DJoin: Differentially Private Join Queries over Distributed Databases

Arjun Narayan
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Abstract

In this paper, we study the problem of answering queries about private data that is spread across multiple different databases. For instance, a medical researcher may want to study a possible correlation between travel patterns and certain types of illnesses. The necessary information exists today – e.g., in airline reservation systems and hospital records – but it is maintained by two separate companies who are prevented by law from sharing this information with each other, or with a third party. This separation prevents the processing of such queries, even if the final answer, e.g., a correlation coefficient, would be safe to release.

We present DJoin, a system that can process such distributed queries and can give strong differential privacy guarantees on the result. DJoin can support many SQL-style queries, including joins of databases maintained by different entities, as long as they can be expressed using DJoin's two novel primitives: BN-PSI-CA, a differentially private form of private set intersection cardinality, and DCR, a multi-party combination operator that can aggregate noised cardinalities without compounding the individual noise terms. Our experimental evaluation shows that DJoin can process realistic queries at practical timescales: simple queries on three databases with 15,000 rows each take between 1 and 7.5 hours.

1 Introduction

A vast amount of information is constantly accumulating in databases (social networks, hospital records, airline reservation systems, etc.) all around the world. There are many good uses to which this data could potentially be put; however, much of this data is sensitive and cannot safely be released because of privacy concerns. Simple solutions, such as anonymizing or aggregating the data before release, are not reliable; experience with cases like the Netflix prize [3] or the AOL search data [2] shows that such data can sometimes be de-anonymized with auxiliary information [26].

Differential privacy [7] has been proposed as a way to solve this problem. By disallowing certain queries, and by adding a carefully chosen amount of noise to the result of others, it is possible to give a strong upper bound on how much an adversary could learn about an individual person's data, even under worst-case assumptions. Several differentially private query processors, including PINQ [23], Airavat [32], Fuzz [16], and PDDP [6], have been developed and are available today.

However, existing query processors assume either that all the data is available in a *single* database [16, 23, 32] or that distributed queries can be broken into several subqueries that can each be answered using only one of the databases [6, 10, 15, 31]. In practice, this is not necessarily the case. For instance, suppose a medical researcher wanted to study how a certain illness is correlated with travel to a particular region. This data may be available, e.g., in a hospital database H and an airline reservation system R, but to determine the correlation, it is necessary to *join the two databases together* – for instance, we must count the individuals who have been treated for the illness (according to H) *and* have traveled to the region (according to R).

We are not aware of any existing method or query processor that can efficiently support join queries with differential privacy guarantees. Joins cannot be broken into smaller subqueries on individual databases because, in order to match up the same persons' data in the two databases, such queries would have to ask about individual rows, which is exactly what differential privacy is designed to prevent. In principle, one could process joins using secure multi-party computation (MPC) [38], but MPC is only practical for small computational tasks, and differential privacy only works well for large databases. The cost of an entire join under MPC would be truly spectacular.

DJoin, the system we present in this paper, is a solution to this problem. DJoin can support SQL-style queries across multiple databases, including common forms of joins. The key insight behind DJoin is that the distributed parts of many queries can be expressed as intersections of sets or multisets. For instance, we can rewrite the query from above to locally select all patients with the illness from H and all travelers to the relevant region from R, then intersect the resulting sets, and finally count the number of elements in the intersection. Not all SQL queries can be rewritten in this way, but

many counting queries can: conjunctions and disjunctions of equality tests directly correspond to unions and intersections of data elements. As we will show, a number of additional operations, such as inequalities and numeric comparisons, can be expressed in terms of multi-set operations.

Protocols for private set operations have been studied by cryptographers for some time [14, 17, 37], but existing solutions compute exact set elements or exact cardinalities, which is not compatible with differential privacy. We present *blinded, noised private set intersection cardinality (BN-PSI-CA)*, an extension of the set-intersection protocol from [17] that supports private noising, as well as *denoise-combine-renoise (DCR)*, an operator that can add or subtract multiple noised subset cardinalities without compounding the corresponding noise terms. DCR relies on MPC to remove the noise terms on its inputs and to re-noise the output, but DCR's complexity grows with the number of parties and not with the number of elements in the sets. For the queries we tried, this step never took more than 20 seconds.

We have implemented and evaluated a prototype of DJoin. Our results show that the costs are substantial but typically feasible. For instance, the elements in a simple two-way join on databases with 32,000 rows each can be evaluated in about 1.8 hours, with 83 MB of traffic, using a single commodity workstation for each database. This is orders of magnitude faster than general MPC. DJoin's cost is too high for interactive use, but it seems practical for applications that can tolerate a certain amount of latency, such as research studies. Our algorithms are easy to parallelize, so the speed could be improved by increasing the number of cores.

To summarize, this paper makes the following four contributions:

- two new primitives, BN-PSI-CA and DCR, for distributed private query processing (Section 4);
- a query planner that rewrites SQL-style queries to take advantage of those two primitives (Section 5);
- the design of DJoin, an engine for distributed, differentially private queries (Section 6); and
- an experimental evaluation of DJoin, based on a prototype implementation (Section 7).

2 Related work

DJoin provides differential privacy [7, 8, 9, 11], which is one of the strongest privacy guarantees that have been proposed so far. Alternatives include randomization [1], k-anonymity [34], and l-diversity [21], which are generally less restrictive but can be vulnerable to certain attacks on privacy [12, 20]. Differential privacy offers a provable bound on the amount of information that an attacker can learn about any individual, even with access to auxiliary information.

Differentially private query processors: PINQ [23], Airavat [32], and Fuzz [16] are query processors that support differential privacy, but they assume a centralized setting in which a single entity has access to the entire data. We are aware of five solutions for distributed settings [6, 10, 15, 31, 33], but these assume that the data is horizontally partitioned (i.e., each individual's data is completely contained in one of the databases), and that the query can be factored into subqueries that are each local to a single database. For instance, [10] computes queries of the form $\sum_i f(d_i)$, i.e., the sum over all rows i in the database after applying a function f to each row. DJoin's data model is more general: multiple databases may contain data for a given individual, and queries can contain joins. We note that some of the other systems have far more sophisticated query languages, but we speculate that DJoin's rewriting and execution engine could be integrated with existing systems, e.g., with PINQ or Fuzz.

Private set operations: The first protocols for private two-party set intersection and set intersection cardinality were proposed by Freedman et al. [14]. Since then, a number of improvements have been proposed; for instance, Kissner and Song [17] extended the protocols to multiple parties, and Vaidya and Clifton [37] reduced the computational overhead. These protocols produce exact results, and are thus not directly suitable for differential privacy. There are specialized protocols for other private multi-party operations, e.g., for decision-tree learning [29], and some of these have been adapted for differential privacy, e.g., [39].

Computational differential privacy: The standard definition of differential privacy is information-theoretic, i.e., it holds even against a computationally unbounded adversary. In contrast, DJoin provides computational differential privacy [25]: it relies on a homomorphic cryptosystem and thus depends on certain computational hardness assumptions. Mironov et al. [25] demonstrated a protocol for this model that privately approximates the Hamming distance between two vectors in a two-party setting. This problem is closely related to that of computing the cardinality of set intersections, which is solved by BN-PSI-CA.

Untrusted servers: Several existing systems enable clients to use an untrusted server without exposing private information to that server. In SUNDR [19], SPORC [13], and Depot [22], the server provides storage; in CryptDB [30], it implements a database and SQL-style queries. This approach is complementary to ours: DJoin's goal is to reveal *some* useful information about the data it stores, but with an upper bound on how much can be learned about a single individual.

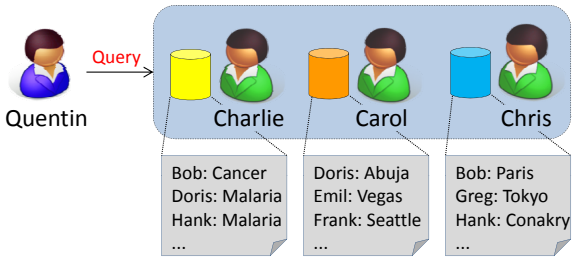


Figure 1: Motivating scenario. Charlie is a physician, and Carol and Chris are travel agents. Quentin would like to know the correlation between treatment for malaria and travel to high-risk areas.

3 Background and overview

3.1 Motivating scenario

Figure 1 shows our motivating scenario. Charlie, Carol, and Chris each have a database with confidential information about individuals; for instance, Charlie could be a physician, and Carol and Chris could be travel agents. We will refer to these three as the *curators*. Quentin asks a question that combines data from each of the databases; for instance, he might want to know the correlation between treatment for malaria and travel to areas with a high risk of malaria infections. We will refer to Quentin as the *querier*.

Our goal is to build a system that can give an (at least approximate) answer to Quentin's question while offering strong privacy guarantee to the individuals whose data is in the databases. In particular, we would like to establish an upper bound on how much *additional* information any participant of the system (queriers or curators) can learn about any individual in the database. The word 'additional' is crucial here, since the curators each have full access to their respective databases. For instance, since Charlie has treated Bob for cancer, our system cannot prevent him from learning this fact, but it can prevent him from learning whether or not Bob has recently traveled to Paris.

3.2 Differential privacy

To formally define the privacy guarantee we want to provide, we rely on *differential privacy* [7]. Differential privacy is a property of randomized queries that take a database as input and return a result that is typically some form of aggregate (such as a number representing a count, a histogram, etc). The database is seen as a collection of rows, and each row contains the data from one individual.

Informally, a randomized function is differentially private if arbitrary changes to a single individual's row (while keeping the other rows constant) result in only statistically insignificant changes in the function's output distribution. Thus, the presence or absence of any individual has a statistically negligible effect. Formally [11], differential privacy is parametrized by a real number ε, which corresponds to the strength of the privacy guarantee; smaller values of ε yield better privacy. Two databases b and b' are considered *similar*, written $b \sim b'$, if they differ in only one row. We then say that a randomized function f with range R is *ε-differentially private* if, for all possible sets of outputs $S \subseteq R$, and for all similar databases b, b', we have

$$Pr[f(b) \in S] \leq e^{\varepsilon} \cdot Pr[f(b') \in S]$$

That is, when the input database is changed in one row, there is at most a small multiplicative difference (e^{ε}) in the probability of *any* set of outcomes S. A slightly weaker variant of this privacy definition is (ε, δ)-differential privacy [10], where δ is a bound on the maximum *additive* (not multiplicative) difference between the probabilities of a given output with and without a particular input row.

Practical solutions for achieving differential privacy typically rely on adding a carefully chosen amount of noise to the result. The required amount of noise depends on the *sensitivity* of the query, i.e., how much the result can change in response to changing the data in a single row [11]. More formally, if q is a function that computes the (exact) result of the query and $|q(b) - q(b')| \leq s$ for any pair of similar databases $b \sim b'$, the query is s-sensitive, and we can construct an ε-differentially private function f by adding noise to s that is drawn from a Laplace distribution with parameter $\lambda = s/\varepsilon$. This corresponds to the intuition that more sensitive queries need more noise to conceal the contributions of any given individual.

3.3 Challenge: Distribution

Answering differentially private queries over a *single* database is a well-studied problem, and several systems [16, 23, 32] are already available for this purpose. In principle, these systems can also be used to answer queries across multiple databases, but this requires that all curators turn over their data to a single trusted entity (e.g., one of the curators), who evaluates the query on their behalf. However, there may not always be a single entity that is sufficiently trusted by all the curators, so it seems useful to have an alternative solution that does not require a trusted entity.

In some cases, distributed queries can be factored into several subqueries that can each be executed on an individual database. For instance, a group of doctors can count the number of male patients in their respective

databases by counting the number of patients in *each* database separately, and then add up the (individually noised) results. This type of distributed query is supported by several existing systems [6, 10, 15, 31, 33]. However, not all queries can be factored in this way. For instance, the above approach will double-count male patients that have been treated by more than one doctor, but a union query (which would avoid this problem) cannot be expressed as a sum of counts. Similarly, any query that involves joining several databases (such as our motivating example) cannot be expressed in this way.

Joins *could* be supported via general-purpose multiparty computation (MPC) [38], but the required runtimes would be gigantic: state-of-the-art MPC solutions, such as FairplayMP [4], need about 10 seconds to evaluate (very simple) functions that can be expressed with 1,024 logic gates. Since the number of gates needed for a join would be at least quadratic in the number of input rows, and since differential privacy only works well for large databases, this approach does not seem practical.

3.4 Approach

The key insight behind our solution is that joins are rarely used to compute full cross products of different databases; rather, they are often used to 'match up' elements from different databases. For instance, in our running example, we can first select all the individuals in R who have traveled to the region of interest, then select all the individuals in H who have been treated for the illness, and finally count the number of individuals who appear in both sets. Thus, the problem of privately answering the overall query is reduced to 1) some local operations on each database, and 2) privately computing the cardinality of the intersection of multiple sets. Not all queries can be decomposed in this way, but, as we will show in Section 5, there is a substantial class of queries that can.

Protocols for private multiset operations (such as intersection and union) are available [14, 17, 37], but they tend to compute *exact* sets or set cardinalities. If we naïvely used these algorithms, Charlie could compute the intersection of the set of the malaria patients in his database with the sets of customers in Carol's and Chris' databases who have traveled to high-risk areas, and then add noise in a collaborative fashion [10]. This would prevent Quentin from learning anything other than the (differentially private) output of the query — but Charlie could learn where his patents have traveled, and Carol and Chris could learn which of their customers have been treated for malaria. Hence, our first challenge is to extend these set-intersection operations to support noising between the data curators.

A second challenge arises because some queries involve multiple set operations. If Charlie simply added the two cardinalities together, the noise terms would compound, and thus (unnecessarily) degrade the quality of the overall result. To avoid this problem, we need a way to de-noise, combine, and re-noise intermediate results without compromising privacy.

4 Building blocks: BN-PSI-CA and DCR

Next, we describe two key building blocks that enable private processing of distributed queries. Each building block performs only one, very specific operation. In Section 5, we will describe how these building blocks can be used in a larger query plan to answer a variety of different queries.

4.1 Background: PSI-CA

Our first building block is related to a primitive called *private set-intersection cardinality (PSI-CA)*, which allows a group of k curators with multisets $S_1, \ldots, S_k$ to privately compute $|\bigcap_i S_i|$, i.e., the (exact) number of elements they have in common, but *not* the specific elements in $\bigcap_i S_i$. PSI-CA is a well-studied primitive [14, 17, 37], albeit not in the context of differential privacy. To explain the intuition, we describe one simple PSI-CA primitive [14] for only two curators with simple sets in the honest-but-curious (HbC) model. The primitive uses a homomorphic encryption scheme that preserves addition and allows multiplication by a constant. Paillier's cryptosystem [28] is an example of a scheme that has this property.

Suppose the two curators are C_1 and C_2 and their sets are $S_1 := \{x_1, \ldots\}$ and $S_2 := \{y_1, \ldots\}$. C_1 defines a polynomial $P(z)$ over a finite field whose roots are his set elements x_i:

$$P(z) := (x_1 - z)(x_2 - z) \cdots = \sum_u \alpha_u z^u$$

Next, C_1 sends homomorphic encryptions of the coefficients α_u to C_2, along with the public key. For each element $y_i \in S_2$, C_2 then computes $\mathrm{Enc}(rP(y_i) + 0^+)$, i.e., she evaluates the polynomial at each of her inputs, multiplies each result by a fresh random number r, and finally adds a special string 0^+, e.g., a string of zeroes. Since the cryptosystem is homomorphic, C_2 can do this even though she does not know C_1's private key. Finally, C_2 sends a random permutation of the results back to C_1, who decrypts them and counts the occurrences of the special string 0^+, which is exactly $|S_1 \bigcap S_2|$.

At first glance, the cost of this algorithm appears to be quadratic: C_2 must compute $\mathrm{Enc}(rP(y_i) + 0^+)$ for each of her $|S_2|$ inputs, which involves computing $\mathrm{Enc}(P(y_i))$ along the way. If this is naïvely evaluated as $\mathrm{Enc}(\sum_{u=0}^{|S_1|} \alpha_u y_i^u)$, C_2 must multiply each of the $|S_1| + 1$

encrypted coefficients with an unencrypted constant (y_i^u), which requires an exponentiation each time, for a total of $O(|S_1| \cdot |S_2|)$ exponentiations. However, [14] describes several optimizations that can reduce this overhead, including an application of Horner's rule and the use of hashing to replace the single high-degree polynomial with several low-degree polynomials. This reduces the computational overhead to $O(|S_1| + |S_2| \ln \ln |S_1|)$ exponentiations.

4.2 BN-PSI-CA: Two-party case

The basic PSI-CA primitive is not compatible with differential privacy because C_1 learns the *exact*, un-noised size of $|S_1 \bigcap S_2|$; moreover, each curator can learn the size of the other curator's set by observing the number of encrypted coefficients, or encrypted return values, that are received from that curator. However, we can extend the primitive to avoid both problems.

First, we need to make the number of coefficients and return values independent of the set sizes. We can do this by adding some extra elements that cannot appear in either of the sets. As long as we can ensure that C_1 and C_2 are adding different elements (e.g., by setting some bit to zero on C_1 and to one on C_2), this will not affect the size of the intersection. In DJoin, we assume that a rough upper bound on the size of each curator's database is known, and we add enough elements to fill up both sets to that upper bound.

Second, we need to add some noise n to the result that is revealed to C_1. We observe that C_2 can increase the apparent size of the intersection by n if she adds n different[1] encodings of the special string 0^+. However, to guarantee ε-differential privacy, we would have to draw n from a Laplace distribution $\mathrm{Lap}(1/\varepsilon)$, and this would sometimes yield $n < 0$ – but C_2 cannot *remove* encodings of 0^+ because she does not have C_1's private key, and thus cannot tell them apart from encodings of other values. Instead, we require C_2 to draw n from $X_2 + \mathrm{Lap}(1/\varepsilon)$ and we cut n at 0 and $2 \cdot X_2$; thus, C_2 can add n encodings of 0^+ and $2 \cdot X_2 - n$ encodings of a random value to keep the overall size independent of n. (Cutting the Laplace distribution can leak a small amount of information when the extremal values are drawn, and thus changes the privacy guarantee to (ε, δ)-differential privacy [10]; however, by increasing X_2, we can make δ arbitrarily small, at the expense of a higher overhead.) We call the resulting primitive *blinded noised PSI-CA (BN-PSI-CA)*.

Note that at the end, C_2 knows the noise term n and C_1 the noised cardinality $|S_1 \bigcap S_2| + n$. Thus, if the latter is used in further computations, we have an opportunity to remove the noise again, as long as we can ensure that neither curator learns both values. This prevents the noise terms from compounding, and it enables us to use a *very* high noise level (and thus a low value of ε) because the noise will not affect the final result.

[1] The Paillier cryptosystem can construct many different ciphertexts for the same plaintext.

4.3 BN-PSI-CA: Multi-party case

Since Freedman's initial work, cryptographers have considerably extended the range of private multiset operations. For instance, the protocol by Kissner and Song [17] also supports set unions, as well as set intersections with more than two parties, and it is *compositional*: the result of a set union or set intersection can be unioned or intersected with further sets, without decrypting it first. [17] can evaluate any function on multisets that can be described by the following grammar:

$$\Upsilon ::= s \,|\, \Upsilon \cap \Upsilon \,|\, s \cup \Upsilon \,|\, \Upsilon \cup s$$

where s is a multiset that is known to some curator C_i.

The protocol from [17] computes $|\bigcap_{i=1,\ldots,k} S_i|$ as follows. First, the k curators use a homomorphic threshold cryptosystem to share a secret key sk amongst themselves, while the corresponding public key pk is known to all curators. Each curator C_i now encrypts a polynomial P_i whose roots are the elements of its local set S_i. The encrypted polynomials are then essentially added together, yielding a polynomial P whose roots are the elements in the intersection. Each curator C_i now evaluates P on the elements e_{ij} of his local set S_i, yielding values $v_{ij} := P(e_{ij})$; however, recall that, because sk is shared, no individual curator can decrypt the v_{ij}. The curators then securely re-randomize and shuffle [27] the v_{ij}, such that each curator learns all the v_{ij} but cannot tell which curator it came from. Finally, the curators jointly decrypt the v_{ij}. If there are n elements in the intersection, this yields $n \cdot k$ zeroes; hence, each curator can compute the final result by dividing the number of zeroes by k.

We can use the same blinding technique as in Section 4.2 to construct a multi-party version of BN-PSI-CA. After computing the v_{ij}, but before the shuffle, each curator draws a noise term n_i as above and adds $2 \cdot X_i$ extra values, n_i of which are 0^+. As above, this adds $\sum_i n_i$ to the resulting cardinality, but the noise can be removed again via DCR, which we discuss next.

4.4 DCR: Adding cardinalities

BN-PSI-CA is sufficient to answer queries that require a single distributed multiset operation. However, in Section 5.2 we will see that some queries require multiple operations, and that the result is then a linear combination of the different cardinalities. In principle, we could

designate a single curator C that collects all cardinalities and computes the overall result; however, this would a) compound all the noise terms and thus decrease the quality of the result, and b) reveal all the intermediate results to C and thus (unnecessarily) reveal some private information.

Instead, we can combine the various cardinalities using secure multi-party computation (MPC) [38]. If we have a number of players with private inputs x_i that are each known to only one of the players, MPC allows the players to collectively compute a function $f(x_1, x_2, \ldots)$ *without* revealing the inputs to each other. Even after decades of research, MPC remains impractical for complex functions or large inputs, but modern implementations, such as [4], can process simple functions in a few seconds or less. Thus, while MPC may be too expensive to evaluate the entire query, we can certainly use it to combine a small number of subquery results.

For instance, suppose the query is for $|S_1 \bigcap S_2| + |S_3 \bigcap S_4|$, and that there are four curators involved: C_1 and C_3 learn the noised results R_1 and R_2 for the first and the second term, respectively, and C_2 and C_4 learn the corresponding noise terms n_1 and n_2. Then we can compute the query result under MPC as

$$q = R_1 + R_2 - (n_1 + n_2) + N$$

where each of the four curators contributes one of the private inputs R_i and n_i, and N is a new, global noise term. Next, we describe how N is computed.

4.5 DCR: Cooperative noising

MPC enables us to safely remove the noise that was added to the individual cardinalities by BN-PSI-CA, but we must add back a sufficient amount of noise N *as part of the MPC*, i.e., before the result is revealed. To prevent information leakage, the new noise N must be such that no individual curator can control it or predict its value.

We follow the algorithm in [10] to generate the noise N, with some implementation modifications. Each curator chooses a random bitstring v_i uniformly at random and contributes it as an input to the MPC. The MPC computes $v := v_1 \oplus v_2 \oplus \cdots$. As long as a curator honestly chooses v_i uniformly at random and does not share this with any other party, she can be certain that no other curator can know anything else about the computed noise string v, even if every single other curator colludes. Finally, the MPC uses the fundamental transformation law of probabilities to change the distribution of v to a Laplace distribution $\text{Lap}(1/\varepsilon)$. This yields the noise term N, which is then added to the query result. We call this primitive *denoise-combine-renoise (DCR)*.

```
query     := SELECT output FROM union
             WHERE predicate
output    := NOISY COUNT(field)
union     := rows | union UNION ALL rows
rows      := join | subquery
join      := db{,db}*
subquery  := SELECT fields FROM join
             WHERE predicate
predicate := term | predicate OR term |
             predicate AND term
term      := val = val | val != val |
             val < val
val       := number | string | db.field
```

Figure 2: DJoin's query language.

5 Distributed query processing

So far, we have described BN-PSI-CA, which can compute differentially private set intersection cardinalities, and DCR, which can privately combine multiple cardinalities. Next, we describe how DJoin integrates these two primitives into larger query plans that can answer SQL-style queries.

5.1 Query language: SPJU

For ease of presentation, we describe our approach using the simple query language in Figure 2, which consists of SQL-style operators for selection, projection, a cross join, and union (SPJU). This query language is obviously much simpler than SQL itself, but it is rich enough to capture many interesting distributed operations. We note that many of the missing features of SQL can easily be added back, as long as queries do not use them to access more than one database at a time.

Each query in our language can be translated into relational algebra, specifically, in a combination of selections (σ), projections (π), joins ($\bowtie$), unions ($\bigcup$), and counts ($|\cdot|$). For instance, the query

```
SELECT COUNT(A.id) FROM A,B
WHERE (A.ssn=B.ssn OR A.id=B.id)
  AND A.diagnosis='malaria'
```

could be written (with abbreviations) as:

$$|\,\sigma_{(\text{A.ssn=B.ssn} \vee \text{A.id=B.id}) \wedge \text{A.diag="malaria"}}(A \bowtie B)\,|$$

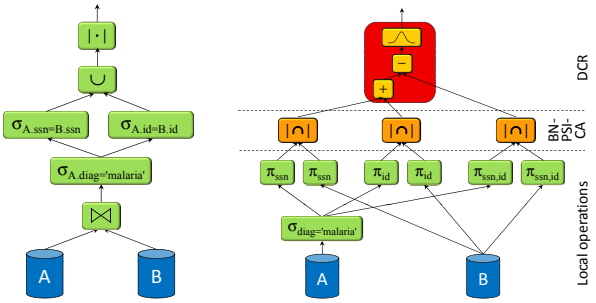
Figure 3(a) shows a graphical illustration of this query.

5.2 Query rewriting

Most distributed queries cannot be executed natively by DJoin because they contain operators (such as $\bowtie$ or $<$) that our system cannot support. Therefore, such queries

		From	To
R1	Local sel.	$\sigma_{P(X)\wedge Q}(X \bowtie Y)$	$\sigma_Q(\sigma_P(X) \bowtie Y)$
R2	Disjunction	$\sigma_{P\vee Q}(X \bowtie Y)$	$\sigma_P(X \bowtie Y) \bigcup \sigma_Q(X \bowtie Y)$
R3	Split	$\lvert\sigma_{X.a=Y.b\wedge(P(X)\vee Q(Y))}(X \bowtie Y)\rvert$	$\lvert\sigma_{X.a=Y.b}(\sigma_P(X) \bowtie Y)\rvert + \lvert\sigma_{X.a=Y.b}(\sigma_{\neg P}(X) \bowtie \sigma_Q(Y))\rvert$
R4	Union	$\lvert X \bigcup Y\rvert$	$\lvert X\rvert + \lvert Y\rvert - \lvert X \bigcap Y\rvert$
R5	Not equal	$\lvert\sigma_{X.a=Y.b\wedge X.c\neq Y.d}(X \bowtie Y)\rvert$	$\lvert\sigma_{X.a=Y.b}(X \bowtie Y)\rvert - \lvert\sigma_{X.a=Y.b\wedge X.c=Y.d}(X \bowtie Y)\rvert$
R6	Comparison	$\lvert\sigma_{X.a=Y.b\wedge X.c>Y.d}(X \bowtie Y)\rvert$	$\sum_{i=0..k-1} \lvert\pi_{\text{a}\,\lvert\lvert\,\text{pre(c,i)}}(\sigma_{\text{bit(c,i)=1}}(X)) \bigcap \pi_{\text{b}\,\lvert\lvert\,\text{pre(d,i)}}(\sigma_{\text{bit(d,i)=0}}(Y))\rvert$
R7	Equality	$\lvert\sigma_{X.a=Y.b\wedge X.c=Y.d}(X \bowtie Y)\rvert$	$\lvert\sigma_{(X.a\lvert\lvert\,\text{pad}\lvert\lvert X.c)=(Y.b\lvert\lvert\,\text{pad}\lvert\lvert Y.d)}(X \bowtie Y)\rvert$
R8	Join	$\lvert\sigma_{X.a=Y.b}(X \bowtie Y)\rvert$	$\lvert\pi_a(X) \bigcap \pi_b(Y)\rvert$

Table 1: DJoin's rewrite rules. These rules are used to transform a query (written in the language from Figure 2) into the intermediate query language from Figure 4, which can be executed natively.



(a) Original plan (b) Rewritten for DJoin

Figure 3: Query example. The original plan (left) cannot be executed without compromising privacy. The rewritten plan (right) consists of three tiers: a local tier, a BN-PSI-CA tier, and a DCR tier.

must be transformed into other queries that are semantically equivalent but contain only operators that our system *can* support, which are a) any SQL queries on a single database that produce a noisy count or a multiset; b) BN-PSI-CA; and c) DCR. Figure 4 shows the language that can be supported natively. DJoin uses a number of rewrite rules to perform this transformation. The most interesting rules are shown in Table 1; some trivial rules, e.g., for transforming boolean predicates, have been omitted.

Local selects: We try to perform as many operations as possible locally at each database, e.g., via rule R1 for selects that involve only columns from one database.

Disjunctions: We use basic boolean transformations to move any disjunctions in the join predicates to the outermost level, where they can be replaced by set unions using rule R2, or split off using rule R3.

Unions: Rule R4 (which is basically De Morgan's law) replaces all the set unions with additions, subtractions, and set intersections.

Inequalities: Rule R5 replaces the $\neq$ operators with an equality test and a subtraction; rule R6 encodes integer comparisons as a sum of equalities. Both rules assume that there is a nearby equality test for matching rows.

```
query    := cardex | cardex + cardex |
            cardex - cardex
cardex   := |setex ∩ setex {∩ setex}*|
setex    := π_fields(sigmaex) | sigmaex
sigmaex  := σ_local_predicate(db) | db
```

Figure 4: DJoin's intermediate language.

Equalities: Once all non-local operations in the join predicates are conjunctions of equality tests, we can use rule R7 to reduce these to a single equality test, simply by concatenating the relevant columns in each database (with appropriate padding to separate columns).

Joins: Once a join cardinality has only one equality test left, rule R8 replaces it with an intersection cardinality.

5.3 Result: Three-tier query plan

If the rewriting process has completed successfully, the rewritten query should now conform to our intermediate language from Figure 4, which implies a three-tier structure: the first tier (`sigmaex` and `setex`) consists of local selections and projections that involve only a single database; the second tier (`cardex`) consists of set intersection cardinalities, and the third tier (`query`) consists of arithmetic operations applied to cardinalities. We refer to the rewritten query as a *query plan*. Figure 3(b) shows a query plan for the query from Figure 3(a) as an illustration.

A query plan with this three-tier structure can be executed in a privacy-preserving way. The first tier can be evaluated using classical database operations on the individual databases; the second tier can be evaluated using BN-PSI-CA (Section 4.2 and 4.3), and the third tier can be evaluated using DCR (Section 4.4 and 4.5).

5.4 Limitations

DJoin has only two distributed operators: BN-PSI-CA and DCR. If a query cannot be rewritten into a query

plan that uses only those operators (and some purely local ones), it cannot be supported by DJoin. For instance, DJoin currently cannot process the query

```
SELECT COUNT(A.id) FROM A,B,C
WHERE ((A.x*B.y)<C.z)
```

because we know of no efficient way to rewrite the predicate into set intersections. Rewriting is generally difficult for predicates that involve computations across fields from multiple databases. The predicates DJoin *can* support include 1) predicates that use only fields from a single database, 2) equality tests between fields from different databases, and 3) conjunctions and disjunctions of such predicates. In addition, DJoin supports operators for which it has an explicit rewrite rule, such as inequalities and numeric comparisons (rules R5 and R6). We do not claim that we have found all possible rewrite rules; if rules for additional operators are discovered, DJoin could be extended to support them as well.

DJoin is currently limited to counting queries: it does not support sum queries, or queries with non-numeric results. Differential privacy can in principle support such queries, e.g., via the exponential mechanism [24], but we have not yet found a way to express them in terms of set intersections.

6 DJoin design

In this section, we present the design of DJoin, our system for processing distributed differentially-private queries using the mechanisms explained so far.

6.1 Assumptions

Our design is based on the following assumptions:

1. All queriers know the schema and a rough upper bound on the total size of each curator's database.
2. The curators are "honest but curious", i.e., they will learn whatever information they can, but they will not deviate from the protocol.
3. Each curator has a "privacy budget" that represents to amount of private information he or she is willing to release through queries.
4. The curators can authenticate each querier.

Assumption 1 is necessary to make BN-PSI-CA and query planning work. Assumption 2 is not inherent (PSI-CA can work in an adversarial model [17]) but helps with efficiency and does not seem unreasonable in practice. Assumption 3 is common for differentially private query processors [16, 23, 32], and assumption 4 can be satisfied, e.g., using cryptographic signatures.

6.2 Overview and roadmap

DJoin consists of a number of *servers*, which run on the curators' machines, as well as at least one *client*, which runs the querier's machine and communicates with the servers to execute queries. Each server has a privacy budget (Section 6.3) and a local database with a schema (Section 6.4) that is known to all clients and servers.

Users can interact with DJoin by issuing a query q and a requested accuracy level ν to their local client. (ν is the parameter of the Laplace distribution from which DCR will draw the final noise term.) The user's client attempts to rewrite the query according to the rules from Section 5.2. If this succeeds, the result is a different query q' that is equivalent to q but can be executed entirely with local queries, BN-PSI-CA, and DCR. The client then submits the query to the servers, and each server performs an analysis (Section 6.5) to determine the sensitivity $S(q, db_i)$ of the query q in that server's local data db_i. In combination with the accuracy level ν, the sensitivity yields the privacy cost ε_i that this server will incur for answering the query.

Next, the client then uses a distributed commit protocol (Section 6.6) to assign an identifier to the query and to ensure that all the servers agree which query is being executed. Once the query is committed, the servers execute the query in three stages (Section 6.7): first, each server completes any subqueries that involve only its local database; next, the servers jointly complete each of the BN-PSI-CA operations; and finally, the servers execute DCR to combine and re-noise their results. The overall result is then revealed to the client.

6.3 Privacy budget

Each server maintains three pieces of local information: A local database, a privacy budget, and a table of pending queries, which is initially empty.

The *privacy budget* is essentially an upper bound on the amount of private information about any individual that the curator owning the server is willing to release through answering queries. It is well known [7] that, if q_1 and q_2 are two queries that are ε_1- and ε_2-differentially private, respectively, the sequential composition of both is $(\varepsilon_1 + \varepsilon_2)$-differentially private. Because of this, servers can simply deduct each query's "privacy cost" from the budget separately, without having to remember previous queries. A similar construction is used in other differentially private query processors, including PINQ [23], Airavat [32], and Fuzz [16]. In the appendix, we briefly sketch a possible approach to choosing the privacy budget.

Recall from Section 4 that DJoin must charge the privacy budget both for intermediate results from BN-PSI-CA operations and for the final result that is revealed by

DCR. To avoid confusion, we use the symbol ε_p to denote the cost of a BN-PSI-CA operation and ε_r to denote the cost of the final result. The total cost of a query with several BN-PSI-CAs is thus $\varepsilon_r + \sum_j \varepsilon_{p,j}$.

6.4 Schemata and multiplicities

The local database is a relational database that can be maintained in a classical, non-distributed DBMS, e.g., mySQL. For simplicity, we will assume that the data from each individual user is collected in a single row of the database; if this is not the case already, a normalization step (e.g., a `GROUP BY`) must be performed first. The database schema may assign an arbitrary type $\tau(c)$ to each column c; however, to make our sensitivity analysis work, we additionally allow each column to be annotated with a *multiplicity* $m(c)$ that indicates how often any individual value can appear in that column (for instance, $m(c) = 1$ indicates a column of unique keys). If no annotation is present, DJoin assumes $m(c) = \infty$.

Multiplicities are important to determine an upper bound on sensitivity of a query. Recall from Section 3.2 that the sensitivity $S(q, db_i)$ of a counting query q in a database db_i is the largest number of rows that a change to a single row in D can cause to be added or removed from the result of q. For instance, consider the query

```
SELECT COUNT(A.x) FROM A,B
WHERE A.x=B.y
```

If the multiplicities are $m(A.x) = 3$ and $m(B.y) = 5$, then a change to a single row in A can add at most five rows to the result – hence, whatever the new value of $A.x$ is, we know that B can contain at most five rows whose y-column matches that value. (The argument for disappearing rows is analogous.) Conversely, the query's sensitivity in B is three because at most three rows in A can have the value $B.y$ in column x. Note that processing such queries as intersections requires an extra encoding step; see the appendix for details.

Clearly, the use of a column with unbounded multiplicity can cause the sensitivity to become unbounded as well. However, it is safe to use such columns in conjunction with others; for instance, the query

```
SELECT COUNT(A.x) FROM A,B
WHERE A.x=B.y AND A.p=B.q
```

has sensitivity 5 in A even if $m(A.p) = m(B.q) = \infty$.

It may seem tempting to let DJoin choose the multiplicity itself, based on how often elements *actually* occur in the database. However, this would create a side channel: queriers could learn private facts about the database by observing, e.g., how much is deducted from the privacy budget after running certain queries. To avoid this problem, DJoin follows the approach from [16] and determines the multiplicity statically, without looking at the data.

6.5 Sensitivity analysis

We now describe how to infer the sensitivity of more complex queries, and specifically on the question how much the number of rows output by a query $\sigma_{\text{pred}}(db_1 \bowtie \cdots \bowtie db_k)$ can change if a single row in one of the db_i is changed.

To explain the intuition behind our analysis, we begin with a few simple examples:

1. $A \bowtie B \bowtie C$
2. $\sigma_{\text{A.x=B.y}}(A \bowtie B \bowtie C)$
3. $\sigma_{\text{A.x=B.y} \wedge \text{ B.y=C.z}}(A \bowtie B \bowtie C)$
4. $\sigma_{\text{A.x=B.y} \wedge \text{ A.p=B.q}}(A \bowtie B \bowtie C)$
5. $\sigma_{\text{A.x=B.y} \wedge \text{ B.y=C.z} \wedge \text{ A.x=C.q}}(A \bowtie B \bowtie C)$

Since query (1) has no predicates, its sensitivity in A is simply $|B| \cdot |C|$. The addition of the constraint $A.x = B.y$ changes the sensitivity to $m(B.y) \cdot |C|$, since each row in A can now join with at most $m(B.y)$ rows in B; similarly, adding $B.y = C.z$ in query (3) reduces the sensitivity to $m(B.y) \cdot m(C.z)$. When there is a conjunction of multiple constraints between the same databases, the most selective one 'wins'; hence, the sensitivity of query (4) is $\min(m(B.y), m(B.q)) \cdot |C|$. When there are multiple 'join paths', the most restrictive one wins. For instance, in query (5), the third constraint reduces the sensitivity in A only if $m(C.q) < m(B.y) \cdot m(C.z)$; otherwise, the sensitivity is the same as for query (3).

To solve this problem in the general case, we adapt a classical algorithm from the database literature [18] that was originally intended for query optimization in the presence of joins. This algorithm builds a *join graph* G that contains a vertex for each database that participates in the join, and a directed edge between each pair (db_1, db_2) of vertices that is initially annotated with $|db_2|$, the size of the database db_2. We then consider each of the predicates in turn and update the edges. Specifically, for each predicate $db_i.f_1 = db_j.f_2$ with $db_i \neq db_j$, we change the annotation $w_{i,j}$ on the edge (db_i, db_j) to $\min(w_{i,j}, m(db_j.f_2))$ and, correspondingly, the annotation $w_{j,i}$ on (db_j, db_i) to $\min(w_{j,i}, m(db_i.f_1))$. Then we can obtain an upper bound on the sensitivity $S(q, db_i)$ of q in some database db_i by finding the min-cost spanning tree that is rooted at db_i, using the product of the edge annotations as the cost function.

If the predicate contains disjunctions, we can rewrite it into DNF and then add up the sensitivity bounds. This is sound because $\sigma_{p \vee q}(X) = \sigma_p(X) \bigcup \sigma_q(X)$. If a row is removed from X and the sensitivities of p and q are s_p and s_q, this can change the cardinalities of the two sets by at most s_p and s_q, and thus the cardinality of the union by at most $s_p + s_q$. The same approach also works for unions of subqueries.

6.6 Distributed commit

Next, we describe how the client submits the query to the servers. It is important to ensure that the servers agree on which query they are executing; without this, a malicious client could trick a server into believing that it is executing a low-sensitivity query, and thus cause an insufficient amount of noise to be added to the result. Note that there is no need to agree on an ordering because all queries are read-only.

When the client accepts a query q with requested noise level ν from the user, it first calculates the sensitivity of q and the corresponding ε; then it tries to rewrite q into an equivalent query q' that uses only the language from Figure 4. If this succeeds, the client chooses a random identifier I and sends a signed PREPARE(I, q, q', ν) message to each server. What follows is essentially a variant of the classical two-phase commit protocol.

Upon receiving the PREPARE message, the server at each C_i verifies that q can be rewritten into q', and that it does not already have a pending query with identifier I. If either test fails, the server responds with a NAK immediately. Otherwise, C_i's server calculates its privacy cost $\varepsilon_i := \varepsilon_{r,i} + \sum_j \varepsilon_{p,ij}$ that it would incur by executing its part of q'. This cost consists of the base cost $\varepsilon_{r,i} := S(q, db_i)/\nu$, which depends on the query's sensitivity in C_i's local data, and an additional charge $\varepsilon_{p,ij}$ for each PSI-CA operation that C_i must participate in to execute q'. If C_i's privacy budget can cover ε_i, its server deducts ε_i from the budget, adds $(I, q', \nu, \varepsilon_i)$ to its pending table, and sends a signed response ACK(I, q, q', ν) back to the client. Otherwise, the server responds with a NAK. This might occur, for instance, if the sensitivity of q is too high or the requested noise level ν is too low.

If the client receives at least one NAK, it sends a signed ABORT(I) message to each server that has responded with an ACK, which causes the reserved parts of the privacy budget to be released. Otherwise the client combines the received ACK messages to form a certificate Γ, and it sends COMMIT(I, Γ) to the servers. The servers verify that all required ACKs are present; if so, they begin executing the query.

6.7 Query execution

Each query is executed in three stages. First, upon receiving the COMMIT message, the server at each C_i computes the parts of the query that require only data from its local database db_i. For some queries, this will yield part of the result directly (e.g., in $|\sigma_{x=0}(A) \bigcup \sigma_{x=1}(B)|$), but more typically the first stage will produce a number of sets on each server that will be used as inputs in the second stage.

The second stage consists of a number of BN-PSI-CA instances. Since all servers agree on the query q', each server can independently determine which BN-PSI-CA instances it should be involved in, and what role in the protocol it should play in each instance. Ties are broken deterministically, and the instances are numbered in order to distinguish different instances that involve the same set of servers. At the end of the second stage, each server has learned a number of noised results and/or noise terms, which are used as inputs to the third stage.

The third stage consists of an invocation of DCR, which de-noises the results from the second stage, combines them as required by q', and then re-noises the combined result using the protocol from Section 4.4. Recall that the re-noising requires an additional input from each server that must be chosen uniformly at random. At the end of the third stage, each server learns the result of the multi-party computation and forwards it back to the client, which displays it to the user.

7 Evaluation

In this section, we report results from an experimental evaluation of DJoin. Our goal is to show that 1) DJoin is powerful enough to support useful queries; and that 2) DJoin's communication and computation overheads are low enough to be practical.

7.1 Prototype implementation

We have built a prototype implementation of DJoin for our experiments. Our prototype uses mySQL to store each curator's data and to execute the purely local parts of each query, and it relies on FairplayMP [4] to execute the secure multi-party computation. We implemented the two-party BN-PSI-CA primitive from Section 4.2, based on the `thep` library [35] for the Paillier cryptosystem. Our implementation includes the optimizations from [14] that were already briefly described in Section 4.1, including the use of bucket hashing to replace the single high-degree polynomial P with a number of lower-degree polynomials. This reduces BN-PSI-CA's $O(|S_1| \cdot |S_2|)$ time complexity to $O(|S_1| + |S_2| \ln\ln |S_1|)$ and makes it highly parallelizable, with synchronization required only for the few elements that hash to the same bucket. Our prototype also supports multi-party BN-PSI-CA based on the protocol from Kissner and Song [17] and the UTD Paillier Threshold Encryption Toolbox [36], but we do not include multi-party results here due to lack of space.

We also built a query planner that implements the rewrite rules from Section 5.2, as well as a backend for FairplayMP that outputs code for DCR (Section 4.5). To our knowledge, DCR is the first implementation of the shared noise generation algorithm described in [10]. Altogether, our prototype consists of 3,560 lines of Java code for the runtime engine, 249 lines of code in FairplayMP's custom language for the DCR primitive, and 6,776 lines of C++ code for the query planner.

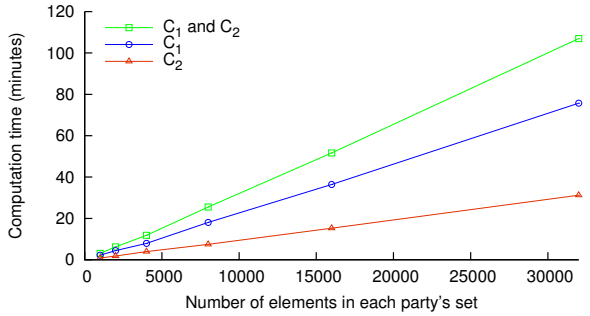


Figure 5: Computation time for PSI-CA. The time is approximately linear in the number of set elements.

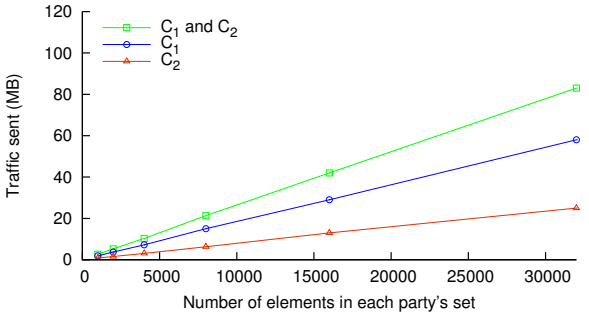


Figure 6: Network traffic sent by the two parties in a BN-PSI-CA run.

7.2 Experimental setup

For our experiments, we used five Dell PowerEdge R410 machines, each with a Xeon E5530 2.4 GHz CPU, 12 GB of memory, and four 250 GB SATA disks. The machines were connected by Gbit Ethernet. Following the recommendations in [5], we used 1,024-bit keys for the Paillier cryptosystem. We chose $\varepsilon_r = 0.0212$ to ensure that the noise for a query with sensitivity $s = 1$ is within ± 100 with probability 95%; we set $\varepsilon_p = 1/8 \cdot \varepsilon_r$, and we chose $\delta = 1/N = 6.67 \cdot 10^{-5}$.

Our experiments use synthetic data rather than 'real' confidential data because our cryptographic primitives operate on hashes of the data anyway, so the actual content has no influence on the overall performance. Therefore, we generated synthetic databases. Each database had $N = 15,000$ rows.

7.3 Microbenchmarks: BN-PSI-CA

First, we quantified the cost of our two main cryptographic primitives. To measure the cost of BN-PSI-CA, we generated two random sets with N elements each, and we ran two-party BN-PSI-CA on them, varying N between 1,000 and 32,000 elements. We measured the computation time on each party and the amount of traffic that was exchanged between the two parties.

Figure 5 shows the time taken by the servers at C_1 and C_2, respectively, to execute BN-PSI-CA using a single core. The time increases almost linearly with the size of the sets; recall from Section 7.1 that the optimizations we applied reduce the computational overhead to $O(|S_1| + |S_2| \ln \ln |S_1|)$. Note that the two servers cannot run in parallel; the total runtime is the sum of the two servers' runtimes. Most of the computation is performed by C_1: 49% of the total time was spent constructing the polynomials at C_1; 29% of the time was spent evaluating the polynomials at C_2; and the remaining 21% were spent decrypting the resulting evaluations at C_1.

Figure 6 shows the total amount of traffic sent by C_1 and C_2. The traffic is roughly proportional to the set sizes. For large sets, approximately 70% of the traffic consists of polynomials sent from C_1 to C_2, and the remaining 30% consists of evaluation results sent back to C_1 for decryption.

To quantify BN-PSI-CA's scalability in the number of cores, we performed a 15,000-element intersection with one, two, and four cores. (This was done on a different machine with a 2.67 GHz Intel X3450 CPU, since our E5530s have only two cores.) The additional cores resulted in speedups of 1.99 and 3.98, respectively. This is expected because BN-PSI-CA is trivially scalable: encryptions, polynomial construction, evaluations, and decryptions can all proceed in parallel on multiple cores, or even multiple machines. Thus, DJoin should be able to handle databases much larger than 32,000 elements, as long as the computation can be spread over a sufficient number of machines.

7.4 Microbenchmarks: DCR

Next, we quantified the cost of the DCR operator. Recall from Section 4.4 that DCR internally consists of two stages: first, the inputs (cardinalities and inverted noise terms) from the various servers are added together, and then a new noise term is drawn from a Laplace distribution and added to the result. To separate the two stages, we measured the time to execute DCR twice, with and without the second stage, and we varied the number of parties from two to four.

Figure 7 shows our results. The times grow superlinearly with the number of parties ([4] reports a quadratic dependency) but are all below 20 seconds. Although MPC is generally expensive, DJoin performs most of its work using a specialized primitive (BN-PSI-CA), so the functionality that remains for DCR to perform is fairly simple. Note that neither the size nor the number of sets affect DCR's runtime because each server inputs just a single number: the sum of all the cardinalities and noise terms it has computed.

	Query	#PSI-CA
Q1	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.y` $\lvert(\pi_x(A) \bigcap \pi_y(B))\rvert$	1
Q2	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.x AND (A.y!=B.y)` $\lvert(\pi_x(A) \bigcap \pi_x(B))\rvert - \lvert(\pi_{x,y}(A) \bigcap \pi_{x,y}(B))\rvert$	2
Q3	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.y AND (A.z="x" OR B.p="y")` $\lvert(\pi_x(A) \bigcap \pi_y(\sigma_{p=\text{"}y\text{"}}(B)))\rvert + \lvert(\pi_x(\sigma_{z=\text{"}x\text{"}}(A)) \bigcap \pi_y(\sigma_{p\neq\text{"}y\text{"}}(B)))\rvert$	2
Q4	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x=B.x OR A.y=B.y` $\lvert(\pi_x(A) \bigcap \pi_x(B))\rvert + \lvert(\pi_y(A) \bigcap \pi_y(B))\rvert - \lvert(\pi_{x,y}(A) \bigcap \pi_{x,y}(B))\rvert$	3
Q5	`SELECT NOISY COUNT(A.x) FROM A,B WHERE A.x LIKE "%xyz%" AND A.w=B.w AND (B.y+B.z>10) AND (A.y>B.y)` $\sum_{i=0..7} \lvert(\pi_{w,(y>>i+1)}(\sigma_{(x \text{ like '\%xyz\%'}) \wedge (y\&2^i=1)}(A)) \bigcap \pi_{w,(y>>i+1)}(\sigma_{((y+z)>10) \wedge (y\&2^i=0)}(B)))\rvert$	8

Table 2: Example queries and the corresponding query plans. The number of BN-PSI-CA operations, which is a rough measure for the complexity of the query, is shown on the right.

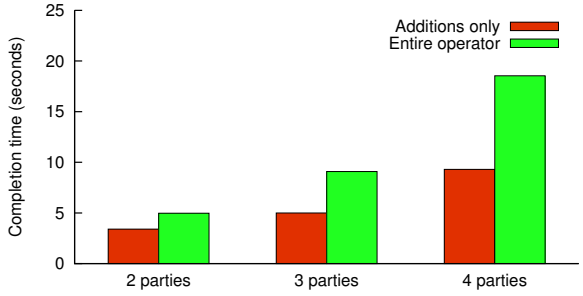


Figure 7: Computation time for DCR with and without the renoising step.

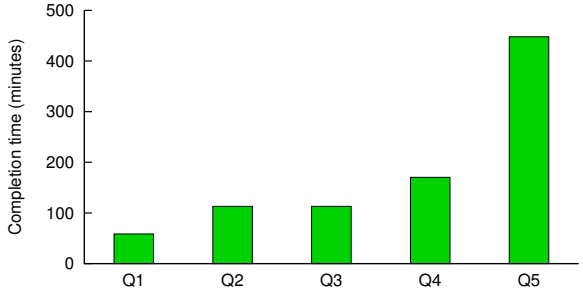


Figure 8: Total query execution time for each of the example queries from Table 2.

7.5 Example queries

To demonstrate that DJoin can execute nontrivial and potentially useful queries, we chose five example queries, which are shown in Table 2 along with the query plan they are rewritten into. Each query illustrates a different aspect of DJoin's capabilities:

- **Q1** is an example of a basic join between two databases, which is transformed into a PSI-CA using rule R8.
- **Q2** adds an inequality, which is rewritten as a difference between two intersections via rule R5.
- **Q3** contains a disjunction with two local predicates, which can be split using rule R3.
- **Q4** contains another disjunction, but with remote predicates; this is rewritten via rule R2.
- **Q5** contains an equality and a numeric comparison between columns in different databases, which can be split via rule R6, as well as several other predicates that can be evaluated locally.

For Q5, the `y` column in both databases contained numbers between 0 and 255. The table also shows the number of BN-PSI-CA operations in each query plan, which (in conjunction with the set sizes) is a rough measure of the effort it takes to evaluate it. The more complex a query is, the more BN-PSI-CAs it requires. Q1 is the least complex query because it translates straight into a BN-PSI-CA; Q5 is the most complex one because the inequality requires one intersection per bit.

7.6 Query execution cost

To quantify the end-to-end cost of DJoin, we ran each of our five example queries over a synthetic dataset of 15,000 rows per database, and we measured the completion time and the overall amount of network traffic that was sent.

Figures 8 and 9 show our results. The simplest query (Q1) took 58 minutes, and the most complex query (Q5) took 448 minutes, or slightly less than seven and a half hours; the traffic was between 42.7 MB and 340 MB. Both metrics should scale roughly linearly with the size of the sets and the number of set intersections in the

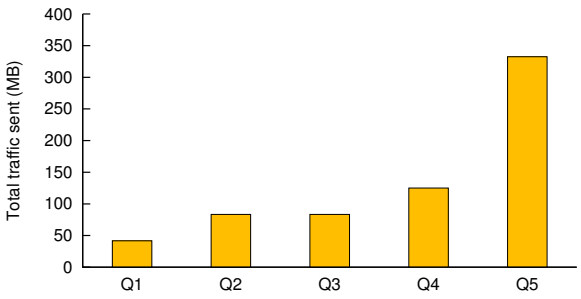


Figure 9: Total network traffic for each of the example queries from Table 2.

query, and a comparison with our microbenchmarks from Section 7.3 confirms this.

The completion times are much higher than the completion times one would expect from a traditional DBMS, but recall that DJoin is not meant for interactive use, but rather for occasional analysis tasks or research studies. For those purposes, an hour or two should be acceptable. Also, recall that the best previously known method for executing such queries is general MPC, which is impractical at this scale.

To illustrate how much DJoin improves performance over straightforward MPC, we implemented our simplest query (Q1) directly in FairplayMP. A version for two databases of just eight (!) rows had 9,700 gates and took 40 seconds to run; we were unable to test larger databases because this produced crashes in FairplayMP. The runtimes we observed increased quadratically with the number of rows, which suggests that this approach is not realistic for the database sizes we consider.

8 Conclusion

In this paper, we have introduced two new primitives, BN-PSI-CA and DCR, that can be used to answer queries over distributed databases with differential privacy guarantees, and we have presented a system called DJoin that can execute SQL-style queries using these two primitives. Unlike prior solutions, DJoin is not restricted to horizontally partitioned databases; it supports queries that join databases from different curators together. The key insight behind DJoin is that many distributed join queries can be rewritten in terms of operations on multisets. Not all SQL queries can be transformed in this way, but many can, including counting queries with conjunctions and disjunctions of equality tests, as well as certain inequalities.

DJoin is not fast enough for interactive use, but, to the best of our knowledge, the only known alternative for distributed differentially private join queries is secure multi-party computation, which is orders of magnitude slower. Also, most of the computational cost is due to BN-PSI-CA, which is trivially scalable and can thus benefit from additional cores.

Acknowledgments

We thank our shepherd, Nickolai Zeldovich, and the anonymous reviewers for their comments and suggestions. We also thank Marco Gaboardi, Benjamin Pierce, Aaron Roth, and Andre Scedrov for helpful comments on earlier drafts of this paper. This work was supported by NSF grants CNS-1065060 and CNS-1054229, ONR grants N00014-09-1-0770 and N00014-12-1-0757, and by a gift from Google.

References

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. SIGMOD*, May 2000.

[2] M. Barbaro and T. Zeller. A face is exposed for AOL searcher No. 4417749. *The New York Times*, Aug. 2006. `http://nytimes.com/2006/08/09/technology/09aol.html`.

[3] R. M. Bell and Y. Koren. Lessons from the Netflix prize challenge. *SIGKDD Explor. Newsl.*, 9(2):75–79, Dec. 2007.

[4] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *Proc. CCS*, Oct. 2008.

[5] J. Bethencourt, D. Song, and B. Waters. New constructions and practical applications for private stream searching (extended abstract). In *Proc. IEEE S&P*, May 2006.

[6] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proc. NSDI*, Apr. 2012.

[7] C. Dwork. Differential privacy. In *Proc. ICALP*, July 2006.

[8] C. Dwork. Differential privacy: A survey of results. In *Proc. TAMC*, Apr. 2008.

[9] C. Dwork. The differential privacy frontier (extended abstract). In *Proc. IACR TCC*, Mar. 2009.

[10] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proc. EUROCRYPT*, May 2006.

[11] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, Mar. 2006.

[12] A. Evfimievski, R. Srikant, R. Agrawal, and J. Gehrke. Privacy preserving mining of association rules. In *Proc. KDD*, July 2002.

[13] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. OSDI*, Oct. 2010.

[14] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Proc. EUROCRYPT*, May 2004.

[15] M. Götz and S. Nath. Privacy-aware personalization for mobile advertising. Technical Report MSR-TR-2011-92, Microsoft Research, Aug. 2011.

[16] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. USENIX Security*, Aug. 2011.

[17] L. Kissner and D. Song. Privacy-preserving set operations. In *Proc. CRYPTO*, Aug. 2005.

[18] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. VLDB*, Aug. 1986.

[19] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, Dec. 2004.

[20] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Proc. ICDE*, Apr. 2007.

[21] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: privacy beyond k-anonymity. In *Proc. ICDE*, Apr. 2006.

[22] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, Oct. 2010.

[23] F. McSherry. Privacy integrated queries. In *Proc. SIGMOD*, June 2009.

[24] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proc. FOCS*, Oct. 2007.

[25] I. Mironov, O. Pandey, O. Reingold, and S. P. Vadhan. Computational differential privacy. In *Proc. CRYPTO*, Aug. 2009.

[26] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. IEEE S&P*, May 2008.

[27] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *Proc. CCS*, Nov. 2001.

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT*, May 1999.

[29] B. Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explor. Newsl.*, 4(2):12–19, Dec. 2002.

[30] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. SOSP*, Oct. 2011.

[31] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proc. SIGMOD*, June 2010.

[32] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proc. NSDI*, Apr. 2010.

[33] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *Proc. NDSS*, Feb. 2011.

[34] L. Sweeney. k-anonymity: A model for protecting privacy. *Int. J. Uncert. Fuzzin. Knowl.-Based Syst.*, 10(5):557–570, Oct. 2002.

[35] The Homomorphic Encryption Project. `http://code.google.com/p/thep/`.

[36] UTD Paillier threshold encryption toolbox. Available from `http://utdallas.edu/~mxk093120/paillier/`.

[37] J. Vaidya and C. Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 13(4):593–622, Nov. 2005.

[38] A. Yao. Protocols for secure computations (extended abstract). In *Proc. FOCS*, Nov. 1982.

[39] N. Zhang, M. Li, and W. Lou. Distributed data mining with differential privacy. In *Proc. ICC*, June 2011.

Appendix

Choosing ε: The choice of ε is essentially a social question and beyond the scope of this paper; however, we briefly sketch one possible approach. Suppose Alice is considering whether or not to allow her data to be included in a database that can later be queried via DJoin, and suppose she is concerned that an adversary might then be able to learn a certain fact about her – for instance, that she has cancer. From Alice's perspective, the worst-case scenario is that the adversary 1) already knows *all* the data in the database (!), except Alice's, that he 2) manages to get access to DJoin, and that he 3) burns the entire privacy budget on a single query q – say, "how many people in the database have cancer?".

Consider the situation from the adversary's perspective. Since we have (very conservatively) assumed that the adversary already knows *all* the data except Alice's, he can construct two "possible worlds": one database b_1 where Alice has cancer, and another database b_2 where she does not. He does not know whether the real database is b_1 or b_2, but he can compute the conditional probability $P_i := P(q(db) = r \,|\, db = b_i)$ that q will return r if the real database is b_i. Thus, once he observes the actual result, he can use Bayes' formula to update his belief that Alice has cancer.

Now recall that, according to the definition of differential privacy from Section 3.2, P_1/P_2 is bounded by e^{ε}. Thus, ε controls how much more confident the adversary can become about Alice's cancer status. If Alice is comfortable with $P_1/P_2 \leq 2$, she can accept values of ε up to $\ln 2 \approx 0.69$. If a benign querier wants to ask queries with sensitivity $s = 1$ and $\sum_j \varepsilon_{p,j} = \varepsilon_r$ on a database with $100,000$ entries and have $c = 95\%$ confidence that the error due to noise is less than $E = 1,000$ (1%), we have

$$N_{max} = \frac{\varepsilon_{max} \cdot \lambda_{max}}{2 \cdot s} = \frac{\varepsilon_{max} \cdot E}{-2 \cdot s \cdot \ln(1-c)} \approx 115$$

In other words, a privacy budget of $\varepsilon_{max} = 0.69$ would be enough to answer up to 115 queries of this type.

Multiset encodings: In some instances, it is necessary to encode the input sets before they can be processed as intersections. For instance, if the underlying PSI-CA primitive supports sets but not multisets, we can encode an element e that appears n times as $\{e||1,\ldots,e||n\}$, with each element included only once [17]. Another example are joins with multiplicities greater than one. Suppose two curators want to evaluate $|\sigma_x(A) \bowtie \sigma_y(B)|$, and $A.x$ and $B.y$ contain n_A and n_B copies of some element e, respectively. Then A's curator can add, $\forall 1 \leq k \leq m(B.y)$, $k \cdot n_A$ encoded elements $e||n_A||k$, and B's curator can add, $\forall 1 \leq k \leq m(A.x)$, $k \cdot n_B$ elements $e||k||n_B$. The intersection then consists of $n_A \cdot n_B$ encoded elements $e||n_A||n_B$.