Shortcut: Making MPC-based Collaborative Analytics Efficient on Dynamic Databases

Peizhao Zhou
zhoupz@mail.nankai.edu.cn
CS, DISSec, Nankai University
Tianjin, China

Xiaojie Guo
xiaojie.guo@mail.nankai.edu.cn
CS, DISSec, Nankai University
Tianjin, China
Shanghai Qi Zhi Institute
Shanghai, China

Pinzhi Chen
chenpinzhi@mail.nankai.edu.cn
CS, DISSec, Nankai University
Tianjin, China

Tong Li
tongli@nankai.edu.cn
CS, DISSec, Nankai University
Tianjin, China

Siyi Lv*
lvsiyi@nankai.edu.cn
CS, DISSec, Nankai University
Tianjin, China

Zheli Liu
liuzheli@nankai.edu.cn
CS, DISSec, Nankai University
Tianjin, China

Abstract

Secure Multi-party Computation (MPC) provides a promising solution for privacy-preserving multi-source data analytics. However, existing MPC-based collaborative analytics systems (MCASs) have unsatisfying performance for scenarios with dynamic databases. Naïvely running an MCAS on a dynamic database would lead to significant redundant costs and raise performance concerns, due to the substantial duplicate contents between the pre-updating and post-updating databases.

In this paper, we propose Shortcut, a framework that can work with MCASs to enable efficient queries on dynamic databases that support data insertion, deletion, and update. The core idea of Shortcut is to materialize previous query results and directly update them via our query result update (QRU) protocol to obtain current query results. We customize several efficient QRU protocols for common SQL operators, including Order-by-Limit, Group-by-Aggregate, Distinct, Join, Select, and Global Aggregate. These protocols are composable to implement a wide range of query functions. In particular, we propose two constant-round protocols to support data insertion and deletion. These protocols can serve as important building blocks of other protocols and are of independent interest. They address the problem of securely inserting/deleting a row into/from an ordered table while keeping the order. Our experiments show that Shortcut outperforms naïve MCASs for minor updates arriving in time, which captures the need of many realistic applications (e.g., insurance services, account data management). For example, for a single query after an insertion, Shortcut achieves up to 186.8× improvement over those naïve MCASs without our QRU protocols on a dynamic database with $2^{16} \sim 2^{20}$ rows, which is common in real-life applications.

*Siyi Lv is the corresponding author of this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690314

CCS Concepts

• **Security and privacy → Privacy-preserving protocols**.

Keywords

Secure multi-party computation; database analytics; data update

ACM Reference Format:
Peizhao Zhou, Xiaojie Guo, Pinzhi Chen, Tong Li, Siyi Lv, and Zheli Liu. 2024. Shortcut: Making MPC-based Collaborative Analytics Efficient on Dynamic Databases. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3658644.3690314

1 Introduction

Analyzing data collected from multiple sources can benefit or even spawn various applications, such as medical studies [8], credit investigation [33], account data managements [36], and commercial supports [1]. However, many data owners are reluctant to disclose raw data to others, due to privacy concerns [30]. Systems, that perform collaborative analytics on data contributed by mutually distrustful data owners without sacrificing their privacy, have attracted much attention from both academic and industrial communities.

Secure Multi-party Computation (MPC) [17] provides a promising solution for this topic. It enables multiple parties to securely evaluate a function together with cryptographic guarantee. A standard MPC-based Collaborative Analytics System (MCAS) (e.g., [8, 19, 22, 33, 39]) can consist of several *secure SQL protocols* that evaluate SQL queries on a database contributed by multiple data owners. Since there is a notable performance gap between MPC-based systems and plaintext systems, known MCASs attempt to mitigate this gap by considering various optimizations or trade-offs. For example, MPC costs can be reduced by using local computation as much as possible [8, 28, 31], reordering operators as per rules [22], optimizing secure operators [5, 19], or allowing more information leakage that is well-bounded by differential privacy (DP) [9, 10].

Table 1: Asymptotic communication per query after an update in existing secret-sharing-based MCASs and Shortcut. Here, n is the table size, $l \le h \le n$, where l is the limit size of Order-by-Limit. ℓ denotes the size of the secret shares.

Methods		Order-by-Limit		Group-by-Aggregate[1]		Join[2]		Select	
		Comm. (bits)	Rounds	Comm. (bits)	Rounds	Comm. (bits)	Rounds	Comm. (bits)	Rounds
Secrecy [22]		$O(\ell n \log^2 n)$	$O(\log \ell \log^2 n)$	$O(\ell n \log^2 n)$	$O(\log \ell \log^2 n)$	–	–	$O(\ell n)$	$O(\log \ell)$
AHK⁺ [5][3]		$O(\ell \lambda n)$	$O(\lambda)$	$O(\ell \lambda n)$	$O(\lambda)$	$O(\ell \lambda n)$	$O(\lambda)$	–	–
Scape [19]		$O(\ell n \log^2 n)$	$O(\log \ell \log^2 n)$	$O(\ell n \log^2 n)$	$O(\log \ell \log^2 n)$	$O(\ell n \log n)$	$O(\log \ell \log n)$	$O(\ell n)$	$O(\log \ell)$
Shortcut	Insert	$O(\ell h)$	$O(\log \ell)$	$O(\ell n)$	$O(\log \ell)$	$O(\ell n)$	$O(\log \ell)$	$O(\ell)$	$O(\log \ell)$
	Delete	$O(\ell h)$	$O(\log \ell)$	$O(\ell n)$	$O(\log \ell)$	$O(\ell n)$	$O(\log \ell)$	$O(\ell n)$	$O(\log \ell)$

[1] The cost of Distinct is consistent with that of Group-by-Aggregate.
[2] Shortcut considers unique-key Join, while Secrecy considers general Join. Hence, we omit their comparison of Join.
[3] λ denotes the size of the ordered, joined, or grouped keys.

Table 2: Asymptotic number of gates per query after an update in existing garbled-circuit-based MCASs and Shortcut. Here, n is the table size, $l \le h \le n$, where l is the limit size of Order-by-Limit. We treat the size of secret shares as constant.

Methods		Number of Gates			
		Order-by-Limit	Group-by-Aggregate[1]	Join[2]	Select
SMCQL [8]		$O(n \log^2 n)$	$O(n \log^2 n)$	–	$O(n)$
Senate [28]		$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$
Shortcut	Insert	$O(h)$	$O(n)$	$O(n)$	$O(1)$
	Delete	$O(h)$	$O(n)$	$O(n)$	$O(n)$

[1] The cost of Distinct is consistent with that of Group-by-Aggregate.
[2] We omit the comparison of Join with SMCQL since it also considers general Join.

However, these attempts are still far from efficient in scenarios with dynamic databases. For example, consider the scenario of *Password Reuse* [22, 28, 36], where many users may reuse passwords across different sites. If one of these sites is hacked, the attacker could compromise the accounts of these users on other sites. As pointed out by [28, 36], sites can arrange for salted hashes of passwords where the salts are deterministically computed from user identifiers, and regularly perform the following query to identify the users with the same password across different sites:

```
SELECT userid FROM R₁ ∪...∪ Rₘ
GROUP BY userid, password
HAVING COUNT(*)>1
```

In real life, the account databases are dynamic: a user can sign up or delete its account at will. So, the regular queries should be launched from scratch each time, leading to heavy MPC overhead.

In fact, there are substantial duplicate contents between the pre-updating and post-updating databases. An intuition to save MPC overhead is to reuse and update the previous query results to obtain the result for the current query. Using this intuition, IncShrink [35] provides a solution to an incremental database, which only allows data insertion. It materializes the results of Join-Select queries and proposes an algorithm to update query results with DP guarantee. However, IncShrink only considers Join-Select queries after data insertion, and its DP guarantee aims to protect the actual size of materialized results (not the actual update manners for databases since only data insertion is allowed therein).

In this paper, we further explore the power of the above intuition and propose Shortcut, an MCAS framework that improves the efficiency of SQL queries on dynamic databases with data insertion, deletion, and update. Our contributions are as follows.

- To the best of our knowledge, Shortcut is the first framework with the following characteristics: (i) it supports dynamic databases with data insertion, deletion, and update, (ii) it enables a wide range of queries, including Order-by-Limit, Group-by-Aggregate, Distinct, Join, Select, Global Aggregate, and their compositions, (iii) it is compatible with existing MPC frameworks [3, 21, 38] (along with the MCASs [22, 28] atop them) to support standard semi-honest and malicious security, and (iv) it can protect actual update manners in an oblivious sense (i.e., the adversary learns nothing about how a dynamic database changes each time).
  To implement Shortcut, we propose several operator-level query result update (QRU) protocols dedicated to dynamic databases. These QRU protocols are constructed for common SQL operators, including Order-by-Limit, Group-by-Aggregate, Distinct, Join, Select, and Global Aggregate.
- We present a constant-round inserting protocol and a constant-round push-down protocol. That is, in contrast to the naïve solution (i.e., the protocol based on bubbling circuits) whose round complexity is linear in table size, the round complexities of our protocols are independent of table size. These protocols are used as the key building blocks of our QRU protocol for Order-by-Limit. Meanwhile, their communication complexities are still linear in table size.
  Notably, our constant-round inserting protocol resolves the problem of securely inserting a new item into an ordered sequence or table while maintaining the order. It is useful and may be of independent interest. For example, in the secure binary search protocol [11] over an incremental dataset, a constant-round inserting protocol can efficiently insert a new record into the sorted dataset, avoiding a full-fledged secure sorting.
- We study the composition between our QRU protocols. This composition is necessary to combine these protocols to implement and accelerate complex SQL queries, which are composed of the above common SQL operators, over dynamic databases. Our composition results complete the picture of these QRU protocols.
- We implement Shortcut and compare it with Secrecy [22] and AHK⁺ [5], the state-of-the-art works also with characteristics (ii) and (iii) on static databases, on several SQL operators and real-life composite SQL queries. Our experiments show that Shortcut outperforms these works, indicating the effectiveness of our QRU protocols. For example, for operator Join, Shortcut achieves 142.4× ~ 186.8× (resp. 14.5× ~ 21×) improvement over Secrecy (resp. AHK⁺) on a dynamic database containing $2^{16} \sim 2^{20}$ rows,

after an insertion. For real-life queries, Shortcut is 236.3× (resp. 26.5×) faster than Secrecy (resp. AHK$^+$) over a 2^{18}-row dynamic database for Credit Scores queries [28] after an insertion.

We compare Shortcut with several standard MCASs. Over dynamic databases, this comparison is essentially between existing secure SQL protocols and the QRU protocols in Shortcut. Secrecy, AHK$^+$, and Scape [19] employ secret-sharing-based MPC schemes. In Table 1, we summarize their asymptotic communication per query after an insertion/deletion. Note that, the cost of an update equals that of an insertion plus a deletion. The communications of Secrecy, AHK$^+$, and Scape for Order-by-Limit, Group-by-Aggregate, and Join are superlinear in table size n, while the communication of Shortcut is linear in n. Moreover, the rounds of all operators in Shortcut are independent of n. In contrast, SMCQL [8] and Senate [28] employ garbled-circuit-based MPC schemes. In Table 2, we summarize their asymptotic number of gates per query after an insertion/deletion. Our improvements over these works are similar to those over Secrecy and Scape.

2 Technical Overview

We focus on MCASs that securely compute SQL queries over dynamic databases with data insertion, deletion, and update. In Shortcut, a data update is directly implemented by a data deletion followed by a data insertion. In a real-life application where a dynamic database varies slightly over time, there are substantial duplicate contents between the pre-updating database and the post-updating one. Intuitively, if every SQL query requires MPC from scratch, these duplicate contents can result in significant redundant MPC overhead and raise performance issues. The core idea of this work is to materialize previous query results, i.e., the tables returned by MPC-based SQL queries, and then directly update them to compute the result for the current query, rather than running MPC protocols on the entire dynamic database. We realize this idea by proposing a series of QRU protocols that update the materialized tables based on the update messages (which consist of data rows and update manners) from data owners. We consider two challenges in the design of QRU protocols: (i) concretely efficient QRU protocols customized for common SQL operators, and (ii) the composition of QRU protocols to support composite SQL queries over dynamic databases. In this section, we first introduce the Shortcut workflow and then outline how we address the above two challenges.

2.1 Shortcut Workflow

Similar to an existing MCAS, Shortcut has three types of roles: (i) data owners, who upload databases and issue update messages, (ii) computing parties, who run MPC protocols, and (iii) analysts, who want to obtain query results. To perform a query, an analyst sends its query to all computing parties, who parse this query as a schedule. Using the databases uploaded by data owners via MPC protocols, these computing parties run MPC protocols to execute the schedule to compute the result of the query. If there are already materialized results in the memories of the computing parties, this execution also takes these materialized results as input. To update an uploaded database, its data owner issues an update message to the computing parties, who invoke the QRU protocols of Shortcut to update related materialized results as per this message. All query results can be revealed to analysts via MPC protocols.

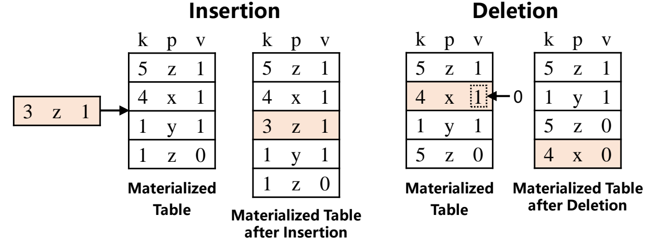


Figure 1: QRU protocols for Order-by-Limit.

2.2 QRU Protocols for Common SQL Operators

Order-by-Limit. On input a table, the operator Order-by-Limit orders this table by a specified column and returns several rows at the front [28]. The QRU protocol of this operator is to incorporate an update message of the given table into a previous query result (also called "materialized table") to make it consistent with the updated table. We consider two fundamental types of update messages: insertion and deletion. We take for example a materialized table with four ordered rows, which is sorted first in the descending order of *validity bit* v and then in the descending order of *order key* k. An insertion example is presented in Figure 1, where we want to insert a new row $(3, z, 1)$ to the underlying table and maintain the materialized table. Meanwhile, a deletion example is presented in Figure 1, where we want to delete an existing row $(4, x, 1)$ from the underlying table and maintain the materialized table. In particular, to prevent information leakage, deletion needs to set bit $v = 1$ of the deleted row to 0, through a linear scan. In the following, we introduce how to maintain the materialized table under its associated Order-by-Limit execution.

In previous works, the order of the materialized table can be maintained via bubbling circuits that require a linear scan of this table. Inserting a new row into the materialized table can be done by a bottom-up bubbling while pushing a deleted dummy row (i.e., that newly with $v = 0$) down in deletion can be done by a top-down bubbling. However, such bubbling circuits suffer from an apparent shortcoming: their depths are linear in the size of the materialized table, yielding protocols with round complexity linear in this size when secret-sharing-based MPC paradigms are used.

We deal with this shortcoming by proposing a constant-depth inserting circuit (CDIC) and a constant-depth push-down circuit (CDPC) for insertion and deletion, respectively. By replacing the above bubbling circuits with CDIC and CDPC, we obtain constant-round protocols to maintain the materialized table. Both CDIC and CDPC consist of a masking stage and a rewriting stage.

For CDIC, (i) in the masking stage, a new row is appended at the end of the materialized table, and the rows whose orders are lower than the new row are all replaced by it, and (ii) in the rewriting stage, the previously replaced rows are rewritten back to their original positions shifted down by one. For CDPC, (i) in the masking stage, the deleted dummy row (if exists) and the rows below it are marked as dummies, and (ii) in the rewriting stage, the previously marked rows, except the deleted dummy row, are rewritten back to their original positions shifted up by one. More details about the two

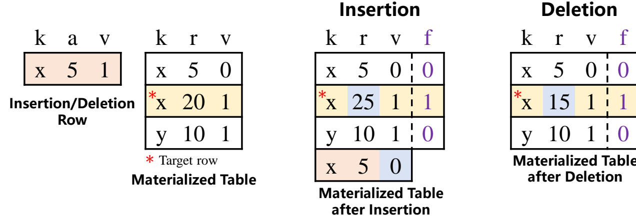


Figure 2: QRU protocols for Group-by-Sum.

constant-depth circuits and the QRU protocol of Order-by-Limit are presented in Section 5.1.

Group-by-Aggregate. On input a table, the operator Group-by-Aggregate [22] groups this table by a specified column (called *group key*), aggregates the values of a specified column (called *aggregate attribute*) within each group, and stores these values in a new column (called *result attribute*) to obtain a materialized table. In the following, we provide an overview of the QRU protocol of Group-by-Sum. This QRU protocol can be slightly modified to obtain the QRU protocols of Group-by-Count/Max/Min and Distinct. Figure 2 illustrates the QRU protocol of Group-by-Sum, where k is the group key, a is the aggregate attribute, r is the result attribute, v is the validity bit, and f is the flag bit.

Let a *target row* in the materialized table denote such a row that (i) has the same group key as that in a valid row to be inserted/deleted, and (ii) has validity bit $v = 1$. In insertion, there are two possible cases: (i) Case 1: if a target row exists, then the value a of the inserted row should be added to the value r of the target row, or (ii) Case 2: if no target row exists, then this inserted row forms a new group in the materialized table. More specifically, the insertion is as follows. First, we compute a flag bit f for each row to locate the target row (i.e., $f = 1$ if and only if for the target row). Then, we append the inserted row to the end of the materialized table. At this time, if Case 1 holds, we obliviously set $v = 0$ in the appended row; otherwise, we keep its v unchanged. Finally, we add the value of a of the appended row to the value of r of the target row indexed by a flag bit $f = 1$ (if exists). Insertion under Case 1 is presented in Figure 2. Meanwhile, deletion is straightforward according to Figure 2, where we locate the target row and subtract the value of a of the deleted row from the value of r of the target row.

Join. Shortcut considers unique-key Join in [8, 28]. On input two table L and R, the join result between L and R on *join key* k is $J = L \bowtie_k R = \{(k, t_L[\neg k], t_R[\neg k]) \mid t_L \in L, t_R \in R, t_L[k] = t_R[k]\}$, where $t[\neg k]$ denotes the row without attribute k. We note that each value of k is unique in each table to identify rows. Shortcut materializes not only query result J but also two inputs L and R.

Consider the QRU protocol for inserting a new row to table L (symmetric to table R). First, we append the inserted row to the end of L, which is a part of the materialized table. Then, we join the inserted row with the row having the same key in table R via a linear scan. Finally, the joined row is appended to the end of J. For the QRU protocol of deleting a row from table L (symmetric to table R), we linearly scan L (resp. J) to set the validity bit to 0 for the deleted row (resp. the joined row w.r.t. the deleted row).

Select. Given an input table R and a predicate $\sigma(\cdot)$, the operator Select [22] computes a validity bit $t[v] = \sigma(t)$ for each row $t \in R$. In the QRU protocol for insertion, the materialized table can be updated by filtering the inserted row with predicate $\sigma(\cdot)$ and then appending the result to the end of the materialized table. In the QRU protocol for deletion, we linearly scan the materialized table to set the validity bit to 0 for the deleted row.

Global Aggregate. On input a table, the operator Global Aggregate [22] aggregates all values of a specified column and returns an aggregated value. Shortcut materializes this aggregated value. For Sum/Count/Max/Min-based aggregation, the QRU protocol for inserting a new row into the underlying table can be realized by additionally aggregating the value in this row into the materialized aggregated value. For Sum/Count-based aggregation, the QRU protocol for deleting a row from the underlying table is to subtract the value in this row from the materialized aggregated value. For Max/Min-based aggregation, we use existing secure SQL protocols to query the underlying table from scratch after deletion.

2.3 Composition of QRU Protocols

In the composition of two QRU protocols, the former protocol can compute an output update message from its associated SQL operator and an input update message. Then, the latter protocol can compute the latest materialized table as per its associated SQL operator and the update message computed from the former protocol. In essence, the output update message of the former protocol describes the difference between the materialized tables before and after the protocol execution. Note that this composition depends on the two SQL operators underlying the two QRU protocols, respectively. We refer readers to Section 4 for the formalized composition rule.

3 Preliminaries

3.1 Notations and Definitions

Given table R and $i \geq 0$, let R_i denote its i-th row, $R[a]$ denote the column of attribute a, and $R_i[a]$ denote the value of attribute a in i-th row. Given row u, let $u[a]$ denote the value of attribute a in row u. Throughout this paper, let $v \in \{0, 1\}$ denote the validity attribute indicating whether a row is valid or not: this row is **valid** if $v = 1$ or **dummy** if $v = 0$. If an attribute column c contains Boolean values, let $\bar{c}$ denote the attribute column that flips each bit of c. Given table R, row t, and element x, we write $R \cup t$ for appending t to the end of table R and $t \cup x$ for appending x to the end of row t. Let κ denote the computational security parameter and s denote the statistical security parameter. We write $[m, n]$ for finite set $\{m, \ldots, n\}$, where $m, n \in \mathbb{Z}$ and $m \leq n$. Given permutation π, we write $\pi(R)$ for applying permutation π to the rows of table R. Two dummy rows a, b are regarded as identical, also denoted by $a = b$ as two valid rows. We also say that two n-row tables A, B are identical, denoted by $A = B$, if $A_i = B_i$ for each $i \in [0, n-1]$. Given table R, let $\mathsf{Compact}(R)$ denote its compaction that removes all dummy rows in R but preserves the order of other rows.

Definition 3.1 (Equivalence of two tables under a query)**.** *Depending on whether query q returns an ordered result or not, the equivalence of two tables R and T under q is defined as follows:*

- *If the result is ordered, the equivalence holds if and only if $R = T$.*
- *Otherwise, the equivalence holds if and only if there exists a permutation π such that* $\mathsf{Compact}(R) = \pi(\mathsf{Compact}(T))$.

Let $R \sim_q T$ denote this equivalence. When it is clear in the context whether the result of q is ordered or not, we simply write $R \sim T$.

This equivalence captures that, if two equivalent tables R and T are returned by the same query whose output is ordered, they are identical row-by-row (where dummy rows are identical to each other). Otherwise, it is sufficient to consider that all valid rows in table R are exactly the valid rows in table T, regardless of their order in either table. In this work, only Order-by-Limit queries will produce ordered outputs. Thus, we mostly use the notation in its simplified form, i.e., $R \sim T$, for ease of notation.

Update message. We define an update message of data owners as $\mathsf{um} : (row, manner)$, where row is an data row and $manner \in \{insert, delete\}$ represents its update manner. In this work, a data update is directly implemented by a data deletion plus a data insertion. We write ums for a queue of update messages.

Plain update. We define PlainUpdate(um, R), a macro that means plainly updating table R using update message um. If um.$manner$ is $insert$, um.row is directly appended to R; if um.$manner$ is $delete$, um.row within R (if exists) is set as dummy.

3.2 Security Model and Guarantees

Security model. We rely on black-box computations of several common circuit evaluation protocols. Therefore, we can support semi-honest/malicious security with an honest/dishonest majority, as long as the underlying MPC protocol allows. A semi-honest adversary can see all the internal states of the corrupted parties but without altering its protocol execution. In contrast, a malicious adversary, in addition to observing the internal states of the corrupted parties, may arbitrarily deviate from the protocol. Typically, we follow the security model of the MCAS that we work with.

Security guarantees. We treat the data schema and query statements as public. It relies on the underlying MPC protocol to protect data throughout the entire lifecycle. The MPC protocol provides two types of guarantees: (i) privacy, meaning that computing parties do not learn anything about the data, and (ii) correctness, meaning that all participants are convinced that the computation output is accurate. The adversary cannot learn anything beyond the size of the input data (which can also be padded by the data owners). Only the designated analyst learns the result of the query.

3.3 Background on MPC

General-purpose MPC [24, 40] supports Boolean and arithmetic operations on encrypted data, or more precisely, shared data. We use $\langle x \rangle$ to uniformly denote data x shared by a general-purpose MPC scheme. In our instantiation, we consider a three-party honest-majority scheme, i.e. at most one corrupted party is allowed, similar to several state-of-the-art MCASs [5, 19, 22]. In particular, we use the semi-honest three-party replicated secret sharing scheme [3] for our instantiation.

Replicated secret sharing. This scheme consists of three parties: P_1, P_2, P_3. A private data $x \in \{0,1\}^\ell$ is split into three random elements $x_1, x_2, x_3 \in \{0,1\}^\ell$, and P_1's share is (x_1, x_2), P_2's share is (x_2, x_3), P_3's share is (x_3, x_1). There are two sharing types with different constraints: arithmetic sharing with $x = x_1 + x_2 + x_3 \mod 2^\ell$, and Boolean sharing with $x = x_1 \oplus x_2 \oplus x_3$ for $x \in \mathbb{Z}_2^\ell$. Any two parties P_i and P_j can reconstruct x from their shares, while any single party learns nothing about x. Conversions between arithmetic share and Boolean share are done by evaluating a ripple-carry adder [2, 24].

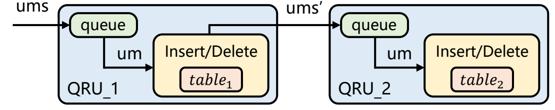


Figure 3: Workflow of QRU protocols.

Circuit evaluation. Given arithmetic sharing data $\langle x \rangle$ and $\langle y \rangle$, parties evaluate the addition circuit (denoted as $\langle x \rangle + \langle y \rangle$) locally. A multiplication circuit (denoted as $\langle x \rangle \cdot \langle y \rangle$) needs one communication round. The equality circuit (=?) and comparison circuit ($<, >, \leq, \geq$) between $\langle x \rangle$ and $\langle y \rangle$ are done on Boolean sharing, and need $O(\ell)$-bit communication in $O(\log \ell)$ rounds for data defined in $\mathbb{Z}_2^\ell$.

Multiplexer. Let $\langle x \rangle$ and $\langle y \rangle$ be input, $\langle s \rangle$ be output, we implement the multiplexer (MUX) as $\langle s \rangle = \langle y \rangle + \langle b \rangle \cdot (\langle x \rangle - \langle y \rangle)$, where $\langle b \rangle$ is the select bit. Specially, minimum circuit is implemented as $\min(\langle x \rangle, \langle y \rangle) = \langle y \rangle + (\langle x \rangle < \langle y \rangle) \cdot (\langle x \rangle - \langle y \rangle)$, and maximum circuit is implemented as $\max(\langle x \rangle, \langle y \rangle) = \langle x \rangle + (\langle x \rangle < \langle y \rangle) \cdot (\langle y \rangle - \langle x \rangle)$.

4 Composition of QRU Protocols

4.1 Single QRU Protocol

The core of MCASs is their secure SQL protocols used to securely evaluate SQL operators. SHORTCUT's QRU protocol aims to directly update the output of the secure SQL protocol, achieving the same goal as using the secure SQL protocol from scratch, i.e., plainly updates the input table first and then invokes secure SQL protocol.

Let SSQL($\cdot$) denote a secure SQL protocol, $outTable$ = SSQL($inTable$), where $inTable$ and $outTable$ represent the input and output table of the protocol respectively. For any update messages ums, let

$$inTable^u = \text{PlainUpdate}(\text{ums}, inTable)$$

$$outTable^u = \text{SSQL}(inTable^u)$$

where superscript "u" indicates that the table has been updated by ums. Then, the associated QRU protocol satisfies:

$$\text{QRU}(\text{ums}, outTable) \sim outTable^u = \text{SSQL}(\text{PlainUpdate}(\text{ums}, inTable)) \quad (1)$$

4.2 Composition Rule

When data owners initiate data updates, SHORTCUT recursively invokes QRU protocols along the route map of the directed acyclic graph to update the materialized tables. If there is more than one SQL operator in the directed acyclic graph, their associated QRU protocols must be composited. Taking an example of a directed acyclic graph with two operators connected end-to-end, Figure 3 shows the workflow of the QRU protocols. $table_i$ is the output table, i.e., the materialized table, of i-th operator. QRU_1($\cdot$) and QRU_2($\cdot$) are the QRU protocols of the first and the second operators respectively. The QRU protocol pops the update message from the queue, and invokes the associated insert protocol or delete protocol based on the update manner to update materialized tables. Note that, update messages uploaded by data owners are semantically targeted towards the underlying databases, i.e., the input tables of the first

operator here. Therefore, the input update messages for the first protocol are those uploaded by data owners, while the input update messages for the remaining protocols should be provided by their former protocol. This means that each QRU protocol, except for the final one, not only updates the materialized tables but also computes the output update messages offered to its next protocol. Informally speaking, we let each protocol inform its next protocol through the output update messages about what changes the current update has made to the input table of the next protocol. Thereby, enabling the composition of any QRU protocols. In addition, the concrete efficiency of QRU protocols can be optimized through a *pipelined execution* when multiple update messages are waiting in the queue.

Throughout this paper, we uniformly denote the output update messages as **ums′**. We introduce the property of ums′ and further show the correctness of composited QRU protocols below.

The correctness of composited QRU protocols. Before discussing the correctness of composited QRU protocols, we provide an important observation of the secure SQL protocol.

Observation 4.1. *The correctness of secure SQL protocol is independent of the row order of its input table. In other words, inputting two equivalent tables under a query into a secure SQL protocol can result in the equivalent output tables. For any table ∼ table′, we have*

$$SSQL(table) \sim SSQL(table') \tag{2}$$

We start with any two operators connected end-to-end. Let SSQL_1(·) and SSQL_2(·) denote the secure SQL protocol of the former and the latter operator respectively, QRU_1(·) and QRU_2(·) denote the QRU protocol of the former and the latter operator respectively. Let $table_1 = \text{SSQL_1}(table_0)$, $table_2 = \text{SSQL_2}(table_1)$. We ensure the ums′ computed by QRU_1(·) satisfies Property 4.1.

Property 4.1. *Let* ums *denote any input update messages of QRU_1(·). The* ums′ *describes the differences between $table_1$ before and after updated by* ums. *Formally, plainly updating $table_1$ by* ums′ *results in an equivalent table to QRU_1(*ums, $table_1$*), that is*

$$PlainUpdate(\text{ums}', table_1) \sim QRU_1(\text{ums}, table_1) \tag{3}$$

Note that, the correctness of the composited QRU protocols holds, if the output table of QRU_2(·) is equivalent to that of using the secure SQL protocols from scratch. In the following, we show that this is guaranteed if ums′ satisfies Property 4. First, for any ums, the process of using the secure SQL protocols from scratch to generate the current results is:

$$table_0^u = \text{PlainUpdate}(\text{ums}, table_0)$$
$$table_1^u = \text{SSQL_1}(table_0^u)$$
$$table_2^u = \text{SSQL_2}(table_1^u)$$

Second, known that ums′ serves as the input update messages for QRU_2(·). Base on Eq 1, Eq 2, and Eq 3, we have what we want:

$$\begin{aligned}\text{QRU_2}(\text{ums}', table_2) &\sim \text{SSQL_2}(\text{PlainUpdate}(\text{ums}', table_1)) \\ &\sim \text{SSQL_2}(\text{QRU_1}(\text{ums}, table_1)) \\ &\sim \text{SSQL_2}(table_1^u) = table_2^u\end{aligned}$$

When compositing more than two QRU protocols, the effects of the update messages uploaded by data owners are transmitted pairwise between adjacent protocols and finally to the materialized tables of the last one.

5 Detailed QRU Protocols

Let um be an input update message of a QRU protocol. Recall that the QRU protocol invokes the associated insert protocol or delete protocol based on um.*manner*. In this section, we present the insert protocol and delete protocol for each SQL operator.

Notation. We denote the data row of an update message as u^r (i.e., um.*row*). We sometimes refer to u^r as an insertion row (resp. deletion row) when um.*manner* is *insert* (resp. *delete*).

5.1 Order-by-Limit

Recall that the secure SQL protocol of Order-by-Limit (OBL) first sorts input table by the descending order of *validity bit v* and then by the ascending/descending (we use descending in this paper) order of *order key k*, to get ordered table R^s, and finally outputs the top l rows of R^s (i.e., the query result) where l is the limit size.

In Shortcut, we can indeed materialize just the top l rows of R^s, if the input update messages for OBL are insertion-only. This is because the valid rows below top l (if any) will never be ranked into top l in this case, so discarding them is reasonable. However, if the incoming updates involve deletions, we cannot materialize just the top l rows of R^s. When the valid rows in top l are deleted, it is possible that the valid rows below the top l need to be ranked into top l, but they are unfortunately discarded. To avoid introducing errors in the query result, we should materialize the entire R^s. If R^s is large, performance concerns arise since securely maintaining an ordered table is not trivial.

Parameterize the size of materialized table. We propose a strategy for OBL to parameterize the size of materialized table. In this strategy, we materialize the top h rows of R^s, where $l \le h = l{+}d \le n$, d is the deletion threshold, n is the size of R^s. We denote the materialized table as D. In subsequent updates, regardless of how D grows, Shortcut only materializes the top h rows. When the number of deletions reaches the threshold d, we use the secure SQL protocol of OBL from scratch to refresh D. With this strategy, data owners can adjust the size of D based on their deletion requirements in different scenarios. The lower the requirement for deletion, the more compact D can be set by reducing d. Specially, the parameter d can be "unlimited". This implies that the number of deletions is unlimited, and Shortcut always materializes the entire table.

We note that as long as D is promptly refreshed before the number of deletions exceeds d, our materialization strategy does not introduce any errors to the query result of OBL. We provide some intuition below to help understand why this is true. If there are $\le d$ deletions on the database, we can attempt to delete these rows in the materialized table without changing the order, ensuring that the resulting table T contains at least l rows (i.e., T is a valid result of OBL with parameter l). However, if the number of deletions exceeds d, this strategy fails because: (i) the resulting table T will contain fewer than l rows, and (ii) the rows (of R^s) that can "patch" this resulting table are unfortunately discarded. A formal analysis is provided in Appendix C.

5.1.1 Insert protocol. The materialized table D is an ordered table. Upon receiving an insertion row u^r, the goal of the insert protocol of OBL is to securely insert u^r into D while keeping D ordered.

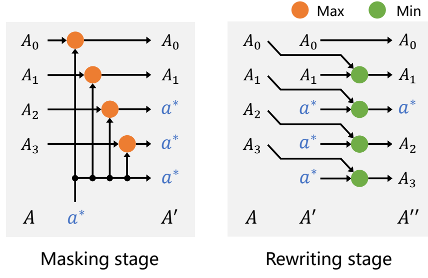


Figure 4: The illustration of constant-depth inserting circuit. Assuming $n = 4$ and $A_0 \geq A_1 \geq a^* > A_2 \geq A_3$.

Before that, if OBL is not the output operator in the directed acyclic graph, we prepare the output update messages ums′ for the next QRU protocol based on the aforementioned Property 4.1 (Protocol 1, line 3-6). Bit $flag$ indicates whether u^r should be ranked into the top l of D. If and only if the $v \parallel k$ of u^r is greater than that of the lowest ranking row in top l (i.e., D_{l-1}), u^r should be ranked into top l. Then, we compute $u^{r'_1} = D_{l-1}$ and $u^{r'_2} = u^r$, if $flag = 1$; otherwise, both $u^{r'_1}$ and $u^{r'_2}$ are set to dummy row. The $u^{r'_1}$ and $u^{r'_2}$ are pushed into ums′ along with *delete* and *insert* manner, respectively.

Intuitively, ums′ contains update messages that describe the changes to be made to the result of OBL (i.e., the top l rows of D), specifically: if u^r has a higher rank than D_{l-1}, u^r should be added to the top l and D_{l-1} should be kicked out; otherwise, nothing should be changed in the top l. These changes will be made in the follow-up steps. In this way, the computation of ums′ satisfies Property 4.1.

Finally, we securely insert u^r into D via an inserting protocol while keeping D ordered, and re-materialize top h rows (see Protocol 1, line 7-8). Now, what remains is the implementation of the inserting protocol. Efficiently inserting a new row into an ordered table while keeping the table ordered is not easy. To prevent privacy leakage through access patterns, the amount of work should be at least equal to the table size (if do not use costly ORAM techniques [15]). Let us start with a simpler case that obliviously inserts a new element a^* into an ordered sequence $A = (A_0, A_1, \ldots, A_{n-1})$, where $A_0 \geq A_1 \geq \ldots \geq A_{n-1}$. A naïve solution is a bubbling circuit that requires a linear scan, which we call bubbling inserting circuit (BIC). BIC involves maximum/minimum circuits linear in sequence size n. Precisely, BIC first do: $A \leftarrow A \cup a^*$, then for $i = n, \ldots, 1$, do: $reg = A_i$, $A_i = \min(A_{i-1}, A_i)$ and $A_{i-1} = \max(A_{i-1}, reg)$. However, a major shortcoming of BIC is its deep circuit depth, because its non-free operations (maximums/minimums) must be performed sequentially. In the context of secret-sharing-based MPC paradigms, this results in communication rounds linear in n, leading to unacceptable time costs as n increases and highlighting the poor scalability of BIC.

Constant-depth inserting circuit. The depth of BIC is discouraging. Fortunately, we observe the fixed pattern of ordered sequences and propose a clever highly-parallel circuit that perfectly solves this problem. We refer to this circuit as constant-depth inserting circuit (CDIC).

The CDIC can be implemented in two stages, each performing n fully parallel maximums/minimums. As shown in Figure 4, CDIC consists of a masking stage and a rewriting stage. In the masking stage, CDIC performs n maximums: for $i = 0, 1, \ldots, n-1$, do $A'_i \leftarrow \max(A_i, a^*)$, and $A'_n \leftarrow a^*$. All maximums can be implemented in parallel, meaning that all maximum protocols can be batched when performing communication. The masking stage marks all elements that are greater than a^* with a^* and appends an additional a^* at the end, to generate a new sequence A'.

In the rewriting stage, CDIC performs n minimums: for $i = 1, \ldots, n$, do $A''_i \leftarrow \min(A_{i-1}, A'_i)$, and $A''_0 \leftarrow A'_0$. All minimums can also be implemented in parallel. The rewriting stage rewrites the previously replaced elements back to their original positions, shifted down by one. Theorem 5.1 shows the correctness of this "sequence-version" CDIC.

Protocol 1: Insert protocol (Order-By-Limit)

```
1 function Insert_OBL(⟨u^r⟩, ⟨D⟩)
2     if OutputOpt ≠ OBL then
3         ⟨flag⟩ ← (⟨u^r[v ∥ k]⟩ > ⟨D_{l-1}[v ∥ k]⟩)
4         ⟨u^{r'_1}⟩ ← ⟨D_{l-1}⟩ · ⟨flag⟩
5         ⟨u^{r'_2}⟩ ← ⟨u^r⟩ · ⟨flag⟩
6         ums′.push({⟨u^{r'_1}⟩, delete}, {⟨u^{r'_2}⟩, insert})
7     ⟨D⟩ ← CDIC(⟨u^r⟩, ⟨D⟩)
8     ⟨D⟩ ← ⟨D_{0,1,...,h-1}⟩
9     return ums′, ⟨D⟩
```

Protocol 2: Constant-depth inserting circuit

```
1 function CDIC(⟨u^r⟩, ⟨D⟩)
      // Masking.
2     for i ∈ [0, h − 1] do
3         ⟨D'_i⟩ ← ⟨D_i⟩ + (⟨D_i[v ∥ k]⟩ < ⟨u^r[v ∥
            k]⟩) · (⟨u^r⟩ − ⟨D_i⟩)
4     ⟨D'_h⟩ ← ⟨u^r⟩
      // Rewriting.
5     ⟨D''_0⟩ ← ⟨D'_0⟩
6     for i ∈ [1, h] do
7         ⟨D''_i⟩ ← ⟨D'_i⟩ + (⟨D_{i-1}[v ∥ k]⟩ < ⟨D'_i[v ∥
            k]⟩) · (⟨D_{i-1}⟩ − ⟨D'_i⟩)
8     return ⟨D''⟩
```

Theorem 5.1. *Let $A = (A_0, A_1, \ldots, A_{n-1})$ be an ordered sequence, i.e., $\forall i \in [0, n-2]$, $A_i \geq A_{i+1}$. Let a^* be an element in the same field, $\forall j \in [0, n-1]$, there are $\forall i \in [0, j-1]$, $A_i \geq a^*$ and $\forall i \in [j, n-1]$, $A_i \leq a^*$. The output sequence A'' of CDIC satisfies: (1) $\forall i \in [0, j-1]$, $A''_i = A_i$, (2) $A''_j = a^*$, (3) $\forall i \in [j+1, n]$, $A''_i = A_{i-1}$.*

PROOF. The masking stage of CDIC computes a sequence A' with $n+1$ elements, such that

(1) $\forall i \in [0, j-1]$, $A'_i = \max(A_i, a^*) = A_i$,
(2) $\forall i \in [j, n-1]$, $A'_i = \max(A_i, a^*) = a^*$,
(3) $A'_n = a^*$.

Next, the rewriting stage of CDIC computes the output sequence A'' with $n+1$ elements, such that

(1) Since $\forall i \in [0, j-1]$, $A'_i = A_i$, then
   - $A''_0 = A'_0 = A_0$,

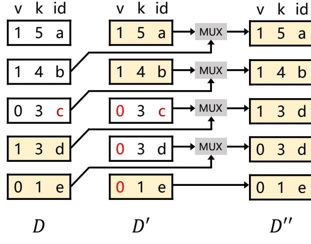


Figure 5: The rewriting stage of CDPC.

- $\forall i \in [1, j-1],\ A''_i = \min(A_{i-1}, A'_i) = \min(A_{i-1}, A_i) = A_i$.

(2) Since $\forall i \in [j, n],\ A'_i = a^*$, then

- $A''_j = \min(A_{j-1}, A'_j) = \min(A_{j-1}, a^*) = a^*$,
- $\forall i \in [j+1, n],\ A''_i = \min(A_{i-1}, A'_i) = \min(A_{i-1}, a^*) = A_{i-1}$. □

Here we treat the size of the element as constant. Both BIC-based protocol and CDIC-based protocol need $O(n)$ bits communication, but the round complexity of BIC-based protocol and CDIC-based protocol are $O(n)$ and $O(1)$ respectively. The round complexity of CDIC-based protocol eliminates the factor n (sequence size) which is a significant improvement.

In Protocol 2, we generalize the sequence-version CDIC to table-version. The core idea of table-version CDIC remains consistent with that of sequence-version CDIC, with the difference being the use of MUX to select rows based on $v \parallel k$. Note that, the comparison strictness of MUXs in two stages must be consistent. In Protocol 2, only the rows with $v \parallel k$ strictly less than that of u^r are masked in the masking stage, and then rewritten back in the rewriting stage. In other words, the rows with $v \parallel k$ equal to $u^r[v \parallel k]$, are treated as the rows with $v \parallel k$ greater than $u^r[v \parallel k]$. The correctness of table-version CDIC can be easily derived from Theorem 5.1.

5.1.2 Delete protocol. There are two cases: (i) Case 1, a valid u^r is within D, then the delete protocol needs to delete u^r from D and then restores its order, (ii) Case 2, no such valid u^r is within D, then the protocol keeps D identical as before. The identifier id (also known as the primary key) is used to uniquely identify rows.

The protocol first computes flag bit f for each row that locates the valid u^r in D, i.e., $f = 1$ if and only if for the valid row that its id equal to $u^r[id]$ (Protocol 3, line 2-3). If OBL is not the output operator in the directed acyclic graph, in lines 5-8 of Protocol 3, we prepare ums′ for the next QRU protocol. The computation of ums′ follows from the following idea. If a valid u^r exists in the top l of D, we should delete it from the top l and insert the candidate row D_l into the top l. Thus we compute $u^{r'_1} = u^r$ and $u^{r'_2} = D_l$. Otherwise, nothing should be changed in the top l, thus we set $u^{r'_1}$ and $u^{r'_2}$ as dummy rows. The $u^{r'_1}$ and $u^{r'_2}$ are pushed into ums′ with *insert* and *delete* manner, respectively. This ensures that ums′ satisfies Property 4.1.

Next, we begin to update the materialized table D. In principle, the protocol should first scan D to delete the row where $f = 1$. Since this operation is implicitly included in the follow-up constant-depth push-down circuit we propose, it can be omitted. Since deleting a row means setting it to a dummy row, D will be disrupted by the generated dummy row. We need a protocol to securely restore the order of D. The naïve solution is also a bubbling circuit that requires a sequential linear scan. Instead of using a bottom-up bubbling like BIC, it performs a top-down bubbling to push the generated dummy row down and restore the order of D. We refer to this circuit as bubbling push-down circuit (BPC). It is evident that BPC shares the same "deep-depth" shortcoming as BIC, which limits its scalability in secret-shared-based MPC paradigms.

Protocol 3: Delete protocol (Order-By-Limit)

1 **function** Delete_OBL($\langle u^r \rangle, \langle D \rangle$)
2   **for** $i \in [0, h-1]$ **do**
3         $\langle D_i[f] \rangle \leftarrow (\langle D_i[id] \rangle \overset{?}{=} \langle u^r[id] \rangle) \cdot \langle u^r[v] \rangle \cdot \langle D_i[v] \rangle$
4   **if** $OutputOpt \neq OBL$ **then**
5         $\langle f_s \rangle = \sum_{i=0}^{l-1} \langle D_i[f] \rangle$
6         $\langle u^{r'_1} \rangle \leftarrow \langle u^r \rangle \cdot \langle f_s \rangle$
7         $\langle u^{r'_2} \rangle \leftarrow \langle D_l \rangle \cdot \langle f_s \rangle$
8   ums′.push($\{\langle u^{r'_1} \rangle, delete\}$, $\{\langle u^{r'_2} \rangle, insert\}$)
9     $\langle D'' \rangle \leftarrow$CDPC($\langle D \rangle$)
10   **return** ums′, $\langle D'' \rangle$

Protocol 4: Constant-depth push-down circuit

1 **function** CDPC($\langle D \rangle$)
      // Masking.
2     $\langle D' \rangle \leftarrow \langle D \rangle$
3   **for** $i \in [0, h-2]$ **do**
4         $\langle D'_{i+1}[f] \rangle \leftarrow \langle D'_i[f] \rangle + \langle D_{i+1}[f] \rangle$
5   **for** $i \in [0, h-1]$ **do**
6         $\langle D'_i[v] \rangle \leftarrow \langle D'_i[v] \rangle \cdot \langle D'_i[\overline{f}] \rangle$
      // Rewriting.
7   **for** $i \in [0, h-2]$ **do**
8         $\langle D''_i \rangle \leftarrow \langle D_{i+1} \rangle + (\langle D'_i[v \parallel k] \rangle \geq \langle D_{i+1}[v \parallel k] \rangle) \cdot (\langle D'_i \rangle - \langle D_{i+1} \rangle)$
9     $\langle D''_{h-1} \rangle \leftarrow \langle D'_{h-1} \rangle$
10   **return** $\langle D'' \rangle$

Since all dummy rows are regarded as identical, we propose a constant-depth push-down circuit (CDPC) by maintaining a "relaxed ordered" D. In relaxed ordered, instead of requiring D strictly ordered by v and then by k, we do not require the dummy rows to be ordered among themselves.

Definition 5.1. *A relaxed ordered table D satisfies: (1) $\forall i \in [0, h-2]$, if D_i is valid row, $D_i[v \parallel k] \geq D_{i+1}[v \parallel k]$, (2) $\forall i \in [0, h-2]$, if D_i is dummy row, then D_{i+1} is dummy row.*

Similar to CDIC, CDPC also consists of two stages. In the masking stage (Protocol 4, line 2-6), CDPC computes a new table D', such that the potentially deleted valid row and the rows below it are marked as dummy. Formally, the masking stage satisfies Claim 5.1. In the rewriting stage (Protocol 4, line 7-9), the dummy row generated from the potentially deleted valid row is pushed down, and the output D'' restores to be relaxed ordered. Figure 5 illustrates an example of the rewriting stage of CDPC. In this example, the row with $id = c$ is deleted and becomes the newly generated dummy

row. The rewriting stage implements $h-1$ parallel MUXs. CDPC is used in Protocol 3 line 9.

Given a relaxed ordered table, Theorem 5.2 states that if a valid row is deleted, CDPC can restore the table to be relaxed ordered. If no row is deleted, Theorem 5.3 states that CDPC keeps the table identical as before.

Claim 5.1. *For the following two cases, the D' computed by the masking stage of CDPC holds different properties.*

C1. If a valid row D_j is deleted from the relaxed ordered D, $D_j[f]=1$. After the masking stage, the D' satisfies:
- *$\forall i \in [0, j-1]$, $D'_i = D_i$ and they are valid rows,*
- *$\forall i \in [j, h-1]$, D'_i are dummy rows.*

C2. If no row is deleted, $\forall i \in [0, h-1]$, there is $D_i[f]=0$. After the masking stage, $D'=D$.

Theorem 5.2. *If a valid row D_j is deleted from the relaxed ordered D. CDPC's output table $D''=(D_0,\dots,D_{j-1},D_{j+1},\dots,D_{h-1},dummy)$.*

Proof. Holding the following two pieces of knowledge:

(a) D still satisfies: $\forall i \in [0,h-2]$, if D_i is valid row, $D_i[v \parallel k] \geq D_{i+1}[v \parallel k]$.

(b) The MUXs circuit in rewriting stage computes: $\forall i \in [0,h-2]$, if $D'_i[v \parallel k] \geq D_{i+1}[v \parallel k]$, $D''_i = D'_i$; otherwise, $D''_i = D_{i+1}$.

Then, we prove Theorem 5.2 from three intervals:

(1) $\forall i \in [0, j-1]$, according to *C1* of Claim 5.1, $D'_i = D_i$ and they are valid rows. Thus we have $D'_i[v \parallel k] = D_i[v \parallel k] \geq D_{i+1}[v \parallel k]$, then $D''_i = D'_i = D_i$.

(2) $\forall i \in [j, h-2]$, according to *C1* of Claim 5.1, D'_i is dummy.
- If D_{i+1} is valid row, there must be $D_{i+1}[v \parallel k] > D'_i[v \parallel k]$, thus $D''_i = D_{i+1}$,
- If D_{i+1} is dummy row, D''_i must be dummy row. $D''_i = D_{i+1}$.

Thus, we have $D''_i = D_{i+1}$ for $\forall i \in [j, h-2]$.

(3) According to *C1* of Claim 5.1, D'_{h-1} is dummy row. CDPC sets $D''_{h-1} = D'_{h-1}$ (line 9). □

Theorem 5.3. *If no row is deleted, D is still a relaxed ordered table. CDPC's output table $D''=D$.*

Proof. Holding the following two pieces of knowledge:

(a) Since D is relaxed ordered, we have
- $\forall i \in [0,h-2]$, if D_i is valid row, $D_i[v \parallel k] \geq D_{i+1}[v \parallel k]$,
- $\forall i \in [0,h-2]$, if D_i is dummy row, then D_{i+1} is dummy.

(b) The MUXs circuit in rewriting stage computes: $\forall i \in [0,h-2]$, if $D'_i[v \parallel k] \geq D_{i+1}[v \parallel k]$, $D''_i = D'_i$; otherwise, $D''_i = D_{i+1}$.

According to *C2* of Claim 5.1, $D' = D$, then

(1) $\forall i \in [0,h-2]$,
- If D'_i is valid row, $D'_i[v \parallel k] = D_i[v \parallel k] \geq D_{i+1}[v \parallel k]$, then $D''_i = D'_i = D_i$.
- If D'_i is dummy row, then D_i, D_{i+1} and D''_i are dummy rows.

Thus we have $D''_i = D_i$ for $\forall i \in [0,h-2]$.

(2) $D''_{h-1} = D'_{h-1} = D_{h-1}$. □

CDPC inherently deletes u^r. Claim 5.1 and Theorem 5.2 imply the fact that CDPC sets the row with $f=1$ as dummy row while pushes it to the bottom of D. Therefore, we can eliminate the preceding linear scan that serves to delete the row with $f=1$ (i.e., sets it as a dummy row), and indeed, we have done so.

Protocol 5: Insert protocol (Group-by-Sum)

1 **function** Insert_GBS($\langle u^r \rangle, \langle G \rangle$)
2   **for** $i \in [0, n-1]$ **do**
3         $\langle G_i[f] \rangle \leftarrow (\langle G_i[k] \rangle =? \langle u^r[k] \rangle) \cdot \langle G_i[v] \rangle \cdot \langle u^r[v] \rangle$
4     $\langle G \rangle \leftarrow \langle G \rangle \cup \langle u^r \rangle$
5     $\langle G_n[v] \rangle \leftarrow \langle G_n[v] \rangle \cdot \overline{\sum_{i=0}^{n-1} \langle G_i[f] \rangle}$
6   **for** $i \in [0, n-1]$ **do**
7         $\langle G_i[r] \rangle \leftarrow \langle G_i[r] \rangle + \langle G_i[f] \rangle \cdot \langle u^r[a] \rangle$
8   **if** *OutputOpt* $\neq$ *GBS* **then**
9         $\langle u^{r'_1} \rangle \leftarrow \sum_{i=0}^{n-1} (\langle G_i[f] \rangle \cdot \langle G_i \rangle)$
10         $\langle u^{r'_2} \rangle \leftarrow \langle u^{r'_1} \rangle$
11         $\langle u^{r'_2}[r] \rangle \leftarrow \langle u^{r'_2}[r] \rangle - \langle u^r[a] \rangle$
12         $\langle u^{r'_3} \rangle \leftarrow \langle G_n \rangle$
13   ums'.push($\{\langle u^{r'_1} \rangle, insert\}, \{\langle u^{r'_2} \rangle, delete\}, \{\langle u^{r'_3} \rangle, insert\}$)
14   **return** ums', $\langle G \rangle$

CDIC can also maintain a relaxed ordered table. If the input update messages involve both insertion and deletion, Shortcut has to implement CDIC and CDPC sequentially. The above discussions indicate that CDPC maintains a relaxed ordered table, while CDIC maintains a strictly ordered table. In fact, CDIC can also maintain a relaxed ordered table to be compatible with CDPC.

Informally, the core observation is that different dummy rows used as the input of MUX do not affect its output. Any two identical tables, where one is relaxed ordered and the other is strictly ordered, only differ in the contents of dummy rows. After computing by CDIC (a set of MUXs), they are still identical. Then we can conclude that CDIC can also correctly maintain any relaxed ordered table.

5.2 Group-by-Aggregate

Let k denote *group key*, a denote *aggregate attribute*, and r denote *result attribute*. Shortcut supports both insert and delete protocols for Group-by-Sum/Count and only supports insert protocol for Group-by-Max/Min. We provide protocols for Group-by-Sum (GBS), which also yield protocols for Group-by-Count if the validity bit v is treated as the aggregate attribute a. Moreover, we discuss how to get the protocols for other SQL operators with minor adjustments. Let G denote a n-row materialized table of GBS.

5.2.1 Insert protocol. Recall that the insert protocol of GBS aims to meet two cases: (i) Case 1: if a target row exists, $u^r[a]$ should be added to the value r of the target row, (ii) Case 2: if no target row exists, u^r forms a new group in G. We first compute flag bit f for each row to locate the target row, i.e., $f=1$ if and only if for the target row (Protocol 5, line 2-3). Then, we append u^r to the end of G and reset its validity bit (Protocol 5, line 4-5). If Case 1 holds, G_n is set to a dummy row; otherwise, G_n remains unchanged. In line 6-7, the protocol adds $u^r[a]$ to the value r of the target row (if exists). If GBS is not the output operator, we prepare ums′ for the next QRU protocol (Protocol 5, line 8-13). When Case 1 holds, $u^{r'_1}$ equals the updated target row and $u^{r'_2}$ equals the target row before

Protocol 6: Delete protocol (Group-by-Sum)

1 **function** Delete_GBS($\langle u^r\rangle, \langle G\rangle$)
2 **for** $i \in [0, n-1]$ **do**
3 $\langle G_i[f]\rangle \leftarrow (\langle G_i[k]\rangle =? \langle u^r[k]\rangle) \cdot \langle G_i[v]\rangle \cdot \langle u^r[v]\rangle$
4 **for** $i \in [0, n-1]$ **do**
5 $\langle G_i[r]\rangle \leftarrow \langle G_i[r]\rangle - \langle G_i[f]\rangle \cdot \langle u^r[a]\rangle$
6 **if** $OutputOpt \neq GBS$ **then**
7 $\langle u^{r'_1}\rangle \leftarrow \sum_{i=0}^{n-1}(\langle G_i[f]\rangle \cdot \langle G_i\rangle)$
8 $\langle u^{r'_2}\rangle \leftarrow \langle u^{r'_1}\rangle$
9 $\langle u^{r'_2}[r]\rangle \leftarrow \langle u^{r'_2}[r]\rangle + \langle u^r[a]\rangle$
10 ums$'$.push($\{\langle u^{r'_1}\rangle, insert\}$, $\{\langle u^{r'_2}\rangle, delete\}$)
11 **return** ums$'$, $\langle G\rangle$

Protocol 7: Insert protocol (Join)

1 **function** Insert_Join($\langle u^r\rangle, \langle A\rangle, \langle B\rangle, \langle J\rangle$)
2 $\langle A\rangle \leftarrow \langle A\rangle \cup \langle u^r\rangle$
3 **for** $i \in [0, m-1]$ **do**
4 $\langle B_i[f]\rangle \leftarrow (\langle B_i[k]\rangle =? \langle u^r[k]\rangle) \cdot \langle B_i[v]\rangle$
5 $\langle u^{r'}\rangle \leftarrow \langle u^r\rangle$
6 $\langle u^{r'}[p_B]\rangle \leftarrow \sum_{i=0}^{m-1}(\langle B_i[p]\rangle \cdot \langle B_i[f]\rangle)$
7 $\langle u^{r'}[v]\rangle \leftarrow \langle u^r[v]\rangle \cdot \sum_{i=0}^{m-1}\langle B_i[f]\rangle$
8 $\langle J\rangle \leftarrow \langle J\rangle \cup \langle u^{r'}\rangle$
9 **if** $OutputOpt \neq Join$ **then**
10 ums$'$.push($\{\langle u^{r'}\rangle, insert\}$)
11 **return** ums$'$, $\langle A\rangle, \langle B\rangle, \langle J\rangle$

updated. When Case 2 holds, $u^{r'_1}$ and $u^{r'_2}$ are dummy rows. The $u^{r'_3}$ always equals the appended row. The update manners of $u^{r'_1}$, $u^{r'_2}$ and $u^{r'_3}$ are *insert*, *delete*, and *insert*, respectively. The computation of ums$'$ satisfies Property 4.1.

5.2.2 Delete protocol. The delete protocol of GBS locates the target row and then subtracts $u^r[a]$ from the value r of the target row (Protocol 6, line 2-5). If GBS is not the output operator, we prepare ums$'$ for the next QRU protocol (Protocol 6, line 6-10). If a target row exists in G, $u^{r'_1}$ equals the updated target row and $u^{r'_2}$ equals the target row before update. Otherwise, $u^{r'_1}, u^{r'_2}$ are dummy rows. The update manners of $u^{r'_1}$ and $u^{r'_2}$ are *insert* and *delete* respectively. The computation of ums$'$ satisfies Property 4.1.

5.2.3 Distinct. Distinct can be seen as a special case of Group-by-Count [22]. We directly reuse the secure SQL protocol and insert protocol of Group-by-Count for this operator. The delete protocol of Distinct is obtained by adding a step (That is, scanning the materialized table to delete the rows with zero counts) at the end of the delete protocol of Group-by-Count.

5.2.4 Group-by-Max/Min. The insert protocol of Group-by-Max/ Min can be obtained by replacing the aggregate operation that updates the target row with Maximum/Minimum. We don't provide the delete protocol for this operator. If the update manner is *delete*, we apply the secure SQL protocol of Group-by-Max/Min from scratch on its plainly updated input table.

Protocol 8: Delete protocol (Join)

1 **function** Delete_Join($\langle u^r\rangle, \langle A\rangle, \langle B\rangle, \langle J\rangle$)
2 **for** $i \in [0, m-1]$ **do**
3 $\langle A_i[v]\rangle \leftarrow \overline{(\langle A_i[k]\rangle =? \langle u^r[k]\rangle) \cdot \langle u^r[v]\rangle} \cdot \langle A_i[v]\rangle$
4 **for** $i \in [0, n-1]$ **do**
5 $\langle J_i[f]\rangle \leftarrow (\langle J_i[k]\rangle =? \langle u^r[k]\rangle) \cdot \langle u^r[v]\rangle \cdot \langle J_i[v]\rangle$
6 **if** $OutputOpt \neq Join$ **then**
7 $\langle u^{r'}\rangle \leftarrow \langle u^r\rangle$
8 $\langle u^{r'}[p_B]\rangle \leftarrow \sum_{i=0}^{n-1}(\langle J_i[p]\rangle \cdot \langle J_i[f]\rangle)$
9 $\langle u^{r'}[v]\rangle \leftarrow \sum_{i=0}^{n-1}\langle J_i[f]\rangle$
10 ums$'$.push($\{\langle u^{r'}\rangle, delete\}$)
11 **for** $i \in [0, n-1]$ **do**
12 $\langle J_i[v]\rangle \leftarrow \overline{\langle J_i[f]\rangle} \cdot \langle J_i[v]\rangle$
13 **return** ums$'$, $\langle A\rangle, \langle B\rangle, \langle J\rangle$

Protocol 9: Insert protocol & Delete protocol (Select)

1 **function** Insert_Select($\langle u^r\rangle, \langle S\rangle$)
2 $\langle u^r[v]\rangle \leftarrow \langle u^r[v]\rangle \cdot \sigma(\langle u^r\rangle)$
3 $\langle S\rangle \leftarrow \langle S\rangle \cup \langle u^r\rangle$
4 **if** $OutputOpt \neq Select$ **then**
5 ums$'$.push($\{\langle u^r\rangle, insert\}$)
6 **return** ums$'$, $\langle S\rangle$

7 **function** Delete_Select($\langle u^r\rangle, \langle S\rangle$)
8 **for** $i \in [0, n-1]$ **do**
9 $\langle S_i[v]\rangle \leftarrow \langle S_i[v]\rangle \cdot \overline{(\langle S_i[id]\rangle =? \langle u^r[id]\rangle)}$
10 **if** $OutputOpt \neq Select$ **then**
11 ums$'$.push($\{\langle u^r\rangle, delete\}$)
12 **return** ums$'$, $\langle S\rangle$

5.3 Join

In operator Join, We denote the materialized table targeted by the input update message as A and the other materialized table as B. Let m be the size of A and B, n be the size of output table J.

5.3.1 Insert protocol. The insert protocol of Join first attaches u^r to the end of A. The flag bit f detects whether there is any valid row in B that joins with u^r, i.e., $f = 1$ if and only if for the valid row that joins with u^r (Protocol 7, line 3-4). Then, the protocol constructs the joined row $u^{r'}$ and attaches it to the end of J (Protocol 7, line 5-8). If Join is not the output operator, the protocol pushes $u^{r'}$ into ums$'$ with *insert* manner (Protocol 7, line 9-10).

5.3.2 Delete protocol. The delete protocol of Join first deletes a valid u^r from A (Protocol 8, line 1-2). The flag bit f detects whether there is a joined row introduced by a valid u^r in J (Protocol 8, line 3-4). Then, if Join is not the output operator, we fetch the joined row and push it into ums$'$ with *delete* manner (Protocol 8, line 6-12). Finally, we delete the joined row from J (Protocol 8, line 13-14).

5.4 Select

Let S denote a n-row materialized table of Select. The insert protocol of Select (Protocol 9, line 1-6) filters u^r by $\sigma(\cdot)$ and appends it to

Protocol 10: Insert protocol & Delete protocol (Global Aggregate)

```
1 function Insert_GA(⟨u^r⟩, ⟨z⟩)
2     ⟨z⟩ ← Agg(⟨z⟩, ⟨u^r⟩) // where Agg(⟨z⟩, ⟨u^r⟩)
        = ⟨z⟩ + ⟨u^r[a]⟩ for Sum, ⟨z⟩ + ⟨u^r[v]⟩ for
        Count, max(⟨z⟩, ⟨u^r[a]⟩) for Max,
        min(⟨z⟩, ⟨u^r[a]⟩) for Min.
3     return ⟨z⟩
4 function Delete_GA(⟨u^r⟩, ⟨z⟩)
5     ⟨z⟩ ← Sub(⟨z⟩, ⟨u^r⟩) // where Sub(⟨z⟩, ⟨u^r⟩)
        = ⟨z⟩ − ⟨u^r[a]⟩ for Sum, Sub(⟨z⟩, ⟨u^r⟩)
        = ⟨z⟩ − ⟨u^r[v]⟩ for Count.
6     return ⟨z⟩
```

the bottom of S. If Select is not the output operator, we push the filtered u^r into ums′ with *insert* manner.

The delete protocol of Select (Protocol 9, line 7-12) scans S to set the validity bit to 0 for the row with the same *id* as u^r. If Select is not the output operator, we push u^r into ums′ with *delete* manner.

5.5 Global Aggregate

We show the insert protocol and the delete protocol of Global Aggregate in Protocol 10. Let a denote *aggregate attribute*, z denote the materialized aggregated value. For Sum/Count/Max/Min-based aggregation, the insert protocol is realized by additionally aggregating the value in u^r into z (Protocol 10, line 1-3). For Sum/Count-based aggregation, the delete protocol subtracts the value in u^r from z (Protocol 10, line 4-6). For Max/Min-based aggregation, we use existing secure SQL protocols to query the underlying table from scratch after deletion. The Global Aggregate is usually not followed by other SQL operators, as its output is a single aggregated value.

5.6 Summary

As shown in Table 1 and Table 2, SHORTCUT's QRU protocols achieve performance improvements over MCAS's secure SQL protocols for one query after an update. However, there is a performance crossover between the QRU protocols and the secure SQL protocols as the input update messages grow, since the secure SQL protocol is more suitable for processing update messages in bulk. The performance crossover occurs when a large number of update messages are uploaded by data owners in a short period causing a big backlog of unprocessed update messages. Therefore, SHORTCUT is more suitable for minor updates arriving in time. Fortunately, due to the nature of Observation 4.1, the materialized table remains compatible with secure SQL protocol regardless of how many times the QRU protocol is invoked. Hence, we can simply turn around to use secure SQL protocol from scratch, when performance crossover occurs. This can be done by a cost-based scheduling. From this perspective, SHORTCUT and MCASs are complementary.

Storage overhead. The initial size of materialized tables is proportional to the size of underlying databases, with a factor close to the number of SQL operators in the directed acyclic graph, which is acceptable. The growth in rows introduced by data updates is proportional to the number of update messages uploaded by data owners. When the number of update messages is not particularly large, this growth is acceptable; otherwise, we can use secure SQL protocol to reinitialize the materialized tables, or reduce the size of materialized tables by providing DP guarantee [9], with either periodic or threshold-based trigger mechanism.

6 Security Analysis

We formalize the security of SHORTCUT using the standard simulation-based security [18, 20]. In the presence of a semi-honest adversary (the case of our instantiation), this is modeled by showing that the real-world joint distribution of the views of corrupted parties and the outputs of honest parties is indistinguishable from its ideal-world counterpart, where the ideal-world views of corrupted parties are generated by a simulator given only the inputs and outputs of corrupted parties. Since the parties in our QRU protocols have no output in plaintext and the overall circuit composed of QRU protocols captures a deterministic functionality, we can use the following simplified definition of semi-honest security, which only considers the views of corrupted parties in both worlds.

Definition 6.1 ([20]). *Let parties $P_1, \dots, P_n$ engage in protocol Π that computes deterministic functionality $f(in_1, \dots, in_n)$, where in_i denotes the input of party P_i. Let $VIEW_\Pi(P_i)$ denote the view of party i during protocol Π, which consists of its input in_i, its internal random coins r_i, and messages m_i that were received by P_i in the execution. Let $C = \{P_{i_1}, P_{i_2}, \dots, P_{i_t}\}$ denote a subset of the parties for $t < n$, $VIEW_\Pi(C)$ denote the combined view of parties in C during protocol Π (i.e., the union of the views of the parties in C), and $f_C(in_1, \dots, in_n)$ denote the projection of $f(in_1, \dots, in_n)$ on the coordinates in C (i.e., $f_C(in_1, \dots, in_n)$ consists of the i_1-th, $\dots$, i_t-th element that $f(in_1, \dots, in_n)$ outputs). We say that protocol Π securely computes deterministic functionality f in the presence of a semi-honest adversary that corrupts parties in C, if there exists probabilistic polynomial time simulator $\mathcal{S}$ for every coalition C such that $\{\mathcal{S}(C, in_C, f_C(in_1, \dots, in_n))\} \equiv \{VIEW_\Pi(C)\}$, where $\equiv$ denotes indistinguishability and $in_C = \bigcup_{P_i \in C}\{in_i\}$.*

In this work, we instantiate our QRU protocols with the 2-out-of-3 secret sharing scheme. Moreover, we prove the security of the QRU protocols in the hybrid model, where parties run a protocol with pairwise communication and also have access to a trusted party computing a sub-functionality for them. The sequential composition theorem of [12] states that security is preserved when such a trusted party is replaced by a sub-protocol securely realizing this sub-functionality. Given a sub-functionality g, a QRU protocol is said to work in the g-hybrid model.

The high-level intuition of the semi-honest security of our protocols is as follows. We rely on a set of functionalities provided by the MPC protocols, which have been proven to be secure in prior works (such as in [3, 24]), to perform data sharing, data reconstruction, and circuit evaluations using 2-out-of-3 secret sharing scheme. In QRU protocols, addition, subtraction, and bit inversion only consist of local computation. Multiplication, equality, and comparison are computed via sub-functionality $\mathcal{F}_{\text{CE}}$. Functionality $\mathcal{F}_{\text{CE}}$ and the ideal functionality $\mathcal{F}_{\text{QRU}}$ of QRU protocols can be found in Appendix D. Loosely speaking, our QRU protocols call the sub-functionalities of circuit evaluations in a modular black-box fashion while all intermediate values between these functionalities

are secret-shared. Then the modular sequential composition theorem directly implies the security of QRU protocols. The semi-honest security is formalized in Theorem 6.1, with the proof postponed to Appendix A.

Theorem 6.1. *According to Definition 6.1, Protocol 1, Protocol 3, Protocol 5, Protocol 6, Protocol 7, Protocol 8, Protocol 9, and Protocol 10 jointly securely compute deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model, where $\mathcal{F}_{QRU}$ and $\mathcal{F}_{CE}$ are given in Appendix D.*

So far, we treat the update manners and the number of update messages as public. When these pieces of information have the potential to leak private data, we should consider them as part of privacy and take measures to protect them. We discuss this below.

Protecting update manner. Two measures can protect the actual update manner in an oblivious sense and prevent privacy leakage through it. First, the most straightforward measure is restricting the update manner uploaded by data owners, such as allowing data owners to initiate update messages involving only insertion. Another measure hides the update manners via dummy padding, i.e., padding the update messages uploaded by data owners as a continuous queue of insert-delete pairs. Such as, padding $\{(t^{u_0}, insert), (t^{u_1}, insert), (t^{u_2}, delete)\}$ as $\{(t^{u_0}, insert), (dummy, delete), (t^{u_1}, insert), (t^{u_2}, delete)\}$.

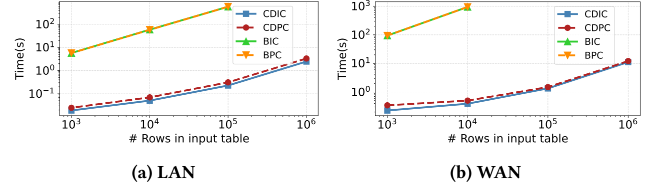
Protecting the number of update messages. The problem of privacy leakage through the number of update messages is first presented in DP-Sync [34]. Data owners can mitigate this problem by padding update messages based on DP, similar to DP-Sync, or even by directly padding update messages to a fixed maximum size. At a high level, the DP-based method in [34] asks data owners to store their update messages in their local storage. When a data update is needed, they retrieve a number of stored update messages, where the number is perturbed by Laplacian noise. If this number is greater than the number of stored update messages, dummy messages will be padded to the retrieved ones.

7 Experiment

This section describes the experimental results of Shortcut. We implement Shortcut on the top of MP-SPDZ [21], a well-known MPC framework that supports various state-of-the-art MPC schemes. We implement Secrecy [22] and the semi-honest version of AHK⁺ [5] as our baselines. Secrecy can be easily extended to other models, e.g. two-party model, because it only uses black-box computations of simple circuits. AHK⁺ achieves the best performance in the three-party honest majority setting among prior MCASs, since it relies on an efficient sorting protocol in this setting [4]. In all experiments, the size of the secret-shared data is 64 bits.

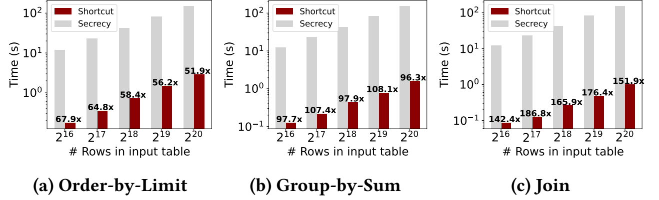
Experimental Setup. We implement the experiments on three connected machines. Each machine offers Intel Xeon Platinum 8375C 2.90GHz CPU and 256GB RAM. These machines perform their computation on a single thread. The network condition of these machines is controlled by *tc* command where the local area network (LAN) has 5Gbps of bandwidth and 0.3ms of RTT, and the wide area network (WAN) has 500Mbps of bandwidth and 20ms of RTT. Our experiments evaluate the end-to-end performance.

Datasets. We conduct all experiments with randomly sampled data. Note that the performance of oblivious protocols is solely dependent on data size and is independent of specific data values. Since all protocols in Shortcut and Secrecy are oblivious, using randomly sampled data is no different than real-life data, and data distribution does not affect protocol overhead.



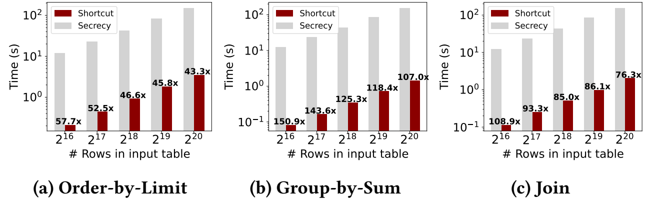
(a) LAN **(b) WAN**

Figure 6: Oblivious inserting circuit & push-down circuit.



(a) Order-by-Limit **(b) Group-by-Sum** **(c) Join**

Figure 7: The performance of SQL operators after an insertion (Shortcut vs. Secrecy).



(a) Order-by-Limit **(b) Group-by-Sum** **(c) Join**

Figure 8: The performance of SQL operators after a deletion (Shortcut vs. Secrecy).

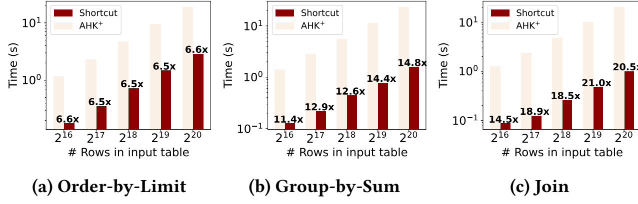
7.1 Single Operator

In this section, (i) we compare our two building blocks, CDIC and CDPC, respectively, with their corresponding naïve solutions, BIC and BPC, (ii) we compare Shortcut with Secrecy and AHK⁺ on three operators, including OBL, GBS, and Join, for one query after an update.

Let the input tables of CDIC and CDPC contain 3 attributes, and the input tables of BIC and BPC contain 2 attributes. Figure 6 shows that CDIC and CDPC significantly surpass BIC and BPC in runtime performance. For instance, CDIC (resp. CDPC) can maintain a 10^6-row table in 2.47 seconds (resp. 3.34 seconds) in the LAN setting, while BIC and BPC need more than 1.63 hours.

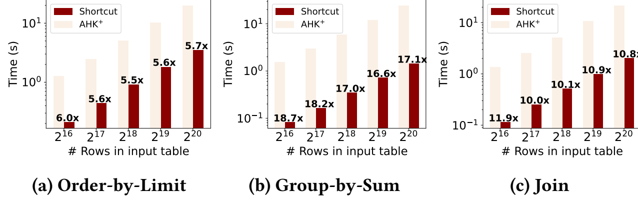
In the following experiments of OBL, GBS, and Join, we let each input table contain 4 attributes and use the LAN setting. We first compare Shortcut with Secrecy on the runtime of one query after an update with a growing table size. The results are shown in Figure 7 and Figure 8. For OBL, Shortcut is up to 67.9× (resp. 57.7×) faster than Secrecy per query after an insertion (resp. a deletion).

In terms of GBS, Shortcut is up to 108.1× (resp. 150.9×) faster than Secrecy for one query after an insertion (resp. a deletion). On a 2^{20}-row materialized table of GBS, Shortcut can complete a



(a) Order-by-Limit (b) Group-by-Sum (c) Join

Figure 9: The performance of SQL operators after an insertion (SHORTCUT vs. AHK⁺).



(a) Order-by-Limit (b) Group-by-Sum (c) Join

Figure 10: The performance of SQL operators after a deletion (SHORTCUT vs. AHK⁺).

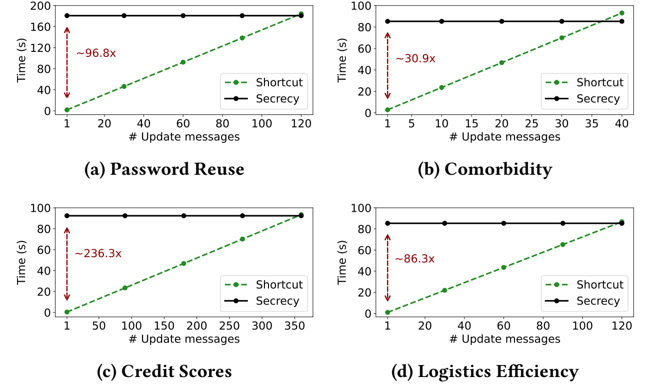
query after an insertion (resp. a deletion) in 1.57 seconds (resp. 1.43 seconds).

We do not implement the nested-loop Join for Secrecy (which it does) due to its high asymptotic cost, but rather implement the sort-compare-shuffle Join [28] which is more efficient in unique-key Join. In terms of Join, SHORTCUT achieves up to 186.8× and 108.9× improvements than Secrecy for one query after an insertion and a deletion, respectively. On a 2^{20}-row materialized table of Join, SHORTCUT can complete a query after an insertion and a deletion in 1 second and 2 seconds, respectively.

Then, we compare SHORTCUT with AHK⁺ on the runtime of one query after an update with a growing table size. The results are shown in Figure 9 and Figure 10. The advantage of SHORTCUT over AHK⁺ is smaller than its advantage over Secrecy. In terms of OBL, SHORTCUT is up to 6.6× (resp. 6×) faster than AHK⁺ for one query after an insertion (resp. a deletion). For GBS, SHORTCUT is up to 14.8× (resp. 18.7×) faster than AHK⁺ for one query after an insertion (resp. a deletion). For Join, SHORTCUT is up to 21× (resp. 11.9×) faster than AHK⁺ for one query after an insertion (resp. a deletion).

7.2 Real-Life Queries

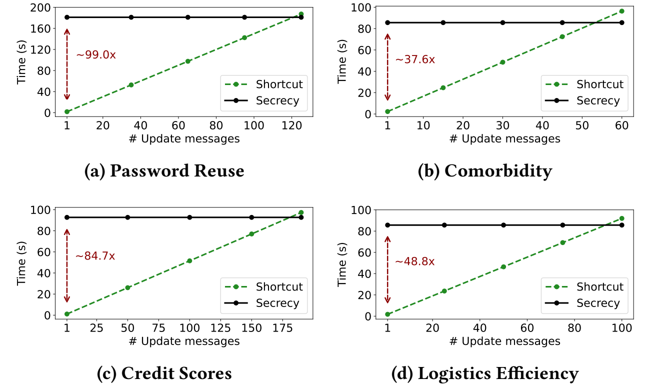
Queries. We test SHORTCUT via several real-life queries. The detailed query statements can be found in Appendix B. The first query is the aforementioned *Password Reuse*, we set the user identifier as 64 bits, the password hash as 256 bits, and the input table contains 2^{18} rows. *Comorbidity* [8] is a medical query that returns the ten most common diagnoses of individuals in a cohort. It first executes Select, followed by GBA, and then OBL. We let the input table contain 2^{18} rows. *Credit Scores* [28] finds persons whose credit scores across different agencies have significant discrepancies in a particular year. This query consists of multiple table Joins and then Select. We use 4 input tables each has 2^{16} rows in our benchmark. We refer to the last query as *Logistics Efficiency*, which is adapted from the use case in [35]. Assuming that a courier company partners with a local retail store to help deliver products. The retail store has its



(a) Password Reuse (b) Comorbidity

(c) Credit Scores (d) Logistics Efficiency

Figure 11: The performance of real-life queries after insertion-only updates (SHORTCUT vs. Secrecy).



(a) Password Reuse (b) Comorbidity

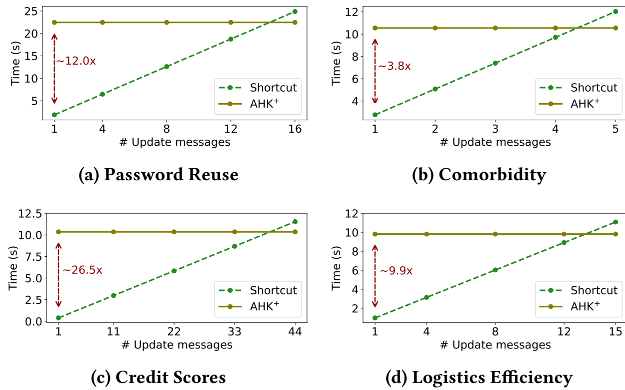
(c) Credit Scores (d) Logistics Efficiency

Figure 12: The performance of real-life queries after deletion-only updates (SHORTCUT vs. Secrecy).

sales data, and the courier company has its delivery records. *Logistics Efficiency* finds the twenty products with the nearest delivery distance among those that are overdue for delivery. It consists of two table Joins, followed by Select and then OBL. We let the two input tables contain 2^{17} rows.

Runtime performance. The above queries involve the composition of QRU protocols, thus we can use pipelined execution to improve the practical efficiency of SHORTCUT. Given that SHORTCUT and MCAS are known to cross performance when the update messages accumulate to a certain threshold, we evaluate the real-life queries with growing update messages. We abuse to denote the ratio of insertions to deletions in a set of updates as *insert : delete*. The evaluations are conducted in LAN setting.

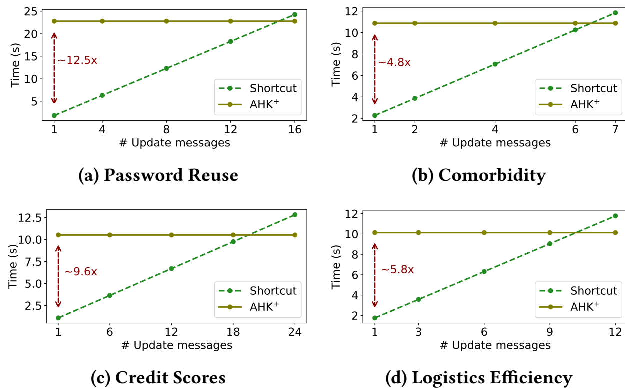
We first compare SHORTCUT with Secrecy. Figure 11 and Figure 12 show the runtime of one query after insertion-only updates (i.e., insert : delete = 1) and deletion-only updates (i.e., insert : delete = 0), respectively. When insert : delete = 1, SHORTCUT achieves the best improvement on *Credit Scores*. In this case, SHORTCUT is 236.3× faster than Secrecy per query after an insertion. Benefiting from pipelined execution, the performance crossover is deferred to the point that the number of accumulated insertion rows is 355. When insert : delete = 0, SHORTCUT achieves the best improvement on *Password Reuse*. In this case, SHORTCUT is 99× faster than Secrecy



(a) Password Reuse (b) Comorbidity

(c) Credit Scores (d) Logistics Efficiency

Figure 13: The performance of real-life queries after insertion-only updates (Shortcut vs. AHK⁺).



(a) Password Reuse (b) Comorbidity

(c) Credit Scores (d) Logistics Efficiency

Figure 14: The performance of real-life queries after deletion-only updates (Shortcut vs. AHK⁺).

after a deletion, and the performance crossover occurs when the number of accumulated deletion rows is 121.

As shown in Figure 13, Shortcut achieves the best advantage over AHK⁺ in *Credit Scores* when insert : delete = 1. In this case, Shortcut is 26.5× faster than AHK⁺ for one query after an insertion. The performance crossover is deferred to the point that the number of accumulated insertion rows is 39. Figure 14 shows that Shortcut achieves the best advantage over AHK⁺ in *Password Reuse* when insert : delete = 0. In this case, Shortcut is 12.5× faster than AHK⁺ for one query after a deletion. The performance crossover occurs when the number of accumulated deletion rows is 15. In addition, Table 3 lists the performance crossover points of Shortcut over Secrecy and AHK⁺ respectively, in different ratios of mixed insertion rows and deletion rows.

Storage overhead. Table 4 lists the storage of Shortcut's materialized tables on real-life queries after inserting 10^3, 10^4, and 10^5 new rows to the underlying databases. Several tens of megabytes of storage will not be a barrier to the application of Shortcut.

8 Related Work

General-purpose MPC. Many general-purpose MPC frameworks are proposed to support secure computation in semi-honest [14, 23] and malicious [21, 24, 38] settings. Shortcut can build upon any general-purpose MPC framework to support different security models since we invoke the basic circuit evaluation protocols in a black-box manner.

Table 3: The performance crossover point of Shortcut over Secrecy or AHK⁺, i.e., the maximum number of accumulated update messages that make Shortcut slower than Secrecy or AHK⁺, in different ratios of insert : delete.

Baseline	Query	Perf. Crossover Point insert : delete 7:3	5:5	3:7
Secrecy [22]	Password R.	119	120	120
	Comorbidity	41	44	47
	Credit S.	275	239	211
	Logistics E.	109	103	99
AHK⁺ [5]	Password R.	14	14	14
	Comorbidity	4	5	5
	Credit S.	30	26	23
	Logistics E.	12	12	11

Table 4: Materialized tables size (MB) after data insertions.

Inserted Rows	Password R.	Comorbidity	Credit S.	Logistics E.
10^3	48.27	36.09	48.18	48.12
10^4	50.74	36.91	49.83	49.22
10^5	75.46	45.15	66.31	60.2

MPC-based collaborative analytics systems. MCASs provide a set of SQL interfaces with cryptographic guarantees. SMCQL [8], Senate [28] and Conclave [33] reduce the usage of MPC by splitting the query plan into plaintext part and ciphertext part. Senate also uses local computation as much as possible, but more importantly, it proposes a technique that decomposes circuits among parties to reduce the size of circuit input. Secrecy [22] optimizes the query cost via reordering SQL operators based on their compositing characteristics. Some works [5, 6, 19, 25] introduce asymptotically efficient protocols for SQL operators. In addition, some works [9, 10] focus on reducing the workload of MPC by allowing more information leakage that is well-bounded by DP. However, these MCASs are originally designed for static databases and exhibit limitations in dynamic database scenarios due to their redundant cost. Our Shortcut framework can work with MCASs to mitigate their limitations and enable efficient collaborative analytics on dynamic databases.

Collaborative analytics systems on incremental databases. Recently, several systems have provided query capabilities for incremental database [13, 35, 41], i.e., the insertion-only dynamic database. IncShrink [35] and Longshot [41] maintain previous query results to achieve efficient queries. These systems support only a few simple query functions and lack the capabilities of data deletion or data update that are provided in Shortcut.

Other techniques. Systems based on trusted hardware [7, 16, 29, 42] suffer from side-channel attacks [32, 37]. Some systems [26, 27] allow data leakage bounded by DP and suffer from query errors.

9 Conclusion

In this work, we present the Shortcut framework that works with MPC-based collaborative analytics systems to enable efficient queries on dynamic databases and support a wide range of query

functions. This is achieved by our composable query result update protocols that support data insertion, deletion, and update on previous query results. Furthermore, the proposed constant-depth inserting circuit and push-down circuit may be of independent interest. We hope this work will inspire researchers to further explore the topic of leveraging intermediate results to accelerate secure computations.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. The work is supported in part by the National Natural Science Foundation of China (No.62302242, No.62272251, No. 62032012).

References

[1] [n. d.]. TPC-H Benchmark. https://www.tpc.org/tpch/
[2] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. 2018. Generalizing the SPDZ compiler for other protocols. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 880–895.
[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 805–817.
[4] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 125–138.
[5] Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. 2023. Secure statistical analysis on multiple datasets: Join and group-by. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3298–3312.
[6] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. 2022. Secret-shared joins with multiplicity from aggregation trees. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 209–222.
[7] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 205–216.
[8] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: Secure Querying for Federated Databases. *Proceedings of the VLDB Endowment* 10, 6 (2017), 673–684.
[9] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018).
[10] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.
[11] Marina Blanton and Chen Yuan. 2022. Binary search in secure somputation. In *NDSS*.
[12] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.
[13] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2450–2468.
[14] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.
[15] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 523–535.
[16] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment* 13, 2 (2019), 169–183.
[17] Dengguo Feng and Kang Yang. 2022. Concretely efficient secure multi-party computation protocols: survey and more. *Security and Safety* 1 (2022), 2021001.
[18] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.
[19] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable collaborative analytics system on private database with malicious security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1740–1753.
[20] Carmit Hazay and Yehuda Lindell. 2010. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media.
[21] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.
[22] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1031–1056.
[23] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376.
[24] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 35–52.
[25] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast database joins and PSI for secret shared data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1271–1287.
[26] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially private join queries over distributed databases. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 149–162.
[27] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. 2017. DStress: Efficient differentially private computations on distributed data. In *Proceedings of the Twelfth European Conference on Computer Systems*. 560–574.
[28] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: a Maliciously-Secure MPC platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*. 2129–2146.
[29] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.
[30] General Data Protection Regulation. 2018. General data protection regulation (GDPR). *Intersoft Consulting, Accessed in October* 24, 1 (2018).
[31] Yongxin Tong, Xuchen Pan, Yuxiang Zeng, Yexuan Shi, Chunbo Xue, Zimu Zhou, Xiaofei Zhang, Lei Chen, Yi Xu, Ke Xu, et al. 2022. Hu-fu: Efficient and secure spatial queries over data federation. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1159.
[32] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.
[33] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.
[34] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2021. Dp-sync: Hiding update patterns in secure outsourced databases with differential privacy. In *Proceedings of the 2021 International Conference on Management of Data*. 1892–1905.
[35] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: architecting efficient outsourced databases using incremental mpc and differential privacy. In *Proceedings of the 2022 International Conference on Management of Data*. 818–832.
[36] Ke Coby Wang and Michael K Reiter. 2018. How to end password reuse on the web. *arXiv preprint arXiv:1805.00566* (2018).
[37] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.
[38] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit.
[39] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*. 1969–1981.
[40] Kang Yang, Xiao Wang, and Jiang Zhang. 2020. More efficient MPC from improved triple generation and authenticated garbling. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1627–1646.
[41] Yanping Zhang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2023. Longshot: Indexing Growing Databases Using MPC and Differential Privacy. *Proceedings of the VLDB Endowment* 16, 8 (2023), 2005–2018.
[42] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.

A Security Proof

In this section, we prove the security of the QRU protocols based on Definition 6.1. To simplify the expression, we assume that the SQL operators associated with each QRU protocol are not output operators. When dealing with output operators, security can be easily obtained by omitting parts of the subfunctionalities calls.

Theorem 6.1. *According to Definition 6.1, Protocol 1, Protocol 3, Protocol 5, Protocol 6, Protocol 7, Protocol 8, Protocol 9, and Protocol 10 jointly securely compute deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

Proof. This theorem is derived from the combination of Lemma A.1, Lemma A.2, Lemma A.3, Lemma A.4, Lemma A.5, Lemma A.6, Lemma A.7, Lemma A.8, Lemma A.9, and Lemma A.10. □

Lemma A.1. *According to Definition 6.1, Protocol 1 securely computes command* insert_obl *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

Proof. In this protocol, the parties hold no private inputs and obtain no output. The implication is that it is required that no information about private update row u^r and table D is revealed to the corrupted party during the protocol execution. This input formulation also means that, upon protocol initiation, the input shares available to the corrupted party are uniformly distributed at random, i.e., they information-theoretically reveal no information about the underlying input. We simulate the input shares upon protocol initiation by distributing random elements to the corrupted party and proceed to build simulator $\mathcal{S}$ as follows:

(1) In line 3, the trusted party computes subfunctionality comp. $\mathcal{S}$ simulates the messages received by the corrupted party from comp.
(2) In line 4 and line 5, the trusted party computes subfunctionality mult once each. $\mathcal{S}$ simulates the messages received by the corrupted party from mult.
(3) In line 7, the trusted party first computes subfunctionalities comp and mult h times, then computes comp and mult h times again. $\mathcal{S}$ simulates the messages received by the corrupted party from comp and mult.

Now, we need to show that the real and simulated views are indistinguishable. The first component of the view is the input shares available to the corrupted party, which has identical distributions (i.e., distributed uniformly at random) in the real and simulated worlds, making this component indistinguishable. Second, according to the modular sequential composition theorem, the messages received by the corrupted party from the subfunctionalities comp and mult have identical distribution to the messages received from the secure protocols computing these subfunctionalities. Therefore, we can say that the simulator-generated view of the corrupted party is indistinguishable from that of a real execution.

Note that because this protocol produces no private output to the corrupted party, there is no need to demonstrate the output correctness in the simulated view. However, if this protocol is followed by another operation that discloses a value to the corrupted party (e.g., an open operation), it will always be possible for the simulator to generate the view that will lead to the corrupted party reconstructing the right output. This is due to the properties of three-party replicated secret sharing, which allow the shares held by a single party to reconstruct to any possible value in the ring. □

The security of other QRU protocols is proved in a similar skill.

Lemma A.2. *According to Definition 6.1, Protocol 3 securely computes command* delete_obl *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

Proof. The parties hold no private inputs and obtain no output. In this protocol, multiplication, comparison, and equality are computed via subfunctionalities. As before, the input shares of a single party are distributed uniformly at random and can be reconstructed to any possible values with the same probability. Thus we begin with simulating the input shares by distributing random elements to the corrupted party.

We proceed to build simulator $\mathcal{S}$ as follows:

(1) In line 3, the trusted party repeats h times: computes eq, mult, and then mult. $\mathcal{S}$ simulates the messages received by the corrupted party from these subfunctionalities.
(2) In line 6 and line 7, the trusted party computes subfunctionality mult once each. $\mathcal{S}$ simulates the messages received by the corrupted party from mult.
(3) In line 9, the trusted party first computes mult h times, then computes comp and mult $h-1$ times. $\mathcal{S}$ simulates the messages received by the corrupted party from these subfunctionalities.

The simulated view now consists of the input shares available to the corrupted party and the messages received by the corrupted party from eq, comp, and mult. As before, the input shares in both real and simulated views are random elements, which are therefore indistinguishable. In addition, because the protocols in the real execution that compute the subfunctionalities are secure, the simulator is guaranteed to generate messages indistinguishable from the real ones. Thus, all components of the view are indistinguishable, and as a result, the overall simulation is indistinguishable as well. □

Lemma A.3. *According to Definition 6.1, Protocol 5 securely computes command* insert_gbs *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

Proof. The parties hold no private inputs and obtain no output. In this protocol, multiplication and equality are computed via subfunctionalities. Simulator $\mathcal{S}$ first distributes random elements to the corrupted party to simulate the input shares, and continues to work as follows:

(1) In line 3, the trusted party repeats n times: computes eq, mult, and then mult. $\mathcal{S}$ simulates the messages received by the corrupted party from these subfunctionalities.
(2) In line 5, the trusted party computes mult and $\mathcal{S}$ simulates the messages sent back.
(3) In line 7, the trusted party computes mult n times. $\mathcal{S}$ simulates the messages sent back.
(4) In line 9, the trusted party also computes mult n times. $\mathcal{S}$ simulates the messages sent back.

The simulated view now consists of the input shares available to the corrupted party and the messages received by the corrupted party from eq and mult. As before, the input shares in both real and simulated views are random elements, which are therefore indistinguishable. Since the protocols computing eq and mult are

secure, the simulator is guaranteed to generate messages indistinguishable from the real ones. Therefore, the overall simulated view of the corrupted party is indistinguishable from that of a real execution. □

Lemma A.4. *According to Definition 6.1, Protocol 6 securely computes command* delete_gbs *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

PROOF. The parties hold no private inputs and obtain no output. As before, we simulate the input shares by distributing random elements to the corrupted party and continue to build simulator $\mathcal{S}$ as follows:

(1) In line 3, the trusted party repeats n times: computes eq, mult, and then mult. $\mathcal{S}$ simulates the messages sent back.
(2) In line 5, the trusted party computes mult n times. $\mathcal{S}$ simulates the messages sent back.
(3) In line 7, the trusted party also computes mult n times. $\mathcal{S}$ simulates the messages sent back.

As before, the input shares in both real and simulated views have identical distributions. Moreover, the simulator is guaranteed to generate messages sent back from eq and mult to be indistinguishable from the real ones. Thus, the simulator-generated view of the corrupted party is indistinguishable from that of a real execution. □

Lemma A.5. *According to Definition 6.1, Protocol 7 securely computes command* insert_join *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

PROOF. The parties hold no private inputs and obtain no output. As before, simulator $\mathcal{S}$ first distributes random elements to the corrupted party to simulate the input shares, and proceeds to work as follows:

(1) In line 4, the trusted party repeats m times: computes eq and mult. $\mathcal{S}$ simulates the messages sent back.
(2) In line 6, the trusted party computes mult m times. $\mathcal{S}$ simulates the messages sent back.
(3) In line 7, the trusted party computes mult and $\mathcal{S}$ simulates the messages sent back.

The input shares in both real and simulated views have identical distributions. In addition, the simulator is guaranteed to generate messages sent back from eq and mult to be indistinguishable from the real ones. Thus, the simulator-generated view of the corrupted party is indistinguishable from that of a real execution. □

Lemma A.6. *According to Definition 6.1, Protocol 8 securely computes command* delete_join *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

PROOF. The parties hold no private inputs and obtain no output. As before, we simulate the input shares by distributing random elements to the corrupted party and continue to build simulator $\mathcal{S}$ as follows:

(1) In line 3, the trusted party repeats m times: computes eq, mult, and then mult. $\mathcal{S}$ simulates the messages sent back.
(2) In line 5, the trusted party repeats n times: computes eq, mult, and then mult. $\mathcal{S}$ simulates the messages sent back.
(3) In line 8 and line 12, the trusted party computes mult n times each. $\mathcal{S}$ simulates the messages sent back.

The input shares in both real and simulated views have identical distributions. In addition, the simulator is guaranteed to generate messages sent back from eq and mult to be indistinguishable from the real ones. Therefore, the simulator-generated view of the corrupted party is indistinguishable from that of a real execution. □

Lemma A.7. *According to Definition 6.1, the line 1-6 of Protocol 9 securely computes command* insert_select *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

PROOF. The parties hold no private inputs and obtain no output. Simulator $\mathcal{S}$ begins with simulating the input shares by distributing random elements to the corrupted party. In line 2, the trusted party computes (1) the subfunctionalities used to implement predicate σ, such as eq or comp, and (2) then mult. $\mathcal{S}$ simulates the messages sent back.

As before, indistinguishability follows from (1) information-theoretic security of the replicated secret sharing scheme in the presence of at most one corrupted party and (2) security of protocols (that compute the subfunctionalities) which must result in indistinguishable views from the simulated ones. □

Lemma A.8. *According to Definition 6.1, the line 7-12 of Protocol 9 securely computes command* delete_select *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

PROOF. Simulator $\mathcal{S}$ begins with simulating the input shares by distributing random elements to the corrupted party. In line 9, the trusted party repeats n times: computes eq and then mult. $\mathcal{S}$ simulates the messages sent back.

As before, indistinguishability follows from (1) information-theoretic security of the replicated secret sharing scheme in the presence of at most one corrupted party and (2) security of protocols (that compute the subfunctionalities) which must result in indistinguishable views from the simulated ones. □

Lemma A.9. *According to Definition 6.1, the line 1-3 of Protocol 10 securely computes command* insert_ga *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

PROOF. The parties hold no private inputs and obtain no output. Simulator $\mathcal{S}$ first distributes random elements to the corrupted party to simulate the input shares. Because the replicated secret sharing scheme is information-theoretically secure in the presence of at most one corrupted party, the simulated and the real input shares are indistinguishable. Next, if the protocol realizes a Max/Min-based aggregation, the trusted party computes comp and then mult. $\mathcal{S}$ simulates the messages sent back. The security of protocols that compute the above subfunctionalities ensures the indistinguishability of the simulated and the real views. Therefore, the whole simulated view is indistinguishable from the real view. □

Lemma A.10. *According to Definition 6.1, the line 4-6 of Protocol 10 securely computes command* delete_ga *of the deterministic functionality $\mathcal{F}_{QRU}$ in the $\mathcal{F}_{CE}$-hybrid model.*

PROOF. The parties hold no private inputs and obtain no output. This protocol only involves local subtraction. Therefore, the

real view contains no information. This protocol is secure because the simulator has nothing to simulate, and the simulated view is indistinguishable from the real view, of cause. ◻

B Real-word Queries

Here we list the query statements used in Section 7.2.

Comorbidity:

```
SELECT diag, COUNT(*) cnt
FROM diagnoses
WHERE patient_id IN cdiff_cohort
GROUP BY diag
ORDER BY cnt LIMIT 10
```

Credit Scores:

```
SELECT c1.ssn
FROM credit_scores|P1 AS c1
...
JOIN credit_scores|Pm AS cm ON c1.ssn = cm.ssn
WHERE GREATEST(c1.credit, ..., cm.credit) -
  LEAST(c1.credit, ..., cm.credit) > threshold
  AND c1.year = 2019 ... AND cm.year = 2019
```

Logistics Efficiency:

```
SELECT PID, distance
FROM Sales JOIN Deliverys ON Sales.PID = Deliverys.PID
WHERE Sales.DeliverDate - Deliverys.CompleteDate >= 10
ORDER BY distance LIMIT 20
```

C Analysis of Materialization Strategy

Recall that SHORTCUT materializes the top h rows of the ordered input table R^s as D, where $h = l + d$, l is the limit size, d is the delete threshold. We aim to prove that, the query result of Order-by-Limit will not introduce any errors, provided the number of invocations of the delete protocol (Protocol 3) does not exceed d. The conditions under which errors are introduced are as follows: (i) there are valid rows are discarded (i.e., not materialized) during the process, and (ii) subsequent updates result in the number of valid rows in D being less than l. Errors are introduced because the discarded valid rows, which should be used to patch the top l rows of D at this point, cannot be included due to their removal.

We can now turn to prove that, when valid rows are discarded, subsequent updates always ensure that the number of valid rows in D remains greater than or equal to l, provided the number of invocations of the delete protocol does not exceed d. In the following, we introduce Theorem C.2 via a gradual progression to prove this property.

Notations. $\|D[v]\|$ denotes the L_1 norm of the vector $D[v]$, which represents the number of valid rows in D. We use $del^{(t)}(D)$ to denote that performing t consecutive delete protocols on D, $ins^{(t)}(D)$ denotes that performing t consecutive insert protocols (Protocol 1) on D, and $upd^{(t)}(D)$ denotes that performing arbitrary t updates involving insertion and deletion on D.

Property C.1. *The delete protocol decreases at most one valid row from D. Thus, after t consecutive deletions, there is $\left\|del^{(t)}(D)[v]\right\| \geq \|D[v]\| - t$.*

Property C.2. *The insert protocol may increase, but not decrease the valid rows on D. Therefore, after t consecutive insertions, there is $\left\|ins^{(t)}(D)[v]\right\| \geq \|D[v]\|$.*

Theorem C.1. *After t_1 consecutive deletions and then t_2 consecutive insertions, there is $\left\|ins^{(t_2)}(del^{(t_1)}(D))[v]\right\| \geq \|D[v]\| - t_1$.*

PROOF. According to Property C.1 and Property C.2, we have

$$\left\|ins^{(t_2)}(del^{(t_1)}(D))[v]\right\| \geq \left\|del^{(t_1)}(D)[v]\right\| \geq \|D[v]\| - t_1$$

◻

Theorem C.2. *Let D be any materialized table at the moment when valid rows are discarded. After arbitrary $a + b$ updates involving insertion and deletion, where $a \leq d$ is the total deletion times and b is the total insertion times, we always have $\left\|upd^{(a+b)}(D)[v]\right\| \geq l$.*

PROOF. At the moment when valid rows are discarded, D consists entirely of valid rows, without any dummy rows, i.e., $\|D[v]\| = h$. The subsequent arbitrary $a + b$ updates can be represented as the composition of consecutive delete protocols followed by consecutive insert protocols:

$$\begin{aligned} D^1 &= ins^{(b_1)}(del^{(a_1)}(D)) \\ D^2 &= ins^{(b_2)}(del^{(a_2)}(D^1)) \\ &\cdots \\ upd^{(a+b)}(D) &= ins^{(b_n)}(del^{(a_n)}(D^n)) \end{aligned}$$

where $a = a_1 + a_2 + ... + a_n$, $b = b_1 + b_2 + ... + b_n$. According to Theorem C.1,

$$\begin{aligned} \left\|upd^{(a+b)}(D[v])\right\| &= \left\|ins^{(b_n)}(del^{(a_n)}(D^n[v]))\right\| \\ &\geq \left\|D^n[v]\right\| - a_n \geq \left\|D^{n-1}[v]\right\| - (a_n + a_{n-1}) \geq \ldots \\ &\geq \|D[v]\| - (a_n + a_{n-1} + \ldots + a_1) \\ &\geq h - a \geq h - d = l \end{aligned}$$

◻

D Functionalities

Functionality $\mathcal{F}_{\text{QRU}}$

Insert_OBL: On input (insert_obl, rowid_0, rowid_1, rowid_2, tabid_0, tabid_1) from all parties, the functionality retrieves (rowid_0, u^r) and (tabid_0, D) from memory, where D is descendingly ordered by v and then descendingly ordered by k. The functionality first stores $(\text{tabid}_1, \text{sort}_{v\uparrow k\uparrow}(D \cup u^r))$, where $\text{sort}_{v\uparrow k\uparrow}$ represents sorting the two-dimensional table by the descending order of v and then by the descending order of k. Then, if $u^r[v \parallel k]$ is greater than $D_{l-1}[v \parallel k]$, the functionality stores $(\text{rowid}_1, D_{l-1})$ and (rowid_2, u^r); otherwise, the functionality stores $(\text{rowid}_1, \textit{dummy})$ and $(\text{rowid}_2, \textit{dummy})$.

Delete_OBL: On input (delete_obl, rowid_0, rowid_1, tabid_0, tabid_1) from all parties, the functionality retrieves (rowid_0, u^r) and (tabid_0, D) from memory, where D is descendingly ordered by v and then descendingly ordered by k. If u^r is a valid row and there exists one and only one valid row $t \in D$, such that $t[id]$ equals $u^r[id]$, the functionality stores $(\text{tabid}_1, (D \setminus t) \cup \textit{dummy})$; otherwise, the functionality stores (tabid_1, D). Then, if u^r is a valid row and there exists one and only one valid row $t \in D_{0,1,\dots,l-1}$, such that $t[id]$ equals $u^r[id]$, the functionality stores (rowid_1, u^r); otherwise, the functionality stores $(\text{rowid}_1, \textit{dummy})$.

Insert_GBS: On input (insert_gbs, rowid_0, rowid_1, rowid_2, rowid_3, tabid_0, tabid_1) from all parties, the functionality retrieves (rowid_0, u^r), (tabid_0, G) from memory.

(1) If u^r is a valid row and there exists one and only one valid row $t \in G$, such that $t[k]$ equals $u^r[k]$, the functionality (i) first stores (rowid_1, t), (ii) then sets $t[r] = t[r] + u^r[a]$ and stores (rowid_2, t), (iii) then sets $u^r[v] = 0$ and stores (rowid_3, u^r). Otherwise, the functionality directly stores $(\text{rowid}_1, \textit{dummy})$, $(\text{rowid}_2, \textit{dummy})$ and (rowid_3, u^r).
(2) Finally, the functionality stores $(\text{tabid}_1, G \cup u^r)$.

Delete_GBS: On input (delete_gbs, rowid_0, rowid_1, rowid_2, tabid_0, tabid_1) from all parties, the functionality retrieves (rowid_0, u^r), (tabid_0, G) from memory.

(1) If u^r is a valid row and there exists one and only one valid row $t \in G$, such that $t[k]$ equals $u^r[k]$, the functionality (i) first stores (rowid_1, t), (ii) then sets $t[r] = t[r] - u^r[a]$ and stores (rowid_2, t). Otherwise, the functionality directly stores $(\text{rowid}_1, \textit{dummy})$ and $(\text{rowid}_2, \textit{dummy})$.
(2) Finally, the functionality stores (tabid_1, G).

Insert_Join: On input (insert_join, rowid_0, rowid_1, tabid_0, tabid_1, tabid_2, tabid_3, tabid_4) from all parties, the functionality retrieves (rowid_0, u^r), (tabid_0, A), (tabid_1, B), (tabid_2, J) from memory. If there exists one and only one valid row $t \in B$, such that $t[k]$ equals $u^r[k]$, the functionality stores $(\text{tabid}_3, A \cup u^r)$, $(\text{tabid}_4, J \cup (u^r \cup t[p]))$ and $(\text{rowid}_1, u^r \cup t[p])$; otherwise, the functionality stores $(\text{tabid}_3, A \cup u^r)$, $(\text{tabid}_4, J \cup \textit{dummy})$ and $(\text{rowid}_1, \textit{dummy})$.

Delete_Join: On input (delete_join, rowid_0, rowid_1, tabid_0, tabid_1, tabid_2, tabid_3, tabid_4) from all parties, the functionality retrieves (rowid_0, u^r), (tabid_0, A), (tabid_1, B), (tabid_2, J) from memory.

(1) If u^r is valid and there exists one and only one valid row $t \in A$, such that $t[k]$ equals $u^r[k]$, the functionality sets t as a dummy row.
(2) If u^r is a valid row and there exists one and only one valid row $s \in J$, such that $s[k]$ equals $u^r[k]$, the functionality stores (rowid_1, s) and then sets s as a dummy row; otherwise the functionality stores $(\text{rowid}_1, \textit{dummy})$.
(3) Finally, the functionality stores (tabid_3, A) and (tabid_4, J).

Insert_Select: On input (insert_select, rowid_0, rowid_1, tabid_0, tabid_1) from all parties, the functionality retrieves (rowid_0, u^r) and (tabid_0, S) from memory. The functionality computes $u^r[v] = u^r[v] \cdot \sigma(u^r)$, then stores $(\text{tabid}_1, S \cup u^r)$ and (rowid_1, u^r).

Delete_Select: On input (delete_select, rowid_0, rowid_1, tabid_0, tabid_1) from all parties, the functionality retrieves (rowid_0, u^r) and (tabid_0, S) from memory. If there exists row $t \in S$, such that $t[id]$ equals $u^r[id]$, the functionality sets t as a dummy row. The functionality then stores (tabid_1, S) and (rowid_1, u^r).

Insert_GA: On input (insert_ga, rowid, varid_0, varid_1) from all parties, the functionality retrieves (rowid, u^r) and (varid_0, z) from memory. The functionality computes $z = z + u^r[a]$ for Sum-based aggregation, $z = z + u^r[v]$ for Count-based aggregation, $z = \max(z, u^r[a])$ for Max-based aggregation, and $z = \min(z, u^r[a])$ for Min-based aggregation. The functionality stores (varid_1, z).

Delete_GA: On input (delete_ga, rowid, varid_0, varid_1) from all parties, the functionality retrieves (rowid, u^r) and (varid_0, z) from memory. The functionality computes $z = z - u^r[a]$ for Sum-based aggregation, and $z = z - u^r[v]$ for Count-based aggregation. The functionality stores (varid_1, z).

Figure 15: The ideal functionality of query result update protocols.

Functionality $\mathcal{F}_{\mathrm{CE}}$

Multiplication: On input ($\mathsf{mult}, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3$) from all parties, the functionality retrieves (varid_1, x), (varid_2, y) from memory and stores ($\mathsf{varid}_3, x \cdot y \bmod \mathbb{Z}_{2^\ell}$).
Equality: On input ($\mathsf{eq}, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3$) from all parties, the functionality retrieves (varid_1, x), (varid_2, y) from memory and stores ($\mathsf{varid}_3, x =? y$).
Comparison: On input ($\mathsf{comp}, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3$) from all parties, the functionality retrieves (varid_1, x), (varid_2, y) from memory and stores ($\mathsf{varid}_3, x < y$), ($\mathsf{varid}_3, x > y$), ($\mathsf{varid}_3, x \leq y$) or ($\mathsf{varid}_3, x \geq y$) according to specific instructions.

Figure 16: The ideal functionality of circuit evaluation protocols.