Alchemy: A Query Optimization Framework for Oblivious SQL

Donghyun Sohn
Northwestern University
donghyun.sohn@u.northwestern.edu

Kelly Jiang
Northwestern University
kellyjiang2022@u.northwestern.edu

Nicolas Hammer
Northwestern University
nicolashammer2021@u.northwestern.edu

Jennie Rogers
Northwestern University
jennie@northwestern.edu

ABSTRACT

Data sharing opportunities are everywhere, but privacy concerns and regulatory constraints often prevent organizations from fully realizing their value. A private data federation tackles this challenge by enabling secure querying across multiple privately held data stores where only the final results are revealed to anyone. We investigate optimizing relational queries evaluated under secure multiparty computation, which provides strong privacy guarantees but at a significant performance cost.

We present Alchemy, a query optimization framework that generalizes conventional optimization techniques to secure query processing over circuits, the most popular paradigm for cryptographic computation protocols. We build atop VaultDB, our open-source framework for oblivious query processing. Alchemy leverages schema information and the query's structure to minimize circuit complexity while maintaining strong security guarantees. Our optimization framework builds incrementally through four synergistic phases: (1) rewrite rules to minimize circuits; (2) cardinality bounding with schema metadata; (3) bushy plan generation; and (4) physical planning with our fine-grained cost model for operator selection and sort reuse. While our work focuses on MPC, our optimization techniques generalize naturally to other secure computation settings. We validated our approach on TPC-H, demonstrating speedups of up to 2 OOM.

PVLDB Reference Format:
Donghyun Sohn, Kelly Jiang, Nicolas Hammer, and Jennie Rogers. Alchemy: A Query Optimization Framework for Oblivious SQL. PVLDB, 18(9): 3021 - 3034, 2025.
doi:10.14778/3746405.3746425

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/vaultdb/alchemy.

1 INTRODUCTION

Over the past decade, a cornucopia of privacy-enhancing technologies (PETs) has matured in the cryptography research community, offering ways to query data across private sources while protecting sensitive inputs securely. Many offer ways to securely query data from one or more private sources such that the input records are not divulged to anyone. These techniques "blindly" compute over their input records, usually only revealing their query answer.

The private data federation (PDF) has emerged as a promising model for enabling organizations to query the union of their private datasets collaboratively while revealing only the query answers. These platforms perform secure query evaluation either over cryptographic protocols [8, 37, 52] or in trusted hardware [15, 57]. This work focuses on SQL over cryptographic protocols.

Many PDFs use secure multiparty computation (MPC) for query evaluation. Members compute queries by passing encrypted messages among themselves to simulate a completely hypothetical trustworthy third party evaluating the query over their combined data. These cryptographic protocols take a two-pronged approach to protecting the privacy of their inputs. First, they protect the data during computation by compiling the query's logic into circuits, the dominant paradigm for cryptographic computing protocols. These circuits encode their wire labels as secret shares to prevent eavesdropping on the query's inputs or intermediate results. Second, the transcripts of MPC circuits are oblivious. Their program traces, network communication, and memory access patterns are independent of their private inputs.

These strong security guarantees exact an extremely steep performance penalty, resulting in query runtimes *several orders of magnitude* slower than computing them insecurely. Worse yet, their impact is cumulative for complex oblivious queries with tall operator trees because they cannot leak the selectivity of intermediate results. They pad individual operator outputs with dummy tuples. For example, a query with two joins where all three input tables have 100 tuples each would have an output cardinality of 1M rows!

Recently, there has been substantial practical interest in bringing MPC into real-world deployments. For example, MPC has been used to identify repeat perpetrators of sexual assault [45], for SQL-style querying of electronic health records for clinical research [46], for quantifying the gender wage gap [33], and for privacy-preserving COVID-19 exposure notification [3]. In addition, industry has invested substantial work in making MPC easy to deploy on the public cloud [17, 23, 29, 49], and this technology is being evaluated in numerous fields [5, 10–12, 27].

There is an impedance mismatch between how cryptographers develop efficient MPC protocols, working in a circuit paradigm, and how relational databases express and optimize their queries as random access machines. Oblivious security guarantees are antagonistic to conventional query optimization goals. For example, we cannot leak row-level information to the computing parties, such as the offsets of tuples from a filter or the result of a comparison.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746425

To bridge this gap, we exploit the federation's shared schema and characteristics of oblivious database operator protocols to make oblivious querying more efficient. Since these queries reveal no information in their transcript, it might seem they have very limited optimization opportunities at first glance. However, we observe that the federation's shared schema–its keys, constraints, and column domains–offers public information we may use to reduce the complexity of our oblivious circuits. For example, we can bound the cardinality of a PK-FK join to the length of the FK relation without divulging new information.

We introduce Alchemy, a query optimization framework for oblivious SQL statements in the circuit paradigm. Alchemy leverages schema metadata to craft efficient oblivious query execution plans using two strategies. First, we push as much work as possible out of circuits (and into local evaluation) while upholding our security guarantees. Second, we reduce the complexity of our circuits for subtrees that call for secure evaluation.

Alchemy builds atop VaultDB [46], our open-source oblivious query execution engine. This distributed analytics system provides a suite of oblivious database operators from the literature. It is circuit-agnostic, and its generic oblivious database operators work modularly with one or more cryptographic backends. VaultDB supports semi-honest 2PC [46], maliciously secure MPC [35], and zero-knowledge proofs [36]. We use the terms MPC and 2PC interchangeably in this text.

Alchemy extends several well-known query optimization principles to circuit-based query optimization. Its rewrite rules migrate predicates to reduce our intermediate cardinality and to push sorts outside circuits. Alchemy maintains oblivious statistics from schema constraints and public metadata to tightly bound sizes of its intermediate result (IR). Its bushy plan generator tames the cumulative pile-up of dummy tuples in oblivious operator trees. Lastly, it uses a circuit-based analytical cost model to guide operator selection and sort reuse across secure protocols. Whereas prior work in oblivious query processing uses heuristics [8, 15, 52] or parallelizes one operator at a time [42], we take a more holistic approach by modeling attributes as they flow through the query tree to incorporate optimizations such as sort reuse, predicate migration, and cardinality bounds input our plans. Our main contributions are:

- An optimization framework that minimizes circuit complexity for oblivious SQL.
- Deriving and validating an analytical cost model for estimating the runtime of oblivious database operators.
- Verifying the concrete efficiency of Alchemy's optimized plans with a robust experimental evaluation.

The rest of this paper is structured as follows. Section 2 reviews the building blocks of Alchemy. Section 3 introduces our system architecture and security model. We detail our optimization framework in Section 4. In Section 5, we present experiments from our prototype on TPC-H, highlighting improvements in performance, scalability, and robustness across various cryptographic protocols. We then survey relevant literature in Section 6 and conclude.

2 BACKGROUND

We now describe key features of secure computation in the context of Alchemy and the building blocks of oblivious query optimization.

2.1 Secure Circuits

Alchemy processes queries in the circuit model, expressing them as a series of gates it evaluates in topological order. Computing parties exchange encrypted messages to process each gate. This gate-by-gate structure, common across secure computation systems such as MPC, simplifies algorithm design by implementing one protocol per gate type (e.g., AND, NOT). This approach is Turing complete and thus suitable for ad-hoc computing like query evaluation.

By default, VaultDB operates under semi-honest security, where parties execute protocols faithfully but may attempt to learn unauthorized information from observing a program execution and IR sizes. While we primarily discuss two-party computation between Alice and Bob, our experimental results demonstrate how Alchemy's optimization techniques extend to settings with additional parties and varied security guarantees. Data providers prepare for secure computation using *oblivious transfer*, a process that generates *secret shares* of the query's private input data. This is analogous to distributing encrypted copies of the private input data where no party holds the key to decrypt it. This sharing process incurs minimal computational overhead, with communication costs proportional to data size.

2.2 Oblivious Query Evaluation

VaultDB prevents information leakage by adapting its operator processing in several ways. It adapts the cardinality of its IRs not to leak side channel information and obliviously pads them with dummy rows as needed. It uses a common data model (CDM) to make all rows in secret-shared tables indistinguishable, padding string fields to maximum length. To prevent leaking statistics like join selectivity or aggregate group counts, it pads IRs with dummy tuples, indicated by a secret-shared dummy tag bit. Operators manipulate these tags during evaluation. For example, setting the tag to true on tuples that do not satisfy a filter predicate. Our operators output their worst-case cardinality by default, although Alchemy exploits schema properties to derive tighter bounds and thus lower the cost of parent operators.

VaultDB abstracts away its expression evaluation over circuits in a `Field` object. We parse expressions into an AST that runs over `Field` instances. VaultDB builds atop EMP Toolkit [53], which provides boolean circuits for comparison ($\neq, =$), logic operators ($\oplus, \wedge, \vee, \neg$), and `mux`. It also offers circuits for some math operations. `Field` offers a wrapper for circuits that is protocol-agnostic, so any library that offers similar logic gates can be adapted to it. `Field` provides comparators ($=, \neq, <, \leq, >\geq$) for integers, floats, strings, and bools. For integers and floats, `Field` overloads math expressions ($+, -, *, /$). For each of these operations, it takes secret shares as inputs and returns them as results.

To translate our operators into circuits, we create parameterized versions of their logic, similar to conventional DBMS ones. There are two levels of circuit conversion. First, to prevent branching on private values that reveal the selected path, we translate conditionals into oblivious compare-and-swap operations. We express this as an oblivious multiplexer gate, `mux`. We demonstrate this functionality below. Second, operators do not read from or write to private offsets. We sidestep this need by sorting so that the objects we compute together are adjacent to the table. This is an important primitive for efficient equi-joins and group-by aggregates.

Algorithm 1 Sketch of circuit logic for sort-merge aggregate.

```
1:  procedure SortMergeAggregate(R, gb, agg)
2:      R is obliviously sorted by gb columns
3:      Init output table S with |R| rows. Copy R's dummy tags to S
4:      r_i is the ith row ∈ R, s_{i,dummy} is s_i's dummy tag
5:      for i do := 1 ... |R|
6:          select := gb_match(r_{i-1}, r_i)
7:          s_{i-1,dummy} := mux(select, 1, s_{i-1,dummy})
8:          a := mux(select, s_{i-1,agg}, 0)
9:          update a with r_i's fields
10:         s_{i,agg} := a
11:     end for
12:     return S;
13: end procedure
```

We demonstrate this process with a sketch of an oblivious sort-merge aggregate in Algorithm 1. We loop through the input table R sorted on its group-by columns. When we visit row r_i, if it belongs to the same group as r_{i-1}, then we mark the $i-1$th output row as a dummy and copy its partial aggregate to a. Otherwise, we reset the value of a. We then update a with the values in r_i and write the new aggregate to s_i.

3 OVERVIEW

In this section, we formalize Alchemy's security guarantees. We then provide an overview of VaultDB's query processing workflow and a roadmap of how Alchemy's optimization framework identifies efficient query execution plans for it.

Notation Table 1 describes our notation. A query references R and S, tables from the CDM. r_i is the ith row in R and $r_{i,j}$ is the jth field in r_i. We refer to the secret shares of R as $[R]$, and all computing parties hold partitions of these shares. When we run a circuit-based database operator, both its inputs and outputs are secret shares. For example, if we ran a filter over $[R]$ in circuits, we would have $[S] := \sigma([R])$. When we evaluate an expression e in circuits, such as a filter predicate, we convert it to gates, $\hat{e}$, where $|\hat{e}|$ represents the length of the expression in gates, a proxy for circuit complexity. Each table R is bounded by maximum cardinality $|R_{\max}|$, and for any column x, we denote the theoretical domain as $\text{dom}(R.x)$ and represent the number of distinct values in x as $|\pi_x(R)|$.

3.1 Security Model

Clients query the data federation with a CDM. This shared schema defines the tables over which a client composes their queries and the format in which data providers input their private records. Users initialize the CDM before running their first query. To ensure effective collaboration among the parties, the CDM describes constraints, including PK-FK, relationships, `CHECK` constraints, and functional dependencies. Alchemy uses this system catalog to derive cardinality bounds for its query plans.

Alchemy's CDM has a security policy that defines what data is visible to the parties participating in the evaluation of Q. This informs what steps it must run obliviously. Stakeholders collaborate to create this policy using the best practices of their domain and regulations. A column is either *public*–visible to all and able to alter a query's control flow –or *private*–accessible at its site of origin alone and computed upon obliviously.

Table 1: Alchemy Notation.

Symbol	Description
R, S	Tables we query
$r_i, r_{i,j}$	ith row in R, jth field in r_i
$[R]$	Secret shares of R
$\mathcal{D}$	Set of input tables $\{R_1, \ldots, R_n\}$
$\mathcal{S}$	Shared schema across tables in $\mathcal{D}$
Q	Query to be run obliviously
$\hat{Q}$	Q's oblivious query plan (in circuits)
$\lvert\hat{e}\rvert$	Length of expression e in gates
$\{R\}$	Length of a row in R in bits
$\lvert R\rvert$	Cardinality of R
$\text{dom}(R.x)$	Domain of attribute $x \in R$

All parties know Q and observe the number of rows each party inputs to it at each leaf. Q is a read-only query, and admitting writes would fundamentally alter the security guarantees of Alchemy. We trust the system to faithfully translate SQL statements into their corresponding secure query execution plan. The cardinality of all tables on each party is public. This informs the size of our circuits and is consistent with the guarantees of other PDFs [8, 42, 52]. We expect data providers to provide correct and complete inputs. By default, the client sees their exact query results. If the data providers desire stronger data privacy guarantees, they may use differential privacy [19] to noise the results of their query in circuits before they reveal them to the client as in [31, 39].

When a query computes on multiple attributes, Alchemy operates at the security level of the most sensitive one. Recall that VaultDB uses semihonest protocols by default. Hence, we trust each data provider to faithfully execute the protocol specified in $\hat{Q}$, although they may try to learn about the input data of others by observing the protocol's execution in its program counters and memory accesses. Naturally, we extend this guarantee to $\hat{Q}$'s observable query metadata, such as IR cardinalities, and to operator algorithm selection.

3.1.1 Security Definition We now describe the security guarantees of our oblivious query evaluation wherein the participating parties do not learn anything about a query's private input data except that which can be deduced from its answer. Say that Alchemy is running query Q over $\mathcal{D} := \{R_1, \ldots, R_n\}$ with shared schema metadata $\mathcal{S}$, including the input cardinalities of each relation $\forall R_i \in \mathcal{D}$. Let $\hat{Q}$ be Q's circuit-based oblivious query execution plan. At runtime, VaultDB generates $\text{Trace}(\hat{Q}, \mathcal{S}, \mathcal{D})$, a transcript recording network traffic, I/O requests, and memory access patterns.

Consider a simulator, Sim, that takes in the same query execution plan and schema metadata but only the length of the private input records, $|\mathcal{D}|$. The simulator, without access to $\mathcal{D}$, produces a transcript indistinguishable from Trace.

Theorem 1. For private input database $\mathcal{D}$ evaluating query plan $\hat{Q}$ with schema metadata $\mathcal{S}$, there exists a polynomial time simulator Sim such that:

$$\text{Trace}(\hat{Q}, \mathcal{S}, \mathcal{D}) \stackrel{c}{=} \text{Sim}(\hat{Q}, \mathcal{S}, |D|)$$

The output traces of VaultDB's real-world query evaluation are *computationally indistinguishable* from the observed behavior of the ideal world execution in the simulator Sim.

This security guarantee holds because Alchemy is designed to expose only authorized metadata during query execution. All transformations applied during optimization preserve the semantics of Q and ensure that any observable behavior is derivable from the known metadata. Specifically, the only leakage permitted is the shared schema $\mathcal{S}$, the cardinalities $\mathcal{D}$, and any domain constraints explicitly marked as public. Since Alchemy's compiled plan $\hat{Q}$ executes entirely within circuits that are oblivious to data values and access patterns, all runtime behavior can be simulated from $(\hat{Q}, \mathcal{S}, |\mathcal{D}|)$. This allows us to construct a simulator Sim that satisfies Theorem 1 under the assumed adversary model.

Security Proof Although Alchemy does not modify the input table definitions or constraints in the CDM, it does make this public information visible to parties computing Q. Since all parties know Q, they learn about what aspects of the data the client is interested in, and if we filter on public attributes (such as `r_name` in Q5), then we divulge how many rows they learn about in their query answer. Owing to the circuit model making Q's logic visible, this is unavoidable. Once we are past the leaf nodes, the only additional information we reveal is from the cardinality bounds and schema constraints to guide operator selection. Because Alchemy only takes in public data to make its optimization decisions, it cannot reveal unauthorized information to the computing parties. Thus, the guarantees of the simulation hold for all Alchemy queries.

3.1.2 Backend Independence Our simulation-based security definition remains consistent across all cryptographic protocols supported by Alchemy. The core guarantee–namely, that the view of any adversary can be simulated from the query plan $\hat{Q}$, schema $\mathcal{S}$, and input cardinalities $|\mathcal{D}|$—holds regardless of circuit implementation. On the other hand, the strength of the adversary and the attacks we defend against depend on the circuit protocol. For instance, when using a zero-knowledge proof (ZKP) backend, we offer malicious security in a two-party setting with a prover and verifier. In contrast, semihonest two-party computation (e.g., SH2PC) assumes both parties follow the protocol faithfully. Our outsourced MPC (OMPC) mode supports two or more computing parties and tolerates a dishonest majority. Alchemy ensures that all plans are generated and executed securely within the guarantees offered by the selected backend.

3.2 System Architecture and Roadmap

Figure 1 shows the life cycle of an Alchemy query. Almost all steps of our optimization workflow draw from the system catalog. It offers public statistics about each table in the federation, and we describe the features that we use in Section 3.1. For our input phase, the user submits a SQL statement, Q, to the Alchemy optimizer. We use Apache Calcite [2] to parse and transform the query into a DAG of database operators. In Step 1 of the diagram, Calcite verifies the query's semantics against the CDM and produces a JSON specification for each logical operator (e.g., filter, join).

Step (2), we first use the SMCQL [8] security type system to identify the minimal covering set of distributed database operators to run in circuits for our query execution plans while producing correct and complete results. This applies labels to each operator, denoting the public and private attributes as they move up the query tree. Next, we apply the query transformations described in Section 4.1 to reduce our reliance on circuits further. These rewrite

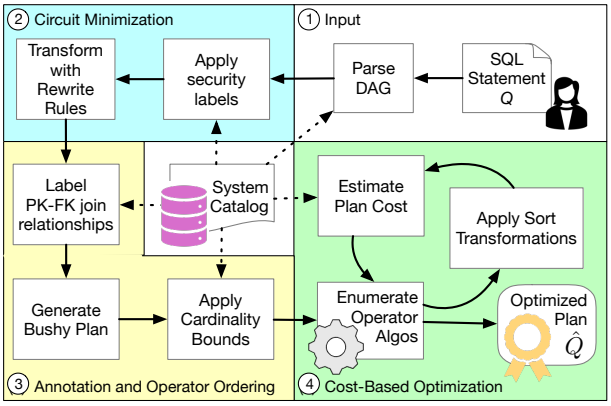


Figure 1: Alchemy optimization workflow.

rules do so by 1) performing single-tuple operations locally, 2) reusing work done in circuits, and 3) revising the in-circuit plan nodes to minimize their circuit complexity.

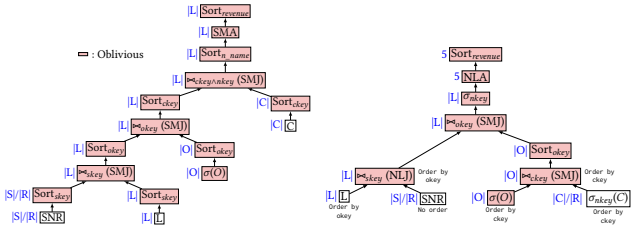
In Step (3), we prepare Q for operator selection. First, we use the system catalog to annotate all PK-FK joins. Then, we use the greedy algorithm in Section 4.2.1 to construct a bushy join tree that further reduces our circuit sizes. After ordering our operators, we use metadata about public attributes to derive an oblivious cardinality bound for each one (Section 4.2.2).

Now, we are ready to select the algorithm for each operator in the query plan using our analytical cost model in Section 4.3.2. We traverse the tree bottom-up, assigning its implementation one at a time. When proposing a physical plan for subtree $q \in Q$, we first apply a suite of transformation rules (Section 4.3.1 that identify opportunities to push sorts out of circuits and reuse them when possible. Recall that an oblivious sort is a common primitive for aligning tuples that we compute on together, such as for sort-merge join (SMJ) or group-by aggregation. We then compare q to competing plans and retain the lowest cost one. After selecting the physical operators in the tree, we obtain the optimized plan, $\hat{Q}$.

3.2.1 Inputting Rows Alchemy ships the optimized secure query plan, $\hat{Q}$, to the data providers as a JSON file. $\hat{Q}$ specifies and parameterizes all circuit operators, and it generates a SQL statement for each tree leaf that all data partners run in parallel. These plaintext steps either: (1) compute on public data (and reveal the true cardinality of their output) or (2) do not require coordination among the parties (such as a filter that dummies out rows that were not selected). We now describe the two methods with which Q receives its secret-shared input tables.

Union If the leaf has unordered inputs, we construct $[R]$ by having each party secret-share their records and concatenate the resulting data. This operation requires no circuits and its communication costs are linear in the input size with extended oblivious transfer [6, 24, 25]. This is the default input method for VaultDB's PDF queries.

Merge If the table is vertically partitioned, we secret share the columns from each party (all parties input the same number of rows), and then copy together the secret shares into a single table object. Because VaultDB is a column store, this process is very efficient. We also `AND` the dummy tags of the input rows to only retain the real ones on all sites. We use this approach to combine



(a) Baseline **(b) Bushy Operator Selection**

Figure 2: Running example query plans. $|S| < |C| < |O| < |L|$.

the locally computed parts of operators (such as aggregate and sort), and we will describe these later in Section 4.1.1.

This system also supports outsourced settings (by secret-sharing all of the input records at setup time) by accepting files of shares as inputs, bypassing the oblivious transfer phase.

3.2.2 Revealing Query Answers The data providers execute $\hat{Q}$'s operators bottom-up. Each operator produces a secret-shared table as output that it passes to its parent. The root node of $\hat{Q}$ outputs the secret-shared query answer, $[A]$, with each party holding a share of it. All parties send their share of $[A]$ to the front-end via an encrypted connection such as `ssh`. The client then assembles the shares to reveal the results. From their perspective, this system behaves like a conventional data federation where one submits a query and receives results over the data of all of its member engines.

3.3 Running Example

We use TPC-H Q5 to illustrate Alchemy's optimization process:

```
SELECT n_name, SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM customer C JOIN orders O ON c_custkey = o_custkey JOIN lineitem L
ON o_orderkey =  l_orderkey JOIN supplier S ON  s_suppkey = l_suppkey
AND c_nationkey = s_nationkey JOIN nation N ON s_nationkey = n_nationkey
JOIN region R on r_regionkey = n_regionkey WHERE r_name = ...
AND o_orderdate >= ... AND o_orderdate < ...  GROUP BY n_name ORDER BY revenue DESC;
```

Throughout the paper, we demonstrate our optimization process using this query with the TPC-H security policy in Section 5.1. To highlight how Alchemy transforms queries, we visualize the query tree before and after optimization in Figure 2.

4 OBLIVIOUS OPTIMIZATION FRAMEWORK

Circuit minimization is the first phase of Alchemy's optimization process. Starting with a DAG of operators derived from the SQL statement, this phase applies various techniques to minimize circuit complexity with logical rewrites and cardinality bounding.

4.1 Query Transformation Rules

Alchemy uses a series of query transformations in Step (2) of its workflow to reduce the circuit cost of Q, formalized in Table 2. Many are analogous to rules used in conventional query optimization [16].

4.1.1 Local Querying We now describe the transformations we use to push steps of Q out of circuits for local, plaintext evaluation.

Filter Pushdown We push down all single-table filters for local evaluation, rewriting ones on private fields to dummy pad the input $[R]$. In our running example, Q5 has a filter on `o_orderdate`, a private column. For this, our SQL generator adds a dummy tag column to its input query, `SELECT ..., !(o_orderdate >=...)`. All other filters reveal their true cardinality from the size of their secret shares. For example, Q5's public join of supplier, nation, and region (filtering on `r_name`) inputs $O(|S|/|R|)$ rows.

Table 2: Oblivious query transformation rules.

Name	Transformation ($\hat{op}$ in circuits, k : key col)
Union	$[R] \rightarrow \bigcup_i R_i$
Sort Pushdown	$\hat{\text{merge}}(\hat{\text{sort}}(\bigcup_i(R_i))) \rightarrow \hat{\text{merge}}(\bigcup_i(\text{sort}(R_i)))$
Concatenate Rows	$[R] \rightarrow \hat{\bigcup}_i(dom(k) \bowtie_K R_i)$
Filter Pushdown	$\hat{\sigma_e}(R) \rightarrow \pi_{R,\neg e}(R)$
Eager Projection	$\hat{op}_{e_2}(\hat{op}_{e_1}(R)) \rightarrow \hat{op}_{e_2}(\hat{\pi}_{R\setminus e_1}(\hat{op}_{e_1}(R)))$
1:1 Merge Join	$[R] \hat{\bowtie}_K S \rightarrow \hat{\sigma}_{dummy\ tag}([R] \hat{\cup} (dom(k) \bowtie S))$
Hybrid Agg.	$\hat{\gamma}_{G,Agg(x)}(R) \rightarrow \hat{\gamma}_{G,Agg(\sum x)}(\hat{\bigcup} i\gamma_{G,\sum x}(R_i))$
Functional Dependencies	$\hat{\gamma}_{G_1,\ldots,G_n,Agg(x)}(R) \rightarrow \hat{\gamma}_{G_k,Agg(x)}(R)$ where $G_k \rightarrow \{G_1, \ldots, G_n\}$
Join Predicate Pullup	$R \hat{\bowtie}_{R.a=S.b} S \hat{\bowtie}_{S.k_1=T.k_1 \wedge R.k_2=T.k_2} T \rightarrow \hat{\sigma}_{R.a=S.b}(S \hat{\bowtie}_{S.k_1=T.k_1} T \hat{\bowtie}_{R.k_2=T.k_2} R)$
Predicate Transfer	$\sigma(R) \hat{\bowtie}_k S \rightarrow K' := \pi_k(\sigma(R)); \sigma(R) \hat{\bowtie}_k \sigma_{k \in K'}(S)$
Merge Common Subexpressions	e.g., $(R \hat{\bowtie}_k S) \ldots \hat{\bowtie} \ldots (R \hat{\bowtie}_k S); \rightarrow$ $CTE := R \hat{\bowtie}_k S; CTE \ldots \hat{\bowtie} \ldots CTE$

Hybrid Aggregate For group-by aggregates, if the domain of the group-by columns is public, each site computes a partial aggregate on its local inputs. We then use the *merge* input method to construct a single table over all parties with one row per group-by bin. After that, we project the partial aggregates of all sites to assemble their output. Because our domain is known, Alchemy adds a left outer join on the group-by domain column to the input query to pad private inputs to the appropriate length. In the input query, we also sort on this domain to ensure all parties' input rows are aligned. This naturally generalizes to aggregates without group-by clauses.

Sort Pushdown For leaf nodes with direct ancestors that are collation-dependent, we partially evaluate their sorts locally before secret sharing them. We obliviously arrange the secret shares into a bitonic sequence by reading one party in ascending order and the next in descending order, and then merging this sequence. The merge performs $log(|R|)$ rounds of comparisons with $|R|/2$ comparisons per round. This reduces our sort complexity by a $log(|R|)$. More importantly, it can make later operations, such as joins and group-by aggregation, less costly.

Merge Join If a join has two children that are leaves in the secure query plan and the domain of the join key is public, we partially construct the join result locally. We cannot simply send all joining rows to one node and compute the join in plaintext as in Conclave [52] because this would preclude the use of *eager projection* and *filter pushdown* since they compute over private attributes. We push down a left join in plaintext on the join key domain, similar to *hybrid aggregate*, and *merge* this vertically partitioned join output. This sidesteps the need for expensive tuple comparisons because we align by join key before entering circuits.

4.1.2 Logical Circuit Reduction Our next group of rewrites reduces the complexity of our circuits by applying logical transformations to the operator tree. The next rewrites simplify our circuit complexity via logical transformations on the operator tree.

Eager Projection Many of our operators use conditional writes to update entire tuples, such as a compare-and-swap for oblivious sorting. Hence, slimming down rows in our IR is critical for performance. We aggressively project unneeded columns after each operator in the oblivious query tree, and perform this projection in the local DBMS before secret-sharing to minimize the data entering

circuits. Lastly, we offload math and logic expressions from circuits for local evaluation when possible. In Q5, we project `revenue`, `l_extendedprice*(1 - l_discount)` in plaintext with this rule.

Join Predicate Pullup If the query has an N:N join condition, we attempt to rewrite it as a filter after our PK-FK joins. This rule is targeted at join graphs that are fully connected, e.g., a triangle or square. In 2b, we could pull this predicate up to the final join ($L \bowtie_{okey,nkey} O$). If we did this, then our gates per tuple comparison would double! By rewriting this as $\sigma_{nkey}(L \bowtie_{okey} O)$, we calculate this check lazily over $|L|$ rows.

Functional Dependencies When we perform tuple comparisons, such as in sorting and group-by aggregation, we reduce our circuit costs by leveraging functional dependencies among the referenced columns. We identify the minimal set of determinant columns for each tuple comparison. For example, TPC-H Q3 groups by `o_orderkey, o_orderdate, o_shippriority`. Since `o_orderdate` and `o_shippriority` are functionally dependent on the primary key `o_orderkey`, we can determine if two rows are in the same group-by bin by comparing only their `o_orderkeys`. The remaining query logic (copying out the entirety of the group-by columns for the parent operator) is unchanged by this rewrite.

4.1.3 Work Reuse Circuit-based execution introduces a performance bottleneck not present in plaintext. To address this, our last suite of rewrites eliminates redundant computation in our circuits.

Merge Common Subexpressions We further reduce our circuit counts by identifying repeated logic in a secure query plan, inspired by multi-query optimization [22, 26]. We identify leaf nodes that appear more than once in Q by comparing their SQL input statements. We then work bottom-up to find the largest common subtree. In Table 2, we give an example of this with a join, $R \hat{\bowtie}_k S$, that appears more than once in a cascade of joins. We create a common table expression (CTE) for this join and add this as a child to the tree nodes that reference it, so we do not need to recompute this join.

Predicate Transfer If we join over a public key, we project tables to their public columns, intersect their key domains across sites, and filter unmatched rows prior to secret sharing—thereby reducing the data entering circuits. Secure Yannakakis [54], Secrecy [37], and SMCQL [8] also use this approach. In our running example, we filter `supplier` publicly by joining it with the filtered `nation` table. Recall that `customer` and `supplier` join on `nationkey`. We project the filtered supplier down to its remaining `s_nationkeys` and use this to reduce the rows we receive from `customer` to $O(|C|/|R|)$ as shown in Figure 2b.

4.2 Operator Ordering

We now move on to Step (3) of our query workflow: annotating the query tree and ordering our operators. We first describe our greedy plan generator for creating bushy join trees. Then, with the logical operator order chosen, we dive into how we derive upper bounds on the cardinalities of our IRs.

4.2.1 Bushy Plan Generator Bushy join trees introduce valuable opportunities for optimizing the execution of SMJ and other collation-sensitive operators. The database community has been interested in bushy plans for distributed DBMSs like MemSQL [13] to minimize IR sizes; this single-dimensional optimization does not fully capture the performance characteristics of our setting. With SMJs, maintaining a collation as rows pass through multiple joins may provide greater performance benefits than minimizing IR sizes alone, as it sometimes eliminates costly re-sorting operations down the line. As demonstrated in Figure 2b, joining smaller relations `O` and `C` before incorporating the larger `L` provides independent sort order choices. In this bushy structure, the subquery $\texttt{C} \bowtie \texttt{O}$ can be optimized with either `custkey` ordering for their join, or `orderkey` ordering for the subsequent join with `LSNR`, without being constrained by `L`'s sort order. This independence in sort order selection becomes increasingly valuable as query trees grow bushier, as each independent subtree presents its own optimization opportunities.

Algorithm 2 Greedy algorithm for bushy join plans.

```
1:  procedure GenerateJoinTree(J)
2:      J := j_1, ..., j_n: List of (F, {P}) pairs in ascending order by |F|.
3:      P := ∅: subtree(s) of bushy query plan.
4:      for i do := 1 ... n // for each (F, {P}) entry
5:          for j do := 1 ... |P_i| // for each PK
6:              F_in = F_i; P_in := P_{i,j}
7:              if p ∈ P contains F_in then
8:                  F_in := p; P := P \ p
9:              end if
10:             if p ∈ P contains P_in then
11:                 P_in := p; P := P \ p
12:             end if
13:             P := P ∪ (P_in ⋈ F_in)
14:         end for
15:     end for
16:     return P;
17: end procedure
```

Since finding optimal bushy trees remains NP-hard [13, 47], Alchemy has a heuristics-driven tree builder. It operates under two simplifying assumptions: uniform predicate length and delaying operator selection until after the bushy plan generator. We compare competing plans in this section using their costs over nested loop joins (NLJ), i.e., $|R| \times |S|$.

The fundamental property $|R_{PK} \bowtie R_{FK}| = |R_{FK}|$ drives our plan builder because a tree of multiple PK-FK joins cannot have an output cardinality that exceeds that of the length of the final FK relation. We denote the FK and PK relations in a join as **F** and **P**, respectively.

At setup time, we rewrite the join graph in two ways. First, we consider the covering set of keyed predicates and select the one with the lowest cost. For example, if a TPC-H query joins $\texttt{S} \bowtie_{skey} \texttt{L} \bowtie_{skey,pkey} \texttt{PS}$, we rewrite to $(\texttt{S} \bowtie_{skey} \texttt{PS}) \bowtie_{pkey,skey} \texttt{L}$, reducing costs from $|L| \times |S| + |L| \times |PS|$ to $|S| \times |PS| + |PS| \times |L|$. Although both plans are $O(n^2)$, the constant factors differ significantly - this rewrite performs the join with smaller tables first rather than immediately joining with the much larger table, substantially reducing the concrete costs of a plan. Second, we set aside many-to-many joins until after keyed ones. This delays Cartesian product IRs into our plans as long as possible, avoiding unnecessary cost propagation to parent joins. Recall that if an N:N join is "connected" by a PK-FK join, we sidestep the need for a n^2 IR entirely!

Algorithm 2 describes our join tree planner. We start by analyzing Q, constructing a list of FK relations paired with the PK relations with which they join: $j_i := (\mathbf{F}_i, \{\mathbf{P}_{i,1}, \ldots \mathbf{P}_{i,j}\})$. For multiple PK tables, the cost is order-agnostic since $|\mathbf{F}_i \bowtie \mathbf{P}_{i,1}| = \ldots = |\mathbf{F}_i \bowtie$

Table 3: Rules for deriving cardinality bounds.

Operator	Output Card	Output Statistics
$R \bowtie_{R.FK=S.PK} S$	$\|R\|$	$\text{dom}(R.FK) \cap \text{dom}(S.PK)$
$R \bowtie_{\wedge_{i=1}^{n} R.x_i=S.y_i} S$	$\|R\| \cdot \|S\|$	$\forall i : \text{dom}(R.x_i) \cap \text{dom}(S.y_i)$
$\sigma_{R.x \in [min,max]}(R)$	$\|R\|$	$min \leq dom(R.x) \leq max$
$\pi_{expr(R.x)}(R)$	$\|R\|$	$x \in \text{dom}(R.x) \mid expr(x)$
$\gamma_{G_1,...,G_n,Agg(x)}(R)$	$\prod_{i=1}^{n} \|\pi(R.G_i)\|$	N/A
$\gamma_{G_1,...,G_n,Agg(x)}(R)$ $G_k \rightarrow \{G_1, ..., G_n\}$	$\|\pi(R.G_k)\|$	N/A

$\mathbf{P}_{i,j} = |\mathbf{F}_i|$. Each of these $\mathbf{F}$ entries serves as a module of joins to add to our query plan $\mathcal{P}$, a DAG of joins. A given table may appear in more than one entry.

The algorithm's input, the list of $(\mathbf{F}, \mathbf{P})$ pairs, is sorted in ascending order by the cardinality of their FK relation. We add these FK tables into $\mathcal{P}$ from smallest to largest. When we add a $\mathbf{F}_i$, we incrementally join it with its related primary key relations from smallest to largest, adding the resulting join to $\mathcal{P}$. If $\mathbf{F}_i$ is in an existing subplan $p \in \mathcal{P}$, we use that subplan as $\mathbf{F}_{in}$ and remove p from $\mathcal{P}$ for replacement later. Similarly, if $\mathbf{P}_{i,j}$ is already in $p \in \mathcal{P}$, then we use the existing subplan p as $\mathbf{P}_{in}$ and remove p from $\mathcal{P}$. We then join $\mathbf{P}_{in}$ with $\mathbf{F}_{in}$ and add this new join to $\mathcal{P}$ in Line 13. This process repeats for each primary key relation, incrementally building up bushy plans in $\mathcal{P}$ while reusing existing subtrees when possible to delay the inclusion of expensive FK relations. By generating bushy plans, we create additional opportunities to optimize sort operators described in Section 4.3.1.

4.2.2 Cardinality Bounds We derive a tight upper bound on the cardinality for each node in Q without leaking information about $\mathcal{D}$ to reduce the complexity of subsequent operators. This generalizes the principles of cardinality estimation, reframing them for oblivious IR sizes.

We start with the leaves in Q's tree and work bottom-up. If a column, $R.x$ has domain constraints or if it is public, we annotate it with its domain, $\text{dom}(R.x)$. As $R.x$ passes through an operator, *op*, we incrementally update $\text{dom}(R.x)$ using its parameters. We infer *op*'s cardinality bound with these updated domain statistics. These are not meant to be exhaustive. For example, we could use multiplicity (the number of times a given value may repeat in a column) to refine these bounds as proposed in prior work [4, 14], but we found that in practice the overhead of eliminating dummies by obliviously sorting and truncating IRs for purely oblivious cardinality bounds was cost-prohibitive. Instead, we integrate these bounds into our cost model in Section 4.3.2.

Our rules for propagating these statistics are in Table 3. This framework deduces that PK-FK joins output |FK| rows, based on cardinality inference techniques [15, 37]. Absent a key constraint, we fall back to a quadratic cardinality bound. Our equi-join outputs update the domain of their key columns to the intersection of the domains of their source keys. Filters restrict their output domains based on their predicates. We apply projection expressions with scalars to all values in a column's domain. Aggregates bound their output cardinality by taking the product of the length of the domain of each group-by column. If some of the group-by columns are functionally dependent on another, $G_k \rightarrow \{G_1, \ldots, G_n\}$, we drop the dependent columns from our bound calculations.

Table 4: Circuit-based operator cost model.

Operator	Gate Cost
Filter: $\sigma_p(R)$	$\|R\| \cdot \|\hat{p}\|$
Project: $\pi_e(R)$	$\|R\| \cdot \|\hat{e}\|$
Sort: $sort(R, c)$, $x = log\|R\|$	$\|R\|/2 \cdot (x(x+1))/2 \cdot (2\{c\} + \{R\})$
NLJ	$\|R\| \cdot \|S\| \cdot \|\hat{p}\|$
PK-FK NLJ	$\|R\| \cdot \|S\| \cdot (\{S\} + \|\hat{p}\|)$
Merge Join	$\|R\|$
SMJ	$O(2 \cdot sort(\|R\|+\|S\|))$
SMA: $\gamma_{G_1,...,G_n,Agg(x)}(R)$	$sort(R, G_1,..., G_n) + \|R\| \cdot (\|\widehat{Agg}\| + \{\sum_{i=1}^{n} G_i\})$
NLA: $\gamma_{G_1,...,G_n,Agg(x)}(R, n)$	$n \cdot \|R\| \cdot (\|\widehat{Agg}\| + \{\sum_{i=1}^{n} G_i\})$

4.3 Cost-Based Optimization

After ordering our operators into a logical plan, we are ready to optimize our physical query execution plan. In Step (4) of Figure 1, Alchemy performs cost-based optimization over the bushy query tree generated in the previous step. This phase systematically explores the space of physical execution plans by leveraging our cost model to select the most efficient combination of operator algorithm and input sorting options for each subtree q. We describe the operator implementation VaultDB offers starting from Section 4.3.3.

4.3.1 Sort Transformations Sorting is a key building block for our circuit-based operators because it enables us to make rows on which we compute together adjacent in a table, such as joining rows or ones in the same group-by bin. Alchemy reduces the circuit burden of sorting in two ways. First, we aggressively push sorts out of circuits as much as possible. This idea was first proposed in Conclave, but our work is the first to formalize it and integrate it into a cost model. Second, we reduce the sorting overhead within circuit operators by exploiting existing input orderings.

For each leaf node in q, we first enumerate the collations that may reduce our overall circuit costs using join keys and group-by expressions. When we create a new proposed plan for a given collation, q', if a non-leaf operator is sort-dependent—such as SMA—then we insert any sorts needed for them. For sorts pushed down to the leaves of q', we rewrite them with the *Sort Pushdown* transformation in Section 4.1.1. Recall that when two parties union their sorted inputs, we short-circuit the first phase of our bitonic sort because the inputs already form a bitonic sequence.

Once we've pushed as many sorts as possible outside circuits, we further reduce sorting cost in SMJ by reusing already sorted input. A SMJ over R and S first sorts over $R \cup S$, performing $O((|R|+|S|)\log^2(|R|+|S|))$ tuple comparisons, and it is the most expensive step of this operator. If at least one input is already sorted, then we sort the other and reduce this step's complexity by $log(|R|+|S|)$. For example, in Q5 (Figure 2b), we reduce the complexity of a SMJ on `C` and `O`, inputs are pre-sorted on `o_custkey` and `c_custkey` respectively, eliminating the need for a full sort at the join's start.

4.3.2 Circuit Cost Model VaultDB's suite of oblivious database operators creates a decision space for Alchemy's cost optimization. This section presents the principles underlying our circuit-based query plan cost estimates, followed by our analytical cost model for each operator. We then examine the trade-offs associated with our library of SPJA operators. Although many of these algorithms are from previous work, we are the first to formalize a cost model for bringing them together for selecting among them. Moreover, we

incorporate schema metadata to guide Alchemy to more efficient query execution plans. Table 4 summarizes our cost model.

Like conventional query optimization, we use a cost model to identify efficient oblivious query execution plans. Since our unit of computation is the circuit, *our cost model estimates the rounds of communication it will take to evaluate a proposed plan.* Our gate-based cost model informs physical operator selection, specifically modeling the number of `AND` gates needed to execute a DAG of oblivious operators. This contrasts with `NOT` and `XOR` gates, we evaluate without communication among the computing parties using the techniques proposed by Kolesnikov and Schneider [30].

Evaluating each `AND` gate requires communication [43]. This makes them the fundamental driver of our cost model because each round trip of networking substantially outweighs its corresponding CPU time. While it is straightforward to reason about the asymptotic gate cost of each operator as a function of its input size, our analytical cost model is made more complicated by parameters such as predicate evaluation and conditional writes.

VaultDB's operators use a few recurring circuit designs. The first is oblivious if-then statements, which are expensive in circuit-based secure computation because the engine prevents leakage by executing both branches. Second, conditional writes and compare-and-swap operations extend this principle–they incur a gate cost proportional to the length of their input in bits because they work on a field one bit at a time. Similarly, our cost of evaluating a comparison is proportional to the length of its inputs in bits. Since compare-and-swap involves both a comparison and a conditional move, this inflates the runtime of our sorting algorithm.

Our expression library offers in-circuit arithmetic operators, comparators, logical connectors, and `CASE` statements. We use this to both create new field values in projections and to evaluate conditionals. It translates the DAG into circuits during setup. Our cost model estimates the gates associated with a given expression by traversing its DAG and summing up the cost of each step.

Circuit-based cost modeling is a natural paradigm for oblivious query optimization. Measuring and modeling from empirical runtimes would be exceptionally time-consuming owing to the overwhelming cost of the circuit overhead, and in some cases, it suffers from high variance due to the availability of network bandwidth acting as a confounding variable. By contrast, estimating gate counts is deterministic and lightweight to model. Owing to Alchemy's rich metadata from the common data model, the optimizer estimates gate cost offline before running anything for a given query. This enables it to explore variations in sort order and operator selection. Similarly, the cost of computing the plaintext inputs for oblivious operators is overshadowed by the runtime of the circuits. In our experimental results, queries run for several minutes to hours, and the parties compute their local subqueries in a few seconds or less. In Section 5.3, we verify that circuit complexity is a robust proxy for query runtime.

4.3.3 Core Operators We now describe the operators that execute over secret-shared tables within circuits, which dominate the runtime of any non-trivial SQL query. VaultDB supports a range of operations, including select, project, inner joins, aggregates (such as group-by), sort, union, top-k queries, and more.

Filter VaultDB's filter, $\sigma_p(R)$, evaluates its predicate one tuple at a time, updating the corresponding output dummy tag accordingly. We lazily delete dummies when we reveal a query answer rather than nulling them out to avoid costly conditional writes to the IRs. Our cost for this operator is $|\hat{p}|\times|R|$ gates.

Project The projection operator is trivially oblivious, as it applies the same transformations on all rows without data-dependent access patterns. Alchemy leverages projection to drop unneeded fields and push complex expressions out of circuits, as described in Section 4.1.2. This reduces the complexity of our circuits for conditional write-heavy operators that access whole rows, such as sorts, joins, and group-by aggregates. Reordering and deleting columns incur no additional gates. Hence, many projections are free in practice. The cost of a projection, $\pi_e(R)$ is $|\hat{e}|\times|R|$.

Sort VaultDB computes its sorts using a bitonic sorter [7]. It first reorders its input rows into a bitonic sequence wrt the sort key using a series of compare-and-swap operations. A bitonic sequence of rows $r_0, \ldots, r_{n-1}$ will have the form $r_0 \leq \ldots \leq r_k \geq \ldots \geq r_{n-1}$ for some k over either the initial sequence or a circular shift thereof. Given two sorted lists of length $n/2$, the algorithm merges them in $log\ n$ rounds with a maximum of $n/2$ comparisons per round. We considered using the AKS sorting network [1], an alternative approach with a complexity of $O(n \log n)$. But it has a high setup cost, and is only profitable when $n > 10^{52}$. Each compare-and-swap has a comparison over $\{c\}$ bits for the sort key c and a conditional move for $\{R\}$ bits, at a cost of $2\{c\} + \{R\}$.

4.3.4 Joins We support two join algorithms from the literature.

Nested Loop Join (NLJ) The basic NLJ algorithm iterates through all pairs of tuples in R and S, producing $|R|\times|S|$ output rows. It copies the input rows to their output positions unconditionally and marks the rows that do not satisfy the join predicate as dummies. The cost of this operator is $|R|\times|S|\times|\hat{p}|$. If this is a PK-FK join, because each FK row matches at most one tuple from the PK relation, we create one output tuple in the outer join loop and conditionally overwrite the second half of this join output once per tuple comparison with `mux`. This tames the size of our IRs, but at a higher gate cost for the conditional write. Our cost is $|R|\times|S|\times(|\hat{p}|+\{S\})$.

Sort-Merge Join (SMJ) VaultDB offers a variant of the oblivious SMJ algorithm proposed by Krastnikov et al. [32] optimized for PK-FK joins. It first materializes $R \cup S$ sorted on the join key in $O((|R|+|S|)\log^2(|R|+|S|))$ gates. The operator then does a linear-time pass over this hybrid table to record a count of each row's join matches. Next, it sorts the array and splits it into its constituent tables with an additional $O((|R|+|S|)\log^2(|R|+|S|))$ gates. After that, it expands S to $|R|$, padding it to the correct number of join matches in $|R|\ \log^2(|R|)$ time. For N:N joins, we repeat this step for R. We then concatenate the rows one at a time.

VaultDB supports NLJ and SMJ because they are compatible with circuit-based SQL execution across any number of parties. There are other oblivious joins, but they either require specific MPC protocols (such as 3PC) or vertical partitioning. We discuss this further in Section 6. We first describe our version of each for PK-FK joins, and then how we also support N:N relationships. Recall that VaultDB also supports a merge join as described in Section 4.1.1 for 1:1 joins. We do not consider this here because we use this circuit-free operator unconditionally when possible.

4.3.5 Aggregates We now describe VaultDB's oblivious aggregation algorithms. For ones with no group-by clause, the engine does

Table 5: Runtime (s) for incrementally-added query rewrites.

	Base-line	Local Eval	Circuit Min	Work Reuse	E2E Speedup
Q1	252	215	0.348	0.348	724.8×
Q3	734	595	433	374	2×
Q5	2,447	1,969	1,786	1,017	2.4×
Q8	3,604	746	570	436	8.3×
Q9	2,291	1,623	1,356	1,054	2.2×
Q18	1,084	140	140	119	9.1×

a linear pass over all rows, incrementally aggregating its single output row as it goes along. If our aggregates are specified in the gates $\widehat{Agg}$, then this operator runs in $|R|\times|\widehat{Agg}|$ time. We now describe two group-by aggregate algorithms over $S := \gamma_{G_1,\ldots,G_n,Agg(x)}(R)$.

Nested Loop Aggregate (NLA) For group-by aggregates with a small output cardinality, we follow a logic similar to that of NLJ. The operator initializes an output table with $|S|$ rows. For each input row, it visits all $|S|$ output rows and either updates the corresponding group-by bin or–if it finds no match–it obliviously initializes the first empty slot. We estimate NLA's cost as $|R|\times|S|\times|\widehat{Agg}|$.

Sort-Merge Aggregate (SMA) A second strategy is to first sort the input relation on its group-by clause, then do a linear pass over the data to emit a single output tuple per input row. This circumvents the need to visit each possible output group-by bin. We estimate the cost of this algorithm as $|R|\times|\widehat{Agg}|+sort(R, G_1, \ldots, G_n)$. SMA's complexity is asymptotically better than NLA when no suitable cardinality bound exists for S. Here, the runtime of NLA degenerates to $O(|R|^2)$. If the aggregate's input arrives sorted on its group-by columns, the runtime of this algorithm is linear time, identical to the case of a scalar aggregate. The divergent performance characteristics of the two aggregate algorithms present a rich decision space for a sort-order aware optimizer.

5 EXPERIMENTAL RESULTS

In this section, we evaluate our oblivious query optimization framework, incrementally introducing its techniques to assess their impact on performance. Starting from the baseline, we apply rewrite rules to reduce circuit complexity. We then present our operator cost model, showing its correlation with runtime to justify its use in plan optimization. Next, we measure the performance gains from bushy plan generation and the application of physical plan optimizations, including operator selection and sort transformations.

5.1 Setup

We conducted experiments on the TPC-H benchmark, implementing and executing all 22 queries. From these, we selected six representative queries—Q1, Q3, Q5, Q8, Q9, and Q18—for in-depth evaluation, as they highlight a broad range of optimization opportunities [18]. Q5, Q8, and Q9's performance is heavily influenced by predicate positioning and join ordering. Q3 and Q18 showcase how exploiting functional dependencies and sort transformations speed up query evaluation. Q1 and Q18 have performance that is sensitive to operator algorithm selection. Our security policy is motivated by an international online shopping use case, as in SAQE [9]. All columns in `lineitem` are private. `nation`, `region`, and `part` are fully public. For other tables, only primary keys and columns referencing public tables (e.g., `s_nationkey`, `c_nationkey`) are public.

To limit the duration of our experiments, we evaluate Alchemy on a scaled-down version of TPC-H. We refer to each dataset by its scale factor (SF). Most tables scale proportionally with the scale factor as specified in the benchmark, except for `nation` and `region`, which have fixed cardinalities. Unless otherwise stated, we test on *TPC-H SF 0.01*, and we compare Alchemy against a baseline generated by a conventional query optimizer that does not have access to private data distributions. This optimizer is aware of table cardinalities, but it does not perform schema-aware transformations.

Alchemy's query executor sits atop PostgreSQL running on Ubuntu. Although we chose PostgreSQL for our experiments, Calcite supports a myriad of other SQL dialects for generating the input queries for $\hat{Q}$. We evaluated our prototype on AWS `EC2 r6i.4xlarge` instances, each running Ubuntu Server 22.04 LTS, equipped with 128 GiB memory, 16 vCPUs, and offering up to 12.5 Gbps connectivity between the parties. Our results show the end-to-end runtime of a query, including its local evaluation and oblivious steps. Unless otherwise noted, our results are computed over EMP Toolkit's `sh2pc` library for semi-honest 2-party computation [53].

5.2 Query Transformation Rules

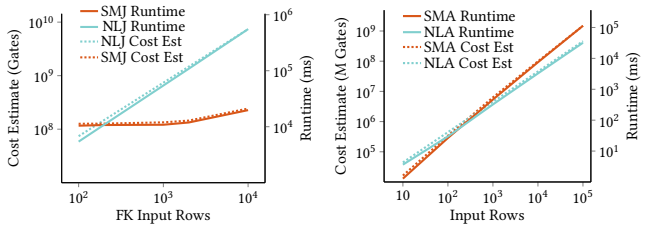
We first evaluate the impact of the query transformations described in Section 4.1.1. They reduce circuit complexity by leveraging local computation and eliminating redundant operations. Table 5 shows our runtimes in seconds as we add our three classes of transformations incrementally. To isolate the impact of these rewrites, we use SMJ and SMA for all joins and aggregates, respectively. To limit the duration of our experiments, we apply PK–FK cardinality bounds to all plans to reduce our output cardinalities from $O(n^2)$ to $O(n)$ with no information leakage.

Although we are not yet deriving the formal cardinality bounds from Section 4.2.2, this naturally arises in many of our oblivious operators. In merge joins, these bounds ensure that both parties provide input tables of equal size. In predicate transfer, they allow for the pruning of join inputs by projecting public key domains across related tables. These forms of cardinality reasoning are foundational to oblivious transformations. This reduction in IR sizes is crucial for reducing the circuit complexity of parent operators.

Although individual query transformations provide measurable improvements, we observe that combining multiple transformations results in greater synergistic effects. For example, Q1 achieves the largest speedup (724×), which is due to the combination of hybrid aggregation and eager projection: it offloads both grouping and complex arithmetic operations to local plaintext evaluation. In Q18, a single transformation—merge join—yields substantial benefit due to the domain knowledge from the schema. Eager projection proves particularly effective for queries containing complex expressions within circuits, while join predicate pullup and functional dependency yield approximately 1.2× improvement in Q5 and Q3, respectively. Finally, aside from Q1 (which has no joins), all queries benefit from local sort reuse, avoiding redundant sorting inside circuits and achieving an average of 1.3× additional speedup.

5.3 Operator Cost Modeling

We confirm the accuracy of our analytical cost models in Figure 3. We test joins and aggregates to: 1) confirm that estimated circuit counts are an accurate proxy for query runtime, and 2) identify the



(a) Joins **(b) Aggregates (NLA K=4)**

Figure 3: Alchemy cost model validation.

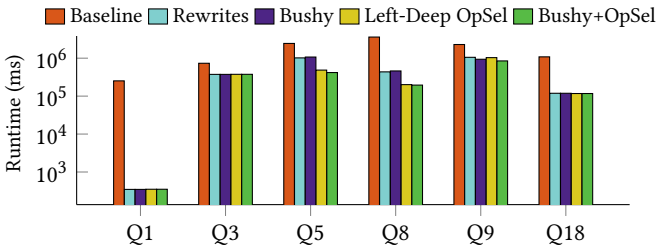


Figure 4: Query runtimes with incremental optimization.

performance envelope for competing operator implementations. We ran on *TPC-H SF 0.1* and manually varied our input cardinalities. For each experiment, we plot the operator runtime (solid lines) and its cost estimate in gates (dotted line).

Joins In Figure 3a, we compare the performance of keyed NLJ versus SMJ. We joined the lineitem and order tables, holding the cardinality of the former (FK) constant at 10k rows. We incrementally scaled up orders, starting with 100 rows and increasing by 10× each time. We did this to examine how their performance changes with the ratio between the two input table cardinalities. Our results show join times proportional to their estimated gate counts with an R^2 of 0.99. This demonstrates excellent agreement between the estimated number of AND gates and wall clock time. In addition, our results confirm that the cost model will enable Alchemy to correctly identify the most efficient join strategy as the relative sizes of the tables change. NLJ performs marginally better than SMJ when the inner loop cardinality is low. However, as the input size increased, the logarithmic scaling of SMJ's performance makes it a more efficient option. This result indicates that the input size must be sufficiently large to amortize the cost of SMJ's expensive sorts.

Aggregates We now experiment with a simplified version of Q1 that computes a count and sum to compare NLA to SMA. We record the cost of each aggregation algorithm as the input cardinality increases. The results of our experiments are in Figure 3b. We vary the input cardinality from 10 to 100K. We fix the cardinality bound of NLA at 4. While SMA is efficient for small inputs where the cost of sorting is low (or, when possible, pushed down to partial plaintext evaluation), the linearly increasing gate count of NLA makes it a more efficient choice for larger input sizes with a sufficiently tight output cardinality such as Q1's K=6. Our results show a strong correlation between gate count and runtime, with an R^2 of 0.99 again. This indicates that gate counts and their corresponding network communication are strong predictors of runtime.

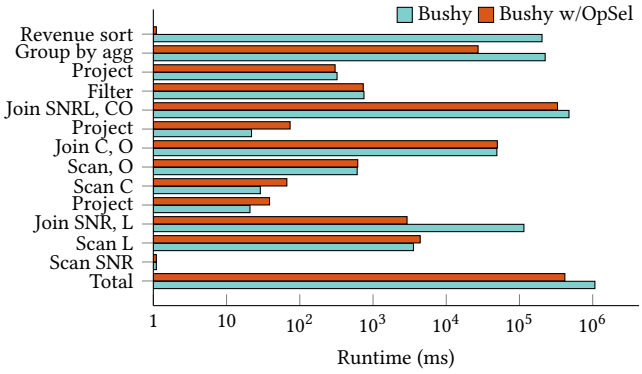


Figure 5: Individual operator performance on Q5.

5.4 Cost-Based Optimization

Our previous optimizations incrementally changed the query plan by switching the order of individual operators or pushing work out of circuits. We will now evaluate the impact of our tree-level optimization techniques. We first plot the baseline and the performance with query transformation rules (*Rewrites*), as shown in Table 2. This gets our query trees into a form where we can focus on minimizing the circuit cost of operators. We then add our bushy plan generator to select Q's join order. After that, we evaluate Alchemy with our cost-based operator selection both for left-deep and bushy plans. Our results are in Figure 4.

5.4.1 Bushy Trees We theorized that distributing join operators over a bushy tree would speed up our query runtimes for two reasons. First, it would delay joins with large input relations, taming the size of IRs as they cascade up the tree. Second, bushy trees create more opportunities for sort transformations, which in turn accelerate SMJs and SMAs. Our *Bushy* result in Figure 4 covers Step (3) in our workflow. Thus, it both uses Algorithm 2 to build the join tree and applies the cardinality bounds in Section 4.2.2.

Unsurprisingly, our results indicate that this optimization is only profitable for complex multi-way join queries, Q5, Q8, and Q9 in our core workload. Although our design was partially motivated by operators that benefit from local sorts, we will evaluate this aspect of our pipeline shortly in the "OpSel" results. Thus, any performance gains in *Bushy* must come from cardinality bounding onto the wider tree. These queries showed only modest gains with their new plans because they are "off-by-one" from their left-deep trees. For example, Q5's bushy tree in Figure 2b only differs in the cost of $C \bowtie O$, where C joins with $|O|$ tuples instead of $|L|$ ones.

5.4.2 Operator Selection To test the impact of operator selection and sort transformations in Step (4) of our query workflow, we measure this workflow with two tree configurations: left-deep and bushy. We do this to tease apart the synergistic effects of the bushy tree and the cost-based operator selection.

Left-Deep OpSel We start by testing our cost-based operator selection on a left-deep tree. This isolates the performance effects of operator selection from join reordering. Compared to the circuit-minimized plan, this variation shows notable performance improvements in several queries, particularly Q5 and Q8, due to smarter choices such as replacing SMA with NLA. These choices reduce IR sizes and circuit costs, as discussed in detail in Section 5.4.3.

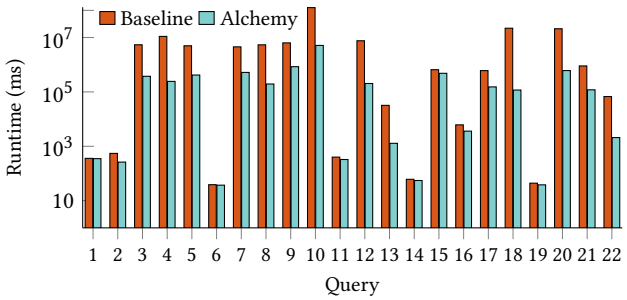


Figure 6: Runtime comparison for 22 TPC-H queries.

Bushy OpSel Our final result in this figure is a fully optimized configuration combining the *Bushy* operator ordering with cost-based operator selection. Now Alchemy uses both the operator choices we saw with *Left Deep* and the reduced IR sizes with *Bushy*. By putting these two together, we can finally realize the full potential of our sort transformations. Compared to left-deep trees, bushy trees provide greater flexibility in preserving and exploiting sort order, as they permit multiple branches to be partially sorted in parallel before merging. As shown in Figure 4, this configuration yields an average 2× improvement over the *Rewrites* plans for queries amenable to bushy planning (Q5, Q8, Q9).

5.4.3 Operator-at-a-Time Performance To better understand the impact of our cost-based operator selection, we compared the performance of *Bushy* and *Bushy OpSel* for Q5 in Figure 4, focusing on individual operator runtimes as shown in Figure 5. In both plans, the join, aggregation, and sort operations account for 98% of their runtime, highlighting the importance of optimizing these steps.

Our cost-based optimization produces a 97% speedup for $L \bowtie SNR$. As in Section 5.3, NLJ is more efficient than SMJ when the inner relation has low cardinality. This decision, visualized in the *Bushy OpSel* plan in Figure 2b, also preserves the sort order of L, which is sorted on `okey`, making the parent SMJ faster, a 30% performance gain. Additionally, by recognizing that the `n_name` attribute has a cardinality bound of five, we select NLA, boosting our performance on this step by 87%. This small IR size also reduces the last sort time by 99%. This demonstrates that our cost-based optimization effectively targets the primary bottleneck operators—join and aggregate—resulting in improved overall runtime.

5.5 Full TPC-H Workload

We evaluate how Alchemy's optimization framework performs across a broad range of analytical queries by extending our experiments to all 22 queries from the TPC-H benchmark. Figure 6 shows the runtime for each query. We introduced a new baseline that uses the rewrite rules featured in prior work outlined in Table 6 to highlight how Alchemy builds on the SOTA.

Alchemy achieved an average speedup of 21.8× over baseline, with the largest improvement reaching 188×. Only four queries showed comparable performance across the two systems, while the remaining 18 benefited significantly from Alchemy's cost-based optimizations. Moreover, plan enumeration resulted in negligible overhead across all 22 queries, remaining consistently below 0.01% of runtime. These findings confirm that our techniques generalize effectively across diverse query structures and complexities.

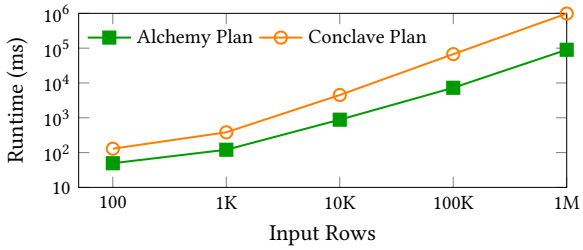


Figure 7: Comparison of Alchemy query execution plan to Conclave's on HealthLNK query.

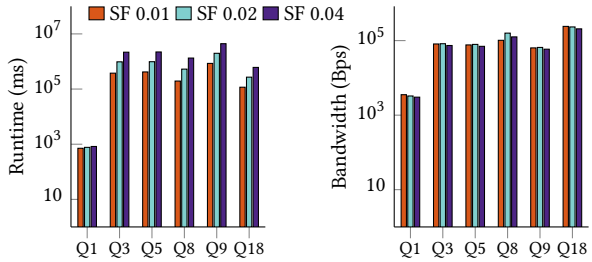


Figure 8: TPC-H performance on data of increasing size.

5.6 Comparison with Conclave Execution Plan

We additionally evaluate Alchemy on the HealthLNK [41] workload to compare with prior secure query systems, specifically Conclave. HealthLNK is a real-world clinical dataset with six million patient records collected from seven Chicago-area healthcare institutions. For comparison, we adopt the `Aspirin Count` query from prior work, which counts patients diagnosed with heart disease and prescribed Aspirin on or after diagnosis.

Conclave operates under a semi-trusted party (STP) model, where a coordinator may observe public attributes and intermediate metadata to accelerate computation. In contrast, Alchemy follows a stricter two-party semi-honest model with no additional trusted party. We could not run Conclave directly, so we manually reimplemented its logical plan without relying on the STP model.

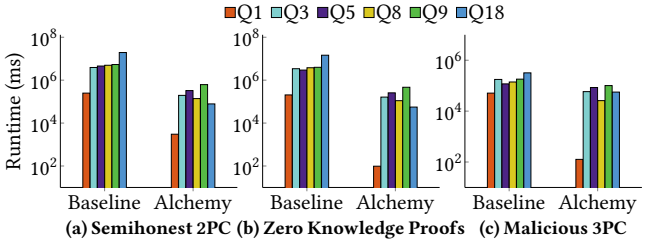
In this setting, absent the STP, the Conclave plan requires a bitonic sort before SMA. Alchemy avoids this cost by preserving row order through dummy-tag filtering and applying sort-aware transformations to eliminate unnecessary sort. As shown in Figure 7, Alchemy consistently outperforms this plan, with performance improvements that grow with data scale, reaching up to an 11× speedup at one million rows. This outcome underscores the efficiency of Alchemy's cost-based operator selection and sort-aware transformations under stronger privacy constraints.

5.7 Scale Up

We next examine Alchemy's scalability with increasing data volumes to *TPC-H SF 0.02* and *TPC-H SF 0.04*. Each experiment roughly doubles the dataset size of its predecessor. Our results are presented in Figure 8. Our findings strongly support Alchemy's scalability. For all queries except Q1, we observed near-linear scaling with runtime increasing by approximately 2.3× when the data size doubled. Q1's circuit cost is constant because it uses hybrid aggregation. Notably, bandwidth usage remained constant despite an increasing number of gates for more data. This indicates we are not saturating the bandwidth between the parties.

Table 6: Comparison of MPC SQL engines. SH = semihonest, MAL = maliciously secure, nPC = n-party MPC.

Name	Comp. Method	Filter Pushdown	Hybrid Agg	Merge Join	Simplify Circuits w/FDs	Join Pullup	Merge Shared Subtrees	Bound IR Size w/Constraints	Obliv Ops for 2+ Backends
Conclave [52]	SH3PC		✓	✓					
Secrecy [37]	SH3PC					✓			
SecYan [54]	SH2PC	✓	✓	✓		✓			
Senate [42]	MAL-MPC	✓	✓	✓					
SMCQL [8]	SH2PC	✓	✓	✓					
Alchemy (ours)	SH || MAL 2+PC, ZKP	✓	✓	✓	✓	✓	✓	✓	✓



(a) Semihonest 2PC (b) Zero Knowledge Proofs (c) Malicious 3PC

Figure 9: Performance with varying cryptographic protocols.

5.8 Broadening Oblivious Query Optimization

Alchemy's optimization framework is grounded in circuit minimization, making it naturally applicable across a range of secure computation protocols. To demonstrate its protocol-agnostic design, we integrated it with two additional cryptographic protocols from EMP Toolkit: zero-knowledge (ZK) proofs [36, 55, 56] and maliciously secure MPC [35] (OMPC), alongside our original semi-honest 2PC (SH2PC) protocols. For SH2PC and ZK experiments, we used *TPC-H SF 0.01*, while for OMPC, due to its higher storage and computational costs, we used *TPC-H SF 0.001* to reduce its runtime. Our results in Figure 9 reveal substantial performance gains across all cryptographic protocols. In SH2PC and ZK protocols, queries with pushed-down aggregates (Q1 and Q18) showed dramatic improvements of up to 2096× and 821× respectively, while other queries showed improvements ranging from 8× to 35×. OMPC showed more modest improvements, with Q1 achieving 401× improvement but others ranging from 1.4× to 5.7×. While operator-level analysis confirms that our optimized joins and aggregations benefit OMPC, the reduced dataset size limits the magnitude of these improvements. These results demonstrate that Alchemy's optimization approach is an effective, protocol-agnostic solution for secure query processing.

6 RELATED WORK

There has been substantial research interest in oblivious query processing in the DB community. Table 6 compares Alchemy to other systems that securely compute over federated data. Conclave [52] provides efficient in-MPC evaluation for collaborative analytics, but it does not exploit schema constraints for optimization. Although Conclave considers sort and filter pushdown, it does not handle intra-operator sorting or address multiple intermediate collations like Alchemy. Senate [42] has a cost-based optimizer for its maliciously secure queries, but it relies on user-provided cardinality bounds, while Alchemy automatically generates these bounds using the public system catalog. Secrecy [37] serves PDF queries with outsourced computation using a mix of boolean and arithmetic circuits. Their plans minimize the cost of switching between these two circuit types. Secure Yannakakis [54] offers oblivious join-aggregate queries with a PSI-based protocol. It only supports vertically partitioned tables where each join column is held by one party, whereas VaultDB also offers joins on horizontally partitioned data.

Alchemy's circuit-based framework is protocol-independent and extensible for more than two parties. It uses boolean circuits alone to make its plans easily portable to other cryptographic protocols without specialized adapters to make them interoperable. Optimizing among multiple plug-and-play protocols was studied in [21, 28, 48, 51]. We speed up queries by reasoning about schema properties and transforming the logical plan by eliminating redundant sub-expressions, exploiting schema constraints to craft tighter cardinality bounds, and by simplifying operator-level circuits.

A second approach to oblivious querying is to use trusted hardware, especially for outsourcing to the cloud. Opaque [57] and ObliDB [20] offer some dummy padding to protect the size of intermediate results, but do not protect memory access patterns or instruction traces within the enclave, making them vulnerable to side-channel attacks from a curious observer. We use full oblivious padding by default and instead leverage hints from the system catalog to reduce our IR sizes. OCQ [15] goes a step further by using PK-FK relationships for reducing the size of join outputs. Alchemy builds on this by targeting generic relational DBMSs, leveraging richer metadata including public attribute domains, functional dependencies, and `CHECK` constraints.

There are numerous oblivious join algorithms as described in recent surveys [34, 50]. Oblivious hash joins [40] presently only support SH3PC. Similarly, PSI-based joins leak the IR size of the join result [42, 44, 54] (or require manual padding specifications), require vertical partitioning [54], or a 3PC protocol [38]. This limits their interoperability with multiple circuit backends or data layouts, making them unsuitable for general-purpose, n-party settings. Consequently, VaultDB adopts NLJ and SMJ for their generality, ease of integration into cost-based optimization, and compatibility with more computing parties.

7 CONCLUSION

In this paper, we introduce Alchemy, an optimization framework for oblivious query processing. Our approach generalizes traditional optimization techniques, pairing them with circuit-aware cost modeling to reduce the overhead of secure computing. Our experimental results demonstrate up to 2 OOM performance gains across various cryptographic protocols, validating both the effectiveness and protocol-agnostic nature of these techniques.

ACKNOWLEDGMENTS

This work was supported in part by NSF awards CNS-1846447 and CNS-2016240.

References

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 1–9.

[2] Apache Software Foundation. [n.d.]. Calcite. https://calcite.apache.org

[3] Apple and Google. 2021. Exposure Notification Privacy-preserving Analytics (ENPA) white paper. (2021).

[4] Arvind Arasu and Raghav Kaushik. 2014. Oblivious query processing. *ICDT* (2014).

[5] David W Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P Smart, and Rebecca N Wright. 2018. From keys to databases—real-world applications of secure multi-party computation. *Comput. J.* 61, 12 (2018), 1749–1771.

[6] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 535–548.

[7] K. E. Batcher. April 1968. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.

[8] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2016. SMCQL: Secure Querying for Federated Databases. *VLDB* 10, 6 (2016), 673–684. https://doi.org/10.14778/3055330.3055334

[9] 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.

[10] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. 2015. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *International conference on financial cryptography and data security*. Springer, 227–234.

[11] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. 2016. Students and Taxes: A Privacy-Preserving Social Study Using Secure Computation. In *Privacy Enhancing Technologies Symposium (PETS)*.

[12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. 2012. Deploying secure multi-party computation for financial data analysis. *Financial Cryptography and Data Security* (2012), 57–64.

[13] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1401–1412.

[14] Shumo Chu, Danyang Zhuo, Elaine Shi, and TH Hubert Chan. 2021. Differentially oblivious database joins: Overcoming the worst-case curse of fully oblivious algorithms. *Cryptology ePrint Archive* (2021).

[15] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2020. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–17.

[16] Bailu Ding, Vivek Narasayya, Surajit Chaudhuri, et al. 2024. Extensible Query Optimizers in Practice. *Foundations and Trends® in Databases* 14, 3-4 (2024), 186–402.

[17] David-Paul Dornseifer and Ben Liderman. 2024. Build secure multi-party computation (MPC) wallets using AWS Nitro Enclaves. https://aws.amazon.com/blogs/web3/build-secure-multi-party-computation-mpc-wallets-using-aws-nitro-enclaves/.

[18] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H choke points and their optimizations. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1206–1220.

[19] Cynthia Dwork. 2006. Differential privacy. *International Colloquium on Automata, Languages and Programming* (2006), 1–12. https://doi.org/10.1007/11787006{_}1

[20] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019), 169–183.

[21] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. 2022. CostCO: An automatic cost modeling framework for secure multi-party computation. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 140–153.

[22] Sheldon Finkelstein. 1982. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. 235–245.

[23] Inc. Google. [n.d.]. Private Join and Compute. GitHub repository: https://github.com/google/private-join-and-compute.

[24] Chun Guo, Jonathan Katz, Xiao Wang, Chenkai Weng, and Yu Yu. 2020. Better concrete security for half-gates garbling (in the multi-instance setting). In *Annual International Cryptology Conference*. Springer, 793–822.

[25] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*. Springer, 145–161.

[26] Matthias Jarke. [n.d.]. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*. Springer, 191–205.

[27] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. 2013. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics* 29, 7 (2013), 886–893.

[28] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. 2014. Automatic protocol selection in secure two-party computations. *International Conference on Applied Cryptography and Network Security* 8479 (2014), 566–584. https://doi.org/10.1007/978-3-319-07536-5-33

[29] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems* 34 (2021), 4961–4973.

[30] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*. Springer, 486–498.

[31] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: a differentially private sql query engine. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1371–1384.

[32] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. *PVLDB* 13, 12 (2020), 2132–2145. https://doi.org/10.14778/3407790.3407814

[33] Andrei Lapets, Eric Dunton, Kyle Holzinger, Frederick Jansen, and Azer Bestavros. 2015. *Web-based multi-party computation with application to anonymous aggregate compensation analytics*. Technical Report. Computer Science Department, Boston University.

[34] Shuyuan Li, Yuxiang Zeng, Yuxiang Wang, Yiman Zhong, Zimu Zhou, and Yongxin Tong. 2024. An experimental study on federated equi-joins. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[35] Xiling Li, Gefei Tan, Xiao Wang, Jennie Rogers, and Soamar Homsi. 2023. RESCU-SQL: Oblivious Querying for the Zero Trust Cloud. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4086–4089.

[36] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. 2023. ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1804–1816.

[37] 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*.

[38] Qian Luo, Yuhang Wang, Wei Dong, and Ke Yi. 2024. Secure Query Processing with Linear Complexity. *arXiv preprint arXiv:2403.13492* (2024).

[39] Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*. ACM, 19–30.

[40] 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.

[41] PCORI. 2015. Exchanging de-identified data between hospitals for city-wide health analysis in the Chicago Area HealthLNK data repository (HDR). *IRB Protocol* (2015).

[42] 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 *USENIX Security Symposium*. 2129–2146.

[43] Michael O. Rabin. 2005. How To Exchange Secrets with Oblivious Transfer. *IACR Cryptol. ePrint Arch.* 2005 (2005), 187. https://api.semanticscholar.org/CorpusID:15222660

[44] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing fast PSI from improved OKVS and subfield VOLE. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2505–2517.

[45] Anjana Rajan, Lucy Qin, David W Archer, Dan Boneh, Tancrede Lepoint, and Mayank Varia. 2018. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*. 1–4.

[46] Jennie Rogers, Elizabeth Adetoro, Johes Bater, Talia Canter, Dong Fu, Andrew Hamilton, Amro Hassan, Ashley Martinez, Erick Michalski, Vesna Mitrovic, et al. 2022. VaultDB: A Real-World Pilot of Secure Multi-Party Computation within a Clinical Research Network. *arXiv preprint arXiv:2203.00146* (2022).

[47] Wolfgang Scheufele and Guido Moerkotte. 1996. *Constructing Optimal Bushy Processing Trees for Join Queries is NP-hard*. Technical Report 96-011. School of Business Informatics and Mathematics, University of Mannheim. https://ub-madoc.bib.uni-mannheim.de/795

[48] Axel Schroepfer and Florian Kerschbaum. 2011. Forecasting run-times of secure two-party computation. In *2011 Eighth International Conference on Quantitative Evaluation of SysTems*. IEEE, 181–190.

[49] Aditya Shastri, David Wu, Lynn Chua, Ruiyu Zhu, Tal Davidi, Yu Sheng, and Josh Mintz. 2022. Private Computation Framework 2.0. https://research.facebook.com/publications/private-computation-framework-2-0/.

[50] Donghyun Sohn, Xiling Li, and Jennie Rogers. 2023. Everything You Always Wanted to Know About Secure and Private Database Systems (but were Afraid to Ask). *IEEE Data Engineering Bulletin* (2023).

[51] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*. 363–378.

[52] 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.

[53] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. 2016. EMP-Toolkit: Efficient Multiparty Computation Toolkit. https://github.com/emp-toolkit.

[54] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *SIGMOD Conference*. 1969–1981.

[55] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 501–518.

[56] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2986–3001.

[57] 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 *NSDI*. 283–298.