Ultraverse: Efficient Retroactive Operation for Attack Recovery in Database Systems and Web Frameworks

Ronny Ko[1,4], Chuan Xiao[2,3], Makato Onizuka[2], Yihe Huang[1], Zhiqiang Lin[4]
[1]Harvard University, [2]Osaka University, [3]Nagoya University, [4]Ohio State University
{hrko,yihehuang,mickens}@g.harvard.edu,{chuanx,onizuka}@ist.osaka-u.ac.jp

Abstract

Retroactive operation is an operation that changes a *past* operation in a series of committed ones (e.g., cancelling the past insertion of '5' into a queue committed at *t=3*). Retroactive operation has many important security applications such as attack recovery or private data removal (e.g., for GDPR compliance). While prior efforts designed retroactive algorithms for low-level data structures (e.g., queue, set), none explored retroactive operation for higher levels, such as database systems or web applications. This is challenging, because: *(i)* SQL semantics of database systems is complex; *(ii)* data records can flow through various web application components, such as HTML's DOM trees, server-side user request handlers, and client-side JavaScript code. We propose Ultraverse, the first retroactive operation framework comprised of two components: a database system and a web application framework. The former enables users to retroactively change committed SQL queries; the latter does the same for web applications with preserving correctness of application semantics. Our experimental results show that Ultraverse achieves 10.5x~693.3x speedup on retroactive database update compared to a regular DBMS's flashback & redo.

1 Introduction

Modern web application services are exposed to various remote attacks such as SQL injection, cross-site scripting, session hijacking, or even buffer overflows [27, 31, 41]. To recover from the attack damages, the application code often needs to be reconfigured or patched, and the application's state polluted by the attack has to be retracted. Retroactive operation is particularly important, especially for the service that hosts many users and manages their financial assets, because their financial data may have been affected by the attack and put into invalid state.

Concretely, Figure 1 shows an example of an online banking web server's user request handler that transfers money from one user's account to another. The server queries the `Accounts` table to check if the sender has enough balance; if so, the server subtracts the transferring `amount` from the sender's balance and adds it to the receiver's balance. But suppose that some `SendMoney` request was initiated by a remote attacker (e.g., via request forgery or session hijacking). This will initially corrupt the server's `Accounts` table, and as time flows, its tampered data will flow to other tables and put the entire database into a corrupted state. To recover the database to a good state, the most naive and intuitive solution is the following steps: *(1)* roll back the database to just before the SQL query's commit time of the malicious user request; *(2)* either skip the malicious query or sanitize its tampered amount to a benign value; *(3)* replay all subsequent queries. But this naive solution suffers two problems: efficiency and correctness.

```
1: function SendMoney (sender, receiver, amount):
2:   var sender_balance = SQL_exec("SELECT balance
3:    FROM Accounts WHERE id = " + sender)
4:   if (sender_balance >= amount):
5:     sender_balance -= amount
6:     SQL_exec("UPDATE Accounts SET balance = "
7:      + sender_balance + " WHERE id = " + sender)
8:     SQL_exec("UPDATE Accounts SET balance += "
9:      + amount + "WHERE id = " + receiver)
```

Figure 1: An online banking server's money transfer handler.

- **Efficiency:** It is critical for any financial institution to investigate an attack and resume its service in the shortest possible time, because every second of the service downtime is its financial loss. Provided this, performing naive rollback & replay of all past queries is an inefficient solution. In fact, it would be sufficient to selectively rollback & replay only those queries whose results are affected by the problematic query. Unfortunately, no prior work has proposed such a more efficient database *rollback-fix-replay* technique covering all types of SQL semantics (i.e., retroactive database system).
- **Correctness:** Even if there existed an efficient retroactive database system, this would not provide the correctness of the application state. This is because replaying only database queries does not replay and re-reflect what has occurred in the application code, which can lead to incorrect database state from the application semantics. In Figure 1, the `SendMoney` handler is essentially an application-level transaction comprised of 3 SQL queries: `SELECT`, $\texttt{UPDATE}_1$, and $\texttt{UPDATE}_2$. Among them, $\texttt{UPDATE}_2$ takes `sender_balance` as an input dynamically computed by the application code (line 5). If we only *rollback-fix-replay* the SQL logs of the DB system, this will not capture the application code's `sender_balance` variable's new value that has to be recomputed during the replay phase, and will instead use the stale old value of `sender_balance` recorded in the prior $\texttt{UPDATE}_1$ query's SQL log.

Since there exists no automated system-level technique to retroactively update both a database and application state, most of today's application service developers use a manual approach of handcrafting compensating transaction [57], whose goal is to bring in the same effect of executing retroactive operation on an application service. However, as a system's complexity grows, ensuring the correctness of compensating transactions is challenging [46], because their dependencies among SQL queries and data records become non-trivial for developers to comprehend. In the SQL level, although materialized views [11] can reflect changes in their base tables' derived computations, they cannot address complex cases such as: *(i)* views have circular dependencies; *(ii)* computations derived from base tables involve timely recurring queries [38]. Indeed, major financial institutions have suffered critical software glitches in compensating transactions [60].

Some traditional database system techniques are partially relevant to addressing these issues: temporal or versioning databases [34,

39] can retrieve past versions of records in tables; database provenance [36] traces lineage of data records and speculates how a query's output would change if inputs to past queries were different; checkpoint and recovery techniques [61] rollback and replay databases by following logs; retroactive data structure [19] can retroactively add or remove past actions on a data object. Unfortunately, all existing techniques are problematic. First, their retroactive operations are either **limited in functionality** (i.e., supports only a small set of SQL semantics, without supporting `TRIGGER` or `CONSTRAINT`, for example) or **inefficient** (i.e., exhaustive rollback and replay of all committed queries). Second, no prior arts address how to perform retroactive operations for web applications which involve data flows outside SQL queries– this includes data flows in application code (e.g., webpage's DOM tree, client's JavaScript code, server's user request handler) as well as each client's local persistent storage/databases besides the server's own database.

To address these problems, we propose Ultraverse, which is composed of two components: a database system and a web application framework. The **Ultraverse database system** (§3) is designed to support efficient retroactive operations for queries with full SQL semantics. To enable this, Ultraverse uses five techniques: *(i)* it records column-wise read/write dependencies between queries during regular service operations and generates a query dependency graph; *(ii)* it further runs row-wise dependency analysis to group queries into different clusters such that any two queries from different clusters access different rows of tables, which means the two queries are mutually independent and unaffected; *(iii)* it uses the dependency graph to roll back and replay *only* those queries dependent to the target retroactive query both column-wise and row-wise; *(iv)* during the replay, it runs multiple queries accessing different columns in parallel for fast replay; *(v)* it uses Hash-jumper which computes each table's state as a hash (to track its change in state) upon each query commit and uses these hashes to decide whether to skip unnecessary replay during retroactive actions (i.e., when it detects a table's hash match between the regular operation and the retroactive operation). Importantly, Ultraverse is seamlessly deployable to any SQL-based commodity database systems, because Ultraverse's query analyzer and replay scheduler run with an unmodified database system.

The **Ultraverse web application framework** (§4) is designed to retroactively update the state of web applications whose data flows occur both inside and outside the database, with support of all essential functionalities common to modern web application frameworks. The Ultraverse framework provides developers with a *uTemplate* (Ultraverse template), inside which they define the application's user request handlers as SQL `PROCEDURE`s. The motivation of *uTemplate* is to enforce all data flows of the application to occur only through SQL queries visible to the Ultraverse database. Using *uTemplate* also fundamentally limits the developers' available persistent storage to be only the Ultraverse database. During regular service operations, each client's user request logs are transparently sent to the server whenever she interacts with the server. The server replays these logs to mirror and maintain a copy of each client's local database, with which the server can retroactively update its server-side database even when all clients are offline. *uTemplates* are expressive enough to implement various web application logic (e.g., accessing the DOM tree to read user inputs and display results on the browser's screen). *Apper* (application code generator) converts each *uTemplate* into its equivalent application-level user request handler and generates the final application code for service deployment.

We have implemented Ultraverse and compared its speed to MariaDB's rollback & replay after retroactive database update (§5). Ultraverse achieved 10.5x∼693.3x speedup across various benchmarks. We also evaluated Ultraverse on popular open-source web services (InvoiceNinja [17]) and machine-learning data analytics (Apache Hivemall [32]), where Ultraverse achieved 333.5x-695.6x speedup in retroactive operation. As a general-purpose retroactive database system and web framework, we believe Ultraverse can be used to fix corrupt states or simulate different states (i.e., *what-if* analysis [21]) of various web applications such as financial service, e-commerce, logistics, social networking, and data analytics.

Contributions. We make the following contributions:

- Designed the first (efficient) retroactive database system.
- Designed the first web application framework that supports retroactive operation preserving application semantics.
- Developed and evaluated Ultraverse, our prototype of retroactive database system and web application framework.

2 Overview

Problem Setup: Suppose an application service is comprised of one or more servers sharing the same database, and many clients. An attacker maliciously injected/tampered with an SQL query (or an application-level user request) of the application service, which made the application's database state corrupted. All application data to recover are in the same database. We identified the attacker-controlled SQL query (or user request) committed in the past.

Goal: Our goal is to automatically recover the application's corrupted database state, by retroactively removing or sanitizing the attacker-controlled SQL query (or user request) committed in the past. This goal should be achieved: *(i) efficiently* by minimizing the recovery delay; and *(ii) correctly* not only from the low-level SQL semantics, but also from the high-level application semantics.

Retroactive Operation: Consider a database $\mathbb{D}$, a set $\mathbb{Q}$ of queries Q_i where i represents the query's commit order (i.e., query index), and Q'_τ is the target query to be retroactively added, removed, or changed at the commit order τ within $\{Q_1, Q_2, ...Q_\tau...Q_{|\mathbb{Q}|}\} = \mathbb{Q}$. In case of retroactively adding a new query Q'_τ, Q'_τ is to be inserted (i.e., executed) right before Q_τ. In case of retroactively removing the existing query Q_τ ($i.e., Q'_\tau = Q_\tau$), Q_τ is to be simply removed in the committed query list. In case of retroactively changing the existing query Q_τ to Q'_τ, Q_τ is to be replaced by Q'_τ. The retroactive operation on the target query Q'_τ is equivalent to transforming $\mathbb{D}$ to a new state that matches the one generated by the following procedure:

(1) **Rollback Phase**: roll back $\mathbb{D}$'s state to commit index $\tau - 1$ by rolling back $Q_{|\mathbb{Q}|}, Q_{|\mathbb{Q}-1|}, \ldots Q_{\tau+1}, Q_\tau$.
(2) **Replay Phase**: do one of the following:
   - To retroactively add Q'_τ, execute Q'_τ and then replay $Q_\tau, \ldots Q_{|\mathbb{Q}|}$.
   - To retroactively remove Q'_τ, replay $Q_{\tau+1}, \ldots Q_{|\mathbb{Q}|}$.
   - To retroactively change Q_τ to Q'_τ, execute Q'_τ and then replay $Q_{\tau+1}, \ldots Q_{|\mathbb{Q}|}$.

Instead of exhaustively rolling back and replaying all $Q_\tau, \ldots Q_{|\mathbb{Q}|}$, Ultraverse picks only those queries whose results depend on the retroactive target query Q'_τ. To do this, Ultraverse analyzes query dependencies based on the read and write sets of each query. Our

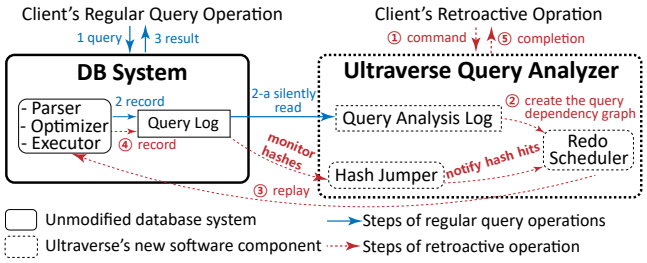


Figure 2: The Ultraverse database system's architecture.

proposed granularity for expressing the read and write sets is table columns, the finest database granularity we can obtain from query statements only. Also, we will consider the effect of schema evolution including `TRIGGER` creation/deletion, as well as `TRANSACTION` or `PROCEDURE` which binds and executes multiple queries together.

3 Ultraverse's Database System

Figure 2 depicts Ultraverse's database architecture. At a high level, Ultraverse's query analyzer runs with a unmodified database system. While the database system serves a user's regular query requests and records them to the query log, Ultraverse's query analyzer reads the query log in the background and records two additional pieces of information: *(a)* read-write dependencies among queries, *(b)* the hash values of each table updated by each committed query. When a user requests to retroactively add, remove, or change past queries, Ultraverse's query analyzer analyzes the query dependency log and sends to the database system the queries to be rolled back and replayed. Ultraverse's replay phase executes multiple non-dependent queries in parallel to enhance the speed, while guaranteeing the final database state to be strongly serialized (as if all queries are serially committed in the same order as in the past).

3.1 A Motivating Example

Figure 3 is a motivating example of Ultraverse's query dependency graph for an online banking service, comprised of 4 tables and 1 trigger. The `Users` table stores each user's id and social security number. The `Accounts` table stores each user's account number and its current balance. The `Transactions` table stores each record of money transfer as the tuple of the sender & receiver's account numbers and the transfer amount. The `Statements` table stores each account's monthly transactions history. The `BalanceCheck` trigger checks if each account has an enough balance before sending out money.

Initially, the service creates the `Users`, `Accounts`, `Transactions`, and `Statements` tables (Q1~Q4), as well as the `BalanceCheck` trigger (Q5). Then, Alice and Bob's user IDs and accounts are created (Q6~Q9). Alice's account transfers $100 to Bob's account (Q10). Charlie creates his ID and account (Q11~Q12). Alice's monthly bank statement is created (Q13). Now, suppose that Q10 turns out to be an attacker-controlled money transfer and the service needs to retroactively remove it. Ultraverse's 1st optimization goal is to rollback & replay only Q5, Q10, Q12, and Q13, skipping Q11, because the columns that Q11 reads/writes (`Users.*`) are unaffected by the retroactively removed Q10.

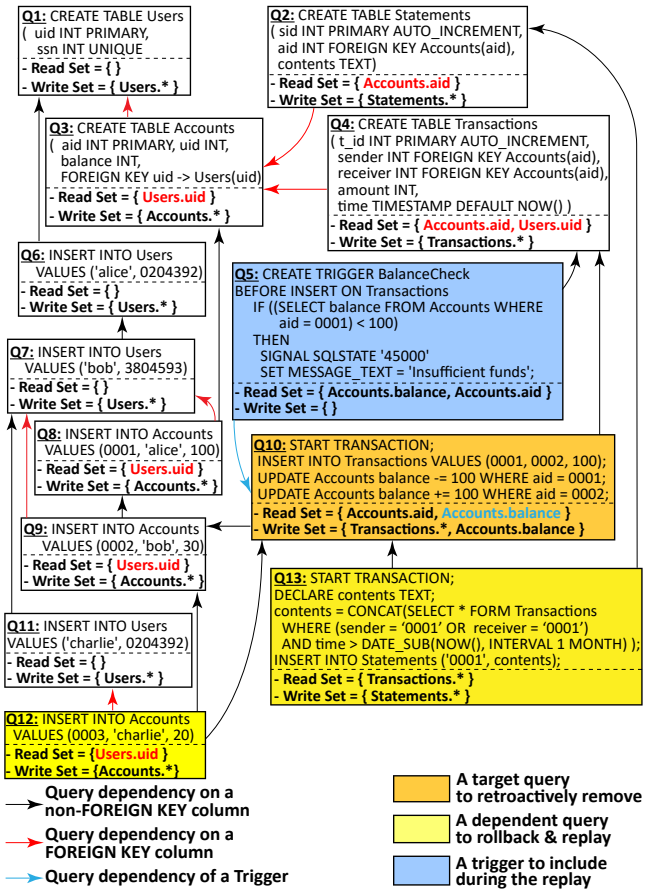


Figure 3: Online banking service's query dependency graph.

3.2 Column-wise Read/Write Dependency

Table A in §A[1] shows all types of SQL queries that Ultraverse supports for query dependency analysis. In the table, each query has a read set (R) and a write set (W), which are used to deduce query dependencies. A query's R set contains a list of columns of tables/views that the query reads during execution. A W set is a list of columns of tables/views that the query updates during execution. Besides the descriptions in Table A, we add the following remarks:

- The R/W set policy for `CREATE/ALTER TABLE` is applied in the same manner for creating/altering `CONSTRAINT` or `INDEX`.
- If an `ALTER TABLE` query dynamically adds a `FOREIGN KEY` column to some table, then the R/W set policy associated with this newly added `FOREIGN KEY` column is applied only to those queries committed after this `ALTER TABLE` query.
- `VIEW`s are updatable. If a query `INSERT`, `UPDATE` or `DELETE` a view, the original table/view columns this view references are also cascadingly included in the query's W set.
- As for branch conditions, it is difficult to statically correctly predict which direction will be explored during runtime, because the direction may depend on the dynamically evolving state of the database. To resolve this uncertainty issue, Ultraverse assumes that each conditional branch statement in a `PROCEDURE`, `TRANSACTION`, or `CREATE TRIGGER` query to explores both directions (i.e.,

[1] Appendix URL: https://drive.google.com/file/d/11plMlhaUv1neCm4WTk6BD4G5r5DLEDo_

Notations	
Q_n	: *n-th* committed query
τ	: The retroactive target query's index
T_x	: Query "`CREATE TRIGGER x`"
$R(Q_n), W(Q_n)$	: Q_n's read & write sets
c	: a table's column
$Q_n \rightarrow Q_m$	: Q_n depends on Q_m
$Q_n \triangleright T_x$	: Q_n is a query that triggers T_x
$A \Longrightarrow B$	: If A is true, then B is true
Column-wise Query Dependency Rule	
1.	$\exists c((c \in W(Q_m)) \wedge (c \in (R(Q_n) \cup W(Q_n)))) \wedge (m < n) \Longrightarrow Q_n \rightarrow Q_m$
2.	$(Q_n \rightarrow Q_m) \wedge (Q_m \rightarrow Q_l) \Longrightarrow Q_n \rightarrow Q_l$
3.	$(Q_n \rightarrow Q_\tau) \wedge (Q_n \triangleright T_x) \Longrightarrow T_x \rightarrow Q_\tau$

Table 1: Ultraverse's column-wise query dependency rules.

Ultraverse merges the R/W sets of the true and false blocks) . This strategy leads to an over-estimation of R/W sets, and thereby the dependency graph's size is potentially larger than optimal. At this cost, we ensure the correctness of retroactive operation.

3.3 Query Dependency Graph Generation

Ultraverse's query analyzer records the R/W sets of all committed queries during regular service operations. This information is used for retroactive operation: serving a user's request to retroactively remove, add, or change past queries and updating the database accordingly. Ultraverse accomplishes this by: *(i)* rolling back only those tables accessed by the user's target query and its dependent queries; *(ii)* removing, adding, or changing the target query; *(iii)* replaying only its dependent queries. To choose the tables to roll back, Ultraverse creates a query dependency graph (Figure 3), in which each node is a query and each arrow is a dependency between queries.

In Ultraverse, if executing one query could affect another query's result, the latter query is said to *depend on* the former query. Table 1 defines Ultraverse's four query dependency rules. Rule 1 states that if Q_m writes to a column of a table/view and later Q_n reads/writes the same column, then Q_n depends on Q_m. In the example of Figure 3, Q12→Q11, because Q12 reads the `Users.uid` (foreign key) column that Q11 wrote to. Note our query dependency differs from the dependency in conflict graphs [64], which includes *read-then-write*, *write-then-read*, and *write-then-write* cases. In contrast, our rule excludes the *read-write* case, because the prior query's read operation does not affect the values that the later query writes. Rule 2 states that if Q_n depends on Q_m and Q_m depends O_l, then Q_n also depends on Q_l (transitivity). In Figure 3, Q12→Q7, because Q12→Q11 and Q11→Q7. Rules 3 and handles triggers. Rule 3 states that if Q_n depends on Q_τ (the retroactive target query), then we enforce T_x (a trigger linked to Q_n) to depend on Q_τ, so that T_x gets reactivated and its statement properly executes whenever its linked query (or queries) is executed during the retroactive operation. In Figure 3, the trigger Q5→Q10, because Q5 is linked to Q10 (i.e., `INSERT ON Transactions`). Note that Ultraverse's dependency graph in Figure 3 omits all queries whose write set is empty (e.g., `SELECT` queries), since they are read-only queries not affecting the database's state. Also note that Figure 3's red arrows represent column read-write dependencies caused by `FOREIGN KEY` relationships – if some column's value is retroactively changed, then the foreign key columns of other tables referencing this column can be also affected. The red arrows ensure to rollback and replay the queries accessing such potentially affected foreign key columns(s).

We provide formal analysis of Ultraverse's column-wise retroactive operation in §E[1].

3.4 Efficient Rollback and Replay

Given the query dependency graph, Ultraverse rollbacks and replays only the queries dependent on the target query as follows:

(1) **Rollback Phase**: Rollback each table whose column(s) appears in some read or write set in the query dependency graph. Copy those tables into a temporary database.
(2) **Replay Phase**: Add, remove, or change the retroactive target query as requested by the user. Then, replay (from the temporary database) all the queries dependent on the target query, as much in parallel as possible without harming the correctness of the final database state (i.e., guaranteeing strongly serialized commits).
(3) **Database Update**: Lock the original database and reflect the changes of *mutated* tables (defined later) from the temporary database to the original database. After done, unlock the original database and delete the temporary database.

During the above process, each table in the original database is classified as one of the following three: 1) a *mutated* table if its column(s) is in the write set of at least one dependent query; 2) a *consulted* table if none of its columns is in the write set of any dependent queries, but its column(s) is in the read set of at least one dependent query; 3) an *irrelevant* table if the table is neither mutated nor consulted. In step 1's rollback phase, Ultraverse rollbacks mutated and consulted tables as well as their any logical `INDEX`es to each of their first-accessed commit time after the retroactive operation's target time, by leveraging system versioning of temporal databases [44]. The reason Ultraverse needs to rollback consulted tables is that their former states are needed while replaying the dependent queries that update mutated table(s). Affected by this, other non-dependent queries that have those consulted tables in their write set will also be replayed during the replay phase. During replay, the intermediate values of a consulted table will be read by replayed queries; at the end of replay, consulted tables will have the same state as before the rollback.

In step 2's replay phase, the past commit order of dependent queries should be preserved, because otherwise, they could lead to inconsistency of the final database state– leading to a different universe than the desired state. To ensure this, a naive approach would re-commit each query one by one (i.e., enforce *strict serializability*) for reproduction. However, serial query execution is slow. Ultraverse solves this problem by leveraging query dependency information and simultaneously executing multiple queries in parallel whose R/W sets do not overlap each other. Such a parallel query execution is safe because if two queries access different objects, they do not cause a race condition with each other. This improves the replay speed while guaranteeing the same final database state as strongly serialized commits.

Figure 4 is Ultraverse's replay schedule for Figure 3's retroactive operation scenario. Red arrows are the replay order. A replay arrow from $Q_n \rightarrow Q_m$ is created if $n < m$ and the two queries have a conflicting operation [55] (a *read-write*, *write-read*, or *write-write*) on the same column of a table/view). Q12 and Q13 are safely replayed in parallel, because they access different table columns.

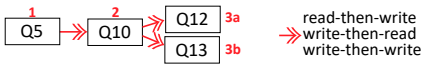


Figure 4: The replay schedule for Figure 3.

During the retroactive operation, Ultraverse simultaneously serves regular SQL operations from its clients, so the database system's front-end service stays available. Such simultaneous processing is possible because the retroactive operation's rollback and replay are done on a temporarily created database. Once Ultraverse's rollback & replay phases (steps 1 and 2) are complete, in step 3 Ultraverse temporarily locks the original database, updates the temporary database's mutated table tuples to the original database, and unlocks it. Should there be new regular queries additionally committed during the rollback and replay phases, before unlocking the database, Ultraverse runs another set of rollback & replay phases for those new queries to reflect the delta change.

Replaying non-determinism: During regular service operations, Ultraverse's query analyzer records the concretized return value of each non-deterministic SQL function (e.g., `CURTIME()` or `RAND()`). Then, during the replay phase, Ultraverse enforces each query's each non-deterministic function call to return the same value as in the past. This is to ensure that during the replay phase, non-deterministic functions behave the same manner as during the regular service operations. If a retroactively added query calls a timing function such as `CURTIME()`, Ultraverse estimates its return value based on the (past) timestamp value retroactively assigned to the query.

A retroactively added/removed `INSERT` query may access a table whose schema uses `AUTO_INCREMENT` on some column value. §C.5 explains how Ultraverse handles this.

3.5 Row-wise Dependency & Query Clustering

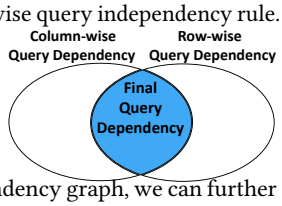
While §3.3 described column-wise dependency analysis, Ultraverse also uses *row-wise* dependency analysis to further reduce the number of queries to rollback/replay. We use the same motivating example in Figure 3 which retroactively removes Q10. The column-wise dependency analysis found the query dependency of {Q5,Q12,Q13}→Q10. However, we could further skip Q12, because Q12 only accesses Charlie's data records and the actual data affected by the retroactive target query (Q10) is only Alice and Bob's data. In particular, Charlie had no interaction with Alice and Bob (i.e., no money transfer), thus Charlie's data records are independent from (and unaffected by) Alice and Bob's data changes. Similarly, later in the service, all other users who have no interaction with with Alice or Bob's data will be unaffected, thus the queries operating on the other users' records can be skipped from rollback & replay. In this observation, each user's data boundary is the table row.

Inspired from this, we propose the row-wise query clustering scheme. Its high-level idea is to classify queries into disjoint clusters according to the table rows they read/write, such that any two queries belonging to different clusters have no overlap in the table rows they access (i.e., two queries are row-wise independent).

Ultraverse identifies the rows each query accesses by analyzing the query's statement. The major query types for query clustering is `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, because they are designed to access only particular row(s) in a table. For `SELECT`, `UPDATE`, and `DELETE` queries, the rows they access are specified in the `WHERE` clause (and in the `SET` clause in case of `UPDATE`); for an `INSERT` query, the rows it accesses are specified in its `VALUES` clause. For example, if a query's statement contains the "`WHERE aid=0001`" clause or the "`(aid,uid,balance) VALUES (0001,'alice',100)`" clause, this query accesses only the rows whose `aid` value is `0001`. We call such a row-deciding column a *cluster key column*. Ultraverse labels each query with a cluster key (or cluster keys if the row-deciding column is specified as multiple values or a range). After every query is assigned a cluster key set (K set), Ultraverse groups those queries which have one or more same cluster keys (i.e., their accessing rows overlap) into the same cluster. At the end of recursive clustering until saturation, any two queries from different clusters are guaranteed to access different rows in any tables they access, thus their operations are mutually independent and unaffected. Table 2 describes this query clustering algorithm in a formal manner as the row-wise query independency rule.

Row-wise Query Independency Rule
$K_c(Q_n)$: A cluster set containing Q_n's cluster keys given c is chosen as the cluster key column
$Q_n \leftrightsquigarrow Q_m$: Q_n and Q_m are in the same cluster
1. $K_c(Q_n) \cap K_c(Q_m) \neq \emptyset \Longrightarrow Q_n \leftrightsquigarrow Q_m$
2. $(Q_m \leftrightsquigarrow Q_n) \wedge (Q_n \leftrightsquigarrow Q_o) \Longrightarrow Q_m \leftrightsquigarrow Q_o$
3. $Q_n \not\leftrightsquigarrow Q_m \Longrightarrow Q_n \nleftrightarrow Q_m$
Choice Rule for the Cluster Key Column
z : The last committed query's index in $\mathbb{Q}$
t : The retroactive target query's index
C : The set of all table columns in $\mathbb{D}$
$c_{choice} := \text{argmin}_{c \in C} \sum_{j=t}^{z} \lvert K_c(Q_j) \rvert^2$

Table 2: Ultraverse's query clustering rules.



Ultraverse regards two queries to have a dependency only if they are dependent both column-wise and cluster-wise, as illustrated in the Venn diagram. Therefore, from §3.2's column-wise query dependency graph, we can further eliminate the queries that are not in the same cluster as the retroactive target query (e.g., Q12 in Figure 3).

The query clustering scheme can be effectively used only if each query in a retroactive operation's commit history window has at least 1 cluster key. However, some queries may not access the cluster key column if they access different tables. For such queries, their other columns can be used as a cluster key under certain cases. Ultraverse classifies them into 2 cases. First, if a column is a `FOREIGN KEY` column that references the cluster key column (e.g., `Accounts.uid`), we define that column as a *foreign* cluster key column, whose value itself can be used as a cluster key. This is because the foreign (cluster) key directly reflects its origin (cluster) key. Second, if a column is in the same table as the cluster key or a foreign cluster key (e.g., `Accounts.aid`), we define it as an *alias* cluster key column, whose concrete value specified in a query statement's `WHERE`, `SET`, or `VALUES` clause will be mapped to its same row's (foreign) cluster key column's value. For example, in Figure 3, Q12's alias cluster key `Accounts.aid` creates the mapping ⟨0003→"charlie"⟩, thus Q12's cluster key is {"charlie"}. The cluster keys of Q10 and Q5 are {"alice", "bob"}. Q13's cluster key is {"alice"}. {"alice", "bob"}∩{"alice"}=¬∅, so Q5, Q10, Q13 are merged into the same cluster under the merged key set {"alice", "bob"}. However, Q12 is not merged, because {"charlie" }∩{"alice", "bob" }= ∅. As Q12 is not in the same cluster as Q10 (the retroactive

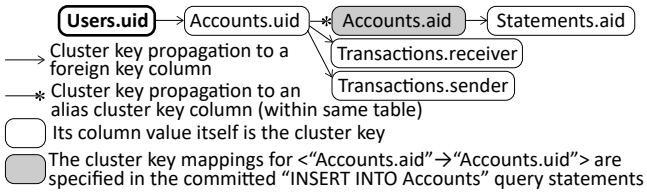


Figure 5: Figure 3's cluster key propagation graph.

target query), Q12 is row-wise independent from Q10, thus we can eliminate Q12 from the rollback & replay list.

Figure 5 shows the online banking example's relationships between the cluster key column (`Users.uid`), foreign cluster key columns (`Accounts.uid`, `Transactions.sender`, `Transactions.receiver`, `Statements.aid`), and an alias cluster key column (`Accounts.aid`). Arrows represent the order of discovery of each foreign/alias key.

In order for the query clustering to be effective, it is important to choose the *optimal* column as the cluster key column. Table 2 describes Ultraverse's choice rule for the cluster key column. Informally speaking, the choice rule is in favor of uniformly distributing the cluster keys across all queries (i.e., minimize the standard deviation), in order to minimize the worst case number of queries to be replayed for any retroactive operation. If some query Q_i does not specify the value of the (foreign/alias) cluster key column of some table t_j that Q_i accesses, then we force $K_c(Q_i) = \neg\emptyset$ (all elements), which makes $|K_c(Q_i)|^2 = \infty$ and thereby the choice rule excludes c.

The K set of a `TRANSACTION/PROCEDURE` query is the union of the `K` sets of of its all sub-queries. The K set of a `CREATE/DROP TRIGGER` query is the union of the K sets of all the queries that are linked to the trigger within the retroactive operation's time window. The `K` set of a `[CREATE/DROP/ALTER] [TABLE/VIEW]` query is the union of the `K` sets of all the queries that access this table/view committed within the retroactive operation's time window. If query clustering is unusable (i.e., the choice rule's computed weight is infinite for all candidate columns in the database), then Ultraverse uses only the column-wise dependency analysis.

In §C.6[1] , we further describe Ultraverse's advanced clustering techniques to enable finer-grained and higher-opportunity clustering: (1) detect and support any *implicit* foreign key columns (undefined in SQL table schema); (2) detect and handle *variable* cluster keys; (3) simultaneously use *multiple* cluster key columns if possible (i.e., multi-dimensional cluster keys). Our experiment (§5) demonstrates the drastic performance improvement enabled by the advanced clustering technique in the TATP, Epinions, and SEATS micro-benchmarks, and the Invoice Ninja macro-benchmark.

We provide formal analysis of the query clustering scheme in §E[1]

3.6 Hash-Jumper

During a retroactive operation, if we can somehow infer that the retroactive operation will not affect the final state of the database, we can terminate the effectless retroactive operation in advance.

Figure 6 is a motivating example, which is a continual scenario of Figure 3. Suppose the service newly creates the `Rewards` table (Q14) to give users reward points for their daily expenses. For the rewards type, Alice chooses '`mileage`' (Q15), Bob chooses '`movie`', and this continues for many future users. In the middle, Alice changes her rewards type to '`shopping`' (Q99). Later, the service detects that Q15 (orange) was a problematic query (triggered by a bug or malicious

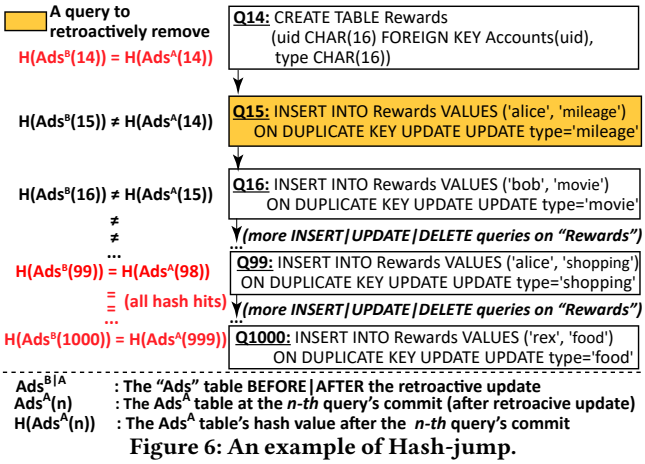


Figure 6: An example of Hash-jump.

activity), and decides to retroactively remove Q15. However, upon rolling back and replaying Q99, the `Rewards` table's state becomes the same as before the retroactive operation at the same commit time, and all subsequent queries until the end (Q1000) are the same as before. Therefore, we deterministically know that the `Rewards` table's final state will become the same as before the retroactive operation, thus it's effectless to replay Q99~Q1000.

Ultraverse's Hash-jumper is designed to capture such cases and immediately terminate the retroactive operation as soon as realizing that replaying the remaining queries will be effectless. For this, Ultraverse's high-level approach is to compute and log the hash value of the modified table(s) upon each query's commit during regular operations. During a retroactive operation's replay phase, Hash-jumper simultaneously runs from the background (not to block the replay of queries) and compares whether the replayed query's output hash value matches its past logged version (before the retroactive operation). During the replay, if hash matches are found for all mutated tables (§3.4), this implies that the mutated table(s)'s final state will become the same as before the retroactive operation, thus Ultraverse terminates effectless replay and keeps the original table(s).

When computing each table's hash, the efficiency of the hash function is crucial. Ideally, the hash computation time should not be affected by the size of the table; otherwise, replaying each query that writes to a huge table would spend a prohibitively long time to compute its hash. Ultraverse designs an efficient table hash algorithm that meets this demand, whose algorithm is as follows.

An empty table's initial hash value is 0. Once the database system executes a requested query and records the target table's rows to be added/deleted to the query log, Ultraverse's query analyzer reads this log and computes the hash value of each of these added/deleted rows by a collision-resistant hash function (e.g., SHA-256), and then either adds (for insertion) or subtracts (for deletion) the hash value from the target table's current hash value in modulo p (size of the collision-resistant hash function's output range, 2^{256} for SHA-256). For each query, the table hash computation time is constant with respect to the target table's size, and linear in the number of rows to be inserted/deleted. Given that the collision-resistant hash function's output is uniformly distributed in $[0, p-1]$, Hash-jumper's collision rate of table hashes, regardless of the number of rows in tables, is upper-bounded by $\frac{1}{p}$ (2^{-256} for SHA-256, which is negligibly smaller

than the CPU bit error rate). See §F[1] for the proof and discussion on false positives & negatives. Nevertheless, Ultraverse offers the option of literal table comparison upon detecting hash hits.

4 Ultraverse's Web Application Framework

A realistic web service's state is manipulated not only by the data flows in SQL queries of its database system, but also by the data flows in its application-level code. While §3 covered the former, this section covers the latter, together which accomplish retroactive updates on a web application's state.

In general, a web application consists of server-side code (i.e., web server) and client-side code (i.e. webpage). When a client browser visits a URL, it sends an HTTP GET request to the web server, which in turn loads the requested URL's base HTML template that is common to all users, selectively customizes some of its HTML tags based on the client-specific state (e.g., learns about the client's identity from the HTTP request's cookie and rewrites the base HTML template's `<h1>` tag's value *"User Account"* to *"Alice's Account"*), and returns the customized HTML webpage to the client browser. Then, the client navigating this webpage can make additional data-processing requests to the server, such as typing in the transfer amount in the `<input>` tag's textbox and clicking the `"Send Money"` button. Then, the webpage's JavaScript reads the user's input from the `<input>` tag's associated DOM node, pre-processes it, and sends it to the web server, who in turn processes it, updates its server-side database, and returns results (e.g., "Success") to the client. Finally, client-side JavaScript post-processes the received result (e.g., displays the result string to the webpage by writing it to its DOM tree's `<p>` tag's' innerHTML). During this whole procedure, JavaScript can store its intermediate data or session state in the browser's local storage such as a cookie. Given this web application framework, when an application-level retroactive operation is needed (e.g., cancel Alice's money transfer committed at *t=9*), Ultraverse should track and replay the data flow dependencies that occur not only within SQL queries, but also within client-side & server-side application code, as well as within the customized client-side webpage's DOM tree nodes. This section explains how Ultraverse ensures the correctness of retroactive operation by carefully enforcing the interactions between SQL queries and application code.

4.1 Architecture

The Ultraverse web application framework is comprised of two components: *uTemplates* (Ultraverse templates) and *Apper* (application code generator). Ultraverse divides a web application's each user request handler into three stages: `PRE_REQUEST`, `SERVE_REQUEST`, and `POST_REQUEST`. `PRE_REQUEST` and `POST_REQUEST` are executed by the client webpage's JavaScript; `SERVE_REQUEST` is by the web server's request handler. For example, when a client types in her input value(s) in some DOM node(s) and sends a user request (e.g., a button click event), the client webpage's JavaScript code executes `PRE_REQUEST`, which reads user inputs and sends an HTTP request to the server. Once the server receives the client's request, it executes `SERVER_REQUEST` to process the request and update the server-side database accordingly. After a response is returned to the client browser, its JavaScript executes `POST_REQUEST` to update, if needed, the client-side local database and updates the webpage's DOM node(s). Ultraverse requires the developer to fill out the *uTemplate*, which is essentially implementing each of these three `*_REQUEST` stages as SQL `PROCEDURE`s. This framework ensures that all data flows in the web application are captured and logged by the Ultraverse database system as SQL queries during regular service operation and replay them during retroactive operation. Importantly, the SQL language used in the *uTemplate* innately provides no built-in interface to access any other system storage than SQL databases. Therefore, using *uTemplate* fundamentally prevents the application from directly accessing any external persistent storage untrackable by the Ultraverse database (e.g., `document.cookie` or `window.localStorage`), while Ultraverse provides a way to indirectly access them via Ultraverse-provided client-side local tables (e.g., `BrowserCookie`) which Ultraverse can track.

PROCEDURE	Caller	INPUT	OUTPUT
PRE_REQUEST	client	DOM node fields	HTTP Req. msg
SERVE_REQUEST	server	HTTP Req. msg	HTTP Res. msg
POST_REQUEST	client	HTTP Res. msg	DOM node fields

Table 3: *uTemplate*'s allowed procedures as user request.

After the developer implements the client-side webpage's user request handler as SQL `PROCEDURE`s in the *uTemplate*, *Apper* converts them into their equivalent application code (e.g., NodeJS). The generated application code essentially passes *uTemplate*'s each SQL `PROCEDURE` statement as a string argument to the application-level SQL database API (e.g., NodeJS's `SQL_exec()`) so that the executed `PROCEDURE` is logged by Ultraverse's query analyzer. In *uTemplate*, the developer can link the `INPUT` and `OUTPUT` of each `PROCEDURE` to client webpage's particular DOM nodes or HTTP messages exchanged between the client and server. The allowed linkings are described in Table 3. At a high level, a webpage's user request reads a user's input from some DOM node(s) as `INPUT` arguments to `PRE_REQUEST`, and writes `POST_REQUEST`'s returned `OUTPUT` back to some DOM node(s). The `INPUT` and `OUTPUT` of `SERVE_REQUEST` are HTTP messages exchanged between the client and server.

Enforcing Data Flow Restriction: By requiring developers to use *uTemplate* and *Apper*, Ultraverse enforces the application's all data flows to be captured at the SQL level by Ultraverse's query analyzer. This way, the inputs to user requests are also captured as `INPUT`s to `PRE_REQUEST`. However, once `POST_REQUEST` writes its output to DOM node(s), Ultraverse's query analayzer cannot track this outgone flow, because a webpage's DOM resides outside the trackable SQL domain. As a solution, the application code generated by *Apper* adds an application logic which dynamically *taints* all DOM node(s) that `POST_REQUEST` writes its output to, and forbids `PRE_REQUEST` from receiving inputs from tainted DOM node(s). This essentially enforces that any data flown from SQL to DOM cannot flow back to SQL (i.e., cannot affect the database's state anymore). Thus, it is safe not to further track/record such flows, and such flows need not be replayed during retroactive operation as well. If a webpage needs to implement the logic of repeatedly refreshing some values (e.g., streamed value) to the same DOM node, then the developer can implement the webpage's `POST_REQUEST` to create a new DOM output node for each update and delete the previously created node, so that each node is transient. Each tainted DOM node gets an additional 1-bit object property named `tainted`. The 1-bit taints in DOM nodes refresh only when the webpage gets refreshed. Note that `POST_REQUEST`s are allowed to write their outputs to already-tainted DOM nodes or to dynamically created new DOM nodes.

```
Webpage: https://online-banking.com/account/send_money.html
Base HTML: <html><head></head><body>
    <h1 id="title">User Account</h1>
    <!-- Apper will auto-generate a <script> tag defining "function SendMoney()" -->
    <form onsubmit="SendMoney()">
      <input type="text" id="receiver" value="" >
      <input type="text" id="amount" value="" >
    </form>
  </body></html>
```

```
Type-1 Request Handler: server=SendMoney_CreateWebpage()
"SERVE_REQUEST"
|- INPUT: [HTTP Request Header's "COOKIE.uname" Field] → @username VARCHAR(32)
|- OUTPUT: @final_html TEXT → [HTTP Response Body]
|- SQL BODY:
   IF username != "" THEN
     DECLARE personalizedTitle TEXT;
     SET @personalizedTitle = username + " 's Account";
    SET @final_html = Ultraverse_UpdateHTML("title", "innerHTML", personalizedTitle);
   END IF;
```

```
Type-2 HTTP Method: PUT, client=SendMoney(), server=SendMoney_Process()
"PRE_REQUEST"
|- INPUT:      [DOM Node id="receiver", field="value"] → @receiver VARCHAR(16)
               [DOM Node id="amount", field="value"] → @amount VARCHAR(16)
|- OUTPUT:     @recv VARCHAR(16), @amt VARCHAR(16) → [HTTP Request Body]
|- SQL BODY:  SET @recv = receiver; SET @amt = amount;
"SERVE_REQUEST"
|- INPUT:  [HTTP Request Header's "COOKIE.uid" Field] → @sender VARCHAR(32)
           [HTTP Request Body's "receiver" Field] → @receiver VARCHAR(32)
           [HTTP Request Body's "amount" Field] → @amount VARCHAR(32)
|- OUTPUT:  @result TEXT → [HTTP Response Body]
|- SQL BODY: DECLARE cur_balance INT;
            SET @cur_balance = SELECT balance from Acccounts WHERE uid = sender;
            IF (cur_balance >= amount) THEN
              UPDATE Accounts SET balance  -= amount WHERE uid = sender;
              UPDATE Accounts SET balance += amount WHERE uid = receiver;
              INSERT INTO Transactions VALUES (sender, receiver, amount);
            END IF
            SET @result = "Successfully sent " + amount + " to " + receiver
"POST_REQUEST"
|- INPUT:  [HTTP Response Body's "result" Field] → @result TEXT
|- OUTPUT:@result TEXT → [new <p> id="result", filed="InnerHTML", append-
To:<body>]
|- SQL BODY: UPDATE BrowserCookie SET last_activity = result;
```

Figure 7: A *uTemplate* that designs Type-1 & Type-2 request handlers of the `send_money.html` webpage.

4.2 Supported Types of User Requests

Modern web applications are generally designed based on three types of user request handlers: *(i)* a web server creates webpages requested by clients; *(ii)* a webpage's JavaScript interacts with the web server to remotely process the client's data; *(iii)* a webpage's JavaScript processes the client's data locally without interacting with the server. The Ultraverse web framework's *uTemplate* allows developers to implement these three types of user request handlers. Figure 7 is a *uTemplate* that designs an online banking service's *"Send-Money"* webpage comprised of Type-1 and Type-2 user requests.

Creation of a client's requested webpage (Type-1): When a client visits `https://online-banking.com/send_money.html`, its web server returns the client's customized webpage. To implement such a user request handler, the developer first associates a new *uTemplate* to the above target URL. Then, the developer implements this webpage's base HTML which is common to all users visiting it. Then, the developer implements `SERVE_REQUEST`'s SQL logic which customizes the base HTML according to each client's HTTP request (i.e., "Type-1 Request Handler" box in Figure 7). When a client browser visits the URL, the web server executes this *uTemplate*'s `SERVE_REQUEST`'s application code generated by *Apper*. Note that `PRE_REQUEST` and `POST_REQUEST` are unused in Type-1 user request handlers, because the client simply navigates to the URL by using the browser's address bar interface or page navigation API.

Remote data-processing between a client webpage's JavaScript and a server (Type-2): After the client loads the `send_money.html` webpage, she types in the recipient's account ID and the transfer amount in the `<input>` textboxes and clicks the *"submit"* button. Then, the client-side JavaScript's `PRE_REQUEST` logic sends the money transfer request to the web server; the web server's `SERVE_REQUEST` logic updates the account balances of the sender and receiver in the server-side database accordingly and sends the result to the client; the client-side JavaScript's `POST_REQUEST` logic displays the received result to the webpage. It is *Apper* which generates the client-side JavaScript code and web server's application code that correspond to each of the 3 SQL `PROCEDURE`s above.

Local data-processing by a client webpage's JavaScript (Type-3): Although not shown in Figure 7, two examples are as follows: (1) A client webpage's JavaScript dynamically updates its refreshed local time to its webpage; for this request, the developer will implement only `POST_REQUEST` which calls `CURTIME()` and writes its return value to an output DOM node. (2) The server locally runs data analytics on its current database; for this request, the developer will implement only the `SERVE_REQUEST`, not interacting with clients.

4.3 Application-level Retroactive Operation

Logging and Replaying User Requests: During the web application service's regular operation, Ultraverse silently logs all the information required for retroactive operation on any user request, which includes the following: each called user request's name and timestamp, the calling client's ID, the webpage's customized DOM node IDs used as the user request's arguments, and interactive user inputs used as arguments (if any). To log them, *Apper*'s generated client-side code has the application logic such that whenever the client sends a user request to the server, it piggybacks the client's execution logs of user requests (e.g., `INPUT` values to `PRE_REQUEST`s). The server merges these logs into the server's *global* log and uses it to build the query dependency graph for retroactive operation. The replay phase has to replay each user request's application-specific logic of updating the browser cookie or any persistent application variables which survive across multiple user requests (e.g., JavaScript global/static variables). *pTemplate* enforces the developer to implement such application-specific logic as SQL logic of updating Ultraverse's specially reserved 2 tables (`BrowserCookie` and `AppVariables`) inside `PRE_REQUEST`, `SERVE_REQUEST`, and `POST_REQUEST`. Thus, when Ultraverse's replay phase replays these `PROCEDURE`s, they replay each webpage's cookie and persistent variables. While replaying them, any customized DOM nodes used as arguments to user requests are also re-computed based on the retroactive updated database state. During the retroactive operation, clients need not be online, because Ultraverse locally replays the user requests of all clients by itself. See §B[1] for in-depth details.

Optimizing Retroactive Operation: Ultraverse treats each type of user request as an application-level transaction, computes its $R/W/K$ sets, and rolls back & replays only dependent application-level transactions both column-wise & row-wise. Hash-jumps are made in the granularity of application-level transaction.

Replaying Interactive Human Decisions: During Ultraverse's retroactive operation, it is tricky to retroactively replay interactive

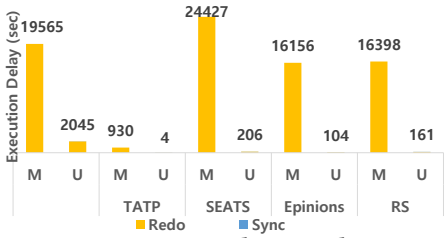


Figure 8: Testcases without Hash-jumps.

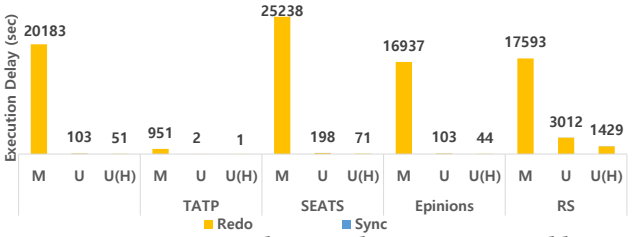


Figure 9: Testcases where Hash-jumps are possible.

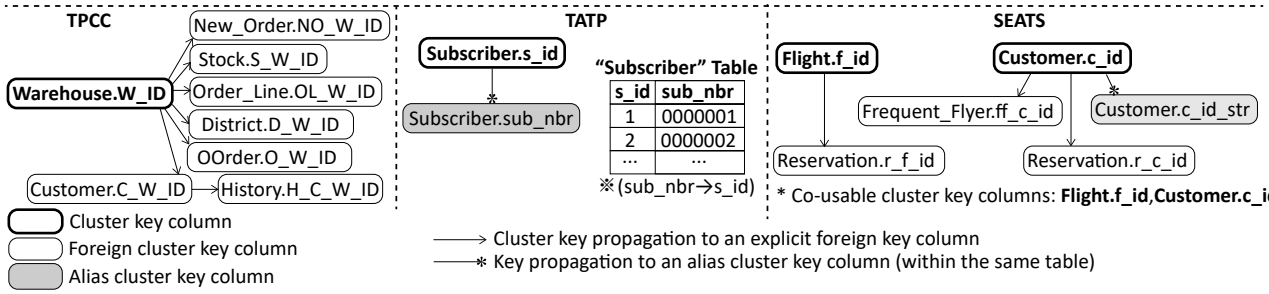


Figure 10: TATP, TPCC, and SEAT's cluster key propagation graphs generated by the row-wise query clustering technique.

user inputs, because Ultraverse cannot replay a human mind. Ultraverse provides 2 options for handling this. First, Ultraverse's retroactive replay uses the same interactive human inputs as recorded in the past user request log. For example, suppose there is a transaction where Alice who initially had $200 sends $100 to Bob. Then, suppose that a retroactive operation changes Alice's initial balance to $50. Then, during the replay, Alice's transaction of sending $100 to Bob will abort due to her insufficient funds. Although this is a correct result in the application semantics, in human's viewpoint, Alice might not have tried to send $100 to Bob if her balance had been lower than that. To address this issue, Ultraverse's 2nd option allows the replay phase to change each user request's `INPUT`s to different (e.g., human-engineer-picked) values to simulate different human decisions. These new values can be either a constant or a return value of `PRE_REQUEST_INTERACTIVE`, an optional `PROCEDURE` in *uTemplate* which executes only during retroactive operation to generate different `INPUT`s to `PRE_REQUEST` (i.e., different user inputs). For example, the developer's `PRE_REQUEST_INTERACTIVE` can implement the logic of simulating a human mind such that if the user's current (i.e., retroactively updated) balance is lower than her intended amount of transfer, then she transfers only the amount she currently has.

Other Design Topics: Due to the space limit, see §C[1] for Ultraverse's other features: handling the browser cookie across user requests (§C.1[1]); preserving secrecy of client's secret values such as password or random seed during replay (§C.2[1]); supporting client-side code's dynamic registration of event handlers (§C.3[1]); handling malicious clients who hack their downloaded webpage's JavaScript to tamper with their user request logic or send corrupt user request logs to the server (§C.4[1]); handling `AUTO INCREMENT` initiated by a user request's `PROCEDURE` (§C.5[1]); the advanced query clustering scheme (§C.6[1]); virtual two-level table mappings to improve column-wise dependency analysis (§C.7[1]); column-specific retroactive operation which allows the user to selectively skip unneeded columns without harming correctness (§C.8[1]);

5 Evaluation

Implementation: Ultraverse can be seamlessly deployed based on any unmodified SQL-based database systems. Our prototype's host database system was MariaDB [42]. We implemented the query analyzer (column-wise & row-wise dependency analysis, replay scheduler, and Hash-jumper) in C. The query analyzer reads MariaDB's binary log [43] to retrieve committed queries and computes each query's $R/W/K$ sets and table hashes. The replay scheduler uses a lockless queue [23] and atomic compare-and-swap instructions [28] to reduce contention among threads simultaneously dequeuing the queries to replay. For database rollback, there are 3 options: *(i)* sequentially apply an inverse operation to every committed query; *(ii)* use a temporal database to stage all historical table states; *(iii)* assume periodic snapshots of backup DBs (e.g., every 3 days, 1 week). Our evaluation chose option 3, as creating system backups is a common practice and this approach incurs no rollback delay (i.e., we can simply load the particular version of backup DB). For the Ultraverse web framework, we implemented *Apper* in Python, which reads a user-provided *uTemplate* and generates web application code for ExpressJS web framework [52]. The Ultraverse software and installation guideline is available at https://anonymous.4open.science/r/ultraverse-8E1D/README.md.

In this section, we evaluate the Ultraverse database system (§5.1) and web application framework (§5.2), and present case studies (§5.3).

5.1 Database System Evaluation

We evaluated Ultraverse on Digital Ocean's VM with 8 virtual CPUs, 32GB RAM and 640GB SSD (NVMe). We compared the speed of retroactive operation of MariaDB (`M`) and Ultraverse (`U`). We used five micro-benchmarks in BenchBase [22]: TPC-C, TATP, Epinions, SEATS, and ResourceStresser (RS). For each benchmark, we ran retroactive operations on various sizes of commit history: 1M, 10M, 100M, and 1B queries. For each benchmark, we designed realistic retroactive scenarios which choose a particular transaction to retroactively remove/add and retroactively update the database. Due

to the space limit, we describe each benchmark's retroactive operation scenarios and Ultraverse's optimization analysis in §D[1]. We ran each testcase 10 times and reported the median.

Both Ultraverse and MariaDB used a backup DB to instantly rollback the database. For the replay, Ultraverse used column-wise & row-wise dependency analysis, parallel replay, and hash-jump described in §3, while MariaDB serially replayed all transactions committed after the retroactive target transaction. For the replay, both Ultraverse and MariaDB skipped read-only transactions (comprised of only `SELECT` queries) as they do not affect the database's state. Since this subsection evaluates only the Ultraverse database system (not its web application framework), we modified BenchBase's source code such that each benchmark's any transaction logic implemented in Java application code is instead implemented in SQL so that it is recorded and visible in the SQL log to properly replay them by both MariaDB and the Ultraverse database system. Each benchmark's default cluster number was set as follows: TPC-C (10 warehouses), TATP (200,000 subscribers), Epinions (2,000 users), ResoureStresser (1,000 employees), and SEATS (100,000 customers).

Figure 8 shows the execution time of retroactive operation scenarios between MariaDB and Ultraverse. Ultraverse's major speedup was from its significantly smaller number of queries to be replayed. Table 4 summarizes Ultraverse's query reduction rate (w/o Hash-jumper), which is between 90% – 99%. For most benchmarks, Ultraverse achieved the query number reduction rate in proportion to the number of clusters (e.g., users, customers, employers, or warehouses). Figure 10 is Ultraverse's cluster key propagation graphs for TPC-C, TATP, and SEATS. In TPC-C, `Warehouse.w_id` was the cluster key column. TPC-C's all tables participating in cluster key propagation (total 8) have explicit foreign key relationship, which were discovered by the basic query clustering scheme (§3.5). In TATP, `Subscriber.s_id` was the cluster key column, and the alias cluster key mappings of `Subscriber.sub_nbr`→`Subscriber.s_id` was discovered by the advanced query clustering technique (§C.6[1]). SEATS also used the advanced clustering technique to simultaneously use 2 cluster key columns (`Flight.f_id` and `Customer.c_id`) with an alias cluster key column (`Customer.c_id_str`). We present the cluster key propagation graphs and column-wise transaction (query) dependency graphs for all benchmarks in §D[1].

In both MariaDB and the Ultraverse database system, reading the query log and replaying queries were done in parallel. However, as the sequential disk batch-reading speed of SSD NVMe was faster than serial execution of queries, the critical path of retroactive operation was the replay delay. Ultraverse additionally had a database synchronization delay, because it replays only column-wise & row-wise dependent queries, and thus at the end of its replay, it updates only the affected table rows & columns to the original database. As shown in Figure 8, this synchronization delay was negligibly small (~1 second).

Figure 9 shows the execution time for different retroactive operation testcases where the Hash-jump optimization is applicable (i.e., a table hash match is found by Hash-jumper). We note `U` as Ultraverse without using Hash-jump, whereas `U(H)` is with Hash-jump enabled. Using hash-jump could achieve additional 101%~185% speedup compared to not using it, by detecting hash matches and terminating the effectless retroactive operation in advance. Our

	TPC-C	TATP	Epinions	SEATS	RS
`U`	90%	99%	99%	99%	94%
`U(H)`	96%	99%	99%	99%	97%

Table 4: Query reduction rate v.s. MariaDB.

	TPC-C	TATP	Epinions	SEATS	RS
`U`	7.0%	0.7%	3.9%	3.9%	6.4%
`U(H)`	9.5%	0.6%	4.9%	4.2%	6.4%

Table 5: Overhead (%) for regular service operations.

	TPC-C	TATP	Epinions	SEATS	RS
Slowdown	8.2%	7.2%	14.1%	16.5%	3.3%

Table 6: Overhead of simultaneously executing a retroactive operation and regular service operations in the same machine.

	TPC-C	TATP	Epinions	SEATS	RS
Log Size (bytes) / Query	110b	48b	12b	35b	48b

Table 7: Ultraverse's average log size per query (bytes).

Queries	TPC-C	TATP	Epinions	SEATS	RS
1 Million	10.0x	240.8x	153.2x	112.0x	10.7x
10 Million	10.5x	253.0x	145.4x	111.9x	9.6x
100 Million	10.1x	232.5x	156.8x	119.2x	12.3x
1 Billion	10.8x	241.1x	153.4x	114.4x	11.8x

Table 8: Speedup for various rollback/replay window sizes.

Size Factor	TPC-C	TATP	Epinions	SEATS	RS
1	10.1x	232.5x	156.8x	119.2x	12.3x
10	50.19x	683.9x	651.9x	693.8x	131.4x
100	106.81x	667.4x	693.3x	674.8x	659.2x

Table 9: Speedup for various database sizes.

Ultraverse prototype's upper-bound of hash collision rate was approximately $1.16x10^{-77}$.

The column-wise & row-wise query dependency analysis and hash-jump analysis incurs additional overhead during regular service operations due to their extra logging activity of $R/W/K$ sets and table hash values for each committed query. We measured this overhead in Table 5, which is between 0.6%~9.5%. However, this overhead can be almost fully reimbursed in practice if Ultraverse's analyzer runs asynchronously in a different machine than the database system, as regular query operations and query analysis can be performed asynchronously.

Table 6 shows how much the retroactive operation running in the background slows down the speed of regular operations running in the foreground when they are executed simultaneously in the same machine. The average overhead varied between 3.3%~16.5%. However, this overhead can be almost fully reimbursed if the retroactive operation's replay runs in a different machine and only the synchronization at the end runs in the same machine. Table 7 shows Ultraverse's average log size per query, which varies between 12~110 bytes, depending on the benchmarks.

Table 8 reports Ultraverse's speedup against MariaDB in retroactive operation across different window sizes of commit history: 1M, 10M, 100M, and 1B queries. Regardless of the window size of queries

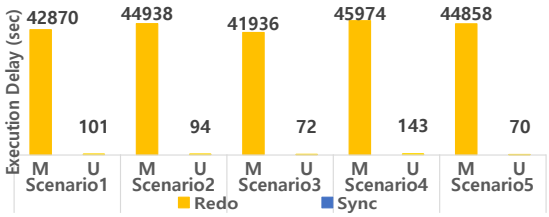


Figure 11: Retroactive operation times for Invoice Ninja.

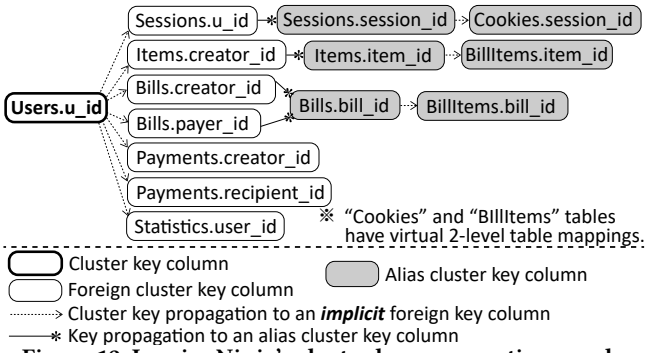


Figure 12: Invoice Ninja's cluster key propagation graph.

to be rolled back and replayed, each benchmark's speedup in retroactive operation stayed roughly constant. This is because each benchmark's transaction weights were constant, so Ultraverse's reduction rates of queries to be replayed were consistent regardless of the window size of committed transactions.

Table 9 reports Ultraverse's speedup of retroactive operation against MariaDB across different database sizes (10MB~10GB), while the window size of commit history is constant (100M queries). Interestingly, Ultraverse's speedup increased roughly in proportion with a database's size factor. This is because a bigger database had more number of clusters (e.g., warehouses, customers), which led to finer granularity of row-wise query clustering. When the database size factor was 100, there were too many query clusters, and thus too few queries to replay, despite the large size of query analysis log. In such cases Ultraverse's replay speedup was upper-limited by the delay of reading and interpreting the query analysis log.

5.2 Web Application Framework Evaluation

For this evaluation, we re-implemented Invoice Ninja [17] based on Ultraverse's web application framework. Invoice Ninja is a Venmo-like open source web application for invoice management of online users. The application provides 31 types of user requests (e.g., creating or editing a user's profile). The application consists of 6 major server-side data tables (`Users`, `Items`, `Bills`, `BillItems`, `Payments`, and `Statistics`) and 2 client-side data tables (`Cookie` and `Session`). Each user's DOM displays a user-specific dashboard such as the invoices to be paid, the current balance, etc. Each user request consists of `PRE_REQUEST`, `SERVE_REQUEST`, and `POST_REQUEST`. An invalid user request gets aborted (e.g., the user's provided credential is invalid, or the user's balance is insufficient). We designed 5 retroactive operation scenarios, each of which retroactively removes an attacker-triggered user request committed in the past. We describe these scenarios and Ultraverse's optimization analysis in §D.6[1].

Figure 12 shows Ultraverse's cluster key propagation graph for Invoice Ninja. All columns propagating the cluster keys had implicit foreign key relationships (i.e., not defined as `FOREIGN KEY` in table schema but used as foreign keys in the application semantics), which were discovered by the advanced query clustering scheme (§C.6[1]). Ultraverse clustered the Invoice Ninja database's all table rows by using `Users.user_id` as the cluster key column.

Transactions	$Scen_1$	$Scen_2$	$Scen_3$	$Scen_4$	$Scen_5$
1 Million	422.3x	495.3x	574.5x	333.5x	594.5x
10 Million	415.5x	489.6x	593.2x	343.4x	661.4x
100 Million	424.5x	478.1x	582.4x	321.5x	640.8x
1 Billion	418.3x	483.5x	581.2x	325.6x	581.5x

Table 10: Speedup for various rollback/replay window sizes.

Size Factor	$Scen_1$	$Scen_2$	$Scen_3$	$Scen_4$	$Scen_5$
1	424.5x	478.1x	582.4x	321.5x	640.8x
10	663.8x	694.4x	671.9x	687.1x	721.6x
100	648.5x	673.5x	664.3x	695.6x	674.6x

Table 11: Speedup for various database sizes in 5 scenarios.

We first compared the performance of Ultraverse and MariaDB on both retroactive operation and regular service operation for a database with 10,000 users for 100M transactions. For retroactive operations, Ultraverse replayed only dependent user requests *both* column-wise and row-wise, whereas MariaDB using the naive strategy did so for all past user requests. Also note that MariaDB's retroactive operation does not provide correctness for application semantics. As depicted in Figure 11, Ultraverse's median speedup over MariaDB was 478.1x. Table 10 shows Ultraverse's speedup against MariaDB for various window sizes of transaction history. While achieving the observed speedup, Ultraverse's average reduction rate of the number of replay queries compared to MariaDB was 99.8%.

Table 11 shows Ultraverse's speedup against MariaDB across various database size factors (10,000~1M users), for the same window size of transaction commit history. The speedup increased with the increasing database size factor, due to the increasing number of clusters (i.e., `Users.u_id`), and eventually upper-limited by the delay of reading and interpreting the query analysis log.

Ultraverse's query analysis overhead during regular service operation (when both running on the same machine) was 4.2% on average. Ultraverse's additional storage overhead for query analysis log was 205 bytes per user request on average.

Data Analytics Evaluation: We evaluated Ultraverse on a popular use case of data analytics (§6), which runs the K-clustering algorithm on 1 million online articles created by 10,000 users and classifies them into 10 categories based on their words relevance, by using the LDA algorithm [4]. The stage for data processing and view meterialization used Hive [47] and Hivemall [32], a machine-learning programming framework based on JDBC. We ported the application into Ultraverse and compared to MariaDB's replay. Ultraverse's average overhead during regular service operation was 1.8%, and its speedup in retroactive operation was 41.7x. The speedup came from eliminating independent queries in the article classification stage, where each user ID was the cluster key.

5.3 Case Study

According to cyberattack reports in 2020 [18, 25], financial organizations take 16 hours on average to detect security breaches (while other domains may take ≥ 48 hours). Recent financial statistics [24]

	TPC-C	TATP	Epinions	SEATS	RS	Invoie Ninja
M	84.6h	95.8h	3.6h	135.6h	101.0h	120.2h
U	8.5h	0.4h	0.1h	1.1h	5.1h	0.3h

Table 12: Retroactive operation time for 1B transactions.

show that everyday the U.S. generates on average 108 million credit card transactions, which account for 23% of all types of financial transactions. This implies that everyday the U.S. generates roughly 108 million÷23%×100%=470 million financial transactions.

Table 12 reports the retroactive operation time of MariaDB and Ultraverse for 1 billion transactions of each benchmark. MariaDB's median retroactive operation time was 95.8 hours, while Ultraverse cut it down to 0.4 hours. Most importantly, a regular database system's SQL-only replay does not provide correctness of the application semantics as Ultraverse's web application framework does. Ultraverse's estimated VM rent cost ($0.41/h) for its retroactive operation is $0.04~$20.9, which is significantly cheaper than hiring human engineers to handcraft compensating transactions for manual data recovery. Ideally, those human engineers can use Ultraverse as a complementary tool to assist their manual analysis of data recovery in financial domains as well as in other various web service domains.

6 Discussion

Data Analytics: Ultraverse can be used to enforce GDPR compliance in data analytics. Modern data analytics architectures (e.g., Azure [45] or IBM Analytics [35]) generally consist of four stages: *(i)* sourcing data; *(ii)* storing them in an SQL database; *(iii)* retrieving data records from tables and processing them to create materialized views; *(iv)* using materialized views to run various data analytics. When a user initiates her data deletion, all other data records derived from it during the data analytics should be also accordingly updated. Thus, GDPR-compliant data analytics should replay the third stage (e.g., data processing) to reflect changes to materialized views. However, this stage often involves machine-learning or advanced statistical algorithms, which are complex, computationally heavy, and do not have efficient incremental deletion algorithm. Ultraverse can be used for such GDPR deletion of user data to efficiently update materialized views. Ultraverse can be easily ported into this use case, because many data analytics frameworks such as Hive [47] or Spark SQL [56] support the SQL language to implement complex data processing logic. We provide our experimental results in §5.2.

***What-if* Analysis:** Ultraverse can be used for *what-if* analysis [21] at both database and application levels with the capability of retroactively adding/removing *any* SQL queries or application-level transactions. For example, one can use Ultraverse to retroactively add/remove certain queries and test if SQL `CONSTRAINT/ABORT` conditions are still satisfied.

Cross-App: We currently support retroactive operation within a single Ultraverse web application. Our future work is to enable cross-app retroactive operation (across many databases/services).

7 Related Work

Retroactive Data Structure efficiently changes past operations on a data object. Typical retroactive actions are insertion, deletion, and update. Retroactive algorithms have been designed for queues [13], doubly linked queues [26], priority queues [20], and union-find [54].

Temporal and Versioning Databases [37] store each record with a reference of time. The main temporal aspects are *valid time* and *transaction time* [15]. The former denotes the beginning and ending time a record is regarded as valid in its application-specific real world; the latter denotes the beginning and ending time a database system regards a record to be stored in its database. A number of database languages support temporal queries [5, 50]. SQL:2011 [40] incorporates temporal features and MariaDB [42] supports its usage. OrpheusDB [34] allows users to efficiently store and load past versions of SQL tables by traversing a version graph. While temporal or versioning databases enable querying a database's past states, they do not support changing committed past operations (which entails updating the entire database accordingly).

Database Recovery/Replication: There are two logging schemes: *(i)* value logging logs changes on each record; *(ii)* operation logging logs committed actions (e.g., SQL statements). ARIES [49] is a standard technique that uses value logging (undo and redo logs) to recover inconsistent checkpoints. Some value logging systems leverage multiple cores to speed up database recovery (e.g., SiloR [65]) or replay (e.g., Kuafu [33]). However, value logging is not designed to support retroactive operation: if a query is retroactively added or removed, the value logs recorded prior to that event become stale. Systems that use operation logging [16, 48, 51, 53, 58] are mainly designed to efficiently replicate databases. However, they either inefficiently execute all queries serially or do not support strong serialization [6], while Ultraverse supports efficient strong serialization.

Attack Recovery: CRIU-MR [63] recovers a malware-infected Linux container by selectively removing malware during checkpoint restoration. ChromePic [59] replays browser attacks by transparently logging the user's page navigation activities and reconstructing the logs for forensics. RegexNet [2] identifies malicious DoS attacks of sending expensive regular expression requests to the server, and recovers the server by isolating requests containing the identified attack signatures. However, the prior works do not address how to selectively undo the damages on the server's persistent database, both efficiently and correctly from application semantics. Warp [9] and Rail [10] selectively remove problematic user request(s) or patch the server's code and reconstruct the server's state based on that. However, all these techniques require replaying the heavy browsers (~500MB per instance) during their replay phase, which is not scalable for large services that have more than even 1M users or 1M transactions to replay. On the other hand, Ultraverse's strength lies in its supreme efficiency and scalability: Ultraverse uses *pTemplate* and *Apper* to represent each webpage as compact SQL code, and replaying a web service's history only requires replaying these SQL queries, which is faster and lighter-weight. Further, the prior arts do not have novel database techniques that reduce the number of replay queries, such as Ultraverse's column-wise query dependency analysis, advanced row-wise query clustering, and hash jumper.

Provenance in Databases: *What-if-provenance* [21] speculates the output of a query if a hypothetical modification is made to the database. *Why-provenance* [7] traces the origin tuples of each output tuple by analyzing the lineage [14] of data generation. *How-provenance* [30] explains the way origin tuples are combined to

generate each output (e.g., ORCHESTRA [29], SPIDER [1]). *Where-provenance* [7] traces each output's origin tuple and column, annotate each table cell, and propagate labels [3] (e.g., Polygen [62], DBNotes [12]). Mahif [8] is a recent work that answers historical *what-if* queries, which computes the delta difference in the final database state given a modified past operation. Mahif leverages symbolic execution to safely ignore a subset of transaction history that is provably unaffected by the modified past operation. However, Mahif is not scalable over the transaction history size: its cost of symbolic execution (i.e., the symbolic constraints on tuples that the SMT solver has to solve) grows unbearably large as the history grows. While Mahif was designed and evaluated for only a small transaction history (only up to 200 insert/delete/update operations used in experiments), Ultraverse has been demonstrated to efficiently handle beyond 1 billion transactions. Finally, note that all prior database provenance works do not preserve application semantics like Ultraverse does.

8 Conclusion

Ultraverse efficiently updates a database for retrospective operation. By using its various novel techniques such as column-wise & row-wise query dependency analysis and hash-jump, Ultraverse speeds up retroactive database update by up to two orders of magnitude over a regular rollback and replay. Further, Ultraverse provides a web application framework that retroactively updates the database with awareness of application semantics.

References

[1] B. Alexe, L. Chiticariu, and W.-C. Tan. Spider: A schema mapping debugger. In *VLDB*, pages 1179–1182, 2006.

[2] Z. Bai, K. Wang, H. Zhu, Y. Cao, and X. Jin. Runtime recovery of web applications under zero-day redos attacks. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1575–1588. IEEE, 2021.

[3] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4):373–396, 2005.

[4] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3(null):993–1022, Mar. 2003.

[5] M. H. Bohlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Trans. Database Syst.*, 25(4):407–456, 2000.

[6] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB J.*, 1(2):181–239, 1992.

[7] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[8] F. S. Campbell, B. S. Arab, and B. Glavic. Efficient answering of historical what-if queries. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1556–1569, New York, NY, USA, 2022. Association for Computing Machinery.

[9] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 101–114, New York, NY, USA, 2011. Association for Computing Machinery.

[10] H. Chen, T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Identifying information disclosure in web applications with retroactive auditing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 555–569, Broomfield, CO, Oct. 2014. USENIX Association.

[11] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[12] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: A post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and Stein. *Introduction to Algorithms*. MIT Press, 2001.

[14] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[15] C. Date and H. Darwen. *Temporal Data and the Relational Model*. Morgan Kaufmann Publishers Inc., 2002.

[16] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726, 2006.

[17] David Bomba. Invoice ninja, 2020. https://github.com/invoiceninja/invoiceninja.

[18] Deep Instinct. Voice of SecOps, 2021. https://www.deepinstinct.com/pdf/voice-of-secops-report-2nd-edition.

[19] E. D. Demaine, J. Iacono, and S. Langerman. Retroactive data structures, 2007.

[20] E. D. Demaine, T. Kaler, Q. Liu, A. Sidford, and A. Yedidia. Polylogarithmic fully retroactive priority queues via hierarchical checkpointing. In *WADS*, pages 263–275, 2015.

[21] D. Deutch, Z. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.

[22] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[23] DPDK Project. Ring Library, 2019. https://doc.dpdk.org/guides/prog_guide/ring_lib.html.

[24] Erica Sandberg. The Average Number of Credit Card Transactions Per Day & Year, 2021. https://www.cardrates.com/advice/number-of-credit-card-transactions-per-day-year/.

[25] Fawad Ali. How Long Does It Take to Detect and Respond to Cyberattacks?, 2021. https://www.makeuseof.com/detect-and-respond-to-cyberattacks/.

[26] R. Fleischer. A simple balanced search tree with O(1) worst-case update time. *Int. J. Found. Comput. Sci.*, pages 137–150, 1996.

[27] D. Fu and F. Shi. Buffer overflow exploit and defensive techniques. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 87–90, 2012.

[28] GCC GNU. Built-in functions for atomic memory access, 2019. https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html.

[29] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *SIGMOD*, pages 1131–1133, 2007.

[30] T. J. Green and V. Tannen. The semiring framework for database provenance. In *PODS*, pages 93–99, 2017.

[31] J. Hasan and A. M. Zeki. Evaluation of web application session security. In *2nd Smart Cities Symposium (SCS 2019)*, pages 1–4, 2019.

[32] Hivemall. Hivemall documentation. https://hivemall.apache.org/userguide/index.html.

[33] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou. KuaFu: Closing the parallelism gap in database replication. In *ICDE*, pages 1186–1195, 2013.

[34] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran. OrpheusDB: bolt-on versioning for relational databases (extended version). *VLDB J.*, 29(1):509–538, 2020.

[35] IBM. IBM Data Analytics. https://www.ibm.com/analytics/journey-to-ai?p1=Search&p4=43700057304811782&p5=b&cm_mmc=Search_Google-_.

[36] James Cheney, Laura Chiticariu and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, pages 379–474, 2009.

[37] C. Jensen. Introduction to temporal database research. *Temporal database management*, pages 1–29, 2000.

[38] A. Jindal, H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, and S. Krishnan. Peregrine: Workload optimization for cloud query engines. In *SoCC*, pages 416–427, 2019.

[39] M. Kaufmann, P. Fischer, N. May, C. Ge, A. Goel, and D. Kossmann. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *ICDE*, pages 471–482, 2015.

[40] K. Kulkarni and J.-E. Michels. Temporal features in SQL:2011. *SIGMOD Rec.*, 41(3):34–43, 2012.

[41] L. Ma, D. Zhao, Y. Gao, and C. Zhao. Research on sql injection attack and prevention technology based on web. In *2019 International Conference on Computer Network, Electronic and Automation (ICCNEA)*, pages 176–179, 2019.

[42] MariaDB. MariaDB Source Code, 2019. https://mariadb.com/kb/en/library/getting-the-mariadb-source-code/.

[43] MariaDB. Overview of the Binary Log, 2019. https://mariadb.com/kb/en/library/overview-of-the-binary-log/.

[44] MariaDB. Temporal Data Tables, 2019. https://mariadb.com/kb/en/library/temporal-data-tables/.

[45] Microsoft. Advanced Analytics Architecture. https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/advanced-analytics-on-big-data.

[46] Microsoft. Compensating transaction pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction.

[47] Microsoft. What is Apache Hive and HiveQL on Azure HDInsight? https://docs.microsoft.com/en-us/azure/hdinsight/hadoop/hdinsight-use-hive#:~:text=Hive%20enables%20data%20summarization%2C%20querying,knowledge%20of%20Java%20or%20MapReduce.

[48] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: transparent high availability for database systems. *VLDB J.*, 22(1):29–45, 2013.

[49] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[50] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: a survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513–532, 1995.

[51] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, pages 155–174, 2004.
[52] Rodion Abdurakhimov. ExpressJS Source Code Repository, 2020. https://github.com/expressjs/express.
[53] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Trans. Knowl. Data Eng.*, 2(1):161–172, 1990.
[54] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.
[55] Sonal Srivastava. Conflict serializability in dbms, 2019. https://www.geeksforgeeks.org/conflict-serializability-in-dbms/.
[56] A. Spark. Spark overview. https://spark.apache.org/sql/.
[57] G. Speegle. *Compensating Transactions*, pages 405–406. Springer US, Boston, MA, 2009.
[58] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.
[59] P. Vadrevu, J. Liu, B. Li, B. Rahbarinia, K. H. Lee, and R. Perdisci. Enabling reconstruction of attacks on users via efficient browsing snapshots. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.
[60] Validata Group. Banks Busted by a Software Glitch during 2021, 2021. https://www.validata-software.com/blog-mobi/item/447-banks-busted-by-a-software-glitch-during-2021.
[61] J. S. M. Verhofstad. Recovery techniques for database systems. *ACM Comput. Surv.*, 10(2):167–195, 1978.
[62] Y. R. Wang and S. E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB*, pages 519–538, 1990.
[63] A. Webster, R. Eckenrod, and J. Purtilo. Fast and service-preserving recovery from malware infections using CRIU. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1199–1211, Baltimore, MD, Aug. 2018. USENIX Association.
[64] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
[65] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, 2014.

A The Table Columns Each Type of Query Reads/Writes

Query Type	Policy for Classifying Table Columns Into Read & Write Sets
`CREATE` || `ALTER TABLE`	R = { The columns of external tables (or views) this query's `FOREIGN KEY`s reference } W = { All columns of the table (or view) to be created or altered }
`DROP` || `TRUNCATE TABLE`	W = { All columns of the target table to be dropped or truncated + all external tables' `FOREIGN KEY` columns that reference this target table's any column }
`CREATE (OR REPLACE) VIEW`	R = { All columns of the original tables (or views) this view references } W = { All columns of the target view to be created }
`DROP VIEW`	W = { All columns of the view to be dropped }
`SELECT`	R = { the columns of the tables (or views) this query's `SELECT` or `WHERE` clause accesses + columns of external tables (or views) if this query uses a `FOREIGN KEY` referencing them + Union of the R of this query's inner sub-queries } , W = { }
`INSERT`	R = { Union of the R of this query's inner sub-queries + the columns of external tables (or views) if this query uses a `FOREIGN KEY` referencing them } W = { All columns of the target table (or view) this query inserts into }
`UPDATE` || `DELETE`	R = { Union of the R of this query's inner sub-queries + the columns of the target table (or view) this query reads + The columns of external tables (or views) if this query uses a `FOREIGN KEY` referencing them + The columns of the tables (or views) read in its `WHERE` clause } W = { Either the specific updated columns or all deleted columns of the target table (or view) + all external tables' `FOREIGN KEY` columns that reference this target table's updated/deleted column }
`CREATE TRIGGER`	R = { Union of the R of all queries within it } W = { Union of all queries within it } ** Also, add these R/W sets to the R/W sets of each query linked to this trigger (since this trigger will co-execute with it)*
`DROP TRIGGER`	R/W = the same as the read/set set of its counterpart `CREATE TRIGGER` query
`TRANSACTION` || `PROCEDURE`	R = { Union of the R of all queries within this transaction or procedure } ** Data flows via SQL's `DECLARE` variables or* W = { Union of the W of all queries within this transaction or procedure } *return values of sub-queries are also tracked*

Table A: Ultraverse's policy for generating a read set (R) and a write set (W) for each type of SQL query.

- In the above table, we intentionally omit all other SQL keywords that are not related to determining R/W sets (e.g., `JOIN`, `LIMIT`, `GROUP BY`, `FOR/WHILE/CASE`, `LABEL`, `CURSOR`, or `SIGNAL SQLSTATE`).

B Apper-generated Web Application

```
// Create the "SERVE_REQUSET" PROCEDURE of the Type-1 user request
    (GET send_money.html)
SQL_Execute('
  CREATE PROCEDURE SendMoney_CreateWebpage
  (IN base__html TEXT, IN username VARCHAR(32), OUT final_html TEXT) AS
  BEGIN
    DECLARE @personalizedTitle AS TEXT;
    IF username != "" THEN
     SET @personalizedTitle = username + " 's Account";
     SET @final_html = Utility__UpdateHTML(base__html, "title",
                          "innerHTML", personalizedTitle);
    END IF;
  END;');

// Create the "SERVE_REQUSET" PROCEDURE of the Type-2 user request
    (PUT send_money.html)
SQL_Execute('
  CREATE PROCEDURE SendMoney_Process
  (IN sender VARCHAR(32),
   IN receiver VARCHAR(32),
   IN amount VARCHAR(32), OUT result TEXT) AS
  BEGIN
    DECLARE cur_balance INT;
    SET @cur_balance = SELECT balance from Acccounts WHERE uid = sender;
    IF (cur_balance >= amount) THEN
      UPDATE Accounts SET balance  -= amount WHERE uid = sender;
      UPDATE Accounts SET balance += amount WHERE uid = receiver;
      INSERT INTO Transactions VALUES (sender, receiver, amount);
    END IF
    SET @result = "result: 'Successfully sent " + amount " to" + receiver + ""'
  END;');

// Request handler for "HTTP GET https://online-banking.com/send_money.html"
function HttpHandler_SendMoney_CreateWebpage (var cookie)
{
  // Log the Ultraverse web application client's ID and her user request
  SQL_Execute(`INSERT INTO Ultraverse__Log
            (request_name, Ultraverse__uid, pre_request_args)
            VALUES ("SendMoney_CreateWebpage", ${cookie.Ultraverse__uid}, "")`)

 // Run the server's SERVE_REQUEST in SQL
  var personalized_html = SQL_Execute('CALL SendMoney_CreateWebpage
                                        (${base__html}, ${cookie.username})');

  // Return the personalized webpage to the client as an HTTP response
  return personalized_html;
}

// Request handler for "HTTP PUT https://online-banking.com/send_money.html"
function HttpHandler_SendMoney_Process (var http)
{
  // Log the Ultraverse web application client's ID and her user request
  SQL_Execute(`INSERT INTO Ultraverse__Log
                (request_name, Ultraverse__uid, pre_request_args)
                VALUES ("SendMoney_Process", ${http.cookie.Ultraverse__uid},
                        ${JSON.parse(http.body).preRequest__args.stringify()})`)

 // Run the server's SERVE_REQUEST in SQL
  var serveRequest__args.0 = JSON.parse(http.body).receiver;
  var serveRequest__args.1 = JSON.parse(http.body).amount;
  var httpResponse__body = SQL_Execute(`CALL SendMoney_Process (
                                        ${http.cookie.uid},
                                        ${JSON.parse(http.cookie.body).recv},
                                        ${JSON.parse(http.cookiebody).amt})`);
  // Return the response in a JSON format
  return "{" + httpResponse__body + "}";
}
```

Figure 13: The server-side web application code generated by *Apper* based on Figure 7's *uTemplate*.

Figure 13 and Figure 14 are JavaScript application code generated by *Apper* based on Figure 7's *uTemplate*. String literals are colored in brown, of which SQL queries are colored in purple. variable names containing double underscores (__) are special variables reserved by Ultraverse, which are generated during *Apper*'s conversion from *uTemplate* to application code. Bold JavaScript/SQL variables store client-specific values (e.g., interactive user input or personalized DOM node). These bold variables may change their values during retroactive operations, and thus Ultraverse is designed to recompute their values during retroactive operation.

```
// The "SendMoney" webpage's base HTML which is auto-generated by Apper
const base__html = '<html>  <head></head>  <body>
  <script>
  // Create the "PRE_REQUEST" PROCEDURE of the Type-2 user request
      (PUT send_money.html)
   SQL_Execute(`CREATE PROCEDURE SendMoney_PRE_REQUEST
                  (IN receiver VARCHAR(16), IN amount VARCHAR(16),
                   OUT recv VARCHAR(16), OUT amt VARCHAR(16)) AS
                  BEGIN
                    SET @recv = receiver;
                    SET @amt = amount;
                  END`);
  </script>
  <h1 id="title">User Account</h1>
  <form onsubmit="SendMoney()">
    <input type="text" id="receiver" value="" >
    <input type="text" id="amount" value="" >
  </form>
  <script>
   function SendMoney()
   {  var preRquest_args = {};
     // Ensure no output data from POST_REQUEST may flow back to PRE_REQUEST
     console.assert(!document.getElementById("receiver").taint__bit);
     console.assert(!document.getElementById("amount").taint__bit);

     // Record interactive user inputs & non-interactive DOM values that are used
     as arguments to PRE_REQUEST. Later, send them to the server for future
     retroactive operation to replay the client's state. Only the arguments that are
     either interactive or personalized need to be refreshed during the replay.
     preRequest__args.0 = {};
     preRequest__args.0.interactive = true;
     preRequest__args.0.personalized = false;
     preRequest__args.0.dom_id = "receiver";
     preRequest__args.0.val = document.getElementById("receiver").value;

     preRequest__args.1 = {};
     preRequest__args.1.interactive = true;
     preRequest__args.1.personalized = false;
     preRequest__args.1.dom_id = "amount";
     preRequest__args.1.val = document.getElementById("amount").value;

     // Call "PRE_REQUEST" of the Type-2 user request (PUT send_money.html)
     var httpRequest__body = SQL_Execute(`CALL SendMoney
                (${preRequest__args.0.val}, ${preRequest__args.1.val})`);
     httpRequest__body.preRequest_args = preRequest__args;
     var httpResponse__body = http_send( host="online-banking.com", port=443,
       path="send_money.html", method="PUT",
       request_body= "{" + httpRequest__body + "}" ); // HTTP body as JSON format

     // "POST_REQUEST" of the Type-2 user request (PUT send_money.html)
     // Print the money transfer result message on the screen and taint
     // the output DOM node to prevent data flow back to PRE_REQUEST
     var postRequest__args = {};
     postRequest__args.result = JSON.parse(httpResponse__body).result;
     var newNode__0 = document.createElement("p");  st
     newNode__0.id = "result";
     newNode__0.innerHTML = postRequest__args.result;
     newNode__0.taint__bit = true;
     document.getElementsByTagName("body")[0].appendChild(newNode__0);

     // "POST_REQUEST": Update the client's browser cookie
     document.cookie.last_activity = result;
   }
   </script>
   </body></html>';
```

Figure 14: The base HTML common to all clients generated by *Apper* based on Figure 7's *uTemplate*.

Logging & Replaying User Requests: Ultraverse reserves one special property in the browser's cookie, which is `cookie.Ultraverse__uid`. This property stores each Ultraverse web application client's unique ID, which can be also used as an application-level unique user ID. Whenever the client makes a Type-2 (client-server remote data processing) or Type-3 (client-only local data processing) request, the webpage's client-side JavaScript (generated by *Apper*) sends the application-specific user request data to the web server. This code also silently piggybacks the following 3 system-level information to the server: *(1)* the client's unique ID (i.e., `cookie.Ultraverse__uid`); *(2)* the user request's name (e.g. *"SendMoney"*); and *(3)* the arguments of the user request's `PRE_REQUEST` call. These 3 pieces of information are essential for Ultraverse to run an application-level

retroactive operation in a multi-client service. Specifically, the arguments used in the PRE_REQUEST call are one of the three types: *(i)* the DOM node that stores the user's interactive input (e.g., the <input id="amount"> tag stores the transfer amount); *(ii)* the DOM node that was customized by the web server based on the client's identity during the webpage creation (e.g., the <h1 id="title"> tag stores Alice's name); *(iii)* all other DOM nodes in the webpage that are constant and common to all users (e.g., the <form onsubmit="SendMoney()"> tag). When Ultraverse runs a retroactive operation, its default mode assumes that the state of the DOM nodes which store a user's interactive inputs is the same as in the past. However, Ultraverse assumes that customized DOM nodes may change their values during retroactive operation, and thus Ultraverse is designed to recompute their values during the replay phase by replaying the webpage's Type-1 webpage creation request (SERVE_REQUEST). Given the standard user request routine comprised of PRE_REQUEST → SERVE_REQUEST → POST_REQUEST, some user request may have only SERVE_REQUEST (e.g., a web server's locally invoked internal scheduler routine) or only POST_REQUEST (e.g., a client's local data processing). For any given user request, Ultraverse only needs to log the arguments of the initial PROCEDURE of the user request, because the arguments of the subsequent PROCEDUREs can be deterministically computed based on the prior PROCEDURE's return value. Therefore, the subsequent arguments are recomputed by Ultraverse while replaying the user request.

Logging & Replaying Browser Cookies: Ultraverse also replays the evolution of the client browser's cookie state by replaying the user request's SQL logic of updating Ultraverse's specially reserved table: BrowserCookie. The developer is required to implement each webpage's customized cookie handling logic as SQL logic of updating the BrowserCookie table in PROCEDUREs in *uTemplate*. See §C.1 for further details on how Ultraverse handles the browser cookie for each of Type-1, Type-2, and Type-3 user requests.

Logging & Replaying Application Code's Persistent Variables: While a client stays in the same webpage, its Type-2 or Type-3 user requests may write to some JavaScript variables in the webpage whose value persistently survive across multiple user requests (e.g., global or static variables). Ultraverse requires the developer to implement all application logic accessing such persistent application variables via Ultraverse's specially reserved table: AppVariables. During a retroactive operation, Ultraverse replays the state of variables stored in this table the similar way it does for replaying the browser cookie with the BrowserCookie table. Note that the developer using the Ultraverse web framework's *uTemplate* fundamentally has no way to directly access the browser cookie or the application's persistent variables, and can access them only through the BrowserCookie and AppVariables tables as SQL logic implemented in PRE_REQUEST, SERVE_REQUEST, and POST_REQUEST.

C Extended Design Features

C.1 Handling the Browser Cookie across User Requests

Modern web frameworks allow a client and server to update the client browser's cookie by using one of 3 ways: *(i)* set an HTTP request's COOKIE field; *(ii)* set an HTTP response's SET-COOKIE field; *(iii)* update client-side JavaScript's document.cookie object. During retroactive operation, all such cookie-handling logic should be retroactively replayed to properly replay the evolution of the client's state. To achieve this, Ultraverse enforces the developer to implement the client-side logic of updating cookies by using SQL so that it can be captured and replayed by Ultraverse's query analyzer. Also, *Apper*-generated JavaScript code silently updates each HTTP message's COOKIE and SET-COOKIE fields before sending it out to client/server.

As explained in §4.2, Type-1 user request handlers create and return a webpage for a client's requested URL. When retroactively replaying a Type-1 user request, Ultraverse assumes that the requested URL and most of the HTTP header fields are the same as in the past, while the following three elements are subject to change: *(a)* the client's HTTP GET request's COOKIE field; *(b)* the HTTP response's SET-COOKIE field; *(c)* the returned webpage's some HTML tags customized by SERVE_REQUEST based on the client's retroactively changed COOKIE and the server's retroactively changed database state. During retroactive operation, replaying all these three elements is important, because they often determine the state of the returned webpage. In order to replay a client's HTTP request's COOKIE field, the same client's all (column-wise & row-wise dependent) Type-2 and Type-3 requests executed before this should be also retroactively replayed, because they can affect the client's cookie state in the database. Ultraverse ensures to replay them while retroactively replaying its global log. Ultraverse achieves this by enforcing the developer to implement the update logic of browser cookies as SQL logic of updating Ultraverse's specially reserved BrowserCookie table in PRE_REQUEST, SERVE_REQUEST, and POST_REQUEST. During retroactive operation, each client's BrowserCookie table is replayed, which is essentially the replay of each client's browser cookie. Given the developer's SQL logic of updating the BrowserCookie table, the *Apper*-generated application code also creates the equivalent application logic that updates Javascript's document.cookie, mirroring the developer's SQL logic implementation, so that each user request's HTTP request's COOKIE field and HTTP response's SET-COOKIE fields will contain the proper value of the browser cookie stored in document.cookie.

Each client webpage manages its own local BrowserCookie table (e.g., BrowserCookie_Alice) comprised of 1 row, whereas the server-side global database has the GlobalBrowserCookie table which is essentially the union of the rows of all clients' BrowserCookie tables. Ultraverse uses virtual two-level page mappings (§C.7) from BrowserCookie_<username>→GlobalBrowserCookie.

C.2 Preserving Secrecy of Client's Secret Values

During retroactive operation, Ultraverse's replay phase, by default, uses the same interactive user inputs as in the past as arguments to Type-2 user request's PRE_REQUEST or Type-3 user request's POST_REQUEST. However, some of the user inputs such as password strings are privacy-sensitive. In practice, modern web servers store a client's

password as hash value instead of cleartext, because exposing the client's raw password breaks her privacy. To preserve privacy, Ultraverse's *uTemplate* additionally provides an optional function called PRE_REQUEST_SECRET, which is designed to read values from privacy-sensitive DOM nodes (e.g., <input id="password">) or do privacy-sensitive computation (e.g., picking a value in an array based on the client's secret random seed). When the client's Type-2 user request is executed during regular operation, PRE_REQUEST_SECRET is first executed and its OUTPUTs (e.g., a hashed password) are pipelined to PRE_REQUEST as INPUTs, which further processes the data and then sends them over to the web server as SERVE_REQUEST's INPUTs. Meanwhile, the *Apper*-generated application code does not record PRE_REQUEST_SECRET's INPUTs (i.e., raw password), and the user request execution log being sent to the web server includes only PRE_REQUEST's INPUTs (i.e., hashed password). Thus, the client's privacy is preserved. During the retroactive operation, Ultraverse will replay the user request by directly replaying PRE_REQUEST by using the hashed password as its INPUT argument (without replaying PRE_REQUEST_SECRET).

C.3 Client-side Code's Dynamic Registration of Event Handlers

A web service's developer may want to design a webpage's JavaScript to dynamically register some event handler. For example, in Figure 7, a developer may want to register the SendMoney function to the <form> tag's onsubmit event listener only after the webpage is fully loaded. The motivation for this is to prevent the client from issuing a sensitive money transfer request before the webpage is fully ready for service (to avoid the webpage's any potential irregular behavior). To fulfill this design requirement, Ultraverse's *uTemplate* used for a Type-2 or Type-3 user request provides an optional section called *Event Registration*, which is comprised of 4 tuples: ⟨*dom_id*, *event_type*, *user_request_name*, *add_or_remove*⟩. Once *Apper* converts this *uTemplate* into application code, it runs in such a way that after the Type-2 or Type-3 user request's POST_SERVE is executed, it scans each row of the *Event Registration* section (if defined), and it dynamically adds or removes *user_request_name* from *dom_id*'s *event_name* listener. For example, in the alternate version of Figure 7, suppose that its base HTML's <form onsubmit="SendMoney()"> is replaced to <form id="form_send"> and its <body> is replaced to <body onload="RegisterSendMoney()">. In this new version of webpage, at the end of the pageload, the RegisterSendMoney() function is called, which in turn registers the SendMoney function to the <form id="form_send"> tag's onsubmit event listener, so that the client can issue her money transfer only after the pageload completion. To implement this webpage's logic in the Ultraverse web framework, Figure 7's Type-1 and Type-2 user requests are unchanged, and there will be an additional Type-3 user request defined for client=RegisterSendMoney(), whose POST_REQUEST is empty and whose *Event Registration* section defines the tuple: ⟨"form_send", "onsubmit", "SendMoney()", "add"⟩. Given this *uTemplate*, the *Apper*-generated JavaScript will execute the Type-3 user request at the end of the pageload, which will in turn call document.getElementById("form_send").addEventListener("onsubmit", SendMoney, true) to register the Type-2 user request handler "SendMoney()" to the <form id="form_send"> tag's onsubmit listener.

To remove an existing event listener, the *Event Registration* tuple's *add_or_remove* value should be set "remove" instead, and then the *Apper*-generated JavaScript code's Type-3 user request will execute document.getElementById("form_send").removeEventListener("onsubmit", SendMoney, true).

C.4 Clients Tampering with Application Code

A malicious client might tamper with its webpage's JavaScript code to hack its user request logic and sends invalid user request data to the web server. Note that a client can do this not only in a Ultraverse web application, but also in any other types of web applications. By practice, it is the developer who is responsible to design his server-side application code to be resistant to such client-side code tampering. However, when it comes to Ultraverse's retroactive operation, such client-side code tampering could result in an inconsistent state at the end of a retroactive operation, because Ultraverse cannot replay the client's same tampered code which is unknown. However, Ultraverse can detect the moment when a client's tampered code causes an inconsistency problem in the server's global database state, because Ultraverse's *uTemplate* has all Type-2 and Type-3 user request code that is to be executed by the client's browser. Besides the web application service's regular operations, Ultraverse's offline *Verifier* can replay each user request recorded in the *global* Ultraverse logs to detect any mismatch between: *(i)* the user request call's associated SERVE_REQUEST's INPUT arguments submitted by the client during regular service operations; *(ii)* the same user request call's associated PRE_REQUEST's OUTPUT replayed by the offline *Verifier*. These two values should always match for any benign user request; a mismatch indicates that the client either had sent a contradicting user request execution log, or had tampered with the client-side JavaScript (originally generated by *Apper*) to produce a mismatching OUTPUT contradicting the one replayed by the server's genuine PRE_REQUEST. The server can detect such contradicting user requests and take countermeasures (e.g., retroactively remove the contradicting user request). This verification is needed only once for each newly committed user request, based on which the server can run any number of retroactive operations. *Verifier* needs to do replay verification for at least those user requests which were committed within the desired retroactive operation's time window. For example, most financial institutions detect a cyberattack within 16 hours from the attack time (§5.3), in which case *Verifier* must have done (or must do) the replay verification for the user requests committed within the latest 16 hours.

C.5 Handling Retroactive AUTO_INCREMENT

Suppose that an online banking service's each money transfer transaction gets a transaction_id whose value is AUTO_INCREMENTed based on its SQL table schema. And suppose that later, some past user request which inserts a row into the Transaction table is retroactively removed. Then, each subsequently replayed money transfer transaction's transaction_id will be decreased by 1. This phenomenon may or may not be desired depending on what the application service provider expects. In case the service provider rather wants to preserve the same transaction_ids of all past committed transactions even after the retroactive operation, Ultraverse provides the developer an option for this. When this option is enabled, Ultraverse marks a tombstone on the retroactively removed

AUTO_INCREMENT value of the Transaction table to ensure that this specific transaction_id value should not be re-used by another query during the retroactive operation. This way, all subsequently replayed transactions will get the same transaction_ids as before. Similarly, when a new query is retroactively added, Ultraverse provides the option of applying tombstones to all queries that were committed in the past, so that the retroactively added query will use the next available AUTO_INCREMENT value (e.g., the highest value in the table) as its transaction_id, not conflicting with the transaction_ids of any other transactions committed in the past.

C.6 Advanced Query Clustering

C.6.1 Support for Implicit/Alias Cluster Key Columns

This subsection requires reading §4 first.

When the query clustering technique is used in the Ultraverse web framework, there are 2 new challenges. We explain them based on Figure 3's example.

Challenge 1: A developer's web application code may use a certain table's column as if it were a foreign key column, without defining this column as an SQL-level FOREIGN KEY in the SQL table's schema. For example, if Figure 3 is implemented as a web application, it is possible that the application code's SQL table schema which creates the Accounts table (Q3) does not explicitly define Accounts.uid as a FOREIGN KEY referencing Users.uid, but the application code may use Accounts.uid and Users.uid as if they were in a foreign key relationship (e.g., whenever a new account is created, the application code copies one value from the Users.uid column and inserts it into the Accounts.uid column, and whenever some Users.uid is deleted, the Accounts table's all rows containing the same Accounts.uid value are co-deleted, which essentially implements the SQL's foreign key logic of ON DELETE CASCADE). Ultraverse defines such a foreign key which is perceivable only from the application semantics as an *implicit* foreign key. It's challenging to determine whether application code uses a particular table column as an implicit foreign key, because this requires the understanding and analysis of the application-specific semantics.

Challenge 2: The application code's generated query statement may use an (foreign/alias) cluster key not as a concrete value, but as an SQL variable. Resolving a variable into a concrete cluster key requires careful analysis, because the optimization and correctness of the row-wise query clustering technique is based on the fundamental premise that the database's each committed query's cluster key set is constant, immutable, and independent from any retroactive operations. If this premise gets violated, using the query clustering technique could break the correctness of retroactive operation. Thus, during the query clustering analysis, when Ultraverse sees a query that uses an SQL variable as a cluster key, Ultraverse has to identify not only the runtime-concretized value of the SQL variable recorded in the SQL log, but also whether the concretized value will be guaranteed to be the same and immutable across any retroactive operations– if not, this query's cluster key set should be treated as $\neg\emptyset$ (i.e., this column cannot be used as a cluster key).

Solution: To solve the 2 challenges, Ultraverse has to scan the application code to identify implicit foreign key relationships and to determine the immutability (i.e., idempotence) of the concretized value of the SQL-variable cluster keys used in query statements. To do this, Ultraverse runs static data flow analysis on the PROCEDUREs defined in the web application's all *uTemplates*. This data flow analysis runs the following two validations:

- **Validation of Implicit Foreign Cluster Key Columns:** This validates if a candidate column in the database is a valid implicit foreign key column . For this validation, Ultraverse checks the following for every data flow in every PROCEDURE in every uTemplate whose sink is the candidate column: (1) the source of the data flow should be the cluster key column; (2) the data flow between the source and the sink is never modified by any binary/unary logic, throughout all control flow branches (if any) that the data flow undergoes. This is to ensure that the data flow sink column's each value always immutably (i.e., idempotently) mirror the same value from the data flow source cluster key column across any retroactive operation's replay scenarios. Meanwhile, there are cases that a data flow's source is a DOM node used as INPUT to PRE_REQUEST, SERVE_REQUEST, or POST_REQUEST. To handle such cases, Ultraverse allows the developer to optionally mark the DOM node(s) (in the base HTML in uTemplate) which stores a cluster key. For example, if Ultraverse's choice rule for the cluster key column (Table 2) reports Users.uid as the optimal cluster key column, then the developer would mark any DOM nodes in *uTemplats* that stores the user's ID (e.g., an <input id='user_id'> tag) as "cluster_key:Users.uid". Note that if the developer mistakenly or purposely omits the marking of cluster key DOM nodes in *uTemplate*, this only makes Ultraverse potentially lose the opportunity of applying the query clustering optimization, without harming Ultraverse's correctness of retroactive operation.
- **Validation of Variable Cluster Keys:** This validates if a runtime-concretized value of an SQL variable (or its expression) used as a cluster key can be considered as a valid cluster key. For this validation, Ultraverse checks that for each data flow whose sink is a (foreign/alias) cluster key column and the value flowing to the sink is an SQL variable (or its expression), the data flow's source(s) should be an immutably (i.e., idempotently) replayable source(s). Specifically, Ultraverse checks if the source(s) is comprised of only the following immutable (i.e., idempotent) elements: (1) a constant literal; (1) a DOM node that stores a constant value across any retroactive operation; (3) another (foreign/alias) cluster key column; (4) an SQL function that is guaranteed to return the same value across any retroactive operation scenario. RAND()/CURTIME() are also valid sources, because their seed values are recorded by Ultraverse during regular operations for idempotent replay (§3.4). This validation is applied to all control flow branches (if any) that the data flow undergoes. A variable cluster key comprised of the above 4 idempotent elements is guaranteed that its runtime-concretized cluster key is also idempotent across any retroactive operation's replay scenarios.

Note that when the developer uses the Ultraverse web framework, Ultraverse only needs to run the above two validations on each *uTemplate*'s PROCEDURE definition merely once before the the service deployment, because all runtime user requests are calls of those same PROCEDUREs. Thus, the data flow analysis does not incur runtime overhead.

In our experiment (§5), there are 2 benchmarks that involve variable cluster keys: TATP (Figure 16) and SEATS (Figure 22). And there

Condition for a Valid Cluster Key Column	
c	: A candidate cluster key column
t_c	: The table that owns the column c
$t_i \rightarrow t_j$	: t_i depends on t_j (i.e., a data flow from t_j to t_i exists in the transaction history of the retroactive operation's time window)
Condition.	$\forall i, j((W(Q_i) \neq \emptyset) \wedge (t_j \in (R(Q_i) \cup W(Q_i)) \wedge (t_j \rightarrow t_c) \Longrightarrow K_c(Q_i) \neq \neg\emptyset)$

Table 14: Generalized condition for valid cluster key columns.

is 1 benchmark that involves implicit foreign key columns: Invoice Ninja web service (Figure 25).

C.6.2 Support for Multiple Cluster Key Columns

Ultraverse can further reduce the granularity of query clustering by using multiple cluster key columns simultaneously. Given a database and its transaction history, suppose that the database's tables can be classified into two groups such that each group's tables have data flows only among them and there is no data flow between two table groups. Then, there is no data interaction between the two table groups, and thus each table group can independently use its own cluster key column to cluster table rows in its table group.

However, there are many cases in practice such that a database's tables cannot be classified into 2 disjoint groups, because the data flows between some tables prevent the formation of disjoint table groups. Yet, even in such cases, there is a way to use multiple cluster key columns simultaneously for finer-grained query clustering. We explain this by example. Figure 22 is the cluster key propagation graph for the SEATS benchmark in BenchBase [22]. In this benchmark, customers create/edit/cancel reservations for their flight. While running this benchmark, its transactions update only 4 tables: `Flight`, `Customer`, `Reservation`, and `Frequent_Flyer`. In SEATS's all transactions, there exist only 3 types of data flows among tables: `Flight`→`Reservation`, `Customer`→`Reservation`, and `Customer`→`Frequent_Flyer`. Further, in SEATS's all transactions that update the database, its each query's `SELECT`, `WHERE` or `VALUE` clause specifies a concrete value of `Flight.f_id`, `Customer.c_id`, or both.

Given SEATS's above transaction characteristics, Ultraverse can apply the following row-wise query clustering strategy:

- The `Flight` table can be row-wise clustered by using `Flight.f_id` as the cluster key column.
- The `Customer` and `Frequent_Flyer` tables can be row-wise clustered by using `Customer.c_id` as the cluster key column.
- The `Reservation` table can be row-wise clustered by using *both* `Flight.f_id` and `Customer.c_id` as the cluster key columns.

Note that the `Reservation` table receives data flows from both the `Customer` and `Flight` tables, which is why clustering the `Reservation` table's rows requires using both `Flight.f_id` and `Customer.c_id` as cluster keys. On the other hand, the `Flight` table can cluster its rows only by using `Flight.f_id`, because it does not get data flows from any other tables. Similarly, the `Customer` and `Frequent_Flyer` tables can cluster their rows only by using `Customer.c_id`, because they don't get data flows from any other tables. As a result, Ultraverse can cluster queries (i.e., transactions) of SEATS by *carefully* using both `Flight.f_id` and `Customer.c_id` as the cluster key columns.

In the query clustering scheme that supports multiple cluster key columns, each cluster key (besides its value) additionally gets the *type* information, where the type is its origin cluster key column. Thus, each cluster key is a 2-dimensional element, comprised of ⟨*cluster key column*, *value*⟩. In the example of SEATS, the two types for its two cluster key columns are `Flight.f_id` and `Customer.c_id`. Each query's K sets contain 2-dimensional cluster keys as elements. When applying the query clustering rule (Table 2), the K sets of two queries are considered to have an intersection only if they have at least 1 common element whose type and value *both* matches.

Table 14 describes Ultraverse's generalized condition for a valid cluster key column, which is general enough to be applied to both cases of using a single cluster key column (§3.5) and multiple cluster key columns. This condition states that for every database-updating query Q_i (or transaction) in the transaction history belonging to the retroactive operation's time window, for every table t_j that Q_i operates on (i.e., t_j belongs to Q_i's R/W sets), if there exists a data flow from the table that owns the cluster key column c (i.e., t_c) to table t_j, then Q_i's `SELECT`, `WHERE` or `VALUE` clause should specify the (foreign/alias) cluster key associated with c. Otherwise, t_j's rows that Q_i operates on cannot be clustered based on the values of the c column, and thus c cannot be used as a cluster key column.

Given Table 14's validation of cluster key columns, the SEATS benchmark has 2 satisfying cluster key columns: `Flight.f_id` and `Customer.c_id` (Figure 22); the Epinions benchmark also has 2 satisfying cluster key columns: `Useracct.u_id` and `Item.i_id` (Figure 18). Our evaluation (§5.1) shows performance improvement of retroactive operation by simultaneously using the 2 cluster key columns in each benchmark.

C.7 Virtual Two-level Table Mappings

In many web applications, there exist a table equivalent to the `Sessions` table, whose each row stores each user's session information, such as a login credential or last active time. When a web server processes a logged-in user's any request, the server first accesses the `Sessions` table to verify the user's login credential and update the user's last active time. This service routine is problematic for column-wise dependency analysis, because all users' requests will end up with mutual column-wise *write-write* dependency on the `Sessions.last_active_time` column, and thus the column-wise dependency analysis alone cannot reduce the number of replay queries for retroactive operation. This problem can be solved by co-using the row-wise dependency analysis (§C.6) to split the `Sessions` table's rows by using `Users.user_id` as the cluster key column. Nevertheless, Ultraverse provides an alternative solution purely based on the column-wise dependency analysis.

Ultraverse's *uTemplate* supports virtual two-level table mappings in SQL query statements, which uses the new syntax: *tableName<columnName>*. For example, the developer's query statement in *uTemplate* can use the syntax `Sessions<'alice'>` to access Alice's session data. Then, when Ultraverse's query analyzer analyzes the column-wise dependency of user requests defined in *uTemplates*, the anlayzer considers `Sessions<'alice'>` as a *virtual* table uniquely dedicated to Alice, named `Sessions_alice`, comprised of 1 row. Therefore, if another user request accesses `Sessions<'bob'>` for example, Ultraverse considers these two queries to access different tables (i.e., `Sessions_alice` and `Sessions_bob` respectively),

thus two user requests are column-wise independent from each other. Therefore, when retroactively replaying Alice's user request, we don't need to replay Bob's user request even if we do not use the row-wise dependency analysis (§C.6), because Bob's query is column-wise independent from Alice's. Meanwhile, in the database system, Alice and Bob's data records are physically stored in the same `Sessions` table in different rows. Ultraverse's *Apper* rewrites `Sessions<'alice'>` in the *uTemplate* such that it reads/writes `Sessions WHERE user_id='alice'`. Therefore, the virtualized table is interpreted only by Ultraverse's query analyzer to improve the performance of column-wise dependency analysis. Another motivation for table virtualization is that for any applications, maintaining a small number of physical tables is important, because creating as many tables as the number of users could degrade the database system's performance in searching user records.

In our macro-benchmark evaluation of the Invoice Ninja web service (§5.2), we applied virtual two-level table mappings to the `Sessions` and `Cookies` tables (Figure 25).

.

C.8 Column-Specific Retroactive Update

For a retroactive operation for data analytics, a user may not need the retroactive result of certain columns, such as bulky debugging table's verbose error message column. Carefully ignoring retroactive updates of such unneeded column(s), say uc_a, can further expedite retroactive operation. Ultraverse supports such an option by safely ignoring the retroactive update of uc_a if no other columns to be rolled back & replayed depend on the state of uc_a. The algorithm is as follows:

(1) The user chooses columns to ignore. Ultraverse stores their names in the *IgnoreColumns* set, and stores names of all other columns of the database in the *IncludeColumns* set.
(2) Ultraverse generates the query dependency graph based on column-wise and cluster-wise analysis.
(3) For each column uc_a in *IgnoreColumns*, Ultraverse moves it to *IncludeColumns* if the following two conditions are true for some query Q_b in the query dependency graph: *(i)* uc_a appears in Q_b's read set; *(ii)* Q_b's write set contains some column in the *IncludeColumns* set. This step runs repeatedly until no more columns are moved from *IgnoreColumns* to *IncludeColumns*.
(4) For each query Q_c in the query dependency graph, Ultraverse safely removes Q_c from the graph if its write set includes only those columns in *IgnoreColumns*.
(5) Ultraverse rollbacks/replays the query dependency graph.

D Retroactive Attack Recovery Scenarios and Ultraverse's Optimization Analysis

In this section, we explain the attack recovery scenarios based on retroactive operations as evaluated in §5 and explains how Ultraverse's optimization techniques are applied. For each benchmark, we also show the column-wise query (transaction) dependency graph, where each node represents a transaction and its R/W sets. Note that we omit read-only transactions from the graphs because they do not affect the database's state during both regular and retroactive operations.

D.1 TATP

TATP's dataset and transactions are designed for mobile network providers to manage their subscribers.

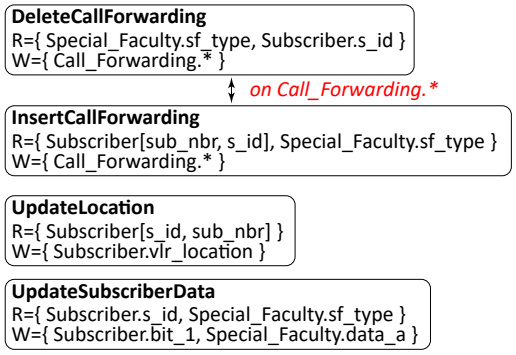


Figure 15: TATP's transaction dependency graph.

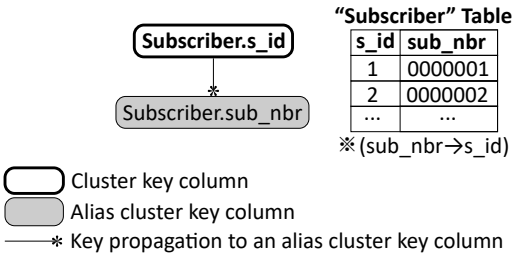


Figure 16: TATP's cluster key propagation graph.

*D.1.1 Attack Recovery Scenario 1*An attacker initiated a tampered `UpdateLocation` user request to provide wrong information about the user's location (e.g., GPS or geographic area). After identifying this transaction, Ultraverse retroactively corrected the data in the query by applying the following optimization techniques.

Column-wise Query Dependency: There was no need to roll back and replay its subsequent `DeleteCallForwarding`, `InsertCallForwarding`, and `UpdateSubscriberData` transactions. This was because these transactions are column-wise independent from the `UpdateLocation` transaction. In particular, these transactions does not contain the `UpdateLocation` transaction's write set element `SUBSCRIBER.vlr_location` in their read/write set.

Row-wise Query Clustering: Ultraverse's query clustering scheme clustered all committed transactions into the number of distinct phone subscribers. Therefore, when a transaction belonging to a particular cluster (i.e., subscriber) was retroactively corrected, all the

other transactions belonging to other clusters (i.e., other subscribers) didn't need to be rolled back and replayed.

*D.1.2 Attack Recovery Scenario 2*An attacker initiated several `InsertCallForwarding` transactions which inserted tampered rows into the `CALL_FORWARDING` table. We used Ultraverse to retroactively remove them based on the column-wise query dependency analysis and row-wise query clustering, as well as the following optimization technique.

Hash-jump: After committing all the previously existing counterpart `DeleteCallForwarding` transactions of the retroactively removed `InsertCallForwarding` transactions, the `CALL_FORWARDING` table's hash value became the same as before the retroactive operation, and thus Ultraverse returned the `Call_Forwarding` table's same state stored before the retroactive operation.

D.2 Epinions

Epinions' dataset and transactions are designed to generate recommendation system networks based on online user reviews.

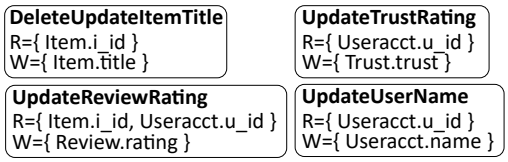


Figure 17: Epinion's transaction dependency graph (none).

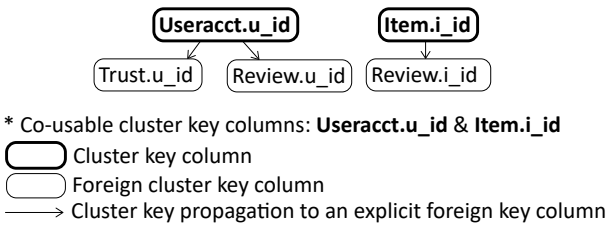


Figure 18: Epinions' cluster key propagation graph.

*D.2.1 Attack Recovery Scenario 1*Several `UpdateReviewRating` user requests turned out to be initiated by a remote attacker's botnet. We retroactively removed those `UpdateReviewRating` transactions by using the following optimization technique.

Column-wise Query Dependency: There was no need to roll back and replay its subsequent `UpdateUserName`, `UpdateUserName`, and `UpdateTrustRating` transactions, because they are column-wise independent from `UpdateReviewRating`. In particular, these transactions does not contain the `UpdateReviewRating` transaction's write set element `review.rating` in their read/write set.

Row-wise Query Clustering: Ultraverse used multiple cluster key columns (`Useracct.u_id` and `Item.i_id`) and only needed to roll back & replay the `UpdateReviewRating` transactions whose cluster key is the same as that of the retroactively removed `UpdateReviewRating` transaction.

*D.2.2 Attack Recovery Scenario 2*An attacker-controlled transaction changed a victim user's account information by initiating a tampered `UpdateUserName` request and changed the `UserAcct` table's state. Later in time, there was the victim user's benign `UpdateUserName` request that changed the his account information to a different state. To correct any potential side effects, we decided to retroactively remove the attacker-initiated `updateUserName` transaction. Ultraverse used the column-wise query dependency analysis and row-wise query clustering, as well as the following optimization technique.

Hash-jump: During the retroactive operation, Ultraverse detected a hash hit upon executing the victim user's benign `UpdateUserName` request that overwrote the attacker-tampered account information. Thus, Ultraverse returned the `UserAcct` table's same state stored before the retroactive operation.

D.3 Resourcestresser

Resource Stresser's dataset and transactions manage employment information (e.g., salary), which are technically designed to run stress-testing on a CPU, disk I/O, and locks from a database system.

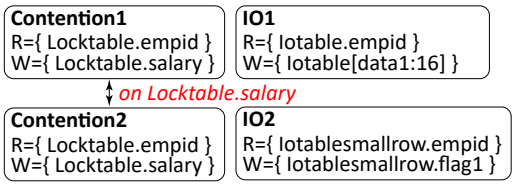


Figure 19: ResourceStresser's transaction dependency graph.

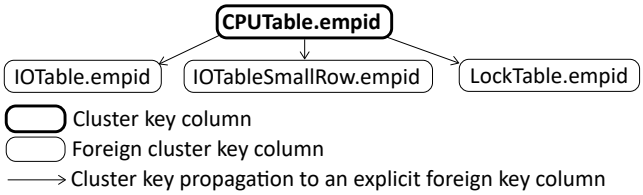


Figure 20: ResourceStresser's cluster key propagation graph.

*D.3.1 Attack Recovery Scenario 1*The attacker injected mal-formed `IO2` transactions. We used Ultraverse to retroactively remove the identified transaction by using the following optimization technique.

Column-wise Query Dependency: There was no need to roll back and replay its subsequent independent `Contention1`, `Contention2`, and `IO1` transactions, because they are column-wise independent from `IO2`. In particular, these transactions did not contain the `IO2` transaction's write set element `iotablesmallrow.flag1` in their read/write set.

Row-wise Query Clustering: Ultraverse used `CPUTable.empid` as the cluster key and only needed to roll back & replay the `IO2` transactions whose cluster key is the same as that of the retroactively removed `IO2` transaction.

*D.3.2 Attack Recovery Scenario 2*The attacker changed the order of certain `IO1` transactions which updated the `iotable` table's state. Once those transactions were identified, we retroactively corrected their commit order. Ultraverse used the column-wise query dependency analysis with the following optimization technique.

Hash-jump: After committing the last `IO1` transaction which was correctly ordered, the `iotable` table's evolution of hash values matched the ones before the retroactive operation. Thus, Ultraverse returned the table's same state stored before the retroactive operation.

D.4 SEATS

SEAT's dataset and transactions are designed for an online flight ticket reservation system.

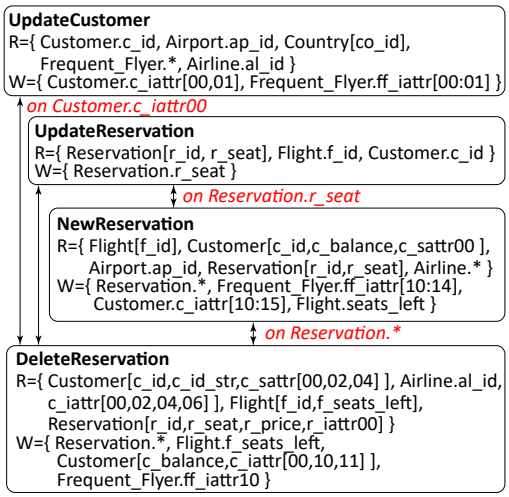
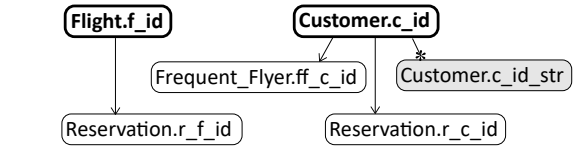


Figure 21: SEAT's transaction dependency graph.



* Co-usable cluster key columns: **Flight.f_id** & **Customer.c_id**

Cluster key column

Foreign cluster key column

Alias cluster key column

Cluster key propagation to an explicit foreign key column

Key propagation to an alias cluster key column (same table)

Figure 22: SEATS's cluster key propagation graph.

*D.4.1 Attack Recovery Scenario 1*An attacker broke into the flight ticket reservation system and tampered with passengers' reservation information by issuing malicious `UpdateReservation` user requests. After the problematic transactions were identified, Ultraverse retroactively updated the database by using the following optimization technique.

Row-wise Query Clustering: Ultraverse used multiple cluster key columns (`Flight.f_id` and `Customer.c_id`), so only needed to rollback & replay the transactions which are in the same cluster as the retroactively updated `UpdateReservation` transaction.

*D.4.2 Attack Recovery Scenario 2*The attacker intercepted and swapped the commit order of a client's two `UpdateCustomer` transactions, both of which updated only the client's `Customer.iattr01` metadata. Ultraverse retroactively corrected their commit order based on the column-wise query dependency analysis and row-wise query clustering, as well as the following optimization technique.

Hash-jump: There is no transaction that reads (i.e., depends on) `Customer.iattr01`, so the other tables' states were unaffected by this attack. After correcting the order of two `UpdateCustomer` transactions, there was another benign transaction which overwrote the client's value of `Customer.iattr01` metadata, after which point the `Customer` table hash for this client's cluster matched the past version. Thus, Ultraverse returned the table's same state stored before the retroactive operation.

D.5 TPC-C

TPC-C's dataset and transactions are designed to manage product orders shippings for online users in an e-commercial service.

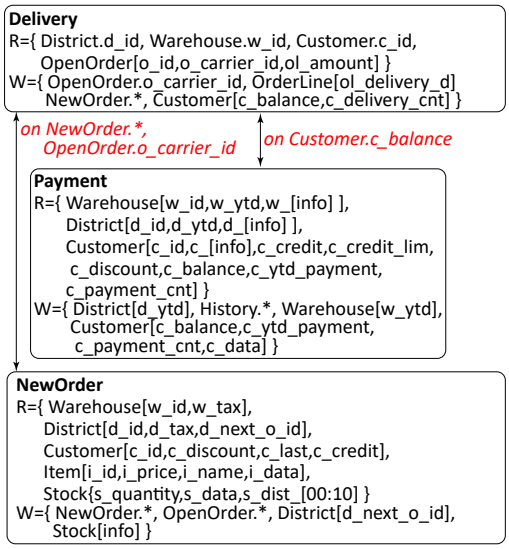
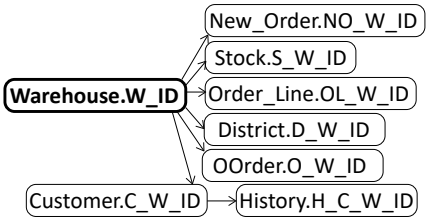


Figure 23: TPC-C's transaction dependency graph.



Cluster key column

Foreign cluster key column

Cluster key propagation to an explicit foreign key column

Figure 24: TPC-C's cluster key propagation graph.

*D.5.1 Attack Recovery Scenario 1*We configured the number of warehouses to 10, which corresponded to the number of clusters in Ultraverse's query clustering scheme. An attacker injected a fabricated `Payment` transaction without an actual payment. After this transaction was identified, Ultraverse retroactively removed the transaction by applying the following optimization technique.

Row-wise Query Clustering: Ultraverse used `Warehouse.w_id` as the cluster key column, so only needed to rollback and reply those transactions which had the same cluster key as the retroactively removed `Payment` transaction.

*D.5.2 Attack Recovery Scenario 2*After the attacker's fabricated a `Payment` transaction, the vendor for this product failed to ship it out due to an issue with logistics, so the delivery was cancelled by the vendor and the victimized client's balance received the refund in the `Customer` table. However, to ensure correctness, we used Ultraverse to retroactively remove the malicious `Payment` transaction. Ultraverse used row-wise query clustering with the following optimization technique.

Hash-jump: After retroactively removing the malicious `Payment` transaction, and after the commit time when its associated `Delivery`

transaction refunded the cost to the customer, the hash value and the subsequent transactions for the `Customer` table matched the ones before the retroactive operation. Thus, Ultraverse returned the table's same state stored before the retroactive operation.

D.6 Invoice Ninja

Invoice Ninja is an online-banking web application service that manages user accounts and transfer of funds between users. We evaluated 5 scenarios where an attacker controlled the following 5 user requests: "Create a Bill", "Modify an Item in a Bill", "Login", MakePayment", and "Logout". We used virtual two-level table mappings (§C.7) for the `Sessions` and `Cookies` tables.

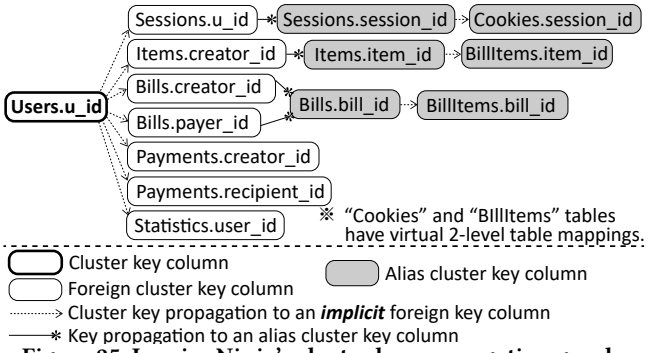


Figure 25: Invoice Ninja's cluster key propagation graph.

*D.6.1 Attack Recovery Scenario 1*A user's bill was maliciously created by an attacker-initiated "Create a Bill" user request. After this user request was identified, Ultraverse retroactively removed it by applying the following optimization technique.

Column-wise Query Dependency: Ultraverse rolled back and replayed only the following transactions: "Add a New Bill", "Delete a Bill", "Add an Item to a Bill", "Modify an Item in a Bill", and "Delete an Item in a Bill". Other transactions such as "Sign Up", "Login", "Log Out", "Reset Password", "Edit My Profile" were not rolled back and replayed because they did not depend on the changed values in the `Bills`, `Items`, or `BillItems` tables.

Row-wise Query Dependency: Ultraverse used `User.u_id` as the cluster key column, so only needed to rollback and replay those transactions that are in the same cluster as the retroactively removed "Create a Bill" transaction.

*D.6.2 Attack Recovery Scenario 2*An attacker tampered with the price of an item by issuing a malicious "Modify an Item in a Bill" user request. We retroactively modified the price in the injected "Modify an Item in a Bill" transaction to a correct value. Ultraverse retroactively updated the database by applying the following optimization technique.

Column-wise Query Dependency: Ultraverse replayed only "Create an Item", "Add an item to a Bill", "Modify an Item in a Bill", and "Delete an Item in a Bill". Other user requests such as "Sign Up", "Login", "Log Out", "Reset Password", "Edit My Profile", "Create a Bill", "Delete a Bill" were not replayed because they didn't depend on the `Items` or `BillItems` tables.

Row-wise Query Dependency: Ultraverse only needed to rollback and replay those transactions that are in the same cluster as the retroactively updated "Modify an Item in a Bill" transaction.

*D.6.3 Attack Recovery Scenario 3*An attacker stole a user's credential and logged into the user's account by issuing a "Login" user request. After this user request was identified, Ultraverse retroactively removed it by applying the following optimization technique.

Row-wise Query Clustering: The `Users.user_id` column was used as the cluster key. When the "Login" user request was removed, Ultraverse only needed to rollback and replay only that user's subsequent requests, while all other users' transactions were skipped who have not had (direct/indirect) interactions with this user, as their queries were in different clusters.

Row-wise Query Dependency: Ultraverse only needed to rollback and replay those transactions that are in the same cluster as the retroactively removed "Login" transaction.

*D.6.4 Attack Recovery Scenario 4*Ultraverse retroactively removed an attacker-initiated "MakePayment" user request by applying the following optimization technique.

Row-wise Query Clustering: Similar to scenario 3's optimization, Ultraverse only needed to rollback and replay that user's and other users' subsequent requests who have had a money flow from this user.

Row-wise Query Dependency: Ultraverse only needed to rollback and replay those transactions that are in the same cluster as the retroactively removed "MakePayment" transaction.

*D.6.5 Attack Recovery Scenario 5*A user finished using the web service by using a public PC and left the seat without logging out. An attacker took the seat and used the service for another hour by using the user's account. After identifying this event via a surveillance camera in the public area, we used Ultraverse to retroactively move the victimized user's "Logout" request to 1 hour earlier when he left the seat. Ultraverse's retroactive operation used the column-wise query dependency analysis and row-wise query clustering, as well as the following optimization technique.

Hash-jump: The attacker's activities only affected the state of the "Sessions" table that exclusively belonged to the user. This table's state was refreshed after the user logged in again next time. As of this point, the user's "Sessions" table's state was the same as before the retroactive operation. Therefore, Ultraverse returned the table's same state stored before the retroactive operation.

E Formal Analysis of Query Dependency and Query Clustering

The formal definition of a retroactive operation is as follows:

Definition 1 (Retroactive Operation). *Let $\mathbb{D}$ a database and $\mathbb{Q}$ a set of all committed queries $Q_1, Q_2, \cdots Q_{|\mathbb{Q}|}$ where the subscript represents the query's index (i.e., commit order). Let $\mathbb{Q}_{\langle i,j \rangle}$ be a subset of $\mathbb{Q}$ that contains from i-th to j-th queries in $\mathbb{Q}$, that is $\{Q_i, Q_{i+1}, \cdots, Q_j\}$ (where $i \leq j$). Let ψ be the last query's commit order in $\mathbb{Q}$ (i.e., $|\mathbb{Q}|$). Let $\mathcal{M} : \mathbb{D}, \mathbb{Q} \rightarrow \mathbb{D}'$ be a function that accepts an input database $\mathbb{D}$ and a set of queries $\mathbb{Q}$, executes queries in $\mathbb{Q}$ in ascending order of query indices, and outputs a resulting database $\mathbb{D}'$. Let $\mathcal{M}^{-1} : \mathbb{D}, \mathbb{Q} \rightarrow \mathbb{D}'$ be a function that accepts an input database $\mathbb{D}$ and a set of queries $\mathbb{Q}$, rolls back queries in $\mathbb{Q}$ in descending order of query indices, and outputs a resulting database $\mathbb{D}'$. Given a database $\mathbb{D}$ and a set of committed queries $\mathbb{Q}$, a retroactive operation with a target query Q'_τ is defined to be a transformation of $\mathbb{D}$ to a new state that matches the one generated by the following procedure:*

(1) Roll back $\mathbb{D}$'s state to commit index $\tau - 1$ by computing $\mathbb{D} := \mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle \tau, \psi \rangle})$.

(2) Depending on the database user's command, do one of the following retroactive operations:

- *In case of retroactively adding Q'_τ, newly execute Q'_τ by computing $\mathbb{D} := \mathcal{M}(\mathbb{D}, Q'_\tau)$, and then replay all subsequent queries by computing $\mathbb{D} := \mathcal{M}(\mathbb{D}, \mathbb{Q}_{\langle \tau, \psi \rangle})$.*
- *In case of retroactively removing Q_τ, skip replaying Q_τ, and replay all subsequent queries by computing $\mathbb{D} := \mathcal{M}(\mathbb{D}, \mathbb{Q}_{\langle \tau+1, \psi \rangle})$.*
- *In case of retroactively changing Q_τ to Q'_τ, newly execute Q'_τ by computing $\mathbb{D} := \mathcal{M}(\mathbb{D}, Q'_\tau)$, and replay all subsequent queries by computing $\mathbb{D} := \mathcal{M}(\mathbb{D}, \mathbb{Q}_{\langle \tau+1, \psi \rangle})$.*

The goal of Ultraverse's query analysis is to reduce the number of queries to be rolled back and replayed for a retroactive operation, while preserving its correctness.

Setup 1 (Ultraverse's Query Analysis).

Input : $\mathbb{D}, \mathbb{Q}, \langle Q'_\tau, add|remove|change \rangle$.
Output : *A subset of $\mathbb{Q}$ to be rolled back and replayed.*

Setup 1 describes the input and output of Ultraverse's query analysis. The input is $\mathbb{D}$ (a database), $\mathbb{Q}$ (a set of all committed queries), Q'_τ (a retroactive target query to be added or changed to), and the type of retroactive operation on Q'_τ (i.e., add, remove, or change it). Note that in case of retroactive removal of the query at the commit index τ, the retroactive target query Q'_τ in the ***Input*** is Q_τ. The output is a subset of $\mathbb{Q}$. Rolling back and replaying the output queries result in a correct retroactive operation.

Ultraverse's query analysis is comprised of two components: query dependency analysis and query clustering analysis. We will first describe query dependency analysis and then extend to query clustering analysis. To show the correctness of performing retroactive operations using query analysis, we first assume that the retroactive operation is either adding or removing a query, and address the case of retroactively changing a query at the end of this section.

E.1 Column-wise Query Dependency Analysis

Terminology 1 (Query Dependency Analysis).

$\mathbb{D}$	: *A given database*
$\mathbb{Q}$	: *A set of all committed queries in $\mathbb{D}$*
Q_n	: *a query with index n in $\mathbb{Q}$*
τ	: *a retroactive target query's index in $\mathbb{Q}$*
Q'_τ	: *The retroactive target query to add or change*
T_x	: *Query* `"CREATE TRIGGER x"`
T_x^{-1}	: *Query* `"DROP TRIGGER x"` *(T_x's counterpart)*
$R(Q_n)$	: *Q_n's read set*
$W(Q_n)$	: *Q_n's write set*
c	: *a table's column*
$Q_n \rightarrow Q_m$	: *Q_n depends on Q_m*
$Q_m \rightsquigarrow Q_n$	: *Q_m is an influencer of Q_n*

Definition 2 (Read/Write Set). *A query Q_i's read set is the set of column(s) that Q_i operates on with read access. Q_i's write set is the set of column(s) that Q_i operates on with write access.*

For each type of SQL statements, its read & write sets are determined according to the policies described in Table A.

Loosely speaking, given $\mathbb{D}$ and $\mathbb{Q}$, we define that Q_i depends on Q_j if some retroactive operation on Q_i *could* change the result of Q_j (i.e., Q_j's return value or the state of the resulting table that Q_j writes to). In this section, when we say query dependency, it always implies column-wise query dependency (discussed in §3.3). We present the formal definition of query dependency in Definition 3.

Definition 3 (Query Dependency). *Given a database $\mathbb{D}$ and a set of all committed queries $\mathbb{Q}$, one query depends on another if they satisfy Proposition 1, 2, 3, or 4.*

Proposition 1. $\exists c((c \in W(Q_m)) \wedge (c \in (R(Q_n) \cup W(Q_n)))) \wedge (m < n) \Longrightarrow Q_n \rightarrow Q_m$

Proposition 1 states that if Q_n reads or writes the table/view's column after Q_m writes to it. Proposition 1 captures the cases where two queries operate on the same column and retroactively adding or removing the prior query could change the column's state that the later query accesses.

Proposition 2. $(Q_n \rightarrow Q_m) \wedge (Q_m \rightarrow Q_l) \Longrightarrow Q_n \rightarrow Q_l$

Proposition 2 states that if Q_n depends on Q_m and Q_m depends O_l, then Q_n also depends on Q_l (transitivity). Proposition 2 captures the cases where two queries, Q_n and Q_l, do not operate on the same column, but there exists some intermediate query Q_m which operates on some same column as each of Q_n and Q_l. In such cases, Q_m acts as a data flow bridge between Q_l's column and Q_n's column, and therefore, a retroactive operation on Q_l could change the column's state that Q_n accesses. Therefore, we regards that Q_n depends on Q_l transitively.

Proposition 3. $(\exists c((c \in W(Q_n)) \wedge (c \in (R(Q_k) \cup W(Q_k))))) \wedge ((((Q_n = T_x) \wedge ((n > k) \vee ((Q_m = T_x^{-1}) \wedge (m > k)))) \vee ((Q_n = T_x^{-1}) \wedge (n > k))) \Longrightarrow Q_k \rightarrow Q_n$

Proposition 4. $(\exists c((c \in W(Q_k)) \wedge (c \in (R(Q_n) \cup W(Q_n))))) \wedge ((((Q_n = T_x) \wedge ((n > k) \vee ((Q_m = T_x^{-1}) \wedge (m > k)))) \vee ((Q_n = T_x^{-1}) \wedge (n > k))) \Longrightarrow Q_n \rightarrow Q_k$

We additionally present Proposition 3 and 4 to handle triggers. At a high level, these two propositions enforce that if a trigger query either depends on or is depended by (i.e., has an incoming or outgoing dependency arrow to) at least one query that depends on the

retroactive target query Q'_τ, then during the replay of the retroactive operation, this trigger is reactivated by replaying its equivalent `CREATE TRIGGER` query. These two propositions conservatively assume that a trigger's conditionally executed body is always executed until the trigger is dropped by its equivalent `DROP TRIGGER` query. Proposition 3 states that if a trigger T_x was alive at the moment Q_k was committed and Q_k reads or writes some column that T_x writes, then Q_k depends on T_x and T_x^{-1}. Proposition 4 covers the reverse of Proposition 3: T_x and T_x^{-1} depends on Q_k if T_x was alive when Q_k was committed and T_x reads or writes a column that Q_k writes.

Definition 4 ($\mathbb{I}$). *$\mathbb{I}$ is an intermediate set of all queries that are selected to be rolled back and replayed for a retroactive target query Q'_τ.*

We define the $\mathbb{I}$ set for three purposes. First, we add the queries dependent on the retroactive target query Q'_τ to $\mathbb{I}$, as candidate queries to be rolled back and replayed. Second, we further add more queries that need to be rolled back and replayed in order to replay *consulted table(s)* (discussed in §3.4). Third, we remove those queries that do not belong to the same cluster as the retroactive target query Q'_τ (discussed in §3.5).

Proposition 5. $(Q_i \rightarrow Q'_\tau) \wedge (W(Q_i) \neq \emptyset) \Longrightarrow Q_i \in \mathbb{I}$.

Proposition 5 states if Q_i depends on the retroactive target query Q'_τ and Q_i's write set is not empty, then Q_i is added to $\mathbb{I}$ (we do not rollback and replay if Q_i's write set is empty, because a read-only query does not change the database's state). Proposition 5 presents our first purpose of using $\mathbb{I}$.

To find the queries to be rolled back and replayed in order to replay *consulted table(s)*, we introduce a new term, *influencer*.

Definition 5 (Influencer). $(Q_i \rightarrow Q'_\tau) \wedge (\exists c, f((c \in (R(Q_i) \cup W(Q_i))) \wedge (f = \text{argmax}_j((j < i) \wedge (c \in W(Q_j)))))) \Longrightarrow Q_f \rightsquigarrow Q_i$

Definition 5 states that if query Q_i depends on the retroactive target query Q'_τ and Q_i *immediately* (back-to-back) depends on Q_f on some column c (i.e., Q_f is the last query that writes to c before Q_i accesses it), then Q_f is defined to be an influencer of Q_i.

Proposition 6. $\exists j, f((Q_j \in \mathbb{I}) \wedge (Q_f \rightsquigarrow Q_j) \wedge (Q_i \rightarrow Q_f)) \Longrightarrow Q_i \in \mathbb{I}$

Proposition 6 states that if Q_i depends on some influencer of some query in $\mathbb{I}$, then Q_i is added to $\mathbb{I}$. Note that Proposition 1, 2, 3, 4, 5, and Proposition 6 repeats to find and add more queries that are required to fully replay all *consulting table(s)* for the retroactive target query Q'_τ.

Once Proposition 1, 2, 3, 4, 5, and 6 complete repetition until no more queries are added to $\mathbb{I}$, query dependency analysis is complete.

Theorem 1. *For a retroactive operation for adding or removing the target query Q'_τ, it is sufficient to do the following: (i) rollback the queries that belong to $\mathbb{I}$; (ii) either execute Q'_τ (in case of retroactively adding Q'_τ) or roll back Q'_τ (in case of retroactively removing Q'_τ); (iii) replay all queries in $\mathbb{I}$.*

Proof. Let the database state after the retroactive operation of adding the target query Q'_τ be $\mathbb{D}' = \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle})$. Let b_i be the i-th oldest query index that satisfies the following: $(b_i > \tau) \wedge (Q_{b_i} \notin \mathbb{I})$. For example, Q_{b_1} is the oldest query in $\mathbb{Q}$ that does not belong to $\mathbb{I}$, and Q_{b_2} is the second-oldest query in $\mathbb{Q}$ that does not belong to $\mathbb{I}$. Note that every query that does not belong to $\mathbb{I}$ also does not depend on any query in $\mathbb{I}$ (otherwise, it should have been put into $\mathbb{I}$ by Proposition 1, 2, 3, 4, 5, and 6). We prove Theorem 1 by finite induction.

Case Q_{b_1}: Let $\mathbb{D}_{b_1} = \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}\}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}\})$, which is equivalent to rolling back all queries in $\mathbb{Q}_{\langle\tau,\psi\rangle}$ except for Q_{b_1}, executing Q'_τ, and replaying all queries in $\mathbb{Q}_{\langle\tau,\psi\rangle}$ except for Q_{b_1}.

Lemma 1. *$\mathbb{D}' = \mathbb{D}_{b_1}$, which is equivalent to:*
$\mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle})$
$= \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}\}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}\})$.

The definition of Q_{b_1} implies that in $\mathbb{Q}_{\langle\tau,\psi\rangle}$, any query committed before Q_{b_1} belongs to $\mathbb{I}$. Proposition 2 and 5 guarantee that Q_{b_1} does not depend on any query in $\mathbb{I}$ (because otherwise, Q_{b_1} would have been put into $\mathbb{I}$). This means that the results of Q_{b_1} (i.e., the resulting state of its write set columns) will be the same as before the retroactive operation. Further, in $\mathbb{Q}$, no query committed before Q_{b_1} depends on the results of Q_{b_1} (because otherwise, there would have been an influencer for such a query, and as both the influencer and Q_{b_1} write to the same column(s), Q_{b_1} would have been put into $\mathbb{I}$ according to Proposition 6). Provided that the results of Q_{b_1} are the same as before the retroactive operation and its results are to be used only by those queries committed after Q_{b_1}, Q_{b_1} needs not be rolled back & replayed while generating $\mathbb{D}'$. Thus, Lemma 1 is true.

Case Q_{b_2}: Let $\mathbb{Q}_{b_1}$ be the set of rolled back & replayed queries for generating $\mathbb{D}_{b_1}$. Let $\mathbb{D}_{b_2} = \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}\}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}\})$, which is equivalent to rolling back all queries in $\mathbb{Q}_{\langle\tau,\psi\rangle}$ except for $\{Q_{b_1}, Q_{b_2}\}$, executing Q'_τ, and replaying all queries in $\mathbb{Q}_{\langle\tau,\psi\rangle}$ except for $\{Q_{b_1}, Q_{b_2}\}$.

Lemma 2. *$\mathbb{D}_{b_1} = \mathbb{D}_{b_2}$, which is equivalent to:*
$\mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}\}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}\})$.
$= \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}\}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}\})$.

The definition of Q_{b_2} implies that in $\mathbb{Q}_{\langle\tau,\psi\rangle}$, any query committed before Q_{b_1} belongs to $\mathbb{I} \cup \{Q_{b_1}\}$. But in case of $\mathbb{D}_{b_1}$, $\mathbb{Q}_{b_1}$ does not contain Q_{b_1}, and thus in $\mathbb{Q}_{b_1}$, any query committed before Q_{b_1} belongs to $\mathbb{I}$. Then, based on the same reasoning used for proving Lemma 1, the results of Q_{b_2} are the same as before the retroactive operation and its results are to be used only by those queries committed after Q_{b_2}. Thus, Q_{b_2} needs not be rolled back & replayed while generating $\mathbb{D}_{b_1}$. Thus, Lemma 2 is true.

Case $Q_{b_{\psi-|\mathbb{I}|}}$: Let $\mathbb{Q}_{b_{\psi-|\mathbb{I}|-1}}$ be the set of rolled back & replayed queries for generating $\mathbb{D}_{b_{\psi-|\mathbb{I}|-1}}$. Let $\mathbb{D}_{b_{\psi-|\mathbb{I}|}} = \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\}), Q'_\tau), \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\})$, which is equivalent to rolling back all queries in $\mathbb{Q}_{\langle\tau,\psi\rangle}$ except for $\{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\}$, executing Q'_τ, and replaying all queries in $\mathbb{Q}_{\langle\tau,\psi\rangle}$ except for $\{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\}$.

Lemma 3. *$\mathbb{D}_{b_{\psi-|\mathbb{I}|-1}} = \mathbb{D}_{b_{\psi-|\mathbb{I}|}}$, which is equivalent to:*

$$\mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|-1}}\}), Q'_\tau),$$
$$\mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|-1}}\})$$
$$= \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\}), Q'_\tau),$$
$$\mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\})$$

The definition of $Q_{b_{\psi-|\mathbb{I}|}}$ implies that in $\mathbb{Q}_{\langle\tau,\psi\rangle}$, any query committed before $Q_{b_{\psi-|\mathbb{I}|}}$ belongs to $\mathbb{I} \cup \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\}$. But in case of $\mathbb{D}_{b_{\psi-|\mathbb{I}|-1}}$, $\mathbb{Q}_{b_{\psi-|\mathbb{I}|-1}}$ does not contain $\{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\}$, and thus in $\mathbb{Q}_{b_{\psi-|\mathbb{I}|-1}}$, any query committed before $Q_{b_{\psi-|\mathbb{I}|}}$ belongs to $\mathbb{I}$. Then, based on the same reasoning used for proving Lemma 1, the results of $Q_{b_{\psi-|\mathbb{I}|}}$ are the same as before the retroactive operation and its results are to be used only by those queries committed after $Q_{b_{\psi-|\mathbb{I}|}}$. Thus, $Q_{b_{\psi-|\mathbb{I}|}}$ needs not be rolled back & replayed while generating $\mathbb{D}_{b_{\psi-|\mathbb{I}|-1}}$. Thus, Lemma 3 is true.

According to Lemma 1, 2 and 3,
$$\mathbb{D}' = \mathbb{D}_{b_1} = \mathbb{D}_{b_2} = \cdots = \mathbb{D}_{b_{\psi-|\mathbb{I}|}}$$
$$= \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{I}), Q'_\tau), \mathbb{I})$$

Therefore, during the retroactive operation of adding the target query Q'_τ, all queries that do not belong to $\mathbb{I}$ need not be rolled back & replayed, and the resulting database's state is still consistent.

If the retroactive operation is removing Q'_τ, then the resulting database's state is $\mathbb{D}' = \mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle}), \mathbb{Q}_{\langle\tau+1,\psi\rangle})$. Then, a similar induction proof used for Lemma 1, 2, 3 can be applied to derive the following:

$$\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle}), \mathbb{Q}_{\langle\tau+1,\psi\rangle})$$
$$= \mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}\}), \mathbb{Q}_{\langle\tau+1,\psi\rangle} - \{Q_{b_1}\})$$
$$= \mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}\}), \mathbb{Q}_{\langle\tau+1,\psi\rangle} - \{Q_{b_1}, Q_{b_2}\})$$
$$\cdots$$
$$= \mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{Q}_{\langle\tau,\psi\rangle} - \{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\}), \mathbb{Q}_{\langle\tau+1,\psi\rangle}$$
$$-\{Q_{b_1}, Q_{b_2}, \cdots Q_{b_{\psi-|\mathbb{I}|}}\})$$
$$= \mathcal{M}(\mathcal{M}^{-1}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{I}), Q'_\tau), \mathbb{I})$$

□

E.2 Row-wise Query Clustering Analysis

Next, we describe query clustering analysis to further reduce queries from $\mathbb{I}$. First, we present additional notations, as described in Terminology 2.

Terminology 2 (Query Clustering Analysis).

ψ	*: The last committed query's index in* $\mathbb{Q}$
C	*: The set of all table columns in* $\mathbb{D}$
$K_c(Q_n)$	*: A cluster set containing* Q_n*'s cluster keys given* c *is chosen as the cluster key column*
$Q_n \leftrightsquigarrow Q_m$	*:* Q_n *and* Q_m *are in the same cluster*

The motivation of defining query clusters is to group queries into disjoint sets such that a retroactive operation on any query in one group does not affect the results of queries in other groups.

Definition 6 (cluster key Column). *A cluster key column is the selected table column in a given database* $\mathbb{D}$*, based on which queries are clustered into disjoint groups.*

Definition 7 (Query Cluster). *A query* Q_i*'s cluster is the set of values or value ranges of the chosen cluster key column* c *that* Q_i *performs read-or-write operation on.*

Proposition 7. *The cluster key column* c_k *is considered to be efficient enough to evenly cluster queries if the following is true:* $c_k := \text{argmin}_{c \in C} \sum_{j=\tau}^{\psi} |K_c(Q_j)|^2$.

Proposition 7 describes how to choose a cluster key column c_k in $\mathbb{D}$ which is efficient enough to evenly cluster queries. Our goal for choosing c_k is not to minimize the number of queries to be rolled back and replayed for every possible scenario of retroactive operation, but to minimize the variance of the number of those queries across all scenarios, based on the observed heuristics applied to the ordered list of committed queries H. $|K_c(Q_j)|$ represents the number of cluster keys that query Q_j is assigned, and its power of 2 adds a penalty if the query has skewedly many cluster keys. In other words, the formula prefers a column where each query belongs to a finer and balanced granularity of clusters.

Now, we describe how to cluster queries.

Proposition 8. $K_{c_k}(Q_n) \cap K_{c_k}(Q_m) \neq \emptyset \implies Q_n \leftrightsquigarrow Q_m$

Proposition 8 states that if two queries have an overlap in their cluster key(s), then the two queries are *merged into* ($\leftrightsquigarrow$) the same cluster. This captures the cases that if two queries access the same tuples, then retroactively adding or removing either query could potentially affect the result of another query.

Proposition 9. $(Q_m \leftrightsquigarrow Q_n) \wedge (Q_n \leftrightsquigarrow Q_o) \implies Q_m \leftrightsquigarrow Q_o$

Proposition 9 states that clustering is transitive. This further captures the cases that even if Q_m and Q_o do not access any same tuples, one could affect another's result if there exists a *bridging* query Q_n between them such that Q_n and Q_m access some same tuple, and Q_n and Q_o access some same tuple. If we repeat transitive clustering until no more clusters are merged together, the final output is a set of mutually disjoint *final* clusters such that queries in different clusters access mutually disjoint set of data tuples.

Theorem 2. *For a retroactive operation for adding or removing the target query* Q'_τ*, it is sufficient to do the following: (i) rollback the queries that belong to* $\mathbb{I}$ *and are co-clustered with* Q'_τ*; (ii) either execute* Q'_τ *(in case of retroactively adding* Q'_τ*) or roll back* Q'_τ *(in case of retroactively removing* Q'_τ*); (iii) replay all queries that belong to* $\mathbb{I}$ *and is co-clustered with* Q'_τ*.*

Proof. In the proof of Theorem 1, we showed that given database $\mathbb{D}$ and committed queries $\mathbb{Q}$, a retroactive operation for adding or removing Q'_τ does not need to roll back and replay all queries in $\mathbb{Q}$, but only those in $\mathbb{I}$. We further break down $\mathbb{I}$ into $\mathbb{I}_K$ and $\mathbb{I}_\emptyset$, where $\mathbb{I}_K = \{Q_i | Q_i \leftrightsquigarrow Q'_\tau\}$, and $\mathbb{I}_\emptyset = \{Q_i | Q_i \not\leftrightsquigarrow Q'_\tau\}$.

Our proof leverages the commutativity rule [64]:

(1) If two transactions are *read-then-read*, *write-then-read*, or *write-then-write* operations on mutually non-overlapping objects, those two transactions are defined to be *non-conflicting*.
(2) If two *non-conflicting* transactions are consecutive to each other, then swapping their execution order does not change the resulting database's state.

Note that each query in $\mathbb{I}_\emptyset$ is *non-conflicting* with all queries in $\mathbb{I}_K$.

Suppose the retroactive operation is to add the retroactive target query Q'_τ. Then, the retroactive operation's result is

$$\mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{I}), Q'_\tau), \mathbb{I})$$
$$= \mathcal{M}(\mathcal{M}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{I}_K), Q'_\tau), \mathbb{I}_K).$$

This is because the commutativity rule allows all queries in $\mathbb{I}_\emptyset$ to be moved to before Q'_τ was committed in the query execution history with harming the consistency of the resulting database, and thus need not be rolled back & replayed.

Suppose the retroactive operation is to remove the retroactive target query Q'_τ. Then, the retroactive operation's result is

$\mathcal{M}(\mathcal{M}^{-1}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{I}), Q'_\tau\}), \mathbb{I})$

$= \mathcal{M}(\mathcal{M}^{-1}(\mathcal{M}^{-1}(\mathbb{D}, \mathbb{I}_K), Q'_\tau), \mathbb{I}_K)$

Therefore, for a retroactive operation, it is sufficient to do the following: *(i)* rollback the queries that belong to $\mathbb{I}_K$; *(ii)* either execute Q'_τ (in case of retroactively adding Q'_τ) or roll back Q'_τ (in case of retroactively removing Q'_τ); *(iii)* replay all queries that belong to $\mathbb{I}_K$. □

We can extend the query analysis to support multiple retroactive target queries. In this case, we run Setup 1 for each retroactive target query, and then take a union of all outputs, which is the final set of queries to rollback & replay. In case of retroactively changing a target query, the desired database state after the retroactive operation is the same as running Setup 1 twice: retroactively removing Q'_τ and then retroactively adding a changed target query Q'_τ. Thus, the retroactive changing is equivalent to performing the two retroactive operations. Because performing retroactive operations using the query analysis is correct for both of them, we have the correctness of retroactively changing a query using the query analysis.

F Collision Rate of Ultraverse's Table Hashes

Ultraverse computes a table's hash by hashing its each row with a collision-resistant hash function and summing them up. By assuming that the collision-resistant hash function's output is uniformly distributed in $[0, p-1]$, we will prove that given two tables T_1 and T_2, Ultraverse's Hash-jumper's hash collision rate is upper-bounded by $\frac{1}{p}$ (i.e., with a probability no more than $\frac{1}{p}$ producing the same hash value for T_1 and T_2, when $T_2 \neq T_1$).

Suppose the Hash-jumper outputs a hash value $h \in [0, p-1]$ for T_1. Without loss of generality, we assume T_2 has n rows. Then we prove by induction.

Case n = 1: Because the collision-resistant hash function's output is uniformly distributed in $[0, p-1]$, it is easy to see that the probability that the Hash-jumper outputs h for any T_2 is $\frac{1}{p}$.

Case n = k: For each row of T_2, let x_i denote the collision-resistant hash function's output. Then there are p^k possibilities of $(x_1, x_2, ..., x_k)$ for the k rows. Because the collision-resistant hash function's output is uniformly distributed in $[0, p-1]$, all these possibilities have the same probability $\frac{1}{p^k}$. Consider the output of the Hash-jumper. For any $h_k \in [0, p-1]$, we assume there are p^{k-1} possibilities of $(x_1, x_2, ..., x_k)$ such that $\sum_i^k x_i = h_k \bmod p$. This holds for $k = 1$, as we have seen for **Case n = 1**.

Case n = k + 1: There are p^{k+1} possibilities of $(x_1, x_2, ..., x_{k+1})$ for the $(k+1)$ rows. Because the collision-resistant hash function's output is uniformly distributed in $[0, p-1]$, all these possibilities have the same probability $\frac{1}{p^{k+1}}$. For any $(x_1, x_2, ..., x_k)$ such that $\sum_i^k x_i = h_k \bmod p$, there exists exactly one x_{k+1} such that $h_k + x_{k+1} = h \bmod p$. By the assumption in **Case n = k**, for each $h_k \in [0, p-1]$, there are exactly p^{k-1} possibilities of $(x_1, x_2, ..., x_{k+1})$ such that the output of the Hash-jumper is h. Because there are p possible h_k values in $[0, p-1]$, there are p^k possibilities of $(x_1, x_2, ..., x_{k+1})$ such that the output of the Hash-jumper is h. Therefore, the probability that that the Hash-jumper outputs h for a table T_2 of $(k+1)$ rows is $\frac{p^k}{p^{k+1}} = \frac{1}{p}$.

Because the above probability is independent of the number of rows n, for any T_2, the Hash-jumper output h with a probability of $\frac{1}{p}$. Therefore, the hash collision rate is upper-bounded by $\frac{1}{p}$ when $T_2 \neq T_1$.

False Positives: From the security perspective, there is yet a non-zero chance that a malicious user fabricates two row hash values (x'_1, x'_2) such that $(x'_1 + x'_2) \bmod p = (x_1 + x_2) \bmod p$ and tricks the database server into believing in a false positive on a table hash hit. To address this, whenever a table hash hit is found, Hash-jumper optionally makes a literal table comparison between two table versions at the same commit time (one evolved during the replay; the other newly rolled back from the original database to this same point in time) and verifies if they really match. If the literal table comparison returns a true positive before finishing to replay rest of the queries, Ultraverse still ends up reducing its replay time.

False Negatives: A subtlety occurs when a query uses the `LIMIT` keyword without `ORDER BY`, because each replay of this query may return different row(s) of a table in a non-deterministic manner,

which may lead Hash-jumper to making a false negative decision and missing the opportunity of a legit hash-jump. However, note that missing the opportunity of optimization does not affect the correctness of retroactive operations.