Secure Multi-Party Traversal Queries over Federated Graph Databases

Nouf Aljuaid[1,2], Alexei Lisitsa[2] and Sven Schewe[2]
[1]*Department of Information Technology, Taif University, Saudi Arabia*
[2]*Department of Computer Science, University of Liverpool, Liverpool, U.K.*

Keywords: Graph Databases, SMPC, Federated Databases, Secure Data Processing, Multi-Party Querying.

Abstract: We introduce and compare two protocols for the execution of secure multi-party traversal queries over federated graph databases. The first, client based protocol uses client-to-client communication; it minimises the exposure of data on a *need-to-know* basis. The second protocol uses a semi-trusted server; it combines the use of private channels for communication between the server and clients with the use of encrypted hashing to prevent the exposure of data to the server. We have conducted experiments to compare the efficiency of the two protocols. The results demonstrated that the execution times for the server-based protocol are around half an order of magnitude higher. This does, however, seem to be down to the security layer provided by encrypted hashing: when switching it off, the running time becomes comparable.

1 INTRODUCTION

With the rapid growth of data volumes collected in databases from various sources, traditional databases struggle to scale to these vast amounts of data. This limitation makes querying a complex and inefficient task (Nayak et al., 2015). In response to these challenges, graph databases were developed to overcome the constraints of relational databases (Salehnia, 2017). They have found application in numerous contexts, including among others

social media platforms (Ciucanu and Lafourcade, 2020a).

A key strength of graph databases lies in their ability to efficiently represent and navigate relationships using traversal queries. This concept is prominently employed in graph databases such as Neo4j (Guia et al., 2017). A typical traversal query follows a pattern where nodes are connected by relationships, creating a path through the graph structure:

$$(node_1) - [: rel_1] - (node_2) - [rel_2] - (node_3) - \ldots - (node_n) - [rel_n] - (node_n)$$

Here, $node_1, \ldots, node_n$ represent the nodes in the graph, and $rel_1, \ldots, rel_n$ are the relationships connecting them. This pattern represents a traversal through n nodes with $n-1$ relationships.

In this work, we study the concept of traversal queries within the realm of federated databases and privacy-preserving processing. When dealing with multiple graph databases owned by different parties, each with its own set of sensitive data, it becomes important to safeguard privacy during collaborative analyses. We aim to develop methodologies that allow the execution of traversal queries across multiple databases without compromising the confidentiality of individual databases.

In doing so, we extend the traditional concept of secure multi-party computation (SMPC) (Evans et al., 2018) to secure multi-party querying. Until now, in the context of data processing, SMPC has primarily been utilized to safeguard relational databases, as seen in systems like Conclave, Senate, and SMCQL (Volgushev et al., 2019; Poddar et al., 2020; Bater et al., 2016). More recently, applications of SMPC for other types of databases with different data models have been considered. This includes systems targeting graph databases like GOOSE (Ciucanu and Lafourcade, 2020b), which employs SMPC at the backend while queries remain single-party only, and SMPQ (Al-Juaid et al., 2022), which implements multi-party queries but does not address traversal queries.

The contribution of our work is as follows: We have developed a system for preserving the privacy of traversal multi-party queries within a federated graph database, particularly in scenarios where the distribution of data relevant for the query is known (clear-cut principle). The system provides the user with two different security protocols:

The first protocol, client-based, is designed to per-

Aljuaid, N., Lisitsa, A. and Schewe, S.
Secure Multi-Party Traversal Queries over Federated Graph Databases.
DOI: 10.5220/0012768000003767

In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 716-721
ISBN: 978-989-758-709-2; ISSN: 2184-7711

form the traversal query with the minimum required exposure of data or such client-to-client communication on a need-to-know basis. The second protocol, a novel approach implemented on the server side, involves hashing the data and masking it with a Diffie-Hellman (DH) key exchanged among all clients. This protects the data against access by the server. The choice between the two protocols is determined based on the preference of clients to use and the amount of information they are willing to share.

In this work we use Neo4j, well-known implementation for graph databases (Guia et al., 2017). Neo4j employs a graph data model composed of nodes and relationships (Miller, 2013). Cypher, the query language used by Neo4j, manages data within these graph databases (Francis et al., 2018). Our system leverages Neo4j Fabric (Gu et al., pear), a federated database solution that executes Cypher queries targeting multiple Neo4j graph databases simultaneously.

The remainder of this paper is organised as follows: Section 2 presents our approach. Next, Section 3 presents the evaluation of the system and the experimental setup, followed by the results of our experiments. Following this, a review of related literature (Section 4). Finally, the paper concludes in Section 5, and we offer directions for future work.

2 SYSTEM OVERVIEW

In this section, we present an overview of how the joint traversal query is distributed and how the system works.

2.1 Query Agreement and Distribution

The system facilitates joint traversal querying among two or more parties. After determining the number of parties involved in performing the traversal query, they should agree on a computation ID. In our current system, this computation ID (ComID) can be considered an agreement as in the JIFF system (Albab et al., 2019).

Following the agreement among the involved parties on the query, a joint traversal Cypher query is submitted.[1] To efficiently solve this collaborative query across multiple parties while safeguarding privacy, it is necessary to transform the query into (sequences of) sub-queries for each party's database.

[1]In our system, the query itself should be known for each party involved to agree on performing it. Query privacy important in other contexts is not relevant here.

The crucial information for dividing the original query into relevant sub-queries pertains to the distribution of data across different parties' databases. Depending on the context, various amounts of such information can be available. Here, we make a strong assumption and consider only traversal queries with a *clear-cut* property. This assumption implies that we assume the distribution of data relevant to the traversal query across the parties is known. More precisely, for a traversal query Q of length k, involving n parties, and partition $k = m_1 + \ldots + m_n$, the clear-cut property $cc(n, [m_1, ..., m_n])$ holds if the correct result of the query may be obtained by: 1) splitting the original query into successive sub-queries of lengths $m_1, \ldots, m_n$; 2) receiving and combining the results of these queries on parties $1, \ldots, n$ databases, respectively. For example, consider the submitted joint traversal query:
$\{(node_1) - [: rel_1] - (node_2) - [: rel_2] - (node_3) - [: rel_3] - (node_4) - [: rel_4] - (node_5)\}$

Now, let's assume the clear-cut property `cc(3, [2, 1, 1])`. This means we want to segment the original query into three parts, with the lengths of relationships assigned to each part as `[2, 1, 1]`.

This will produce three distinct sub-queries corresponding to each party's database:
Query 1:
$\{(node_1) - [: rel_1] - (node_2) - [: rel_2] - (node_3)\}$
Query 2:
$\{(node_3) - [: rel_3] - (node_4)\}$
Query 3:
$\{(node_4) - [: rel_4] - (node_5)\}$

2.2 Query Workflow

After generating the sub-queries corresponding to each party's database, the query can be executed on either the client-based or the server-based. In the following sections, we will explain the query workflow for each base. Before delving into that, we will explain all the necessary operations that could be used on both sides to execute the traversal query.

2.2.1 Preliminaries

In both protocols we need to deal with the results of the sub-queries, which would be in the form:

$\{(node_1, node_2), ... \}$, where $node_1, node_2, \ldots$ are strings. In order to deal with that we introduce some notations that will be used later in both protocols as needed.

- $\nu(s) := \{y \mid (\bar{x}, y) \in s\}$
- $\mu(s) := \{x \mid (x, \bar{y}) \in s\}$

- $\tau(s_1, s_2) := \nu(s_1) \cap \mu(s_2)$
- $F_1(w, s) := \{(\bar{x}, \bar{y}) \mid \bar{x} \in w, (\bar{x}, \bar{y}) \in s\}$
- $F_2(s, w) := \{(\bar{x}, \bar{y}) \mid (\bar{x}, \bar{y}) \in s, \bar{y} \in w\}$
- $C(s_1, s_2) := \{(\bar{x}, y, \bar{z}) \mid (\bar{x}, y) \in s_1, (y, \bar{z}) \in s_2)\}$

The first two are ν and μ, which can be used to project either the last node or the first node in the query result. The τ function is used to find the intersection between the two datasets (the result of the sub-queries).The two functions F_1 and F_2 are used to filter results based on the result of τ; either the first or last node is the connection node between the two queries.

The last one, C, is used to combine the final result of the traversal path among all involved parties.

2.2.2 Protocol 1: Client-Based

The client-based protocol primarily relies on two fundamental operations: projection and intersection. In this process, each party projects its final node from the result of its respective sub-query and intersects it with the subsequent party in the traversal query. Thus, the information is exchanged on a a *need-to-know* basis with the aim to reduce the exposure of the private data. Algorithm 1 illustrates the phases involved in client-based query execution.

The following example illustrates the client-based protocol for the case of three parties, using the notation detailed in subsection 2.2.1:

Party 1: $R_1 := \mathbf{r}(Q_1)$; **Send**$(P_2, \nu(R_1))$
Party 2: $R_2 := \mathbf{r}(Q_2)$; $t_1 := \tau(\nu(R_1)), \mu(R_2)))$;
$R_2 := F_1(t_1, R_2)$; **Send**$(P_3, \nu(R_2)))$
Party 3: $R_3 := \mathbf{r}(Q_3)$; $t_2 := \tau(\nu(R_2)), \mu(R_3)))$;
$R_3 := F_1(t_2, R_3)$; **Send**$(P_2, R_3))$
Party 2: $C_1 := C(R_2, R_3)$; **Send**$(P_1, C_1))$
Party 1: $C_2 := C(R_1, C_1)$; **Send**$([P_2, P_3], C_2))$

This process illustrates a step-by-step interaction among the parties involved, highlighting the sequence of actions and communication exchanges necessary to execute the traversal query within the framework of the client-based protocol.

2.2.3 Protocol 2: Server-Based

In the server-based protocol, our goal is to minimize the exposure of data between parties by employing two key operations: projection and intersection on the server. After each party obtains the result of a sub-query, they send it to the server in a protected form. This protection involves leveraging Diffie-Hellman key exchange (DH) during their connection to the system and hashing the sub-query result using HMAC and the DH key. The ability of the server to perform intersection operation on protected values relies crucially on HMAC being a deterministic function. Algorithm 2 presents the phases involved in server-based query execution. The following example illustrates the server-based protocol for the case of three parties.

Party 1: $R_1 := \mathbf{r}(Q_1)$; **Send**$(S, Hmac(R_1))$
Party 2: $R_2 := \mathbf{r}(Q_2)$; **Send**$(S, Hmac(R_2))$
Party 3: $R_3 := \mathbf{r}(Q_3)$; **Send**$(S, Hmac(R_3))$
Server: $C_1 := C(R_1, R_2)$; $C_2 := C(C_1, R_3)$;
Send$([P_1, P_2, P_3], C_2))$
Party 1: $F_1 := \mathbf{Hmac}^{-1}(C, DH)$; **Send**$(S, (F_1))$
Party 2: $F_2 := \mathbf{Hmac}^{-1}(C, DH)$; **Send**$(S, (F_2))$
Party 3: $F_3 := \mathbf{Hmac}^{-1}(C, DH)$; **Send**$(S, (F_3))$
Server: $C_3 := C(F_1, F_2)$; $C_4 := C(C_3, F_3)$;
Send$([P_1, P_2, P_3], C_4))$

This process illustrates a step-by-step interaction among the parties involved, highlighting the sequence of actions and communication exchanges necessary to execute the traversal query within the framework of the server-based protocol.

Algorithm 1: Client-based Query Execution.

Require: Q: Joint Traversal query
Require: P: Set of parties P_1, $P_2, \ldots, P_n$ for sharing computations.
Require: *ComID*: Shared computation ID as proof of agreement.
Require: k: The length of relationships
Connect to the System
1. Decide the number of parties (P_i) involved in the computation.
2. Use shared *ComID* to agree on the computation.
Preprocessing Phase
1. One of P_i submits Q to the system.
2. Parse Q into Q_i based on m_i for each P_i.
3. Execute Q_i on each P_i's Neo4j database.
Computation Phase
1: **for** $i = 1$ **to** $n - 1$ **do**
1. Each P_i **sends** $\nu(R(Q_i))$ to P_{i+1}
2. Each P_{i+1} computes
$\tau(\nu(R(Q_i)), \mu(R(Q_{i+1})))$
3. Each P_{i+1} filters his $R(Q_{i+1})$ by:
$F_1(\tau(R(Q_i), R(Q_{i+1})), R(Q_{i+1}))$
2: **end for**
Reconstruction Phase
3: **for** $i = n$ **to** 1 **do**
1. Each P_i **sends** the updates $R(Q_{i+1})$ to P_{i-1}.
2. Each P_{i-1} computes $C(R(Q_i), R(Q_{i+1}))$
4: **end for**
3. Once P_1 computes the final C, distribute the final result of Q to all parties.

Algorithm 2: Server Based Query Execution.

Require: Q: Joint Traversal query.
Require: P: Set of parties P_1, $P_2, \ldots, P_n$ for sharing computations.
Require: *ComID*: Shared computation ID as proof of agreement.
Require: k: The length of relationships.
Require: *Dh*: Key generated and exchanged using the Diffie-Hellman algorithm.
Require: *Hmac*: Function to hash result using *Dh*

Connect to the System
1. Decide the number of parties (P_i) involved in the computation.
2. Use shared *ComID* to agree on the computation.
3. After (P_i) connects to the system, generate *Dh* and exchange it between P_n.

Client Side
1. One of P_i submits Q to the system.
2. Parse Q into Q_i based on m_i for each P_i.
3. Execute Q_i on each P_i's Neo4j database.
4. $R(Q_i)$ for each P_i are hashed using $Hmac(R(Q_i), Dh)$ and store in table.
5. Each P_i **sends** the hashed R(Q_i) to the server.

Server Side
1: **for** $i = 1$ **to** $n-1$ **do**
1. Computes $C(R(Q_i), R(Q_{i+1}))$
2: **end for**
7. **Send** the final result of C to each P_i.

Client Side
1. Each P_i recover its parts of C using maps table of $\text{Hmac}^{-1}(C, DH)$
2. Each P_i **sends** the decrypted parts of C to the server.

Server Side
3: **for** $i = 1$ **to** n **do**
1. Computes $C(R(Q_i), R(Q_{i+1}))$
4: **end for**
2. **Send** the final result of C to each P_i.

3 EXPERIMENTATION AND EVALUATION

We evaluated and implemented our proposed system using JavaScript, drawing inspiration from the concept of constructing the system with the JIFF library (Albab et al., 2019). It is designed to be highly flexible, prioritizing usability, and it can run in both browser and Node.js environments. Our model has been developed to facilitate joint traversal queries across multiple graph databases owned by various parties. We investigate its efficiency to answer the following questions:

- **Q1**: How effective is our system in ensuring secure joint traversal multiparty queries, and what is its efficiency in terms of performance?
- **Q2**: Does the data size impact the system's efficiency?
- **Q3**: How does our model performance compare to executing traversal query on a Single Neo4j database?

3.1 Experimental Setup

The experiments were carried out on a desktop PC running Windows 11 with an Intel Core i7 processor clocked at 1.5 GHz and 16.00 GB of RAM, utilising a local database for each party. Execution times, representing the duration to obtain query results, were measured using the *performance.now()* function in JavaScript.

3.2 Dataset and Joint Traversal Query

For our experiment, we utilized the 'POLE' dataset, a large-scale dataset of open crime data for Manchester, UK, spanning August 2017 (Hunger, 2020). This dataset comprises 61,521 nodes interconnected by 105,840 relationships. Our experimental setup involved executing traversal queries using different numbers of parties and applying the clear-cut algorithm to segment the queries. We aimed to evaluate the security and performance of two-based protocols across various configurations. The traversal query employed is as follows:

```
MATCH (o:Officer {badge_no: '26-5234182'})
<-[:INVESTIGATED_BY]-(c:Crime {type: 'Drugs'})
<-[:PARTY_TO]-(p1:Person)-[:KNOWS]-
(p2:Person)-[:KNOWS]-(p3:Person)
-[:PARTY_TO]->(:Crime {type: 'Drugs'})
RETURN o.name, c.type, p1.name, p2.name,
p3.name, c.type
```

In the first experiment, we ran the traversal query within two parties using the clear-cut algorithm `cc(2,[3,2])`, resulting in two distinct sub-queries. In the second experiment, involving three parties, we employed the clear-cut algorithm `cc(3,[2,2,1])` to segment the query, leading to three distinct sub-queries. For the traversal query involving four parties, we utilized the clear-cut algorithm `cc(4,[2,1,1,1])`, resulting in four distinct sub-queries. Lastly, in the traversal query involving five parties, we applied the clear-cut algorithm `cc(5,[1,1,1,1,1])` for segmentation, resulting in five distinct sub-queries.

3.3 Experimental Results

Table 1 presents the execution times of various experiments conducted under different configurations. We conduct the same experiments using client-based protocol and server-based protocol. Additionally, we run the same experiments on the server-based protocol without using encryption (DH key with HMAC hash table) to show the overhead of adding this layer on the server-based protocol.

Table 1: Execution times for E1– E4 when using Client based and Server based protocols.

Experiments	Client based	Server Based (with Encryption)	Server Based (without Encryption)
E1	73	204	70
E2	101	425	166
E3	118	450	179
E4	140	589	224

As evident from Table 1, the client-based protocol offers better performance but exposes more information between parties. On the other hand, the server-based protocol with encryption provides a higher level of security but exhibits higher execution times than the client-based protocol due to rounds of encryption and decryption and collecting the final result.

The server-based protocol with encryption incurs additional overhead compared to the client-based protocol. However, the server-based protocol without encryption demonstrates lower execution times compared to the server-based protocol with encryption for all experiments. This shows that the encryption process adds significant overhead to the server-based protocol.

The choice between the two protocols is based on the preference of the user, either high performance or high security. If the clients prioritize high performance and the data being transmitted is not highly sensitive, the client-based protocol is the better choice. However, if security is of utmost importance and cannot be compromised on data protection, the server-based protocol may be the better choice despite the higher execution times.

According to this analysis, we suggest a third protocol, a hybrid protocol, to take advantage of the performance of the client-based protocol and the security of the server-based protocol. The idea is, instead of sharing all the results with the server, we share the last node as in the client-based protocol but in encrypted form and let the server find the intersection only. Then, the clients perform the filtering of data and complete the query.

4 RELATED WORK

There have been recent efforts to integrate SMPC with databases to enhance data security. The following section summarizes some of the existing works and provides comparisons between them based on various factors.

Model of Data: Most existing works have integrated SMPC with the relational database model, such as Conclave (Volgushev et al., 2019), SMCQL (Bater et al., 2016), Senate (Poddar et al., 2020), SAQE (Bater et al., 2020), Secure Yannakakis (Wang and Yi, 2021), Shrinkwrap (Bater et al., 2018), Secrecy (Liagouris et al., 2021), VaultDB (Rogers et al., 2022), Scape (Han et al., 2022), Sequre (Smajlović et al., 2023), SDB (He et al., 2015), and Hu-Fu (Tong et al., 2022). Recently, there has been exploration into applying SMPC to different data models beyond the traditional relational model, as seen in GOSSE (Ciucanu and Lafourcade, 2020b) and SMPQ (Al-Juaid et al., 2022), both of which deal with graph databases.

Number of parties supported: Most of these works support SMPC within two parties, including SMCQL, Senate, SAQE, Secure Yannakakis, and VaultDB. On the other hand, Scape and Sequre specifically support three-party scenarios. Furthermore, Conclave, Hu-Fu, and SMPQ apply SMPC protocols for more than two parties.

SMPC backend: Some of these works utilize existing implementations of SMPC as a backend to perform joint queries. For example, JIFF (Albab et al., 2019) employs secret sharing protocol and serves as the backend for both Conclave and SMPQ. Similarly, OblivM (Liu et al., 2015)is used as a backend for systems supporting two parties, such as SMCQL, SAQE, and Shrinkwrap. In contrast, both SDB and GOOSE use SMPC as a backend over a database, but they do not support multi-party user queries.

To the best of our knowledge, this research is the first work to develop a system for preserving the privacy of traversal multi-party queries within a federated graph database.

5 CONCLUSION

We have developed a framework for securing traversal queries between multiple private graph databases. Our system provides users with two protocols, the choice between which is determined based on the client's preferences and the amount of information

they are willing to share. We have conducted experiments to compare the efficiency of the two protocols. The results demonstrate that the execution times for the server-based protocol are approximately half an order of magnitude higher. In future work, we plan to extend the system to secure traversal queries for cases where our assumption of a clear-cut scenario does not apply. Additionally, we aim to expand the system to handle secure multi-party queries over heterogeneous federated databases supporting different data models.

REFERENCES

Al-Juaid, N., Lisitsa, A., and Schewe, S. (2022). Smpg: Secure multi party computation on graph databases. In *ICISSP*, pages 463–471.

Albab, K. D., Issa, R., Lapets, A., Flockhart, P., Qin, L., and Globus-Harris, I. (2019). Tutorial: Deploying secure multi-party computation on the web using JIFF. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 3–3. IEEE.

Bater, J., Elliott, G., Eggen, C., Goel, S., Kho, A., and Rogers, J. (2016). SMCQL: secure querying for federated databases. *arXiv preprint arXiv:1606.06808*.

Bater, J., He, X., Ehrich, W., Machanavajjhala, A., and Rogers, J. (2018). Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3):307–320.

Bater, J., Park, Y., He, X., Wang, X., and Rogers, J. (2020). SAQE: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment*, 13(12):2691–2705.

Ciucanu, R. and Lafourcade, P. (2020a). GOOSE: A secure framework for graph outsourcing and sparql evaluation. In *34th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec'20). Accepté, à paraître*.

Ciucanu, R. and Lafourcade, P. (2020b). GOOSE: A secure framework for graph outsourcing and sparql evaluation. In *34th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec'20). Accepté, à paraître*.

Evans, D., Kolesnikov, V., and Rosulek, M. (2018). A pragmatic introduction to secure multi-party computation. *Found. Trends Priv. Secur.*, 2:70–246.

Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor, A. (2018). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445.

Gu, Z., Corcoglioniti, F., Lanti, D., Mosca, A., Xiao, G., Xiong, J., and Calvanese, D. (to appear). A systematic overview of data federation systems. *Semantic Web*.

Guia, J., Soares, V. G., and Bernardino, J. (2017). Graph databases: Neo4j analysis. In *ICEIS (1)*, pages 351–356.

Han, F., Zhang, L., Feng, H., Liu, W., and Li, X. (2022). Scape: Scalable collaborative analytics system on private database with malicious security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1740–1753. IEEE.

He, Z., Wong, W. K., Kao, B., Cheung, D., Li, R., Yiu, S., and Lo, E. (2015). SDB: A secure query processing system with data interoperability. *Proc. VLDB Endow.*, 8:1876–1879.

Hunger, M. (2020). neo4j-graph-examples/pole. https://github.com/neo4j-graph-examples/pole/. Accessed: 2023-12-28.

Liagouris, J., Kalavri, V., Faisal, M., and Varia, M. (2021). Secrecy: Secure collaborative analytics on secret-shared data. *arXiv preprint arXiv:2102.01048*.

Liu, C., Wang, X. S., Nayak, K., Huang, Y., and Shi, E. (2015). ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376.

Miller, J. J. (2013). Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324.

Nayak, K., Wang, X. S., Ioannidis, S., Weinsberg, U., Taft, N., and Shi, E. (2015). Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*, pages 377–394. IEEE.

Poddar, R., Kalra, S., Yanai, A., Deng, R., Popa, R. A., and Hellerstein, J. M. (2020). Senate: A maliciously-secure mpc platform for collaborative analytics. *arXiv e-prints*, pages arXiv–2010.

Rogers, J., Adetoro, E., Bater, J., Canter, T., Fu, D., Hamilton, A., Hassan, A., Martinez, A., Michalski, E., Mitrovic, V., et al. (2022). Vaultdb: A real-world pilot of secure multi-party computation within a clinical research network. *arXiv preprint arXiv:2203.00146*.

Salehnia, A. (2017). Comparisons of relational databases with big data: a teaching approach. *South Dakota State University Brookings, SD 57007*, pages 1–8.

Smajlović, H., Shajii, A., Berger, B., Cho, H., and Numanagić, I. (2023). Sequre: a high-performance framework for secure multiparty computation enables biomedical data sharing. *Genome Biology*, 24(1):1–18.

Tong, Y., Pan, X., Zeng, Y., Shi, Y., Xue, C., Zhou, Z., Zhang, X., Chen, L., Xu, Y., Xu, K., et al. (2022). Hu-fu: Efficient and secure spatial queries over data federation. *Proceedings of the VLDB Endowment*, 15(6):1159.

Volgushev, N., Schwarzkopf, M., Getchell, B., Varia, M., Lapets, A., and Bestavros, A. (2019). Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–18.

Wang, Y. and Yi, K. (2021). Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1969–1981, New York, NY, USA. Association for Computing Machinery.