity, session creation latency quickly deteriorates as the number of active user sessions increases.

5.2.2 View reuse and partial materialization

Figure 5-2 shows MultiverseDB's memory usage as the number of active user sessions increases.

Partial materialization minimizes memory overhead by not materializing keys that clients haven't read yet, while view reuse also reduces memory usage by preventing duplication of views. While partial materialization establishes the lower bound for memory usage, view reuse establishes the upper bound.

With view reuse, user universes share the materializations that the public posts policy creates, which reduces the memory overhead by 24%. However, memory usage still increases linearly with the number of sessions due to internal materializations inside each user universe.

Partial materialization attenuates this by materializing only keys that are read by clients. When clients read no keys, MultiverseDB with partial materialization has a 2$\times$ memory overhead over the base tables due to the metadata from the empty per-universe data-flow nodes.

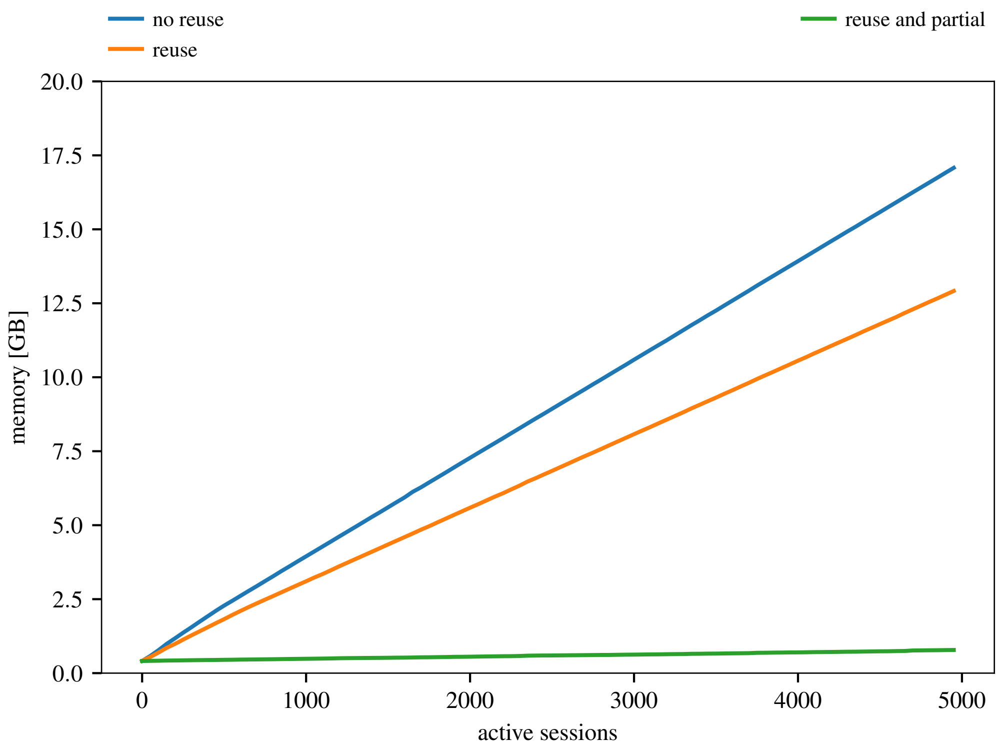


Figure 5-2: View reuse and partial materialization reduce space overhead.

In the worst case scenario, where clients read and materialize all keys, partial materialization has no effect. View reuse bounds the memory overhead to $31\times$ over base tables for 5k active user sessions (versus $41\times$ without view reuse). In practical settings where clients read some, but not all keys in their views, MultiverseDB's memory usage falls somewhere in between the lower bound set by partial materialization and the upper bound established by view reuse.

View reuse and partial materialization also improve session creation latency. View reuse shortens replay paths by replaying from reused views instead of base tables, while partial materialization reduces the amount of data flowing through the graph.

Figure 5-3 shows the effects of view reuse and partial materialization on session creation latency. View reuse doesn't improve session creation latency for the first user session because there are no other user universes to reuse from. However, view reuse benefits all sessions created after that and reduces the average session creation time by 17%. View reuse and partial materialization combined make session creation almost instantaneous, as the new universe's views start out empty.

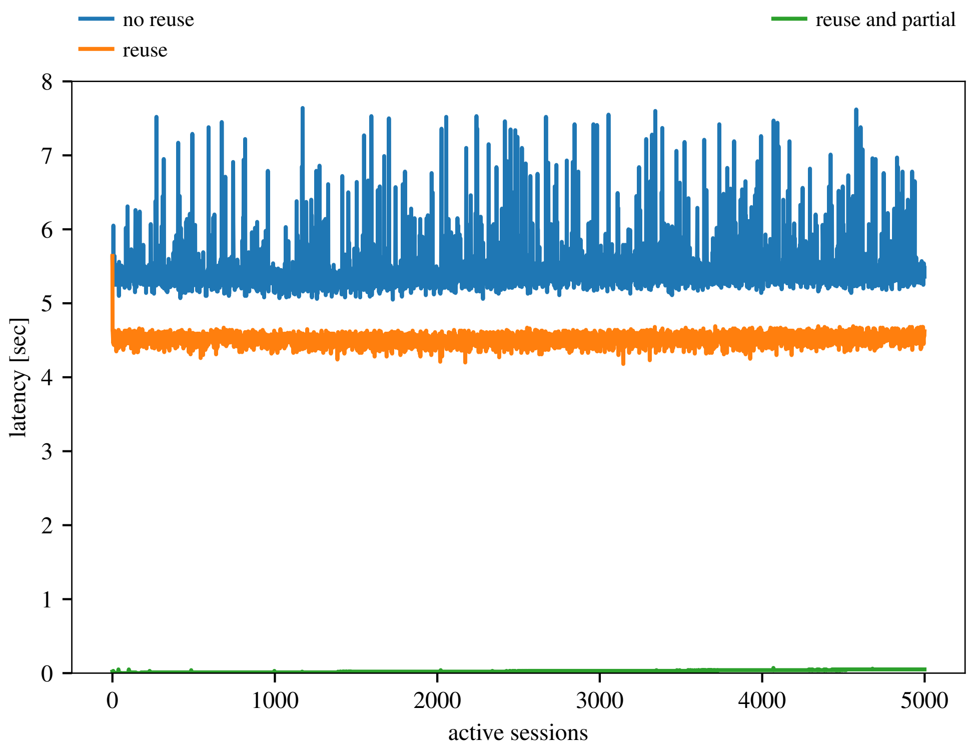


Figure 5-3: View reuse almost halves session creation latency, while reuse and partial materialization make it almost instantaneous, independently of the number of user sessions.

5.3 COMPLEX security configuration

The COMPLEX security configuration can be expressed without groups (using 3 global policies) or with groups (using one global policy and two security groups: students and TAs).

Using this latter configuration, I measured the impact of using group policies versus not using them and compared MultiverseDB's read and write performance with that of a MySQL backend.

5.3.1 Security groups

Memory usage

Figure 5-4 shows MultiverseDB's space overhead as the benchmark creates more active user sessions.

Initially, group policies have a 20% space overhead over not using them. This is because

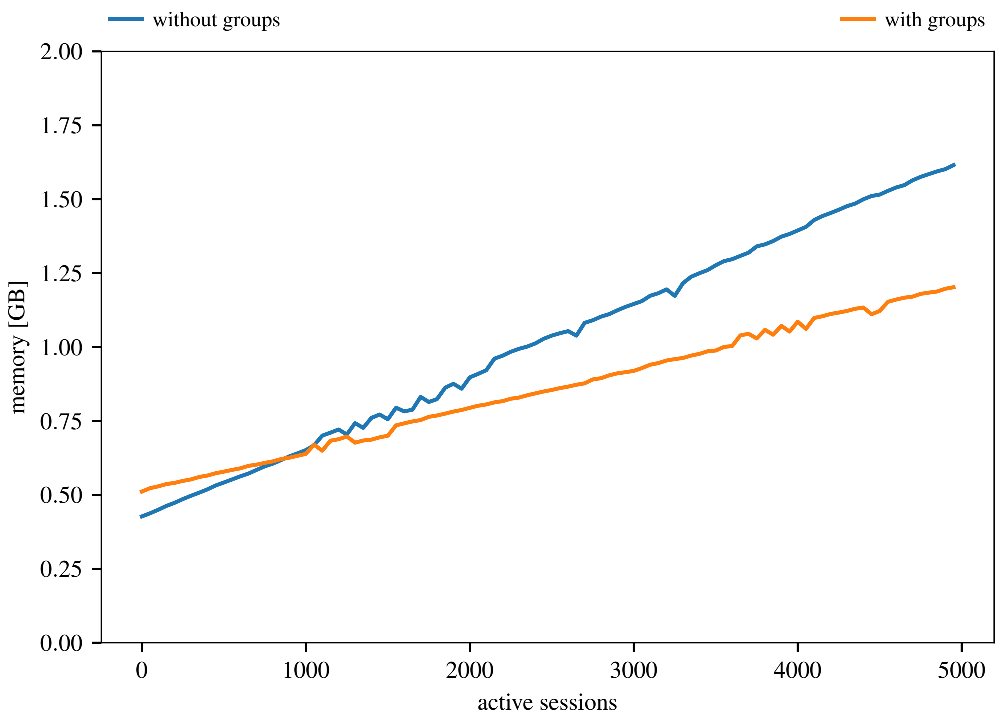


Figure 5-4: Group policies are more efficient at reducing space overhead as the number of sessions increases.

groups universes (and their internal materializations) are long-lived and are present in the system even when no user sessions make use of them. However, as the benchmark creates more user sessions, group policies quickly bridge the gap and with 5k active sessions they reduce memory usage by 25%. Without groups, students and TAs for the same classes duplicate data across their private universes. Groups allow users in the same class to share data and materialized views from a single group universe.

Session creation latency

Similarly to view reuse, security groups also shorten replay paths, since user universes replay data from the group's version of the query and not from the base tables. Moreover, because group universes are long-lived, most of the work done by the aggregation nodes happens at the time MultiverseDB creates the group universes. Therefore, user universes just pay the cost of reconciling the results from different group universes. Figure 5-5 shows that security groups reduce session creation latency by 66% for this benchmark.

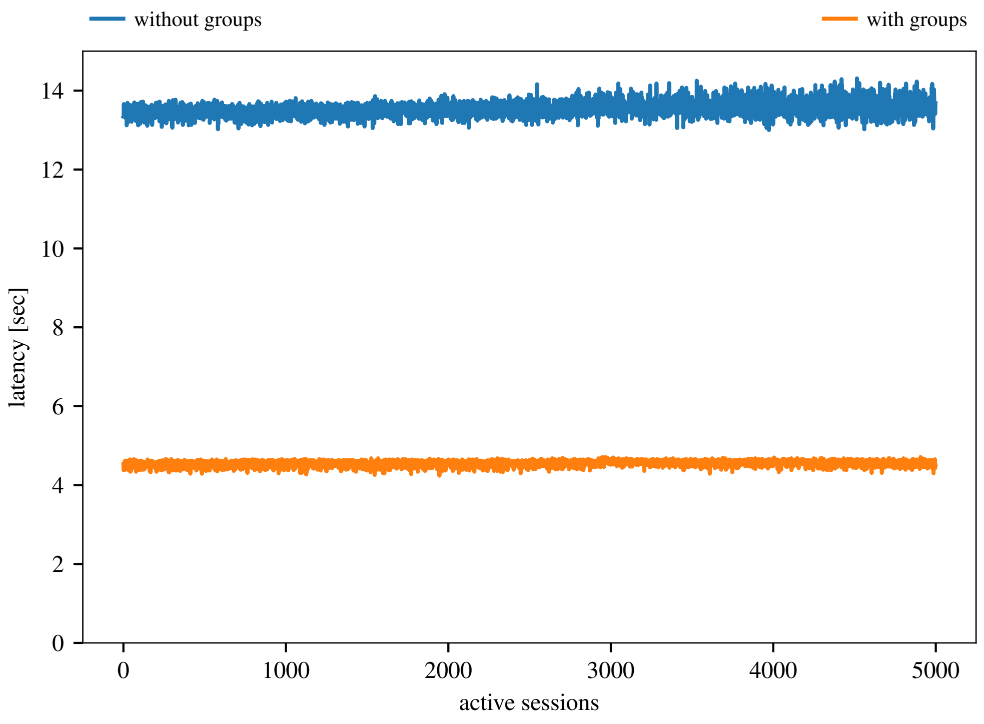


Figure 5-5: Group universes reduce session creation latency, independently of the number of active user sessions.

Read and write throughput

Figure 5-6 illustrates how MultiverseDB's read and write performance scale as the number of user sessions increases.

When the number of active sessions increases, the number of nodes in the data-flow graph also increases. With a limited number of processing threads, more nodes in the data-flow graph means each node has less time to process incoming writes. Therefore, as the number of user sessions increases, write throughput decreases. The number of nodes in the graph doesn't affect read performance, which remains constant as more user sessions start.

Security groups are long-lived and initially create thousands of extra nodes. While this hurts write throughput when there are few active user sessions, security groups reduce the number of nodes that future user universes need to create. As the number of user sessions increases, the setup without groups sees a sharp decline in write throughput, while security groups shows a much lower reduction and scale better as the number of active user sessions increases. As with memory usage, group universes have an initial overhead, but eventually pay off.

View reuse also improves write throughput, since it prevents nodes from doing redundant computation. As Figure 5-6a shows, with 5k active user sessions, view reuse improves write throughput by 2.6×.

5.3.2 Comparison with MySQL

I compared the read and write performance of the MultiverseDB prototype to the performance of a MySQL backend. The MySQL backend uses either the original *post_count* query or a modified version of the query that enforces security policies, similar to the query rewriting approach described in Chapter 2, or to queries manually adapted by the application logic.

Figure 5-7 shows how MultiverseDB's performance with 5k active user sessions compares to a MySQL backend.

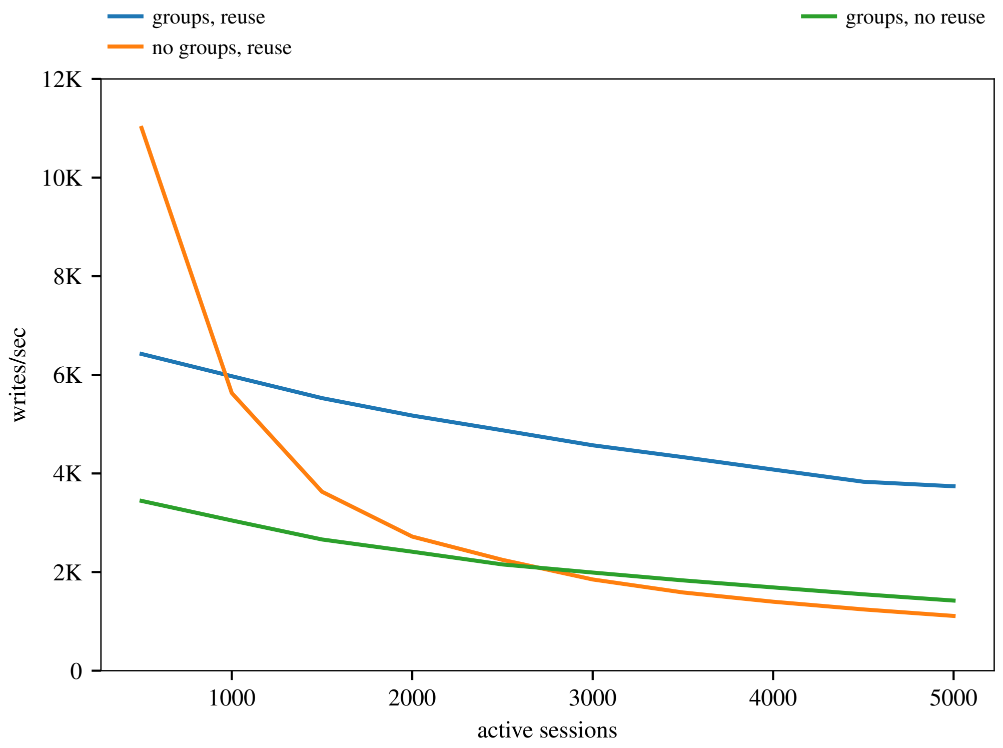
MultiverseDB is designed for read-heavy web applications and for each active user, it pre-computes the query results with security policies already applied. Because MultiverseDB pushes the processing effort from reads to writes, MultiverseDB's write throughput is only 42% that of MySQL [1]. However, MultiverseDB greatly outperforms MySQL in read performance: MultiverseDB reads are 12× faster than MySQL with the original query.

With the secure version of the query, MySQL's read performance decreases by almost 90%. With a complex security configuration, computing policies involves performing expensive sub-queries and unlike MultiverseDB, which enforces security policies during write processing, MySQL recomputes these policies on every read. Hence, MultiverseDB reads are 114× faster than MySQL using the rewritten secure version of the query.

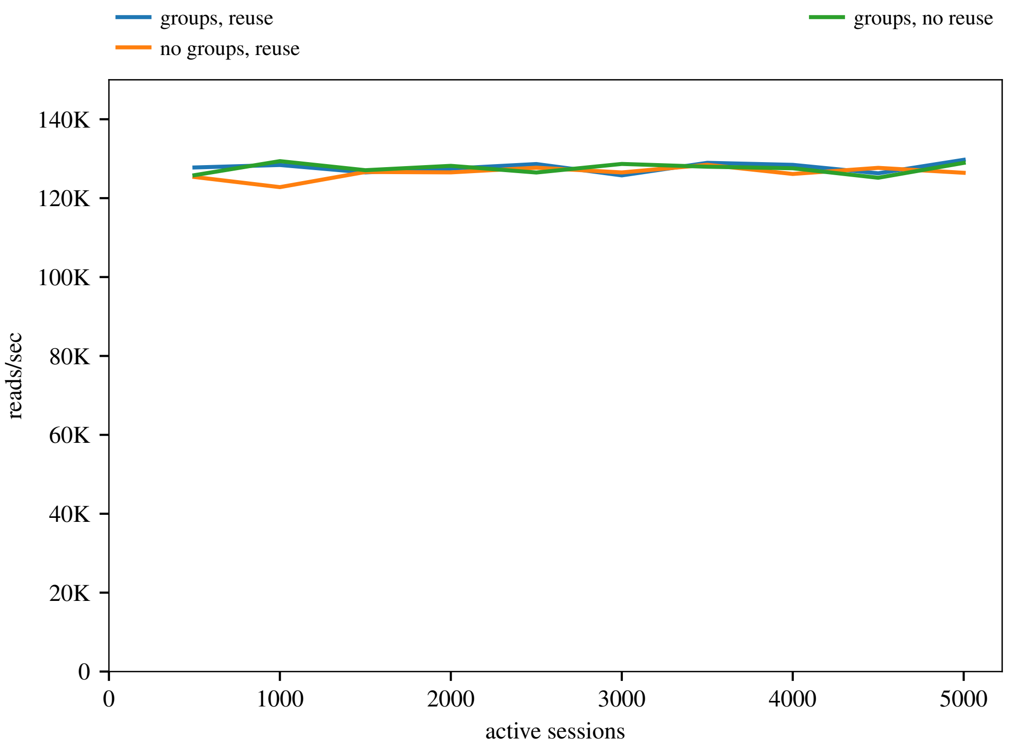
5.4 Conference management system

To evaluate the expressiveness of MultiverseDB, I analyzed the JConf application, a conference management system backend implemented in Jeeves [12] and examined which of

[1] MultiverseDB is limited to a single write processing thread. With several threads, it would achieve higher write performance.

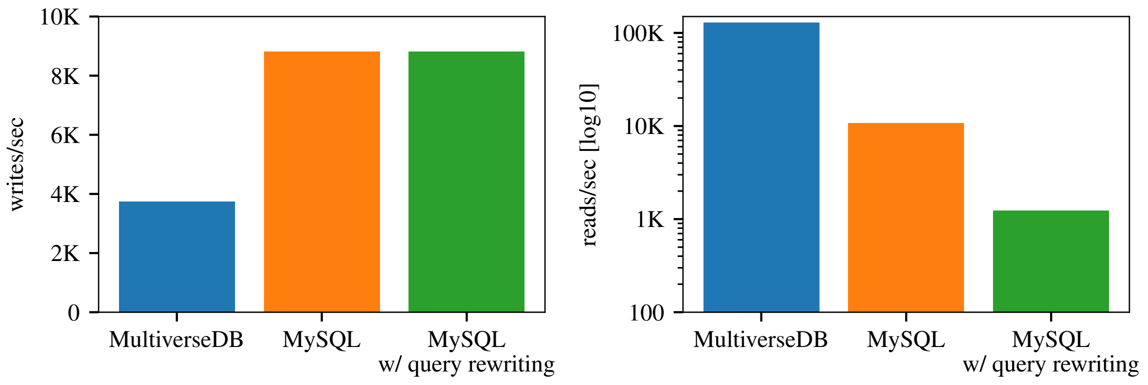


(a) Write performance



(b) Read performance

Figure 5-6: Read performance stays constant, while write throughput declines as the number of active session increases. Security groups and view reuse mitigate the decline in write throughput.



(a) Write performance (b) Read performance

Figure 5-7: With 5k active sessions, MultiverseDB greatly outperforms MySQL read performance, while achieving almost half of MySQL's write throughput. N.B.: log-scale y-axis in (b).

its security policies MultiverseDB is able to express. Table 5.1 shows the results of this analysis.

MultiverseDB supports 12 out of the 13 JConf policies. Policy R.3 requires an `EXIST` data-flow operator that MultiverseDB currently doesn't support.

Policies marked with an asterisk reference a configuration variable representing the phase (submission, rebuttal, review or public) of the submission system. JConf reads this variable from a static configuration file. MultiverseDB supports policies that refer to values not stored in the database, but changes to these values require the system administrator to deploy a new security configuration to reflect the new values.

	Policy	*Supported*	*Group*	*Type*
E.1	Everyone can view accepted papers during the public phase	✓*	✗	Row
E.2	Reviews are not available to the public	✓	✗	Row
R.1	Reviewers can view papers they are assigned to review	✓	✓	Row
R.2	Reviewers can view author names only after the rebuttal phase	✓*	✓	Column
R.3	Reviewers can see reviews for a paper after the review phase and if they submitted a review for that paper	✗	✓	Row
R.4	Reviewers can't see other reviewers identities	✓	✓	Column
P.1	PC members can see all the papers	✓	✓	Row
P.2	PC members can only view author names after the rebuttal phase	✓*	✓	Column
P.3	PC members can see all reviews	✓	✓	Row
P.4	PC members can view reviewers identities	✓	✓	Column
A.1	Authors can see their own papers	✓	✓	Row
A.2	Authors can see reviews for their own papers, but only after the review phase	✓*	✓	Row
A.3	Authors can't see reviewers identities	✓	✓	Column
Total		12 (out of 13)		

Table 5.1: Analysis of the JConf security policies MultiverseDB supports

Chapter 6

Conclusion

This thesis introduces and validates the MultiverseDB approach to secure databases for web applications. MultiverseDB can express complex security policies required by modern web applications and introduces techniques for enforcing security policies in a data-flow system that maintains materialized views. MultiverseDB also maintains per-user views of the database and quickly creates these views while scaling to thousands of active users sessions. Finally, MultiverseDB achieves high read performance with reasonable memory overhead, making it an viable alternative for read-heavy web applications that benefit from a secure database.

Bibliography

[1] Kristy Browder and M. Davidson. The virtual private database in Oracle9ir2. *Oracle Technical White Paper*, 2002.

[2] Surajit Chaudhuri, Tanmoy Dutta, and S. Sudarshan. Fine grained authorization through predicated grants. In *Proceedings of the IEEE 23rd International Conference on Data Engineering*. IEEE, April 2007.

[3] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 105–118, Berkeley, CA, USA, 2010. USENIX Association.

[4] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122–133. ACM, 2010.

[5] Dorothy E. Denning, Selim G. Akl, Matthew Morgenstern, Peter G. Neumann, Roger R. Schell, and Mark Heckman. Views for multilevel database security. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 156–156. IEEE, 1986.

[6] Sheldon Finkelstein. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 235–245, June 1982.

[7] Meteor Development Group. Meteor developer guide. `https://guide.meteor.com/`.

[8] Ashish Gupta and Iderpal Singh Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[9] Eddie Kohler. Hot crap! In *Proceedings of the Conference on Organizing Workshops, Conferences, and Symposia for Computer Systems*, pages 11:1–11:6, Berkeley, CA, USA, 2008. USENIX Association.

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

[11] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 551–562. ACM, 2004.

[12] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–96, New York, NY, USA, 2012. ACM.

[13] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 291–304, New York, NY, USA, 2009. ACM.