Independent Project on Secure Yannakakis

XINYI ZHU, Hong Kong University of Science and Technology

1 BACKGROUND

In this section, the background of two fundamental technologies, multi-party computation and join aggregate query, utilized in paper Secure Yannakakis[2] are illustrated with some explanations of other knowledge.

1.1 Multi-party Computation (MPC)

Multi-party Computation (MPC) also known as *secure multi-party computation (SMC)* is widely used nowadays in many areas like database, machine learning, internet of things and so on. Proposed by Andrew Yao, SMC works by using complex encryption to distribute computation between multiple parties. The model can be described in Fig. 1: a given number of participants,$p_1, p_2, ..., p_N$, each have private data, respectively $d_1, d_2, ..., d_N$. Participants want to jointly compute a public value on these private data: $F(d_1, d_2, ..., d_N)$ while keeping their own inputs secret.

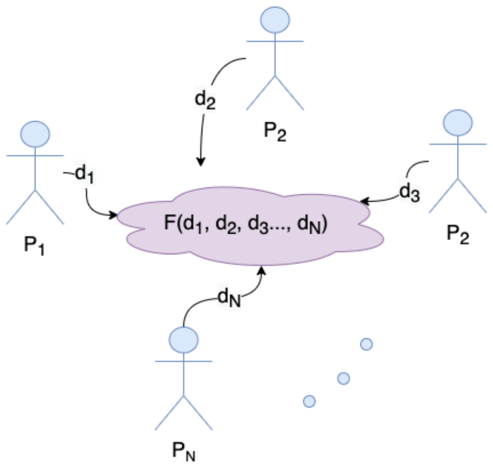
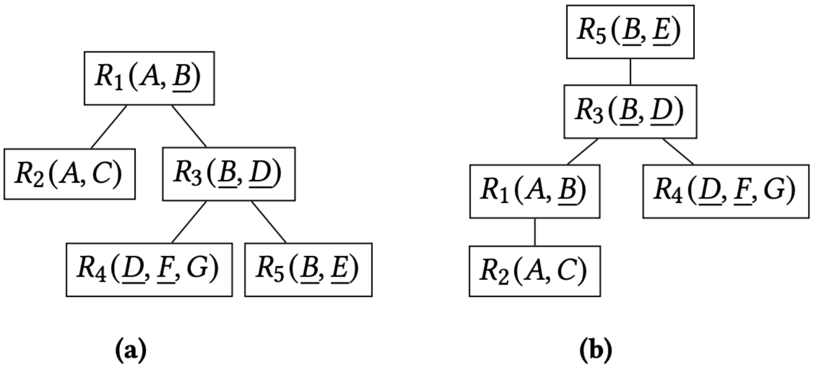


Fig. 1. The SMC Model.

Specifically, in the *2-party computation (2PC)* model, the most popular approach is Yao's garbled circuit, which is a generic protocol that can be used to

Author's address: Xinyi ZHU, Hong Kong University of Science and Technology .



$R_1(A, \underline{B})$
$R_2(A, C)$
$R_3(\underline{B}, \underline{D})$
$R_4(\underline{D}, \underline{F}, G)$
$R_5(\underline{B}, \underline{E})$
(a)
$R_5(\underline{B}, \underline{E})$
$R_3(\underline{B}, \underline{D})$
$R_1(A, \underline{B})$
$R_4(\underline{D}, \underline{F}, G)$
$R_2(A, C)$
(b)

Fig. 2. An acyclic join with different join trees in paper Secure Yannakakis.

evaluate any function by expressing it as a Boolean circuit. There is also a method called *SMCQL* which is the first secure query processing engine that supports joins in the 2PC model. For a join over k relations, generalized Yannakakis (a non-private model algorithm) requires computation cost of only $O(IN + OUT)$, but the complexity of *SMCQL* is $\Omega(IN^k)$.

Secure Yannakakis also uses the 2-party setting (2PC model). Assuming that the global schema of the database is known publicly, but each relation is held either by Alice or by Bob as her/his private data. The aim is to design secure protocols for evaluating any *free-connex join-aggregate* query with the complexity of $O((IN + OUT) \cdot \log(IN + OUT))$

1.2 Join Aggregate Query

Hypergraph and Joins. A natural join takes two relations which have some common attributes as input and generates all tuples with same values on the common attributes[1]. It can be modeled as a hypergraph, where the vertices corresponds to attributes and hyperedge corresponds to a relation.

Join Tree. If the hypergraph of a join can be expressed in the form of a join tree, then it is said to be *acyclic.* Take Fig. 2 as an example, trees in (a) and (b) can express the same acyclic join .

Join Aggregate Query. Generally, a join aggregate query is a specific query that has select, join and aggregate operations. However, there are some restrictions of join aggregate query. Take TPC-H Query 3 as an example.

```
select c_custkey, c_name, c_nationkey,
    sum(l_extendedprice*(1-l_discount))revenue
from customer, orders, lineitem
where c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= date '1993-08-01'
    and o_orderdate < date '1993-11-01'
    and l_returnflag = 'R'
group by c_custkey, c_name, c_nationkey;
```

The aggregate operation such as *sum* in the Query 3, is applied on one specific relation's different attributes. And the filter *where* operation is also applied on a specific relation's attributes, and these filter are connected by *AND* operations.

Free-Connex Join Aggregate Query. According to the definition, a acyclic join has a join tree. A *free-connex* join can be easily explained that all outputs attributes should be placed above the non-outputs attributes in the join tree.

2 SECURE YANNAKAKIS ALGORITHM

2.1 Generalized Yannakakis

The generalized Yannakakis computes the *free-connex* join aggregate query in $O(IN + OUT)$ time. It consists of three phases, *Reduce*, *Semijoin* and *Full join.*

(1) *Reduce.* This phase aims to remove all non-output attributes. It is implemented in a bottom-up order through joins and aggregations.

(2) *Semijoin.* This phases consists of two parts, a bottom-up pass and a top-down pass to remove dangling tuples.

(3) *Full join.* This phase is implemented in a bottom-up order to compute the join and their annotations.

Example: take Fig. 2 (b) as an example. Its join aggregate query can be expressed as:

```
select B, D, E, F
from R1 join R2 join R3 join R4 join R5
```

Firstly, *Reduce* phase works in a bottom-up manner. This phase can be expressed as followed:

$$\begin{aligned} R_2(A) &\leftarrow \pi_A(R_2), \\ R_1(A,B) &\leftarrow R_1 \bowtie R_2, \\ R_2 \ &now\ has\ been\ removed. \end{aligned} \tag{1}$$

$$\begin{aligned} R_1(B) &\leftarrow \pi_B(R_1), \\ R_3(B,D) &\leftarrow R_3 \bowtie R_1, \\ R_1 \ &now\ has\ been\ removed. \end{aligned} \tag{2}$$

$$R_4(D,F) \leftarrow \pi_{D,F}(R_4) \tag{3}$$

R_4, R_5, R_3 now remain.

Secondly, *Semijoin* phase first works in a bottom-up manner then a top-down manner.

$$\begin{aligned} R_3 &\leftarrow R_3 \ltimes^{\otimes} R_4, \\ R_5 &\leftarrow R_5 \ltimes^{\otimes} R_3, \end{aligned} \tag{4}$$

$$\begin{aligned} R_3 &\leftarrow R_3 \ltimes^{\otimes} R_5, \\ R_4 &\leftarrow R_4 \ltimes^{\otimes} R_3, \end{aligned} \tag{5}$$

Now, all dangling tuples have been removed.

Finally, *Full join* computes $R_4 \bowtie R_3 \bowtie R_5$ as the result.

2.2 Secure Yannakakis

To ensure security of the *free-connex* join aggregate query based on *SMC* model in Sec. 1, Secure Yannakakis makes improvement on each relational operators used in Generalized Yannakakis. Then the algorithm just needs to assemble them together.

(1) *Oblivious Projection-Aggregation.* The first one is $\pi_F^{\oplus}(R)$ and the second one is $\pi_F^{1}(R)$. For $\pi_F^{\oplus}(R)$, Alice locally sorts R by F, so that tuples with the same value on F are consecutive. To make the algorithm oblivious, garbled circuit is used then. For $\pi_F^{1}(R)$, Alice also sorts R by F and permute the shares accordingly. Then Alice uses merge gates and the garbled circuit.

(2) *Oblivious Semijoin.* There are also two types of semijoins. The first type is actually an annotated join $R = R_F \bowtie^{\otimes} R_{F'}$. And the second type is an annotated semijoin $R = R_F \ltimes^{\otimes} R_{F'}$. The complexity of quadratic time can be decreased by using PSI (private set intersection) to compute the indicators, using OEP (oblivious extended permutation) to get the indicators and using PSI with (shared) protocol to deal with the annotations. The second type is computing $R = R_F \ltimes^{\otimes} R_{F'}$. It first computes $\pi^1_{F \cap F'}(R'_F)$ by oblivious projection-aggregation and then computes $R = R_F \bowtie^{\otimes} \pi^1_{F \cap F'}(R'_F)$ by the first type oblivious semijoin.

(3) *Oblivious Join.* As Alice will always obtain the final results. And the outputs of oblivious semijoin only contain non-dangling tuples. Thus, Alice can compute the result using general join operation.

Finally, Secure Yannakakis algorithm can be assembled and conducted in the following three phases: (a) *Reduce.* It also works in a bottom-up manner over the join tree while computing $R_{F_p} \leftarrow R_{F_p} \bowtie^{\otimes} \pi^{\oplus}_{F'}(R_F)$, with constraint $F^{'} \subseteq F_p$. This phase can be further splitted into two steps: computing $\pi^{\oplus}_{F'}(R_F)$ (by oblivious projection-aggregation) then the oblivious semijoin; (b) *Semijoin.* Different from Generalized Yannakakis, Secure Yannakakis marks dangling tuples as dummy (setting annotations to 0 which can be computed by oblivious semijoin); (c) *Full Join.* As only output attributes remain and dangling tuples are all zero-annotated, Secure Yannakakis invokes oblivious full join and reveals the outputs directly to Alice.

3 IMPLEMENTATIONS

This section, a new query is defined and implementations are illustrated on this query via Secure Yannakakis algorithm.

3.1 Query Definition

It is required to fill an attribute in the "?" place to make the query a *free-connnex* one. The given query is:

```
select c_name, c_custkey, ?, l_returnflag, sum(ps_supplycost*l_quantity)
from PART,SUPPLIER,LINEITEM,PARTSUPP,ORDERS,CUSTOMER
where s_suppkey = l_suppkey and ps_suppkey = l_suppkey
    and c_custkey=o_custkey and ps_partkey = l_partkey
    and p_partkey = l_partkey and o_orderkey = l_orderkey
    and p_name like '%green%' and s_nationkey = 8
group by c_name, c_custkey, o_orderkey, l_returnflag;
```

I completed the query using the attribute *o_orderkey* and drew the *free-connnex* tree as followed:

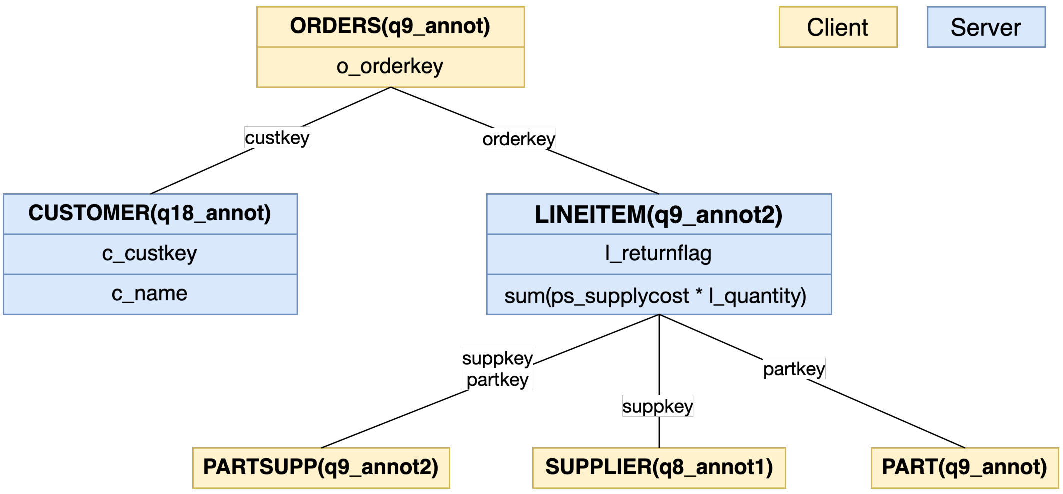


Fig. 3. My Free-connex Query Design.

3.2 Dataset Description

There are five different size of datasets, which are 1MB, 3MB, 10MB, 33MB and 100MB. Each size of dataset contains six tables, which are customer table, lineitem table, orders table, part table, partsupp table and supplier table. All TPC-H queries used in the paper are implemented on some or all these tables.

partkey	suppkey	availqty	supplycost	comment	q9_annot1	q9_annot2
1	2	3325	77.16	, even...	1	77
1	4	8076	99.35	ven ideas...	1	99
1	6	3956	33.71	after the...	1	34
1	8	4069	35.78	al,...	1	36
2	3	8895	37.85	nic accounts...	1	38

Table 1. First Five Rows in Partsupp Table.

To be specific, each table contains its attributes and annotations for different queries. Take partsupp table's first five rows in Table. 1 as an example. Among the TPC-H queries, only Q9 utilizes information in the partsupp table. Thus, for each table, the corresponding annotations are generated only when a query is needed. Also, there are two types of annotations, which are Boolean and numerical respectively.

Algorithm 1 My Query Implementations via Secure Yannakakis

```
(1) Give definition on Relation consists of each table and its annotations;
(2) Load data into each table's Relation;
(3) Do Aggregate() on "False" annotated Relation;
(4) Do Semijoin() on LIINEITEM with PARTSUPP, PART and SUPPLIER on their edges' attributes in the join tree;
(5) Copy LINEITEM into LINETITEM_COPY and do Aggregate() on LINETITEM_COPY;
(6) Do Semijoin() on ORDERS with LINETITEM_COPY on their edges' attribute in the join tree and do Aggregate() on ORDERS;
(7) Do Semijoin() on ORDERS with CUSTOMER on their edges' attribute in the join tree;
(8) Copy ORDERS and do Project(o_custkey), AnnotOrAgg() on it;
(9) Copy ORDERS and do Project(o_orderkey), AnnotOrAgg() on it;
(10) Do Semijoin() on CUSTOMER with ORDERS_COPY by step (8);
(11) Do Semijoin() on LINEITEM with ORDERS_COPY by step (9);
(12) Do RevealTuples() on ORDERS;
(13) Do AnnotOrAgg(), RemoveZeroAnnotatedTuples(), RevealTuples() on CUSTOMER;
(14) Do Join() of ORDERS and CUSTOMER;
(15) Do Project() on output attributes of CUSTOMER and do AnnotOrAgg(), RemoveZeroAnnotatedTuples(), RevealTuples() on LINEITEM;
(16) Do Join() of ORDERS and LINEITEM;
(17) Do RevealAnnotToOwner() on ORDERS;
```

3.3 Query Implementations

In my experiment, the filtering conditions (corresponding annotations) in my given query are all generated before. Thus, I find all corresponding annotations in previous queries for each table which are also shown in Fig. 3.

Dataset size(MB)	Running time(s)	Communication cost(MB)
1	7.689	223.537
3	15.153	715.632
10	38.383	2562.44

Table 2. Evaluation result of different dataset size.

According to Secure Yannakakis algorithm and the Github repository, this query can be implemented as **Algorithm 1**.

4 EVALUATION

4.1 Experiment Setup

To illustrate the performance of this new query in Secure Yannakakis protocol, I evaluate the running time and communication cost for the new query on above datasets. I deploy the whole Secure Yannakakis protocol on a Azure cloud virtual machine equipped with two Intel vCPU, 8 GiB RAM, and a 30 GiB SSD(with max throughput 25Mbps). The software runs on Debian 10.0.

I have forked the repository and added my code in New Query Extension Repository.

4.2 Result Analysis

The internal process of running dataset of size 30MB and 100MB requires more than 10GiB system memory, which exceeds the quota of my VM memory.Due to the limitations of the experimental environment, here I only runned and generated the results of datasets with size 1MB, 3MB and 10MB.

Evaluation on my experiments also support that Secure Yannakakis is both practical and efficient. For the first aspect, the outputs of the query are correct using Secure Yannakakis. For the second aspect, the running time which is illustrated in Table. 2 can prove the efficiency of this algorithm.

5 FUTURE DIRECTIONS

Now Secure Yannakakis only supports two-party computation. Thus, future work can be developed on multi-party (parties > 2) ones. Furthermore, it can also be developed to support more types of queries. For example, now only acylic join is supported. Hence, some explorations on cyclic join problem may also be practical and interesting. Last but not least, now all the implementations are done manually. It will be great to generate Secure Yannakakis codes automatically by a system (maybe using some deep learning strategies to aid).

REFERENCES

[1] M. R. Joglekar, R. Puttagunta, and C. Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91--106, 2016.

[2] Y. Wang and K. Yi. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1969--1981, 2021.