Towards Multiverse Databases

Alana Marzoev Lara Timbó Araújo[†] Malte Schwarzkopf Samyukta Yagati
Eddie Kohler[‡] Robert Morris M. Frans Kaashoek Sam Madden
MIT CSAIL [†] *MIT CSAIL and Airbnb* [‡] *Harvard University*

Abstract

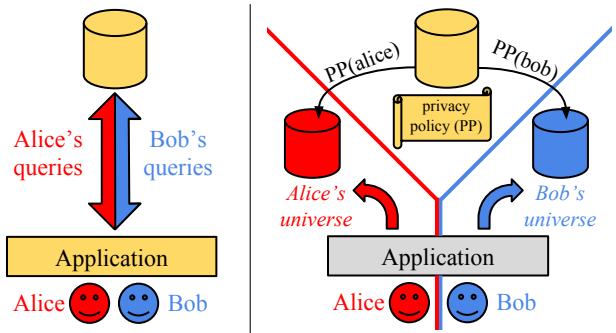
A *multiverse database* transparently presents each application user with a flexible, dynamic, and independent view of shared data. This transformed view of the entire database contains only information allowed by a centralized and easily-auditable privacy policy. By enforcing the privacy policy once, in the database, multiverse databases reduce programmer burden and eliminate many frontend bugs that expose sensitive data.

Multiverse databases' per-user transformations risk expensive queries if applied dynamically on reads, or impractical storage requirements if the database proactively materializes policy-compliant views. We propose an efficient design based on a joint dataflow across "universes" that combines global, shared computation and cached state with individual, per-user processing and state. This design, which supports arbitrary SQL queries and complex policies, imposes no performance overhead on read queries. Our early prototype supports thousands of parallel universes on a single server.

ACM Reference Format:
Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. 2019. Towards Multiverse Databases. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3317550.3321425

1 Introduction

Most web services store users' private, sensitive information in shared backend stores. Any frontend can access the whole store, regardless of the application user consuming the results. Therefore, frontend code is responsible for permission checks and privacy-preserving transformations that protect users' data. This is dangerous and error-prone, and has caused many real-world bugs in applications like HotCRP [25], WordPress [27], and Facebook [2, 4]: any omitted or incorrect check can leak private data, so the trusted computing base (TCB) effectively includes the entire application.



(a) Today. **(b)** Multiverse database.

Figure 1. Currently, web applications' entire frontend code is trusted (amber) and can query any backend data *(left, (a))*. A multiverse database applies a privacy policy (PP) to generate transformed, logical per-user databases, in which the untrusted applications' queries can only see permissible data *(right, (b))*.

It would be safer and easier to specify and transparently enforce access policies *once*, at the shared backend store interface. Although state-of-the-art databases have security features designed for exactly this purpose, such as row-level access policies and grants of views, these features are too limiting for many web applications. Application privacy policies are often data-dependent in ways incompatible with current row- and column-level access control, or allow the exposure of aggregate or transformed information that access control prevents. Prior research solutions based on query interposition and information flow control are slow, complex, or require impractical changes to the application programming model (§2).

In this paper, we make the case for *multiverse databases*, a flexible, easy-to-use, and performant paradigm that enforces declarative privacy policies within the store. The database applies policies for each user, filtering and transforming the base data to form a user-specific "parallel universe" database that contains only data that the user is allowed to see (Figure 1). Application code executing for a user can safely execute *any* query against the user's parallel universe database without risk of seeing (and perhaps inadvertently leaking) forbidden data. In essence, multiverse databases limit the TCB to the privacy policies and the database code enforcing them, defending against a threat model of inadvertently buggy application queries and privacy checks.[1]

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321425

[1] Defending against actively malicious applications would require considering side-channels, which we omit in this paper.

The security policies for a multiverse database are rooted in application-specific notions of data visibility. Consider a class discussion forum (e.g., Piazza [20]) that allows students to post questions that are anonymous to other students, but not anonymous to instructors. A multiverse database might express this privacy policy as follows:

```
table: Post,
-- user sees public posts and her own anonymous posts in full
-- (`ctx`, a universe-specific context, holds the user's ID)
allow: [ WHERE Post.anon = 0,
         WHERE Post.anon = 1 AND Post.author = ctx.UID ],
-- hide author of anonymous posts unless user is class staff
rewrite: [
  { predicate: WHERE Post.anon = 1 AND Post.class
      NOT IN (SELECT class FROM Enrollment
              WHERE role = "instructor" AND uid = ctx.UID),
    column: Post.author,
    replacement: "Anonymous" } ],
```

Given this policy, application code executing on a user's behalf can issue arbitrary queries without risking data leakage: unless the user is on the class staff, all anonymous posts consistently appear to have author "Anonymous" in every query. When a user issues multiple queries—e.g., one selecting and one counting a users' posts—the multiverse database returns semantically consistent results based on the contents of the user's universe. This removes, for example, a real-world inconsistency observed in Piazza, where students' total post count includes private posts invisible to the user [13]. Since multiverse databases transparently apply transformations in the database, application code need not be aware of them and can assume that it is talking to a conventional database.

Multiverse database privacy policies are general and user-extensible. Like row-level security [22] and view grants, they can hide rows and columns from individual users; like role-based access control, they can apply policies to groups of users; and like column masks [15], they can transform column values. But multiverse databases potentially also support powerful policies beyond the capabilities of existing solutions: for example, we are exploring policies that expose only differentially-private information about underlying tables.

A key challenge for the multiverse approach is query performance and space efficiency with many users. Web applications require fast reads, and applying policies on all data at query execution time is therefore unattractive. Applying the policies to the entire database ahead of time, however, explodes the space footprint with many users, and requires an update strategy when the underlying data changes. Fortunately, recent research on dataflow systems provides the key missing enabler for a multiverse database: dynamic, partially-stateful dataflows [11]. Stateful dataflow systems scale well when precomputing complex functions of dynamically changing data, and efficiently apply incremental updates. *Partial* state allows a dataflow to execute without materializing the full results and internal dataflow state. This permits selectively deferring parts of the computation to later read processing, and enables caching, therefore maintaining fast common-case reads at a modest, rather than explosive, space overhead. A *dynamic* dataflow can add new queries to the existing computation at runtime, offering the same query flexibility as classic databases within a streaming, incremental dataflow computation.

We describe a multiverse design that builds on this technology and realizes multiverse storage as a joint dataflow computation (§4). It transparently shares computation and policy-compliant intermediate data between users' universes, and relies on partial statefulness to grant the system freedom to choose what to precompute and cache, and what to compute on query execution. Initial results with an early prototype are encouraging (§5), and the multiverse approach and our design raise interesting questions for future research (§6).

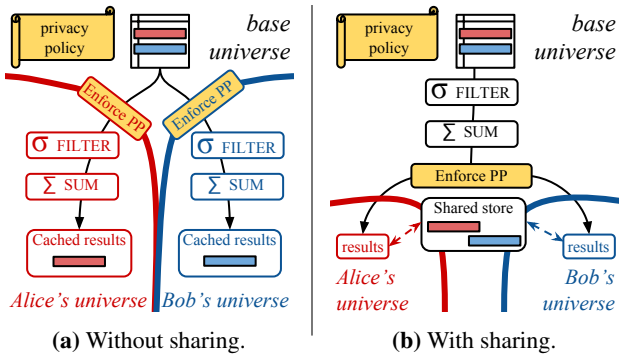
2 Existing approaches

Multiverse database universes, in effect, are per-user *views* of the database defined by privacy policies. Databases have relied on views as a security primitive since at least the 1970s [10, 14], but existing database views cannot substitute for the multiverse model. Defining views that specific users can access affords flexibility, but has the drawback that application developers must understand the view definitions and know which views to query. Algorithms to restate user queries in terms of *authorized views* mitigate this burden, but cannot map all queries and may spuriously reject queries even though the views support them [23]. Transparent *query rewriting* approaches, by contrast, avoid predefined views. Instead, they dynamically insert restrictions congruent with access policies into queries on execution [5, 8, 17]. Both view-based and query rewriting approaches increase the final query's complexity, slowing it down (e.g., by 3–10× in Qapla [17]).

Given the lack of performant database mechanisms, researchers have devised other techniques to apply privacy policies in web applications. General *information flow control* (IFC) provides a powerful approach that makes applications correct by construction [16, 30], but significantly complicates development. Domain-specific IFC systems can statically reject application code in violation of privacy policies [9] or extract policy-compliant implementations using program synthesis techniques [21], but couple privacy policies to a single program, while database techniques allow arbitrary queries. Other solutions embed multi-valued ("faceted") execution in common languages like Scala and Python, resolving data to concrete, policy-compliant values only on output [28, 29]. This has the advantage of keeping the application code policy-agnostic, but comes with substantial memory and runtime overheads, as the execution evaluates all alternative outcomes.

3 The multiverse approach

A multiverse database consists of a *base universe*, which represents the database without any read-side privacy policies applied, and many *user universes*, which are transformed copies of the database. Each user universe corresponds to the database view of a specific principal, typically an end-user authenticated



(a) Without sharing. **(b)** With sharing.

Figure 2. A multiverse database realized as a joint dataflow. For efficiency, universes can share computation and state (§4.2), as (b) shows for an identical query issued by Alice and Bob (here, without any group universes).

to a web application. The application code executing for this principal can only query its user universe, which appears to the application as a fully-fledged database. This makes multiverse databases easy to use—their application query interface is identical to that of normal databases—but requires maintaining many user universes, a potential performance bottleneck.

Multiverse databases maintain read query performance by *precomputing* per-user universes. These transformed databases have the privacy policies already applied, so queries to them execute as quickly as if the application applied the policies. However, precomputation requires storing and maintaining many user universes. Naïvely duplicating the entire database for each user would require prohibitive amounts of storage space, so the multiverse database must precompute only data that users actually need. Moreover, the multiverse database must efficiently update user universes' cached data when the underlying base universe data changes. In the following, we focus on how to efficiently apply read-side access policies that restrict and transform the data cached in user universes. §6 will discuss write access policies that restrict writes to the base universe; applications cannot write to user universes directly.

4 Making multiverse databases practical

A space- and compute-efficient multiverse database clearly cannot materialize all user universes in their entirety, and must support high-performance incremental updates to the user universes. It therefore requires partially-materialized views that support high-performance updates. Recent research has provided this missing key primitive [1, 11, 19]. Specifically, scalable, parallel streaming dataflow computing systems now support partially-stateful and dynamically-changing dataflows [11]. These ideas make an efficient multiverse database possible.

Our multiverse database design combines base-universe tables, privacy policies, and user universes into a single, joint dataflow. The database tables are the dataflow's root vertices, situated in the base universe (the source of ground truth). As the base universe is updated, records move through the dataflow into user universes, where subgraphs compute privacy policies and cache query results (Figure 2a). On each edge that crosses the universe boundary, the system interposes extra *enforcement operators*: special dataflow vertices that compute and apply the privacy policy's effect (e.g., pass, discard, transform) for each record that flows through them. The multiverse database's semantic consistency follows from the fact that enforcement operators for all applicable policies exist on any dataflow edge that crosses into a user universe. In other words, if a record written into the base universe flows into a user universe via multiple independent paths, the multiverse database enforces the same policies on each path.

When the application queries the multiverse database, it specifies an authenticated principal's ID alongside its query, and the system retrieves results from the matching user universe. If the system receives a query for the first time, it dynamically extends the user universe's dataflow with the required subgraph. Once a query is installed, its vertices remain in the dataflow; this facilitates caching, although the system can remove the query when it is no longer needed. Updates only ever flow from the base universe into other universes through enforcement operators, and information never flows back into the base universe or "sideways" across different user universes.

4.1 Specifying privacy policies

In principle, our design supports any policy expressible as an incremental, streaming dataflow computation. This chiefly requires that the policy be a deterministic function of a given update's record data and the database contents. Importantly, this permits data-dependent policies, which are common: consider, for example, class enrollment in Piazza.

Defining the right policy language is an important part of the multiverse database design. The policy language must strike a balance between expressivity and meaningful composition of policies, as well as ease of correct use for policy writers.

In our current prototype, privacy policies comprise expressions defined in a language similar to the security rules of Google's Cloud Firestore [12]. This language includes row suppression policies (the `allow` rules in §1's example) and column rewrite policies (`rewrite`) specified on the granularity of individual tables in the database schema. Policies decide whether to filter or mutate a record based on a relational predicate over the database contents. The language also supports group policies, which support role-based access control and save space and computation, and aggregation policies, which restrict user universes to seeing certain tables or columns only in aggregated or differentially-private form. Each policy expression in this language applies to a single database table T, which makes it easy to ensure semantic consistency. Specifically, the system must add enforcement nodes for all policies over T that apply to principal p on any path into p's user universe that records generated by updates to T can traverse. The system can determine these placement requirements through static analysis of the dataflow.

4.2 Sharing state and computation

A key challenge for multiverse databases is limiting the computation required on writes (which risks growing with the number of system users) and the space required to store user universes (which risks growing as fast or faster). Fortunately, the partially-stateful dataflow model can express optimizations that share computation and cached data between universes. Expressing the multiverse database as a joint partially-stateful dataflow is crucial to harnessing these optimizations. In the following, we showcase some promising optimizations that we have found to be beneficial, but this list is likely neither exhaustive nor sufficient for all applications and policies.

Group policies. Applications often have roles that cover multiple users, such as "students" and "instructors". These *user groups* have their own privacy policies. A group policy might e.g., allow teaching assistants (TAs) to see anonymous posts in classes they teach:

```
group: "TAs",
-- define TA group for each class
membership: SELECT uid, class_id AS GID FROM Enrollment
            WHERE role = "TA",
policies: [
-- show anonymous posts to TAs
-- (`ctx` is a group universe context, holds the group ID)
  { table: Post,
    allow: WHERE Post.anonymous = 1 AND ctx.GID = Post.class } ],
```

Note that this policy is a data-dependent group template: it defines one TA group per class (via `GID` from `membership`), i.e., adding a new class to `Enrollment` creates a new group.

Instead of computing this policy once on the boundary to each group member's user universe, the system applies the privacy policy once for all members. To achieve this, the system creates a *group universe* for each group, and routes inputs to queries affected by the group policy through this universe. From the group universe, records flow into each group member's user universe; at that boundary, user-specific policy operators may further restrict an individual's view, or a union with another path that applies a complementary user-specific policy may widen access. Using a group universe requires only one copy of the enforcement operators for the group's policies (rather than as many copies as the group has members), and shares cached, policy-compliant data in the group universe.

Sharing between queries. Web applications often issue similar queries for many users, and these queries share at least some privacy policies and results. By reasoning about all users' queries as a joint dataflow, the system can detect such sharing: when identical dataflow paths exist, they can be merged. All users and queries share computation in the base universe, so the system aims to maximize the computation in that universe by pushing the universe boundary as far "down" the queries' dataflow as possible while still maintaining correctness. Figure 2b shows how the system shares filter and sum operators in an identical query issued for both Alice and Bob. In this example, a privacy policy (which depends only on data in the group columns preserved by the sum) can be applied late, keeping most of the query's dataflow in the base universe.

Sharing across universes. Often, applications actually issue *identical* queries on behalf of many users, such as a query to retrieve the ten most recent posts to a class. Due to privacy policies, the queries' results may vary for different users, but they often overlap in part (e.g., all public posts). Instead of storing copies of identical records in many universes, the system can share these records across universes. It achieves this by backing logically distinct—but, in query terms, functionally equivalent—dataflow vertices with a shared physical record store. If a record reaches a vertex backed by such a store in universe u, the record's arrival indicates that u has access to it, so the system exposes the shared copy to universe u.

Partial materialization. As web applications are read-heavy, precomputing privacy policies and query results on write processing is more efficient than recomputing them on each read. To save space, however, the multiverse database may choose to precompute only part of a query (e.g., privacy policies only). The partially-stateful dataflow model allows the system to choose dynamically what results to precompute and cache, and how much computation to perform during read query execution. To achieve this, the system can decide which stateful operators in a given query to materialize, and which to compute on the fly on reads using the deferred evaluation supported by partially-stateful dataflow (through "upqueries" [11]). Partially-stateful dataflow also supports evicting records from operators' state, which helps further restrict cached results to frequently-read records. The specific choice of what to materialize may vary according to a query's popularity, overall system load, and the available memory.

4.3 Dynamic universe creation

At any time, many users of a web application are likely inactive. During those times, the multiverse database need not maintain a universe for these users. Instead, it should create and destroy user universes on demand—e.g., on application-level session creation and termination. For an interactive user experience, the creation and destruction of user universes must be fast and permit other users to concurrently interact with the database.

Partially-stateful dataflow supports downtime-free dataflow changes, which make this feasible: a new user universe starts out with empty state and populates itself as queries execute. This bootstrapping can be fast, as the user universe can often derive its data efficiently from cached intermediate results in the base or group universes. Creation and destruction of group universes relies on the same live dataflow change mechanisms.

4.4 Consistency

Enforcing privacy policies to all records that cross into a user universe makes the multiverse database *semantically* consistent: different queries will not expose contradictory results that are impossible to see with a classic database.

	reads/sec	writes/sec
Multiverse database	129.7k	3.7k
MySQL (with AP)	1.1k	8.8k
MySQL (without AP)	10.6k	8.8k

Figure 3. Our prototype achieves high read throughput compared to MySQL queries that execute privacy policies inline. Write throughput is lower than MySQL's as the multiverse database does more work on writes.

The actual consistency observed by clients reading from the multiverse database's caches at runtime, however, depends on the guarantees offered by the underlying dataflow implementation. Dataflow systems can guarantee strong consistency—i.e., that updates take effect in all queries at the same logical time and reads always see a consistent snapshot—using progress tracking protocols [18]. Global progress tracking requires coordination between parallel processors, which reduces scalability and may be costly for a multiverse database's large dataflows. Uncoordinated, eventually-consistent dataflow scales well [11, §8.3], but makes no guarantees as to when different universes and queries see the effects of an update to the base universe. Hence, data-dependent policies may temporarily expose data to a user universe in such a regime: a new record might race with an update that makes a data-dependent policy hide it.

A multiverse database can somewhat restrict coordination, however: since no client ever combines data from different user universes, the system can allow state in different user universes to diverge. Dataflow models that support such local coordination are an interesting direction for future research.

5 Proof of concept

We have implemented an early prototype multiverse database based on our design as an extension to Noria [11]. The prototype implements row suppression, rewrite, and group policies, and relies on Noria's automatic reuse of dataflow operators [11, §5.1] to realize the sharing described in §4, with exception of the shared record store. We added about 2,000 lines to Noria's Rust implementation and ran experiments at revision `15f0492`.

We measure the prototype's performance for a Piazza-style class forum and a privacy policy that allows TAs to see anonymous posts on a database containing 1M posts and 1,000 classes. For reads, the benchmark repeatedly queries all posts authored by different users, and write operations insert new posts into a class. We compare: *(a)* our prototype with 5,000 active user universes; *(b)* MySQL running the same workload with privacy policies inlined in the query; and *(c)* MySQL without any privacy policies. Our prototype currently materializes the full query results in memory, and its base database tables are stored in RocksDB [24].

Figure 3 shows the results. As expected, reads from the multiverse database's precomputed, cached results are fast. By contrast, evaluating the privacy policy as part of the query slows down MySQL reads by 9.6× compared to issuing a straight query; with simpler policies, such as one that merely filters other users' anonymous posts, MySQL sees a smaller slowdown. The write throughput of our prototype is roughly half of what MySQL supports, and though the precise difference is largely an implementation artifact, this is an encouraging result. A multiverse database fundamentally must do more work on writes than MySQL's inserts, as writes must propagate through the dataflow. In this experiment, the dataflow fully updates 5,000 user universes; making some state partial would increase write throughput at the expense of slower reads.

Finally, we measured process memory use as we increased the number of active universes from one to 5,000. The memory footprint increased from 0.5 GB with one universe to 1.1 GB with 5,000 universes; this 600 MB footprint is about half of the 1.2 GB needed without group universes. However, this overhead can be reduced further: for example, a separate microbenchmark showed that using a shared record store for identical queries reduces their space footprint by 94%.

These results are encouraging, but a realistic multiverse database must further reduce memory overhead and efficiently run millions of user universes across machines. Neither Noria nor any other current dataflow system support execution of the huge dataflows that such a deployment requires. In particular, changes to the dataflow must avoid full traversals of the dataflow graph for faster universe creation.

6 Discussion and research directions

Our prototype shows that the promise of multiverse databases is within reach, even for challenging applications with high performance requirements. Flexible, application-specific privacy policies are expressed within the store, and therefore obeyed transparently by applications, with an easy query interface and little to no query execution overhead. Furthermore, though challenges remain, the partially-stateful dataflow model can express key optimizations that limit multiverses' space overhead. We believe the multiverse model can make even complex web applications robust to accidental information exposure and therefore faster to safely build. Moreover, the multiverse concept suggests interesting directions for future research.

Write authorization policies. Our current prototype only applies privacy polices to read queries and allows all users to write to database tables without enforcing any privacy policies. But applications need write-side policies, too: otherwise users might, for instance, change their own role. For example, Piazza may need a write policy specifying that only instructors can enroll other users as instructors or TAs:

```
table: Enrollment,
-- only allow existing instructors to make other users instructors
write: [ {
  column: Enrollment.role,
  values: [ "instructor", "TA" ]
  predicate: WHERE ctx.UID IN (SELECT uid FROM Enrollment
                               WHERE role = "instructor"), } ]
```

A multiverse database might apply such write authorization policies in several ways. The simplest is perhaps to check permissions when applying writes to tables, just like today's databases do. This allows policies that prohibit writes that users might exploit to raise their privileges, and simple filters on the data written or current table contents are sufficient to support policies like the Piazza one. An alternative approach with more expressive power might feed writes through a policy dataflow *before* applying them to the base universe. This supports write authorization policies dependent on data in other tables and policies that require complex computation. Such an approach raises consistency challenges, however: an eventually-consistent write authorization dataflow might erroneously admit writes because the policy evaluation *itself* might observe temporarily inconsistent or intermediate state via data-dependencies. Hence, a write authorization dataflow may require transactional abstractions that atomically process updates until they are admitted to the base universe or rejected.

Differentially-private aggregations. A privacy policy may permit users to run aggregate queries over sensitive records that they cannot see individually. For example, a medical web application might allow a user to query the number of patients with diabetes by ZIP code, even if this user is not authorized to view individual records:

```
SELECT COUNT(*) FROM diagnoses
    WHERE diagnosis = "diabetes" GROUP BY zip;
```

The policy might further desire that revealing such aggregates leaks no information about whether any individual, hidden patient record is part of the aggregate. A multiverse database can rewrite any aggregation that matches such a privacy policy into a differentially-private (DP) aggregation. DP adds noise to the output to hide the impact of individual records, and in the multiverse database setting must allow for continuous updates to underlying data. The continuous, event-based DP algorithm by Chan *et al.* [7] is suitable for a streaming dataflow setting, and we implemented a prototype `COUNT` operator using this algorithm. In microbenchmark experiments, the operator's output was within 5% of the true count after processing about 5,000 updates. Yet, open research questions remain: for example, how do DP aggregation policies compose with other policies? Does a DP policy prohibit other, unrelated queries (e.g., joins)? How should the system handle multiple DP aggregations over the same table?

User-defined policy operators. Our prototype currently defines privacy policies using SQL expressions. This is sufficient for common policies, such as matching against a privacy control list (ACL) stored in a database table. Some applications' privacy policies, however, may rely on external information (e.g., an ACL file) or custom behavior (e.g., a user-defined function). The multiverse database's policy language ought to permit such custom functions, but the right API is an open question: custom operators must satisfy dataflow operator requirements (e.g., determinism), and they must correctly compose with other privacy policies. A domain-specific language for writing such operators, offering access to a limited API to external state, might be a promising starting point.

Policy correctness. Both write-side and read-side privacy policies must be consistent and complete. Developer error can yield policies with non-obvious internal contradictions, or with gaps that leak information. For applications that have large policies consisting of many clauses, checking the policies by hand is impractical, and automated tools will be required. Such policy tools should detect impossible (i.e., contradictory), and incomplete policies (i.e., those not covering all cases). We believe that developing a sound policy-checker for a multiverse database, perhaps using ideas similar to Amazon's SMT-based policy checker for AWS [3], is an interesting challenge.

Verified policy compilation. The multiverse database's TCB includes the privacy policy, as well as the logic that compiles it into dataflow and injects enforcement operators into queries. Ideally, these transformations would be formally verified to ensure that the final dataflow indeed enforces the privacy policy. Recent advances in constructing formally-verified just-in-time compilers [26] may provide some ideas, although the multiverse database must also reason about the existing dataflow. For example, transforming a new policy into a functionally-correct joint dataflow may require adding new dataflow nodes into existing queries.

Universe peepholes. Applications sometimes let users assume other users' identities, but this begets bugs such as Facebook's recent access token exposure [6], which allowed users to view other users' access tokens via the site's "View Profile As" feature. In a multiverse database, it might be tempting to support such a "View As" feature by temporarily allowing Bob to access Alice's universe, but this is dangerous: Alice's access tokens are visible inside her universe (and only there)! Plausible solutions might involve creating a temporary "extension universe" to Alice's universe, and applying a privacy policy that blinds the tokens at that boundary.

7 Conclusion

Multiverse databases are a promising approach that makes common bugs in today's web applications harmless. Our initial results indicate that a large, dynamic, and partially-stateful dataflow can support practical multiverse databases that are easy to use and achieve good performance and acceptable overheads. We are excited to further explore the multiverse database paradigm and associated research directions.

Acknowledgements

We thank the anonymous reviewers and members of the MIT PDOS group for helpful comments on earlier versions of the paper. This work was funded through NSF awards CNS-1301934, CNS-1704172, and CNS-1704376.

References

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views". In: *Proceedings of the VLDB Endowment* 5.10 (June 2012), pages 968–979.

[2] Warwick Ashford. *Facebook photo leak flaw raises security concerns*. URL: https://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns (visited on 01/04/2019).

[3] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. "Semantic-based Automated Reasoning for AWS Access Policies using SMT". In: *Proceedings of the 18th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Austin, Texas, USA, Oct. 2018.

[4] Tomer Bar. *Notifying our Developer Ecosystem about a Photo API Bug*. URL: https://developers.facebook.com/blog/post/2018/12/14/notifying-our-developer-ecosystem-about-a-photo-api-bug/ (visited on 12/14/2018).

[5] Kristy Browder and Mary Ann Davidson. *The virtual private database in Oracle9iR2*. Oracle Technical White Paper, Oracle Corporation. 2002.

[6] Pedro Canahuati and Guy Rosen. *Security Update – Additional Technical Details*. URL: https://newsroom.fb.com/news/2018/09/security-update/#details (visited on 12/14/2018).

[7] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. "Private and Continual Release of Statistics". In: *ACM Transactions on Information and System Security* 14.3 (Nov. 2011), 26:1–26:24.

[8] Surajit Chaudhuri, Tanmoy Dutta, and S. Sudarshan. "Fine Grained Authorization Through Predicated Grants". In: *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE)*. Istanbul, Turkey, Apr. 2007, pages 1174–1183.

[9] Adam Chlipala. "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, Oct. 2010.

[10] Dorothy E. Denning, Selim G. Akl, Mark Heckman, Teresa F. Lunt, Matthew Morgenstern, Peter G. Neumann, and Roger R. Schell. "Views for Multilevel Database Security". In: *IEEE Transactions on Software Engineering* SE-13.2 (Feb. 1987), pages 129–140.

[11] 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". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018, pages 213–231.

[12] Google, Inc. *Cloud Firestore Documentation: Writing conditions for Cloud Firestore Security Rules*. URL: https://firebase.google.com/docs/firestore/security/rules-conditions (visited on 01/15/2019).

[13] Matthew Green. *Piazza offers anonymous posting, but does not hide each user's total number of posts. Discuss.* Twitter post. URL: https://twitter.com/matthew_d_green/status/925053953330634753 (visited on 03/08/2019).

[14] Patricia P. Griffiths and Bradford W. Wade. "An Authorization Mechanism for a Relational Database System". In: *ACM Transactions on Database Systems* 1.3 (Sept. 1976), pages 242–255.

[15] 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 01/15/2019).

[16] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. "Information Flow Control for Standard OS Abstractions". In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, 2007, pages 321–334.

[17] 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)*. Vancouver, British Columbia, Canada, Aug. 2017, pages 1463–1479.

[18] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 439–455.

[19] Milos Nikolic, Mohammad Dashti, and Christoph Koch. "How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates". In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. San Francisco, California, USA, 2016, pages 511–526.

[20] Piazza Technologies, Inc. *Piazza*. URL: https://piazza.com/ (visited on 01/08/2019).

[21] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. "Type-Driven Repair for Information Flow Security". In: *CoRR* abs/1607.03445 (2016). arXiv: 1607.03445.

[22] 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 01/14/2019).

[23] Shariq Rizvi, Alberto Mendelzon, S. 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*. Paris, France, 2004, pages 551–562.

[24] Facebook Open Source. *A persistent key-value store for fast storage environments*. Apr. 2018. URL: http://rocksdb.org/ (visited on 04/20/2018).

[25] Ben Stock. *Search leaks hidden tags*. URL: https://github.com/kohler/hotcrp/issues/135 (visited on 01/08/2019).

[26] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. "Jitk: A Trustworthy In-Kernel Interpreter Infrastructure". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014.

[27] Wordpress Vulnerability Database. *CVE 2016-5835: Authenticated Revision History Information Disclosure*. URL: https://wpvulndb.com/vulnerabilities/8519 (visited on 01/04/2019).

[28] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, Dynamic Information Flow for Database-backed Applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, California, USA, June 2016, pages 631–647.

[29] 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 (POPL)*. Philadelphia, Pennsylvania, USA, Jan. 2012, pages 85–96.

[30] 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 (OSDI)*. Big Sky, Montana, USA, 2009, pages 291–304.