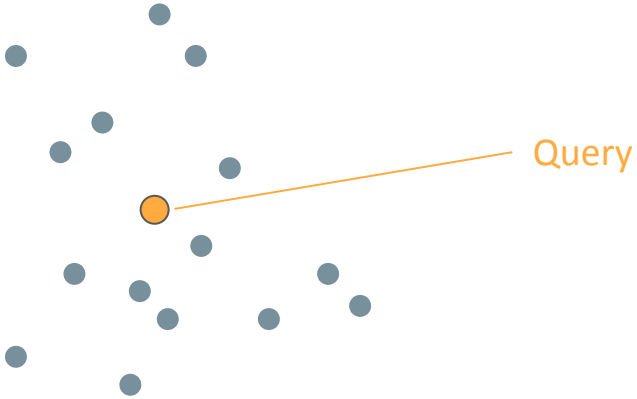
Private Approximate Nearest Neighbor Search with Sublinear Communication

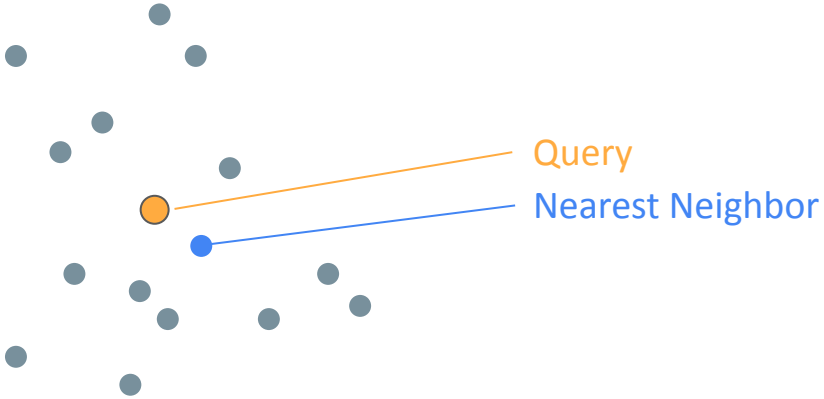
Sacha Servan-Schreiber
with Simon Langowski and Srini Devadas

MIT CSAIL

Nearest Neighbor (NN) search

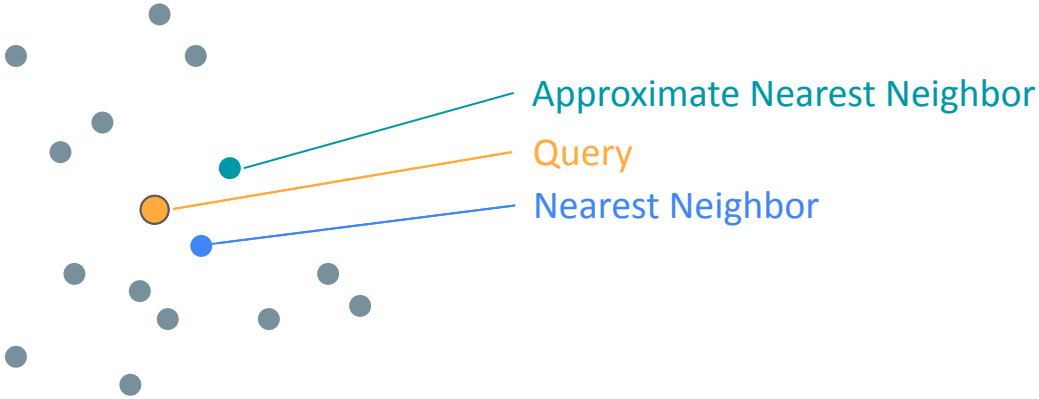


Nearest Neighbor (NN) search



Approximate Nearest Neighbor (ANN) search

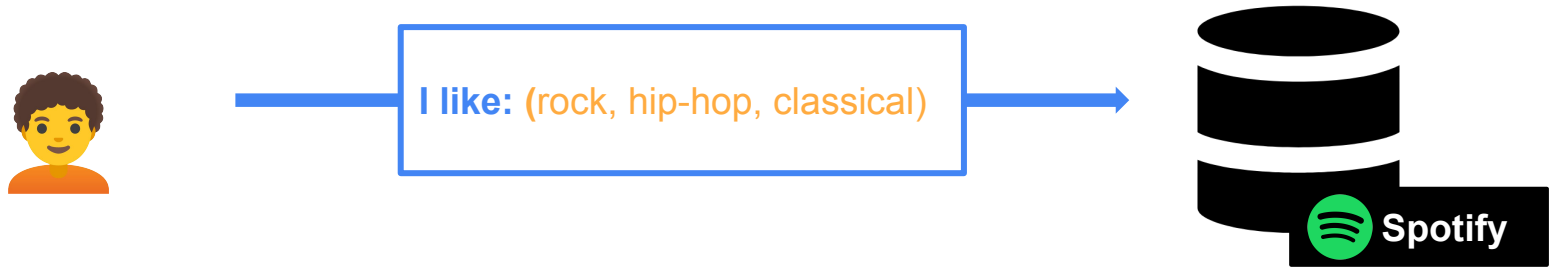
(standard relaxation used in practice)



Example: music recommendations [1]

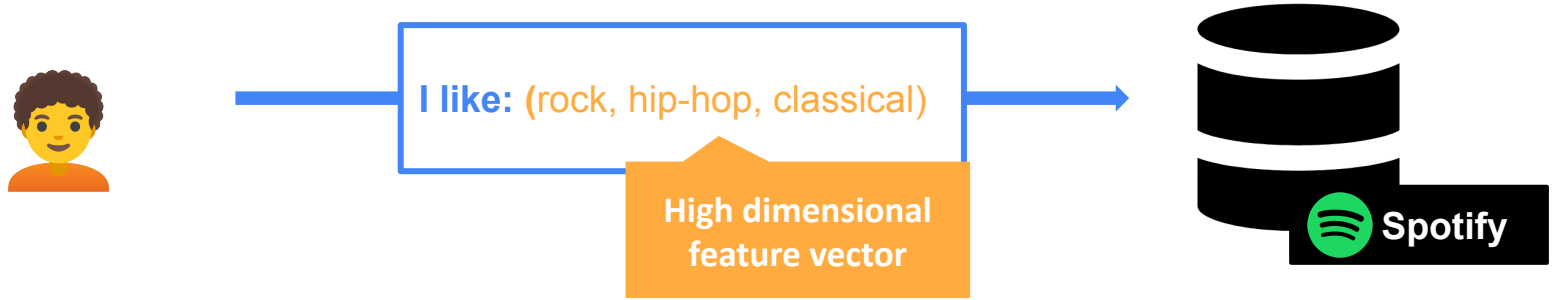
[1]: https://github.com/spotify/annoy

Example: music recommendations [1]



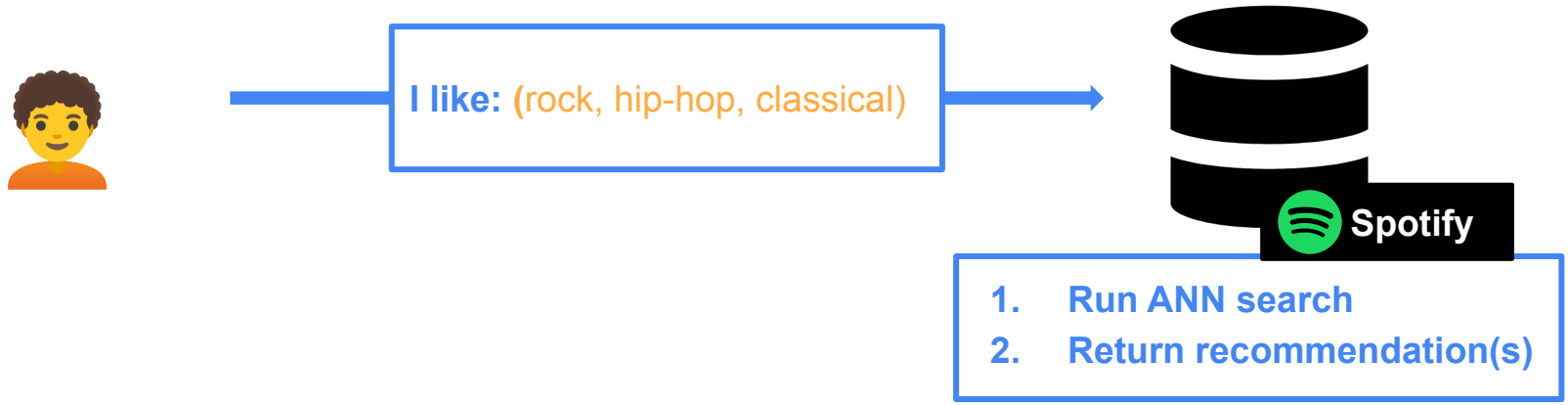
[1]: https://github.com/spotify/annoy

Example: music recommendations [1]



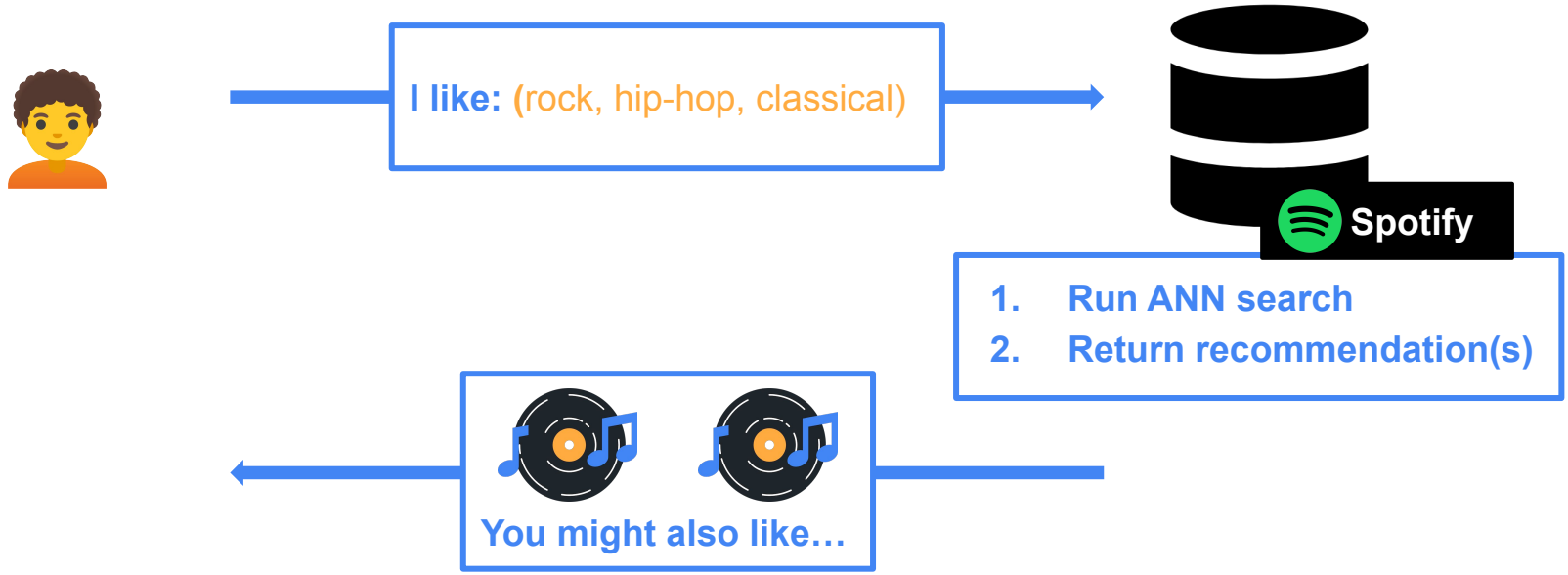
[1]: https://github.com/spotify/annoy

Example: music recommendations [1]



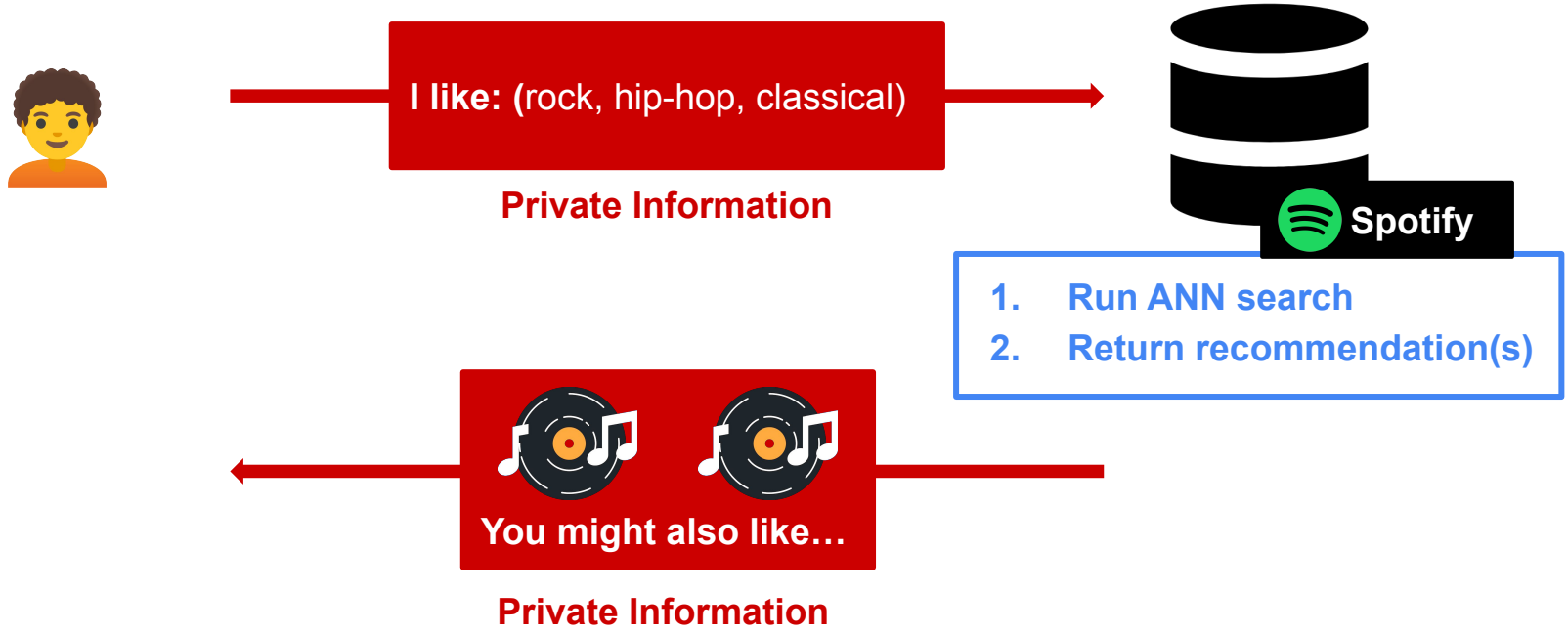
[1]: https://github.com/spotify/annoy

Example: music recommendations [1]



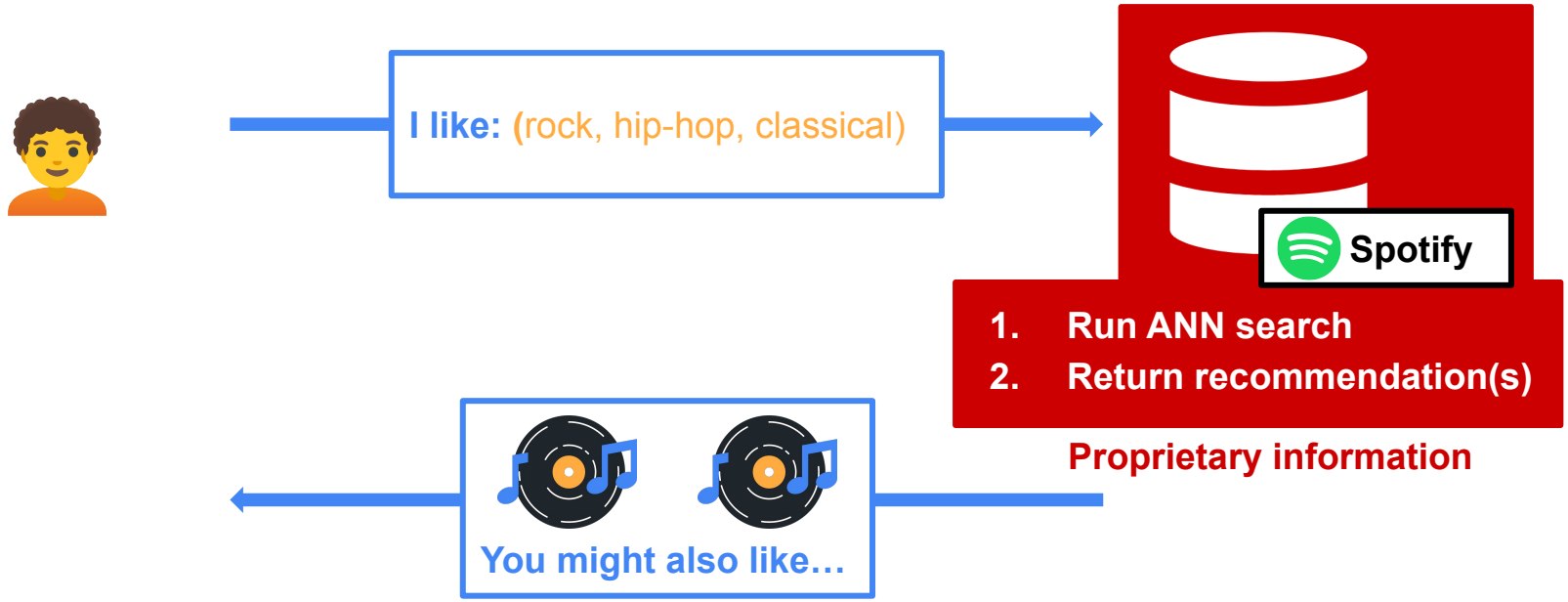
[1]: https://github.com/spotify/annoy

Example: music recommendations [1]



[1]: https://github.com/spotify/annoy

Example: music recommendations [1]



[1]: https://github.com/spotify/annoy

Prior work on private ANN search

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

Fully-homomorphic Encryption

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

- **Not** lightweight:
  - **1 to 5 GB of communication** with databases of 1,000,000 items [1].

Fully-homomorphic Encryption

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

- **Not** lightweight:
  - **1 to 5 GB of communication** with databases of 1,000,000 items [1].
- **No** malicious security:
  - A **malicious** database can deviate from protocol and learn the query.
  - A **malicious** client can deviate from protocol and recover the database.

Fully-homomorphic Encryption

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

- **Not** lightweight:
  - **1 to 5 GB of communication** with databases of 1,000,000 items [1].
- **No** malicious security:
  - A **malicious** database can deviate from protocol and learn the query.
  - A **malicious** client can deviate from protocol and recover the database.

Fully-homomorphic Encryption

- **Is** lightweight for the client.

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

- **Not** lightweight:
  - **1 to 5 GB of communication** with databases of 1,000,000 items [1].
- **No** malicious security:
  - A **malicious** database can deviate from protocol and learn the query.
  - A **malicious** client can deviate from protocol and recover the database.

Fully-homomorphic Encryption

- **Is** lightweight for the client.
- **Is** *trivially* maliciously-secure.

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

- **Not** lightweight:
  - **1 to 5 GB of communication** with databases of 1,000,000 items [1].
- **No** malicious security:
  - A **malicious** database can deviate from protocol and learn the query.
  - A **malicious** client can deviate from protocol and recover the database.

Fully-homomorphic Encryption

- **Is** lightweight for the client.
- **Is** *trivially* maliciously-secure.
- **Not** lightweight for the database.
  - Takes **hours** with small databases of e.g., 500 to 2,000 items [2].

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Prior work on private ANN search

Two-party Computation

Fully-homomorphic Encryption

Our goals:

(1) **low communication** for the client,

(2) concrete **efficiency** for the database,

(3) **privacy** for the client and the database,

(4) and **malicious security**.

[1]: Chen, Hao, et al. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." *29th USENIX Security Symposium (USENIX Security 20)*. 2020.
[2]: Shaul, Hayim, Dan Feldman, and Daniela Rus. "Secure-ish Nearest Neighbors Classifier." *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020): 42-61.

Our work

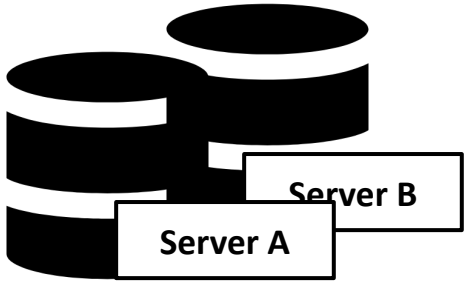
The setting: two non-colluding database servers.

Q: Why do we need non-colluding servers?

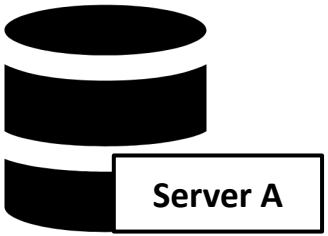
A: For efficient, symmetric-key cryptography only.[1]

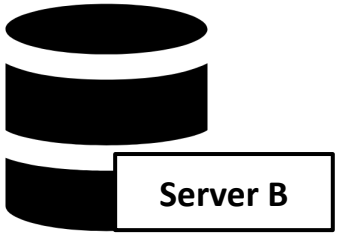
[1] See Appendix E of the full version of our paper [1] for a **single-server protocol** that is less concretely efficient but doesn't require any trust assumptions.

[1]: Full version of our paper: https://eprint.iacr.org/2021/1157.pdf.

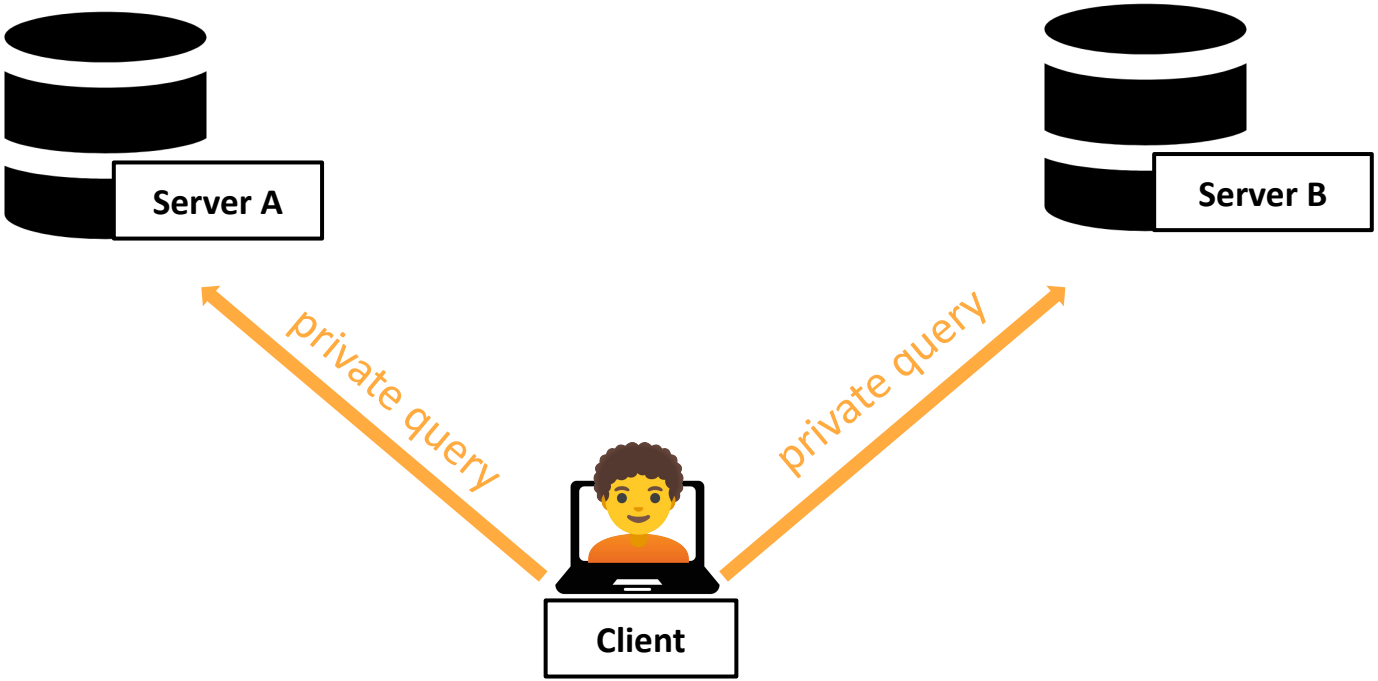


Our work

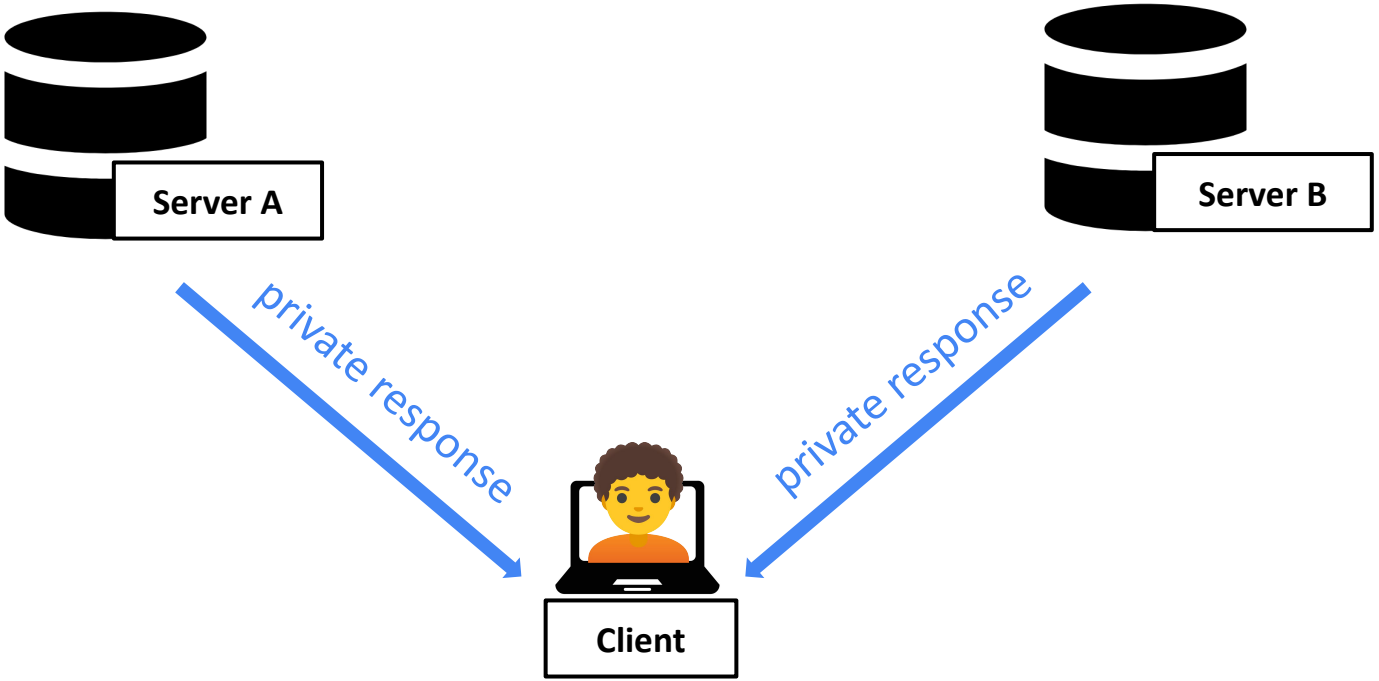




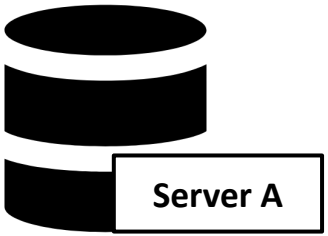
Our work

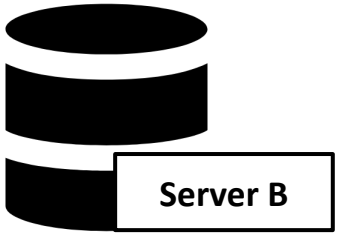


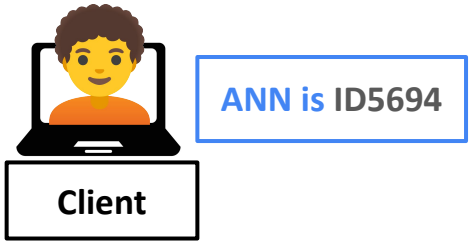
Our work


Server A
Server B
private response
private response
Client

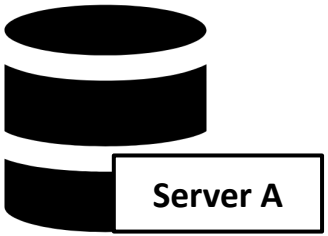
Our work

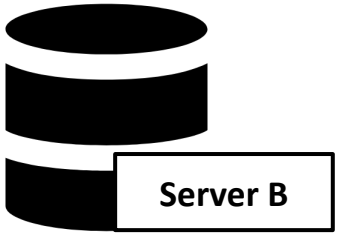


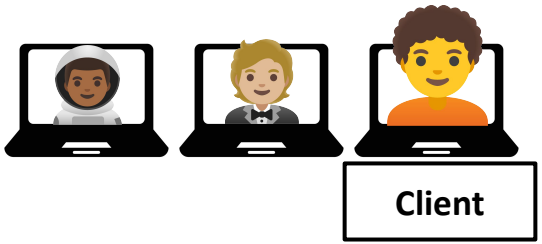




Our work

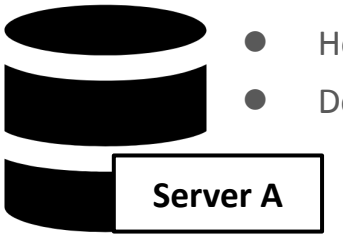




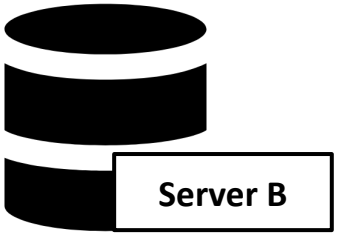


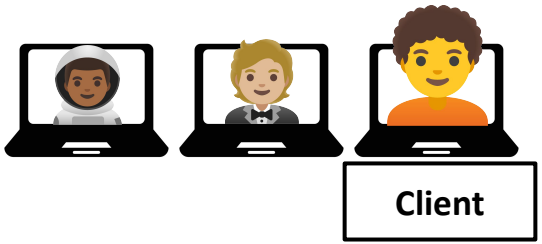
Threat model

Servers



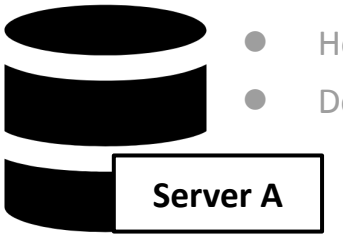
- Hold replicas of the database.
- Do not collude with clients or one another.



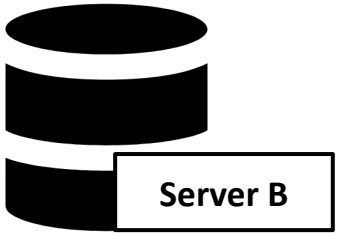


Threat model

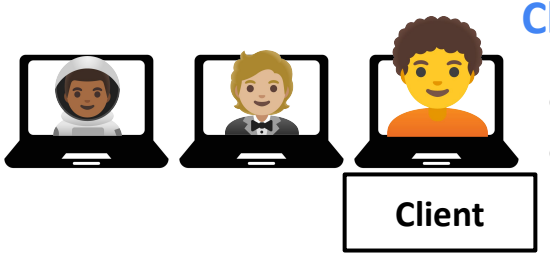
Servers



- Hold replicas of the database.
- Do not collude with clients or one another.



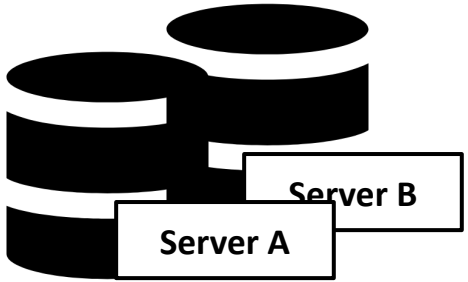
Clients

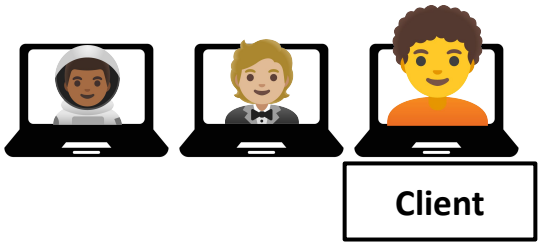


- Will try to learn as much as possible about the database.
- May collude with other malicious clients.

Threat model

Guarantees

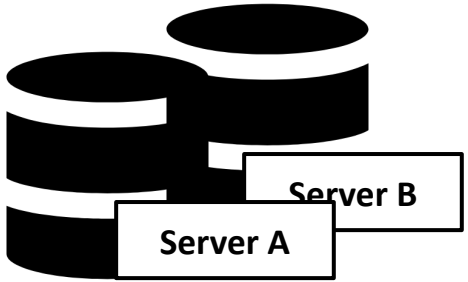


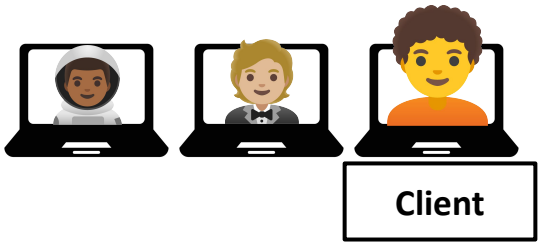


Threat model

Guarantees

- **Accuracy** if both servers follow the protocol.

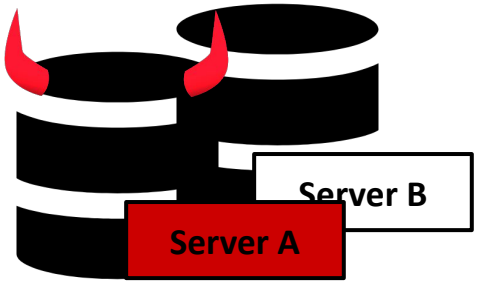


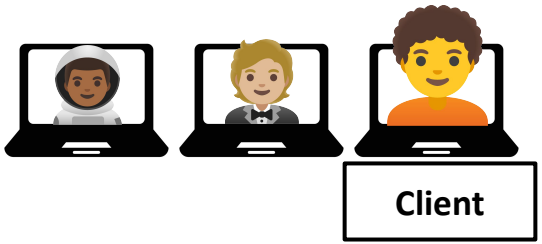


Threat model

Guarantees

- **Accuracy** if both servers follow the protocol.
- **User privacy** even if a server is malicious.

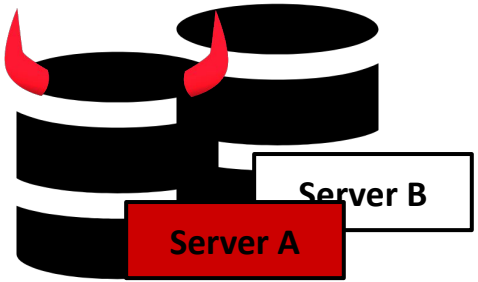


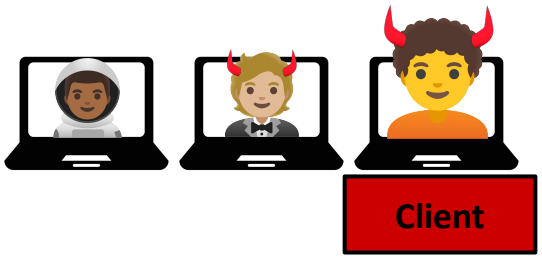


Threat model

Guarantees

- **Accuracy** if both servers follow the protocol.
- **User privacy** even if a server is malicious.
- **Database privacy** even if a subset of clients are malicious.

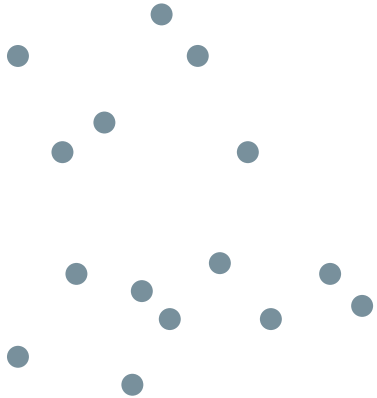




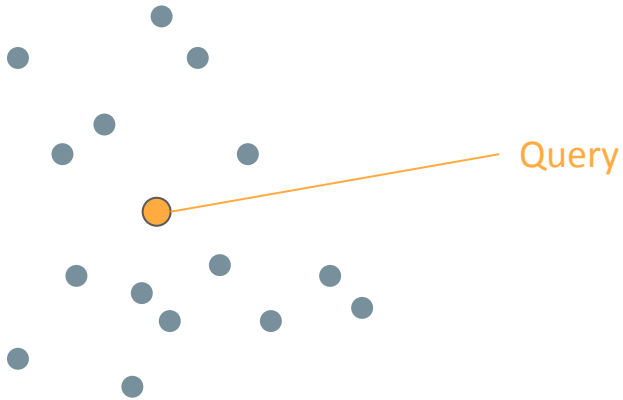
Finding Nearest Neighbors

(non-privately, using Locality-Sensitive Hashing)

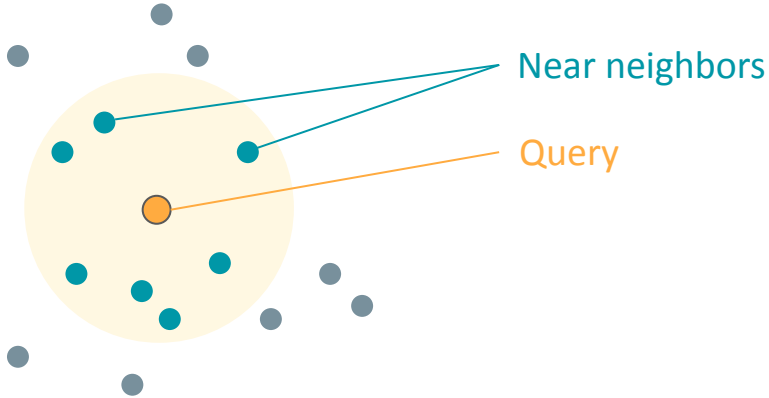
Finding the ANN using LSH (non-privately)



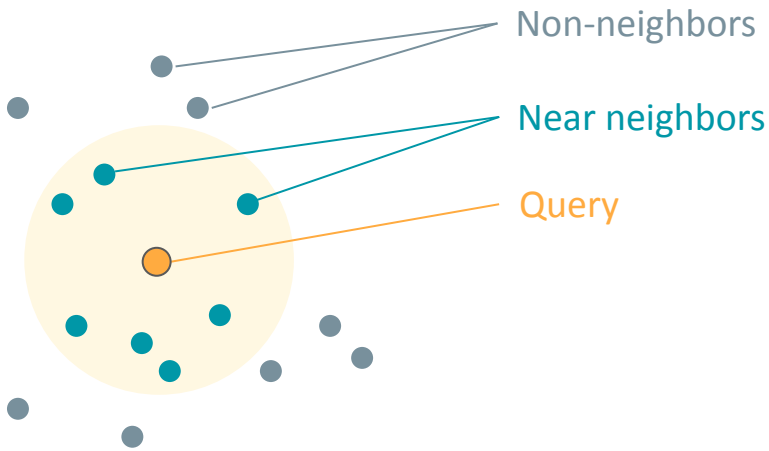
Finding the ANN using LSH (non-privately)



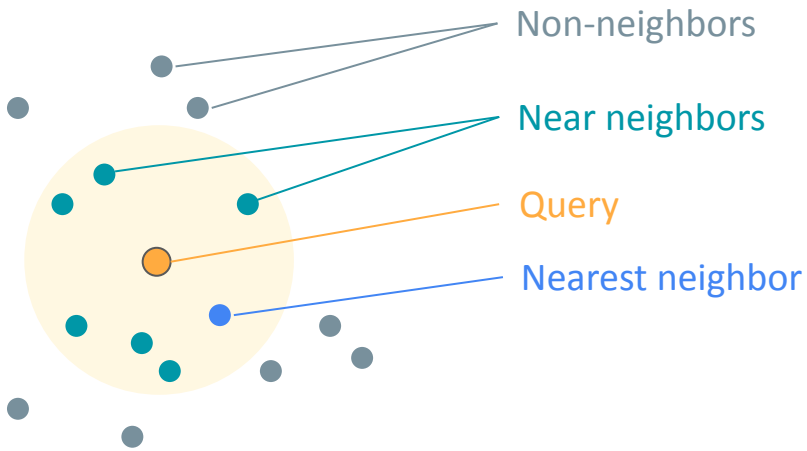
Finding the ANN using LSH (non-privately)



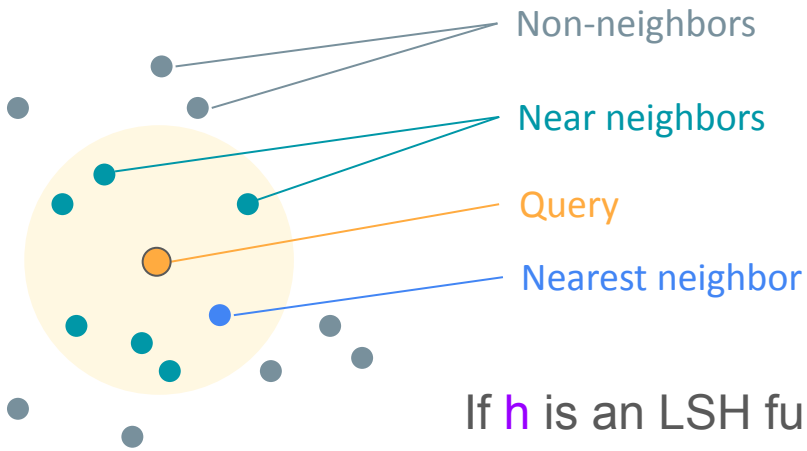
Finding the ANN using LSH (non-privately)



Finding the ANN using LSH (non-privately)

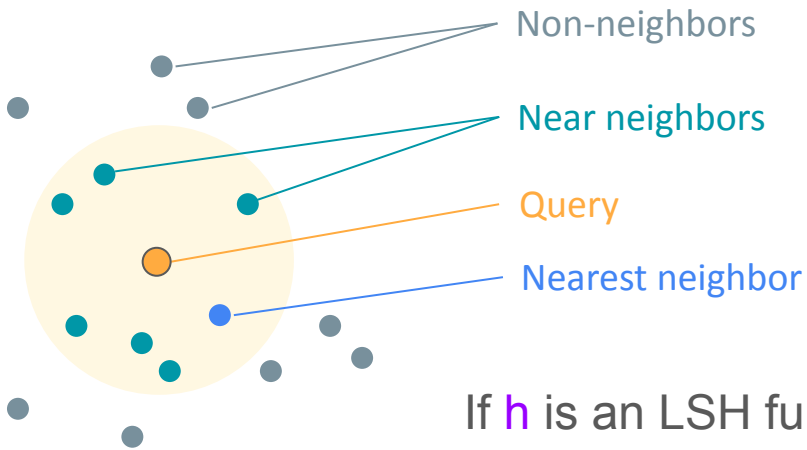


Finding the ANN using LSH (non-privately)



If h is an LSH function:

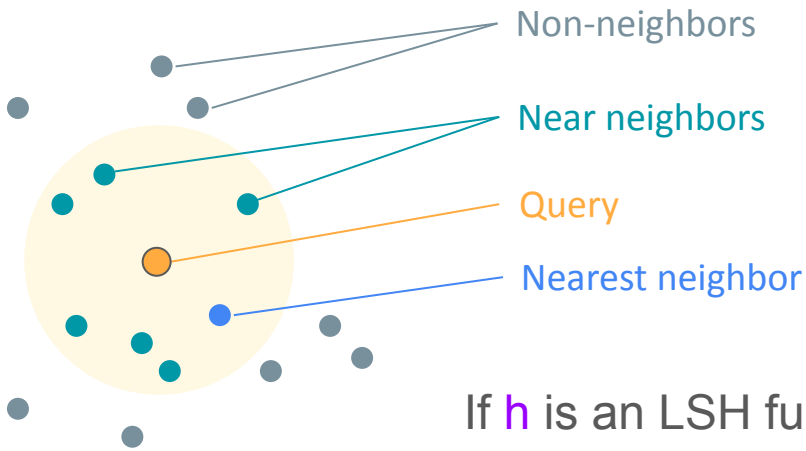
Finding the ANN using LSH (non-privately)



If h is an LSH function:

Pr[h(●) = h(●)] is high

Finding the ANN using LSH (non-privately)

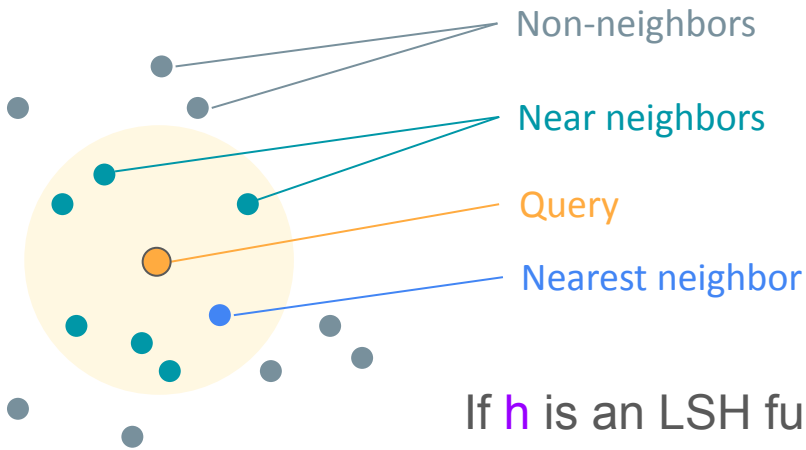


If h is an LSH function:

Pr[h(●) = h(●)] is high

Pr[h(●) = h(●)] is low

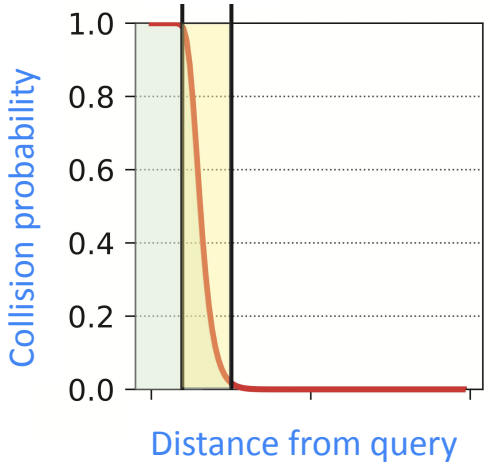
Finding the ANN using LSH (non-privately)



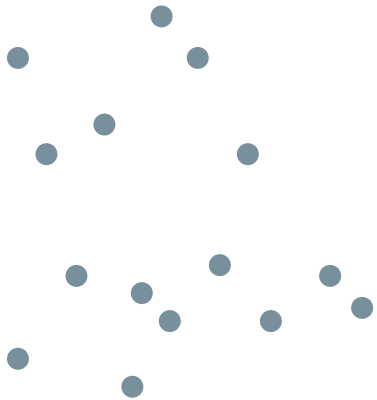
If h is an LSH function:

Pr[h(●) = h(●)] is high

Pr[h(●) = h(●)] is low

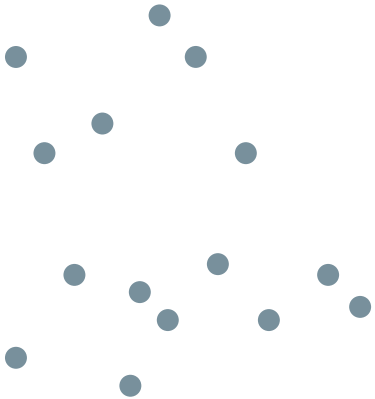


Step 1: Build an LSH hash table using h.



Data

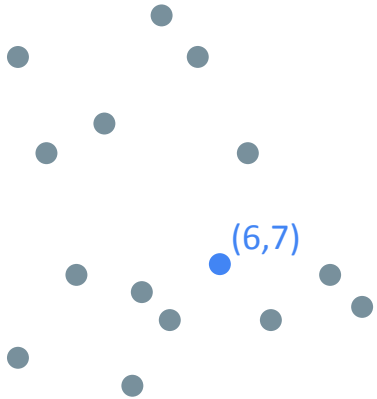
Step 1: Build an LSH hash table using h.



Data

Hashkey	Value(s)

Step 1: Build an LSH hash table using h.

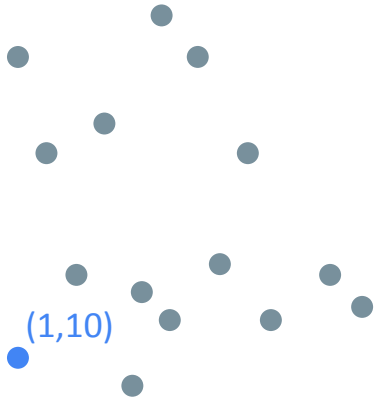


Data

h(●)

Hashkey	Value(s)
09ac34fd	ID3: (6,7)

Step 1: Build an LSH hash table using h.

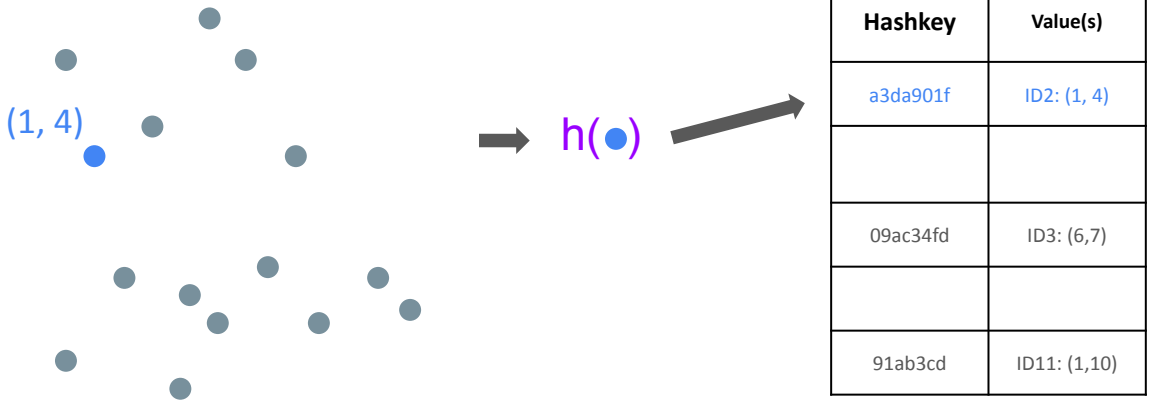


Data

h(●)

Hashkey	Value(s)
09ac34fd	ID3: (6,7)
91ab3cd	ID11: (1,10)

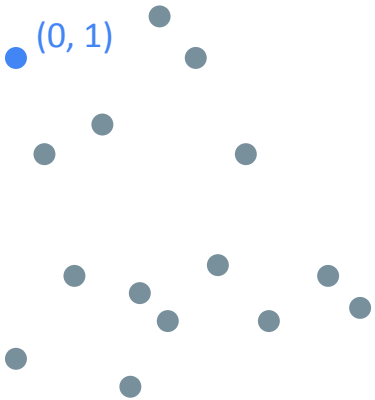
Step 1: Build an LSH hash table using h.



Hashkey	Value(s)
a3da901f	ID2: (1, 4)
09ac34fd	ID3: (6,7)
91ab3cd	ID11: (1,10)

Data

Step 1: Build an LSH hash table using h.



Data

h(●)

Hashkey	Value(s)
a3da901f	ID2: (2, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
91ab3cd	ID11: (1,10)

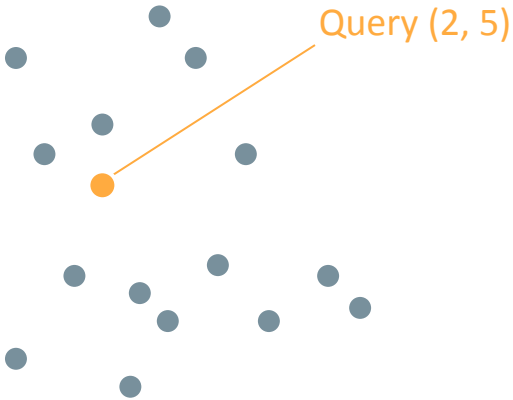
Step 1: Build an LSH hash table using h.

Data

Hashkey	Value(s)
a3da901f	ID2: (1, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
⋮	⋮
91ab3cd	ID11: (1,10)

Step 1: Build an LSH hash table using h.

Step 2: Query the hash table.

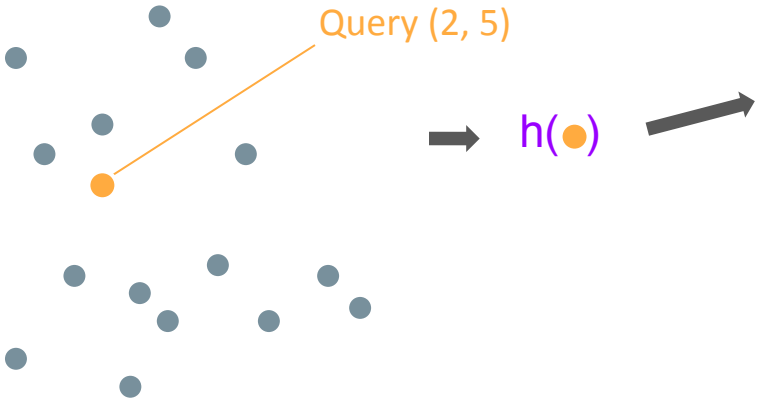


Data

Hashkey	Value(s)
a3da901f	ID2: (1, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
⋮	⋮
91ab3cd	ID11: (1,10)

Step 1: Build an LSH hash table using h.

Step 2: Query the hash table.

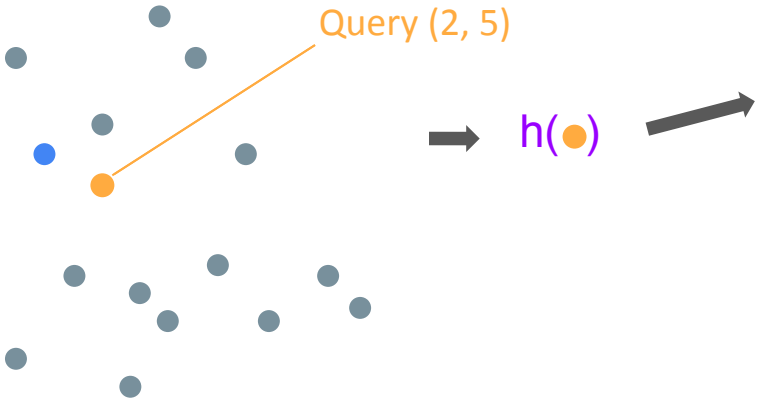


Hashkey	Value(s)
a3da901f	ID2: (1, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
⋮	⋮
91ab3cd	ID11: (1,10)

Data

Step 1: Build an LSH hash table using h.

Step 2: Query the hash table.



Hashkey	Value(s)
a3da901f	ID2: (1, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
⋮	⋮
91ab3cd	ID11: (1,10)

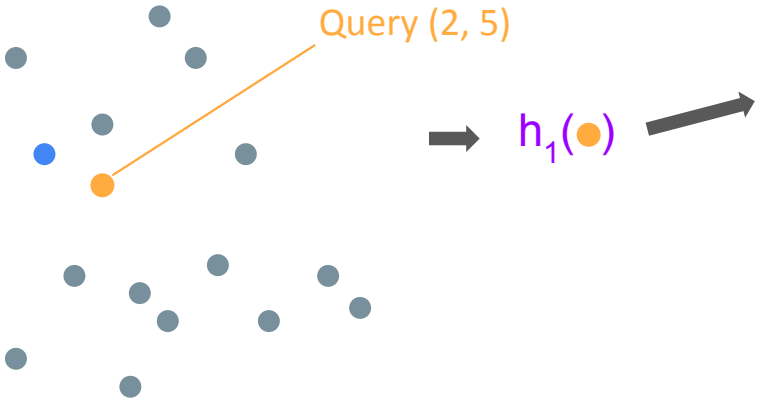
Data

Candidate set: {ID2: (1,4)}

Step 1: Build an LSH hash table using h.

Step 2: Query the hash table.

Step 3: Repeat with many different hash tables.



Hashkey	Value(s)
a3da901f	ID2: (1, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
⋮	⋮
91ab3cd	ID11: (1,10)

Hashkey	Value(s)
beda11fe	ID1000000
f12fbe10	ID101, ID949
ac33445a	ID5694
⋮	⋮
91ab3cd	ID11, ID9191

Hashkey	Value(s)
aa423ef	ID900
10acdff	ID10
ff1e3432	ID514
⋮	⋮
f2f2e12	ID141

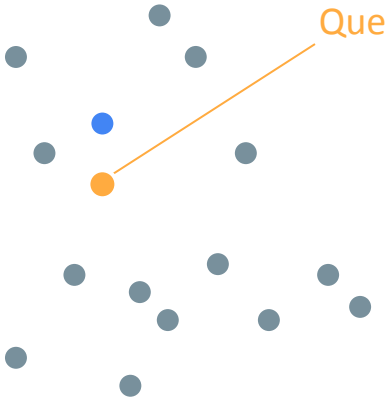
Data

Candidate set: {ID2: (1,4)}

Step 1: Build an LSH hash table using h.

Step 2: Query the hash table.

Step 3: Repeat with many different hash tables.



Data

$h_2(\bullet)$

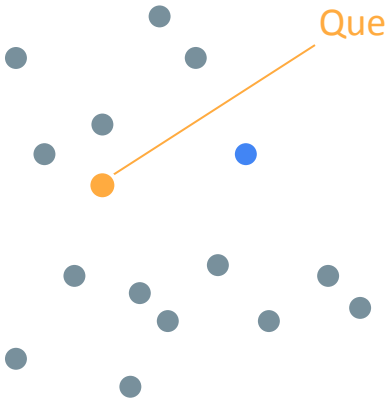
Hashkey	Value(s)
beda11fe	ID10: (9,10)
f12fbe10	ID101: (5,0)
ac33445a	ID5694: (2,4)
⋮	⋮
91ab3cd	ID11: (4,8)

Candidate set: {ID2: (1,4), ID5694: (2,4)}

Step 1: Build an LSH hash table using h.

Step 2: Query the hash table.

Step 3: Repeat with many different hash tables.



Data

$h_3(\bullet)$

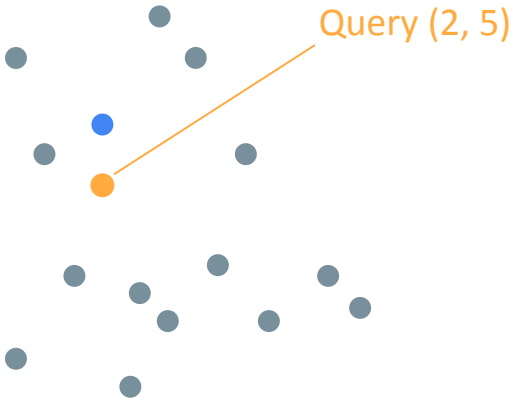
Hashkey	Value(s)
a3da901f	ID2: (1, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
⋮	⋮
91ab3cd	ID11: (1,10)

Hashkey	Value(s)
beda11fe	ID10: (9,10)
f12fbe10	ID101: (5,0)
ac33445a	ID5694: (2,4)
⋮	⋮
91ab3cd	ID11: (4,8)

Hashkey	Value(s)
aa423ef	ID900: (3,4)
10acdff	ID10: (9,10)
ff1e3432	ID514: (8,8)
⋮	⋮
f2f2e12	ID141: (10, 8)

Candidate set: {ID2: (1,4), ID5694: (2,4), ID900: (3,4)}

Step 1: Build an LSH hash table using h.
Step 2: Query the hash table.
Step 3: Repeat with many different hash tables.
Step 4: Find closest neighbor in the candidate set.



Data

Hashkey	Value(s)
a3da901f	ID2: (1, 4)
c26fab1d	ID100: (0,1)
09ac34fd	ID3: (6,7)
⋮	⋮
91ab3cd	ID11: (1,10)

Hashkey	Value(s)
beda11fe	ID10: (9,10)
f12fbe10	ID101: (5,0)
ac33445a	ID5694: (2,4)
⋮	⋮
91ab3cd	ID11: (4,8)

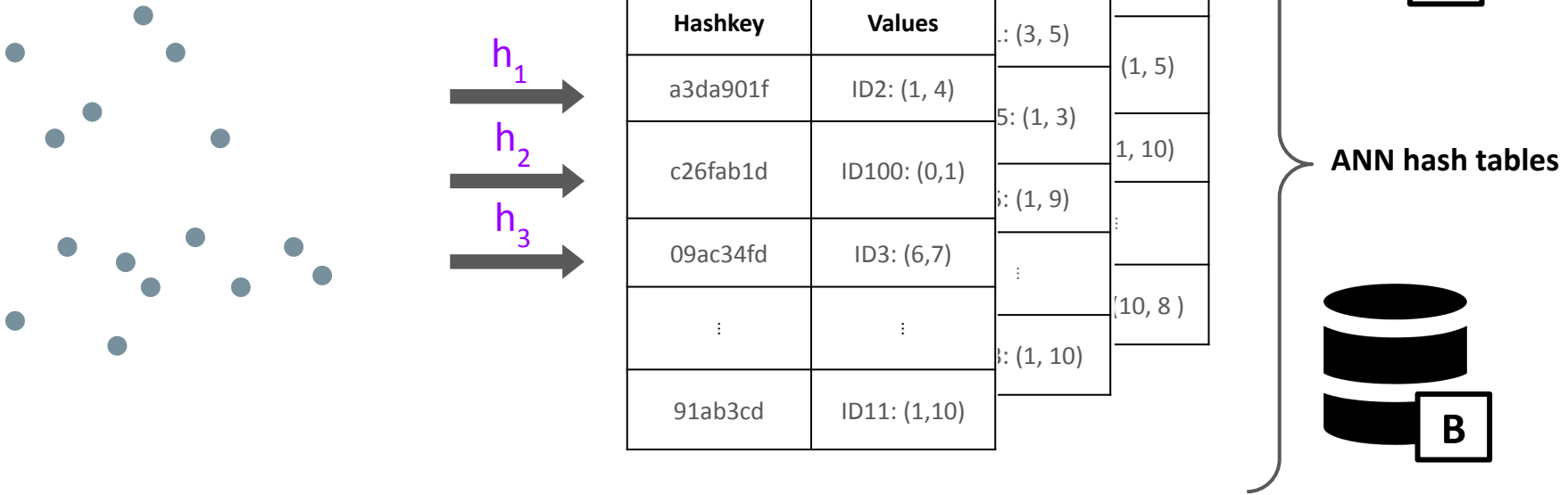
Hashkey	Value(s)
aa423ef	ID900: (3,4)
10acdff	ID10: (9,10)
ff1e3432	ID514: (8,8)
⋮	⋮
f2f2e12	ID141: (10, 8)

Candidate set: {ID2: (1,4), **ID5694**: (2,4), ID900: (3,4)}

A strawman protocol

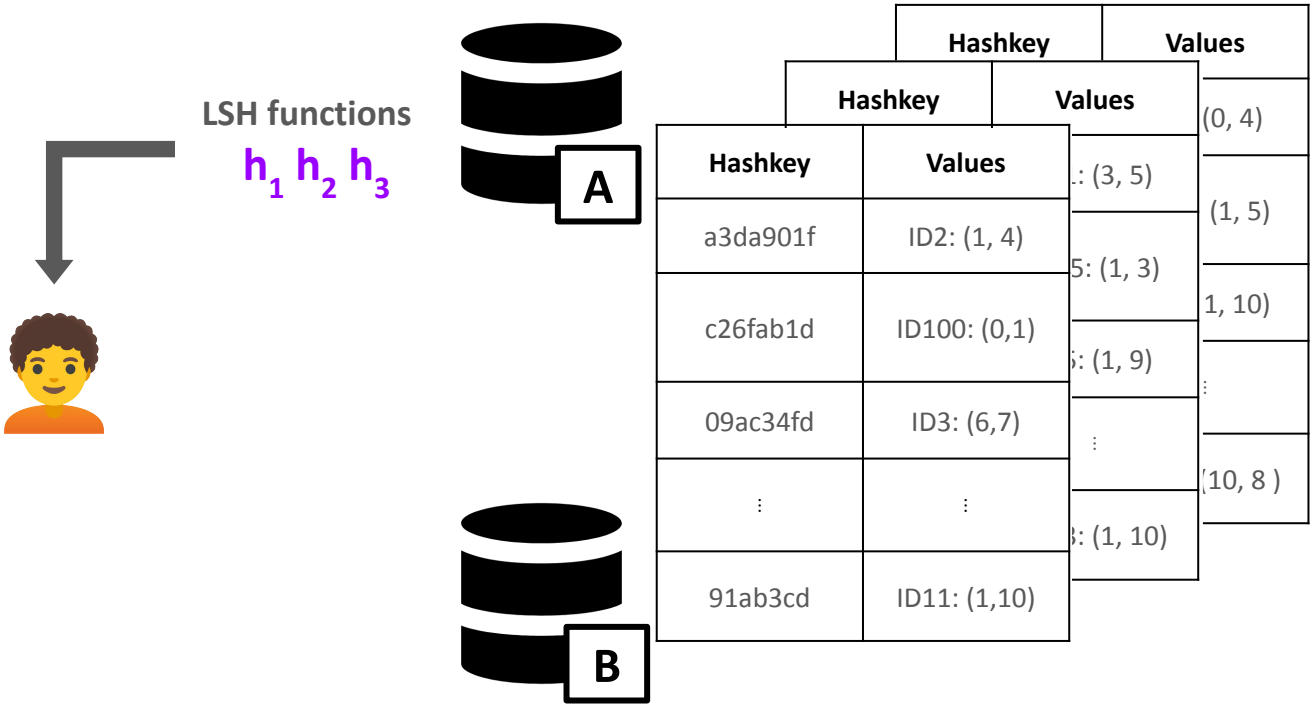
A strawman protocol

One-time setup: Construct LSH tables



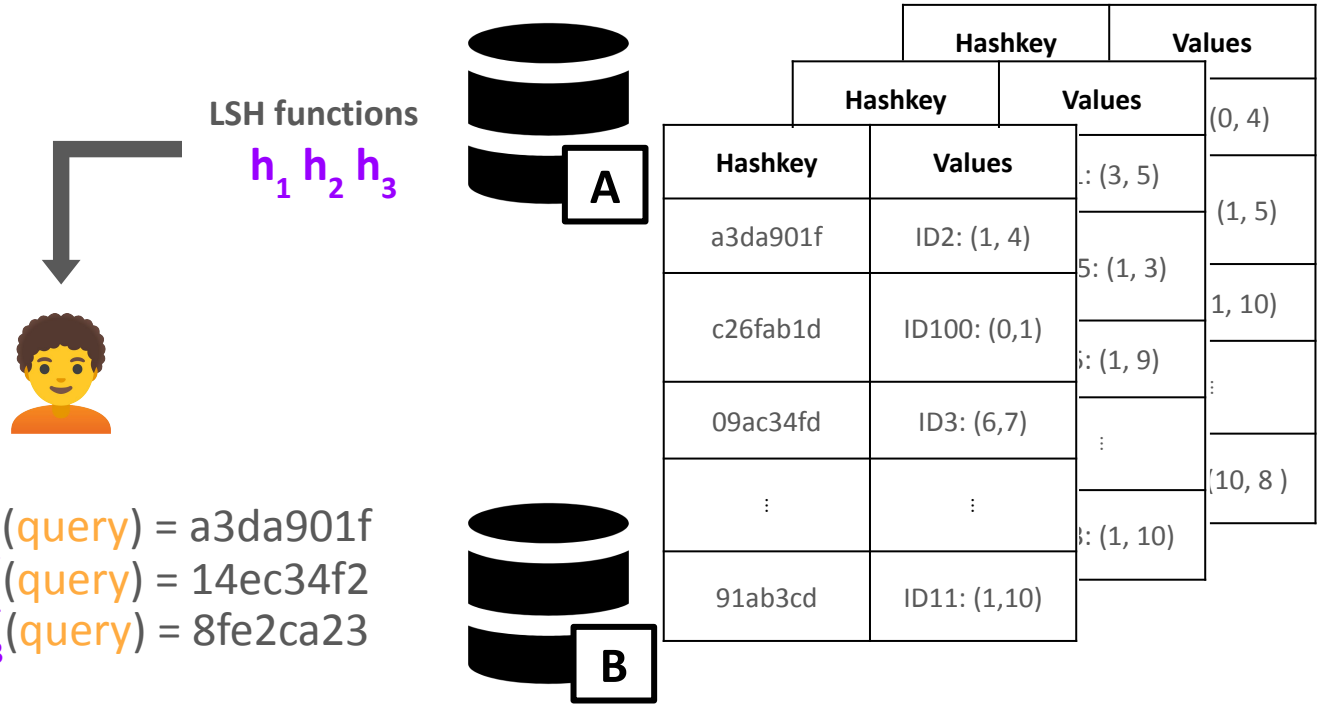
A strawman protocol

Step 1: Client uses LSH functions to find the hashkey



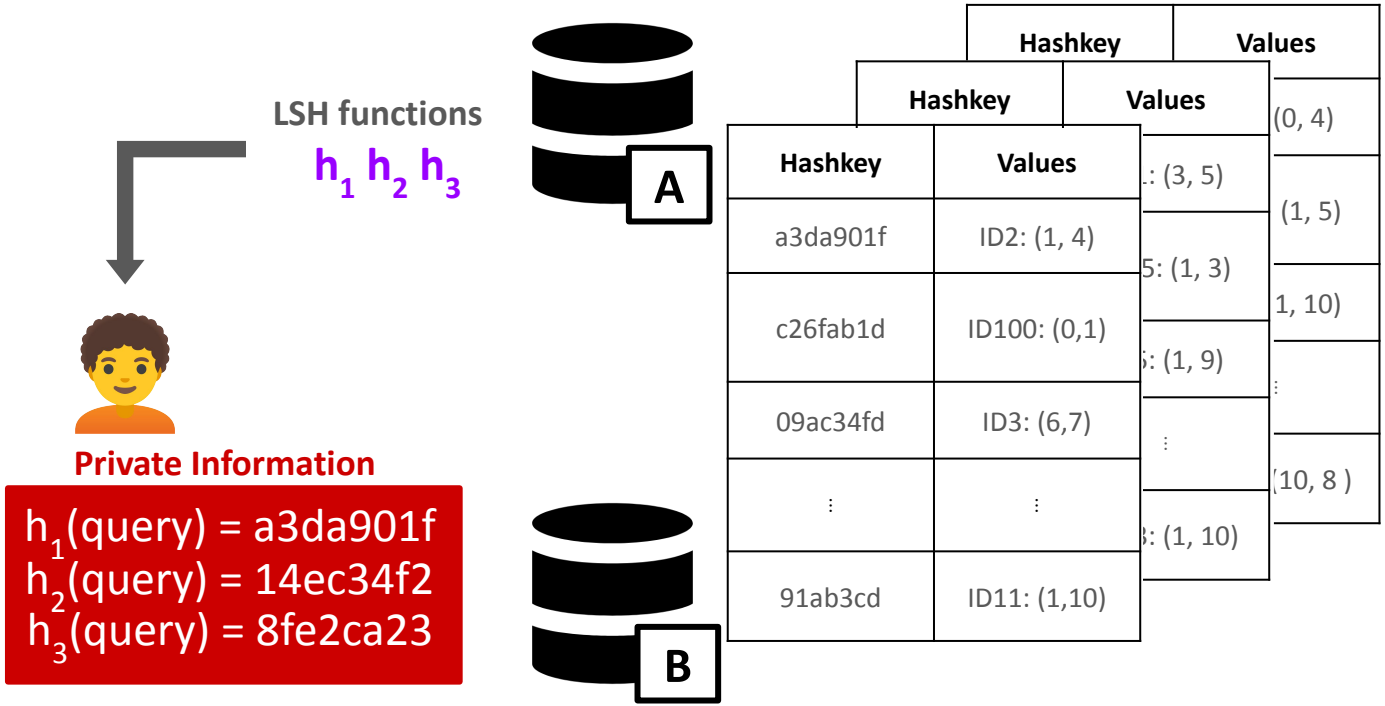
A strawman protocol

Step 1: Client uses LSH functions to find the hashkey



A strawman protocol

Step 1: Client uses LSH functions to find the hashkey



Challenge: querying the hash table privately

Q: How can the client retrieve the candidate from each table privately?

A: Using private information retrieval (PIR) [1].

- We use Distributed Point Functions [2] (DPFs) for efficiently querying hash tables in a two-server setting.

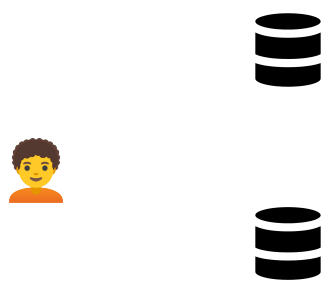
[1]: Gertner, Yael, et al. "Protecting data privacy in private information retrieval schemes." *Journal of Computer and System Sciences* 60.3 (2000): 592-629.
[2]: Gilboa, Niv, and Yuval Ishai. "Distributed point functions and their applications." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2014.

Challenge: querying the hash table privately

Q: How can the client retrieve the candidate from each table privately?

A: Using private information retrieval (PIR) [1].

- We use Distributed Point Functions [2] (DPFs) for efficiently querying hash tables in a two-server setting.



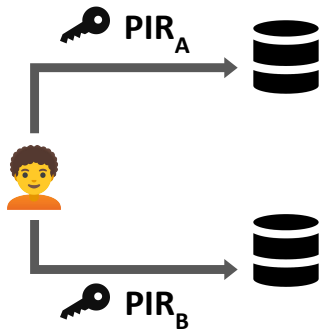
[1]: Gertner, Yael, et al. "Protecting data privacy in private information retrieval schemes." *Journal of Computer and System Sciences* 60.3 (2000): 592-629.
[2]: Gilboa, Niv, and Yuval Ishai. "Distributed point functions and their applications." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2014.

Challenge: querying the hash table privately

Q: How can the client retrieve the candidate from each table privately?

A: Using private information retrieval (PIR) [1].

- We use Distributed Point Functions [2] (DPFs) for efficiently querying hash tables in a two-server setting.



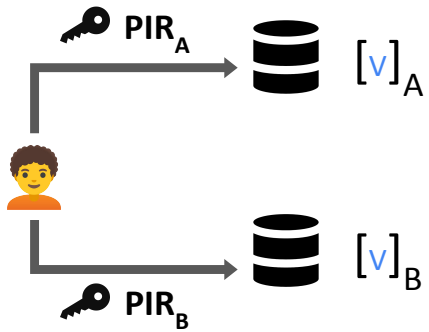
[1]: Gertner, Yael, et al. "Protecting data privacy in private information retrieval schemes." *Journal of Computer and System Sciences* 60.3 (2000): 592-629.
[2]: Gilboa, Niv, and Yuval Ishai. "Distributed point functions and their applications." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2014.

Challenge: querying the hash table privately

Q: How can the client retrieve the candidate from each table privately?

A: Using private information retrieval (PIR) [1].

- We use Distributed Point Functions [2] (DPFs) for efficiently querying hash tables in a two-server setting.



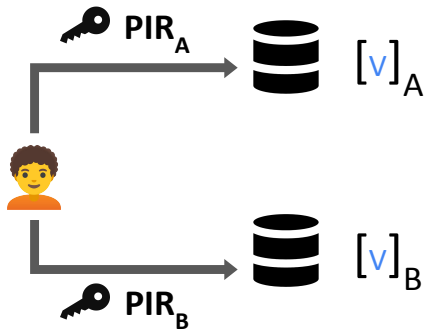
[1]: Gertner, Yael, et al. "Protecting data privacy in private information retrieval schemes." *Journal of Computer and System Sciences* 60.3 (2000): 592-629.
[2]: Gilboa, Niv, and Yuval Ishai. "Distributed point functions and their applications." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2014.

Challenge: querying the hash table privately

Q: How can the client retrieve the candidate from each table privately?

A: Using private information retrieval (PIR) [1].

- We use Distributed Point Functions [2] (DPFs) for efficiently querying hash tables in a two-server setting.



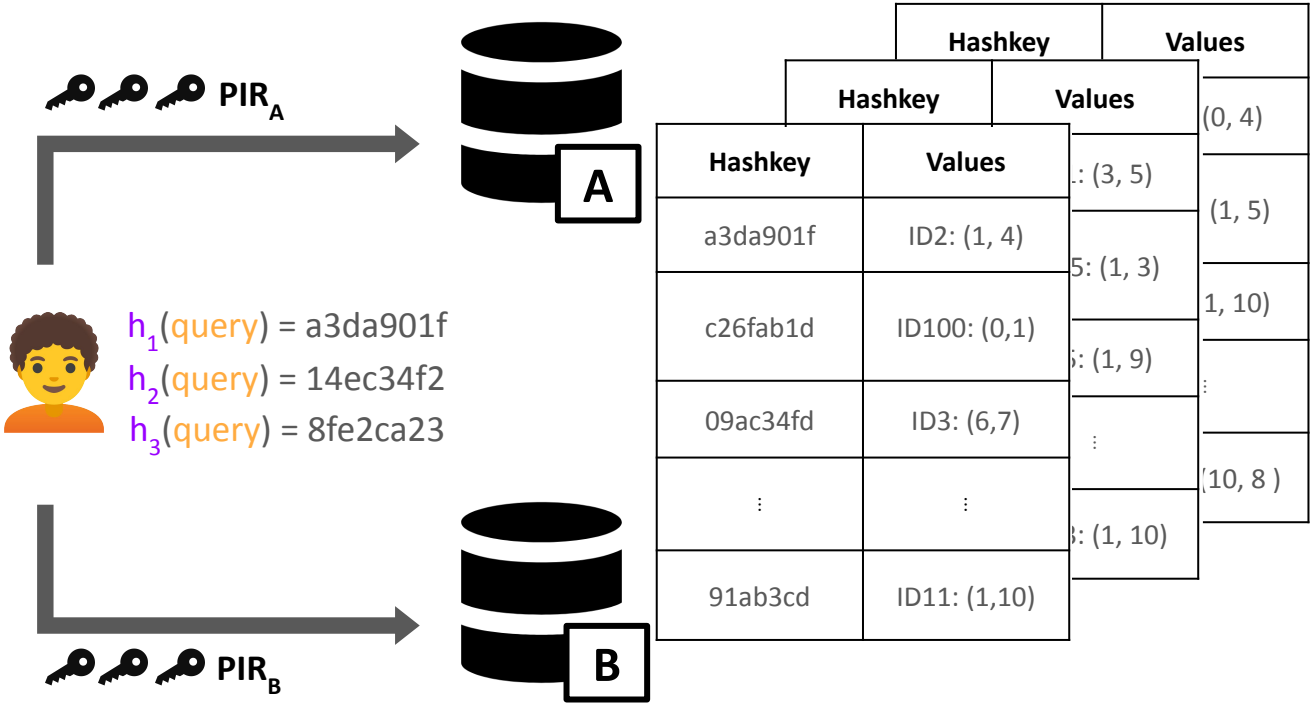
$$[v]_A + [v]_B = v \quad \text{(additive secret shares)}$$

[1]: Gertner, Yael, et al. "Protecting data privacy in private information retrieval schemes." *Journal of Computer and System Sciences* 60.3 (2000): 592-629.
[2]: Gilboa, Niv, and Yuval Ishai. "Distributed point functions and their applications." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2014.

A strawman protocol

Step 1: Client uses LSH functions to find the hashkey

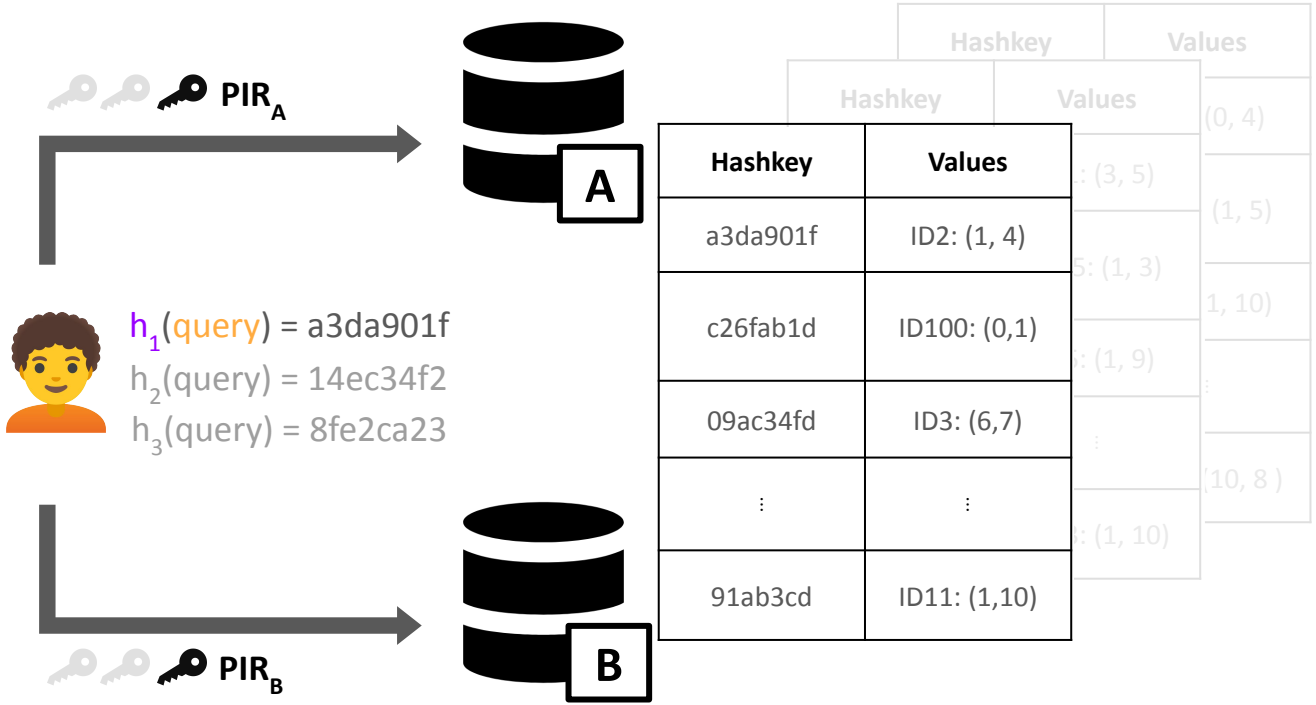
Step 2: Privately query hash tables using PIR



A strawman protocol

Step 1: Client uses LSH functions to find the hashkey

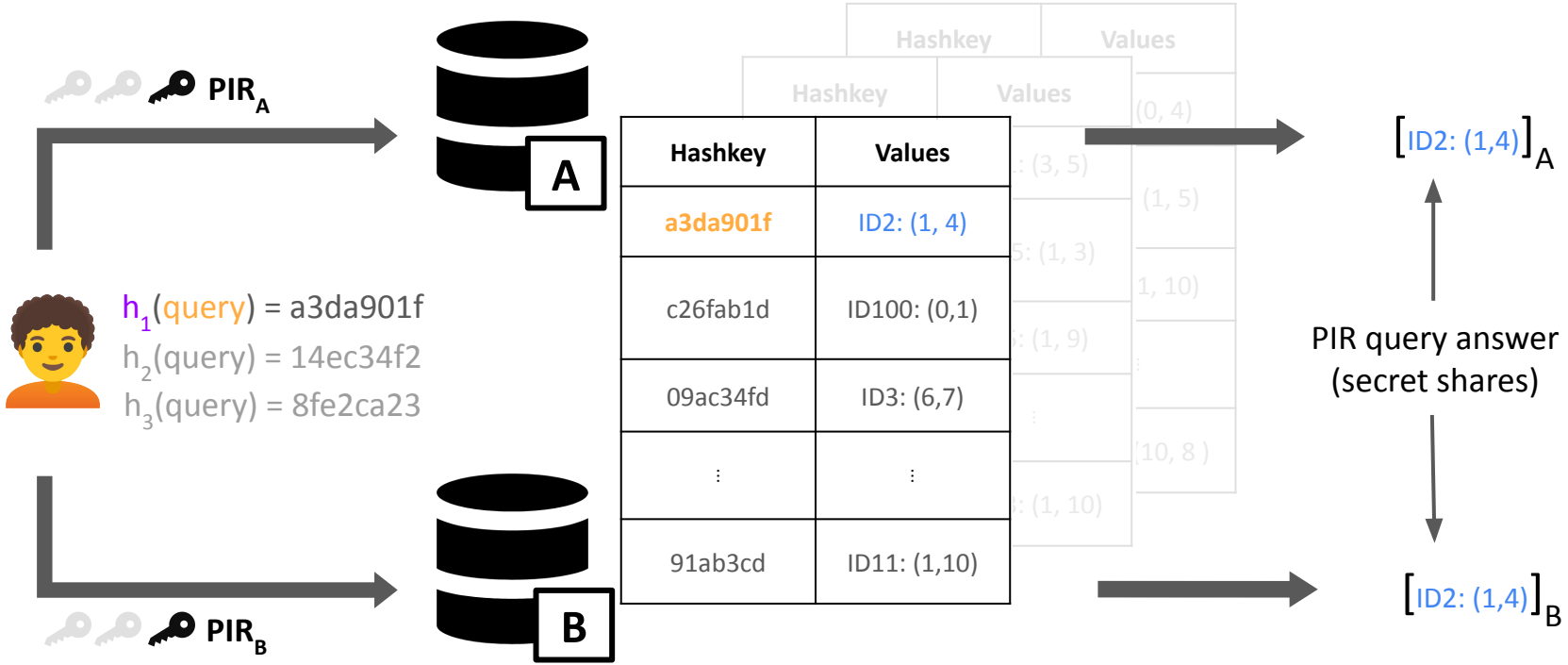
Step 2: Privately query hash tables using PIR



A strawman protocol

Step 1: Client uses LSH functions to find the hashkey

Step 2: Privately query hash tables using PIR



A strawman protocol

Step 1: Client uses LSH functions to find the hashkey
Step 2: Privately query hash tables using PIR
Step 3: Client recovers the result

Query: (2,5)

Candidate set: { **ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5) }

(union over all tables)

A strawman protocol

Step 1: Client uses LSH functions to find the hashkey
Step 2: Privately query hash tables using PIR
Step 3: Client recovers the result

Query: (2,5)

False positives

Candidate set: { **ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5) }

(union over all tables)

A strawman protocol

Step 1: Client uses LSH functions to find the hashkey
Step 2: Privately query hash tables using PIR
Step 3: Client recovers the result

Query: (2,5)

Candidate set: { **ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5) }

(union over all tables)

***Near* Neighbors**

A strawman protocol

Step 1: Client uses LSH functions to find the hashkey
Step 2: Privately query hash tables using PIR
Step 3: Client recovers the result

Query: (2,5)

Candidate set: { **ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5) }

(union over all tables)

Nearest **Neighbor**

Challenge: Database privacy

Candidate set: {**ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5)}

Challenge: Database privacy

Candidate set: {**ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5)}

⚠️ **The candidate set leaks a lot about the database to the client!**

Challenge: Database privacy

Candidate set: {**ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5)}

⚠️ **The candidate set leaks a lot about the database to the client!**

The client learns:

- *All* near neighbors and their feature vectors.

Challenge: Database privacy

Candidate set: {**ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5)}

⚠️ **The candidate set leaks a lot about the database to the client!**

The client learns:

- *All* near neighbors and their feature vectors.
- Other feature vectors in the database (false positives).

Challenge: Database privacy

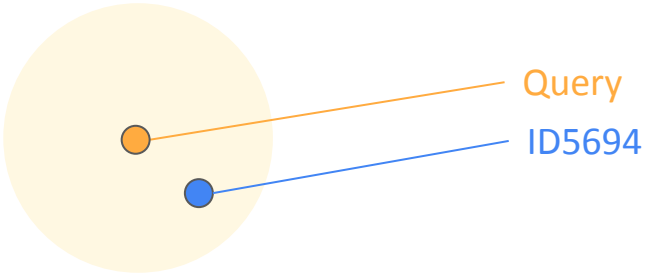
Candidate set: {**ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5)}

⚠ **The candidate set leaks a lot about the database to the client!**

The client learns:

- *All* near neighbors and their feature vectors.
- Other feature vectors in the database (false positives).

Baseline leakage



Challenge: Database privacy

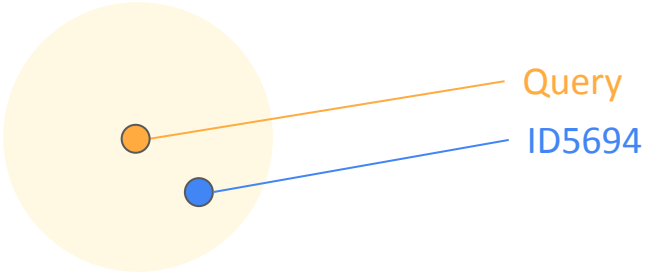
Candidate set: {**ID2:** (1,4), **ID5694:** (2,4), **ID900:** (3,4), **ID101:** (6,5)}

⚠ **The candidate set leaks a lot about the database to the client!**

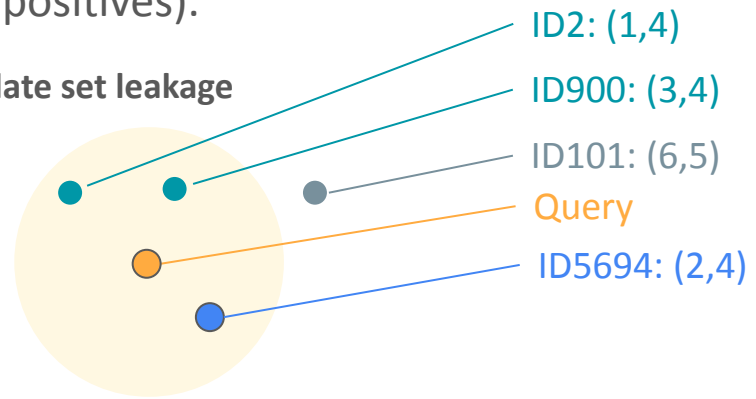
The client learns:

- *All* near neighbors and their feature vectors.
- Other feature vectors in the database (false positives).

Baseline leakage



Candidate set leakage



Database Privacy with

Radix sorting

(hide all feature vectors by pruning without comparisons)

Pruning without comparisons

Main idea: use radix sorting for comparison-free sorting

Pruning without comparisons

Main idea: use radix sorting for comparison-free sorting

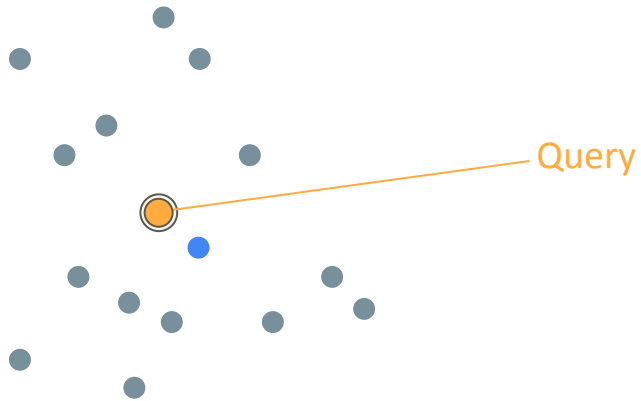
(i.e., let the hash function do the work of sorting by distance)

Pruning without comparisons

Main idea: use radix sorting for comparison-free sorting

(i.e., let the hash function do the work of sorting by distance)

Table 1: radius = 0.1



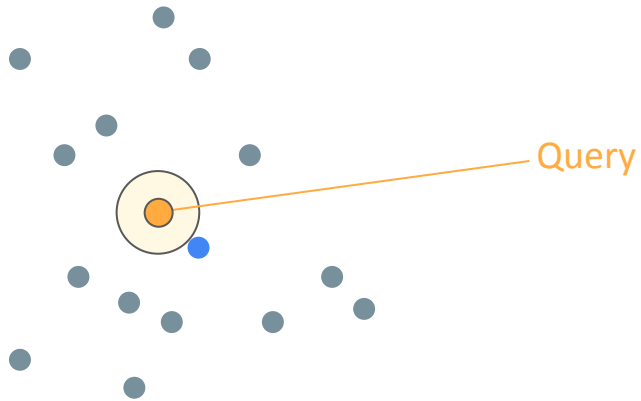
Pruning without comparisons

Main idea: use radix sorting for comparison-free sorting

(i.e., let the hash function do the work of sorting by distance)

Table 1: radius = 0.1

Table 2: radius = 0.2



Pruning without comparisons

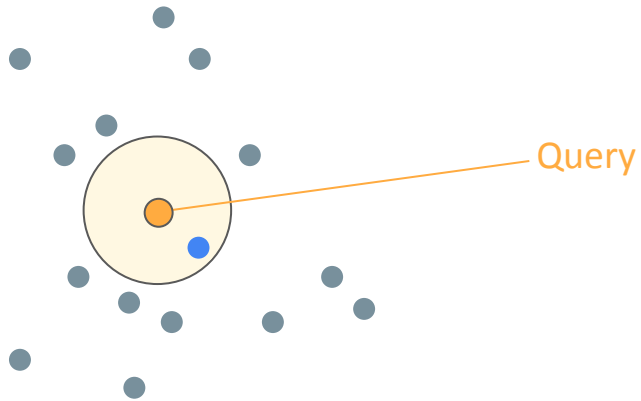
Main idea: use radix sorting for comparison-free sorting

(i.e., let the hash function do the work of sorting by distance)

Table 1: radius = 0.1

Table 2: radius = 0.2

Table 3: radius = 0.3



Pruning without comparisons

Main idea: use radix sorting for comparison-free sorting

(i.e., let the hash function do the work of sorting by distance)

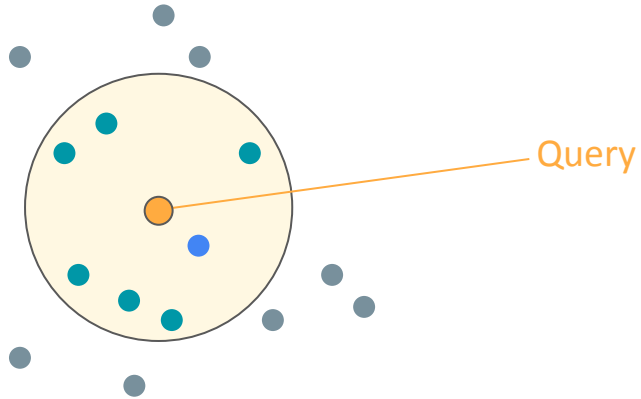
Table 1: radius = 0.1

Table 2: radius = 0.2

Table 3: radius = 0.3

…

Table 20: radius = 2.0



Pruning without comparisons

Candidates are now sorted by distance from the query!

Candidate Set = {0, 0, 0, ID5694, ID2, ID900, 0, ID101}

Pruning without comparisons

Candidates are now sorted by distance from the query!

Candidate Set = {0, 0, 0, ID5694, ID2, ID900, 0, ID101}

Nearest Neighbor

Pruning without comparisons

Candidates are now sorted by distance from the query!

Candidate Set = {0, 0, 0, ID5694, ID2, ID900, 0, ID101}

Nearest Neighbor

We no longer need to include the feature vectors (only the IDs)!

Pruning without comparisons

Candidates are now sorted by distance from the query!

Candidate Set = {0, 0, 0, ID5694, ID2, ID900, 0, ID101}

Nearest Neighbor

We no longer need to include the feature vectors (only the IDs)!

⚠️ **Still leaks many IDs from the database to the client.**

More database privacy with

Oblivious masking

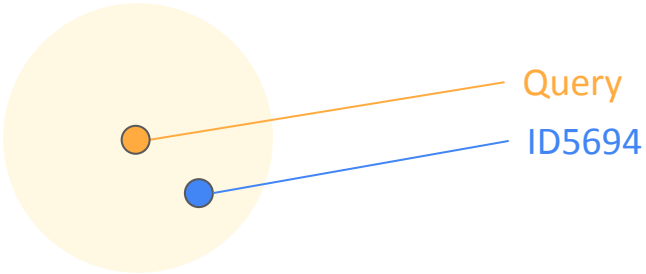
(hide all candidates except for the nearest neighbor)

Oblivious masking

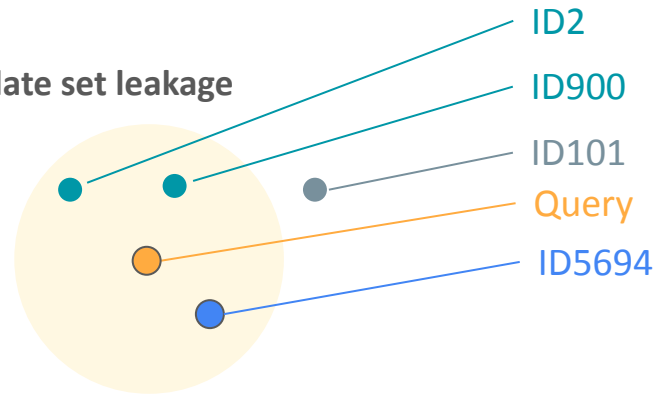
After processing the PIR queries, each server has a share of the candidate set:

$$[\text{Candidate Set}] = \{\ [0], [0], [0], [ID5694], [ID2], [0], [ID900], [ID101]\ \}$$

Baseline leakage



Candidate set leakage

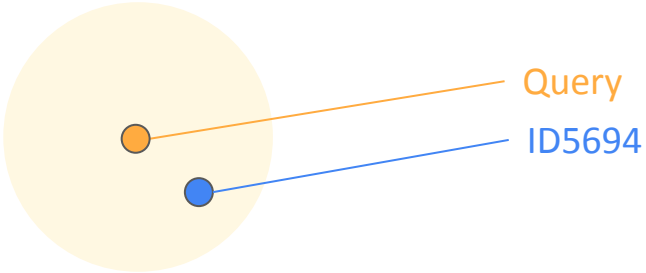


Oblivious masking

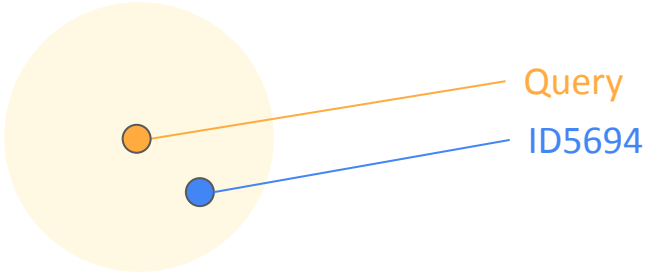
After processing the PIR queries, each server has a share of the candidate set:

[Candidate Set] = { [redacted] [ID5694], [redacted] }

Baseline leakage



Candidate set leakage



Oblivious masking

Observation: the nearest neighbor is always preceded by shares of 0s…

$$[\textbf{Candidate Set}] = \{ [0], [0], [0], [ID5694], [ID2], [0], [ID900], [ID101] \}$$

Oblivious masking

Observation: the nearest neighbor is always preceded by shares of 0s…

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

Algorithm 1: ObliviousMasking

Input: Secret-shared vector $[\boldsymbol{v}] \in \mathbb{F}_p^L$ and randomness rand.

Output: Secret-shared vector $[\boldsymbol{y}] = ([y_1], \ldots, [y_L]) \in \mathbb{F}_p^L$.

Procedure:

1: **for** $i \in \{1, \ldots, L\}$**:**

1.1: Sample $r_i \leftarrow$ rand.

1.2: Set $[\boldsymbol{y}_i] \leftarrow [\boldsymbol{v}_i] + r_i \cdot \left(\sum_{j=0}^{i-1} [\boldsymbol{v}_j] \right)$.

2: Output $\boldsymbol{y} \in \mathbb{F}_p^L$.

Oblivious masking

Step 1: Servers agree on common randomness ($r_1, r_2, \ldots, r_n$) e.g., with a short PRG seed.

$$[\textbf{Candidate Set}] = \{ [0], [0], [0], [ID5694], [ID2], [0], [ID900], [ID101] \}$$

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ \}$$

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ \quad \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add [SUM]$\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0] \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add [SUM]$\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0], [0] + [0]r_2 \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add $[\text{SUM}]\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0], [0], [0] + [0+0]r_3 \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add $[\text{SUM}]\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}] + [0+0+0]r_4 \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add $[\text{SUM}]\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}] + [0+0+0+\text{ID5694}]r_5 \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add $[\text{SUM}]\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}] + [(0+0+0+\text{ID5694})r_5] \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add $[\text{SUM}]\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\#\$@!\#*] \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add [SUM]$\cdot r_i$ to the new element in **[Candidate Set]**.

Oblivious masking

Step 1: Servers agree on common randomness (r_1, r_2, …, r_n) e.g., with a short PRG seed.

Step 2: Servers copy (and randomize) shares from **[Candidate Set]** to **[Candidate Set]**.

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\#\$@!\#*], [\$\%@!\&], \ldots \}$$

Rule: For each copied share, sum all preceding elements in **[Candidate Set]** and add [SUM]·r_i to the new element in **[Candidate Set]**.

Oblivious masking

All elements *after* the first non-zero element are **random**!

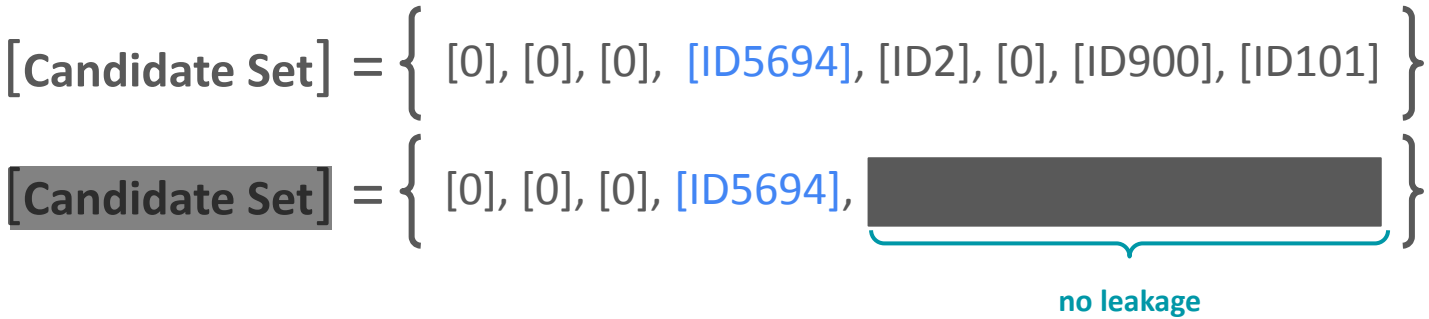
$$[\text{Candidate Set}] = \{\ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}]\ \}$$

$$[\text{Candidate Set}] = \{\ [0], [0], [0], [\text{ID5694}], [\#\$@!\#*], [\$\%@!\&], \ldots\ \}$$

Oblivious masking

All elements *after* the first non-zero element are **random**!

Result: The client learns nothing beyond the first non-zero element (i.e., the ANN)!



Oblivious masking

All elements *after* the first non-zero element are **random**!

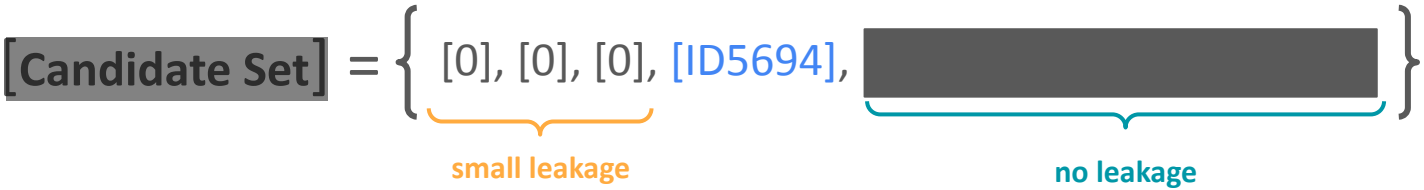
Result: The client learns almost nothing beyond the first non-zero element (i.e., the ANN)!

$$[\text{Candidate Set}] = \{ [0], [0], [0], [\text{ID5694}], [\text{ID2}], [0], [\text{ID900}], [\text{ID101}] \}$$

$$[\text{Candidate Set}] = \{ \underbrace{[0], [0], [0]}_{\text{small leakage}}, [\text{ID5694}], \underbrace{\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare}_{\text{no leakage}} \}$$

Oblivious masking

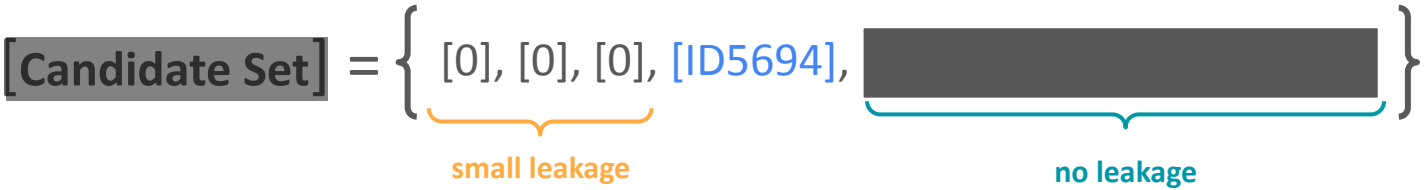
The radix bucket of the ANN leaks the approximate distance from the query.



Oblivious masking

The radix bucket of the ANN leaks the approximate distance from the query.

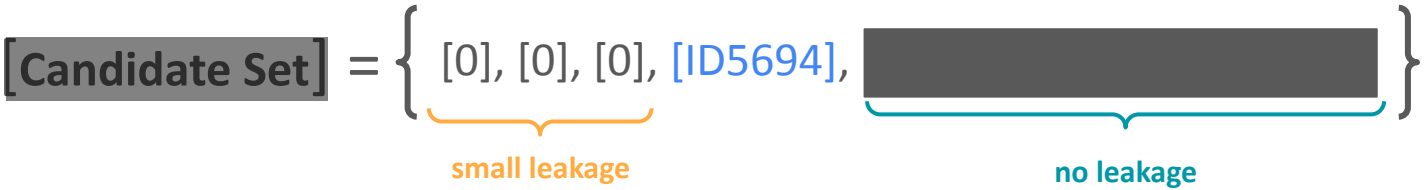
- **Asymptotically** optimal leakage!



Oblivious masking

The radix bucket of the ANN leaks the approximate distance from the query.

- **Asymptotically** optimal leakage!
- **In practice:** 2 to 15x more than baseline leakage.

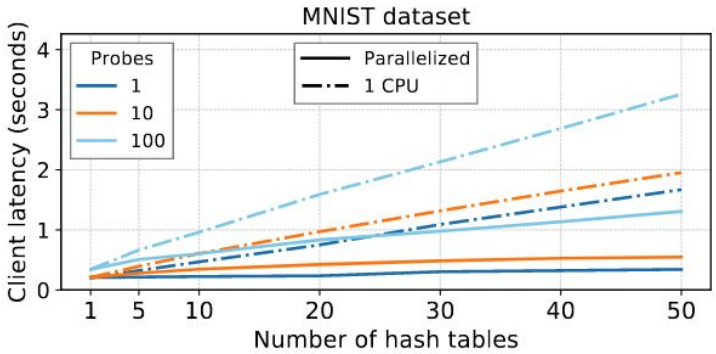


Evaluation

We evaluate on four real world datasets

- **Deep1B** (10 million items, 96 dimensions).
- **SIFT** (1 million items, 128 dimensions).
- **GIST** (1 million items, 916 dimensions).
- **MNIST** (60,000 items, 784 dimensions).

Efficiency on small datasets (MNIST; 60,000 items)

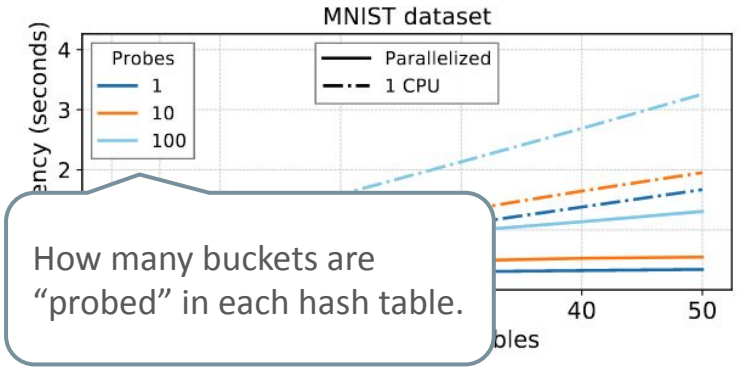


Server computation for > 95% accuracy (32 core servers)**:**

- **300 milliseconds per query.**

Communication: 800 KB between client and both servers.

Efficiency on small datasets (MNIST; 60,000 items)

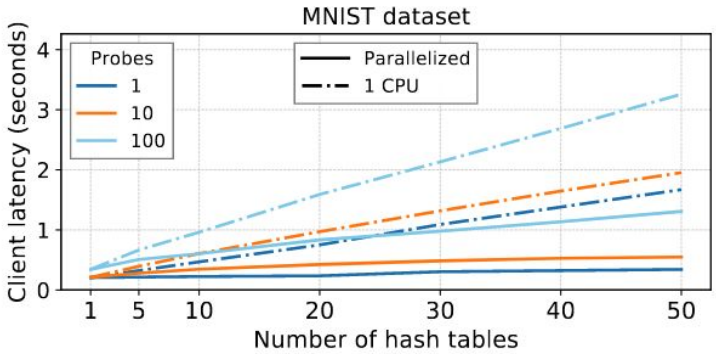


Server computation for > 95% accuracy (32 core servers)**:**

- **300 milliseconds per query.**

Communication: 800 KB between client and both servers.

Efficiency on small datasets (MNIST; 60,000 items)

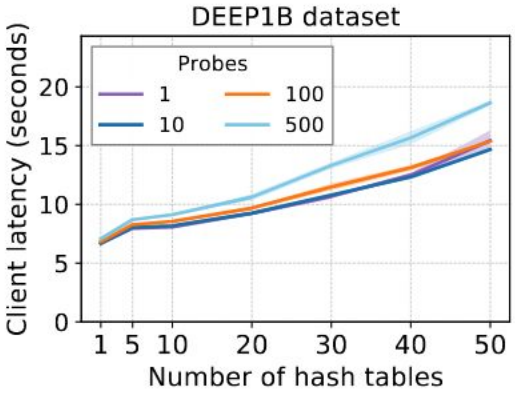


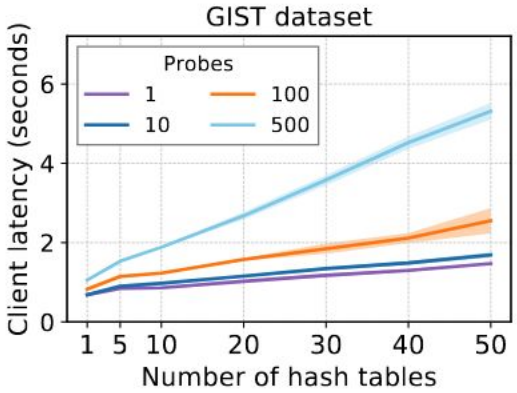
Server computation for > 95% accuracy (32 core servers)**:**

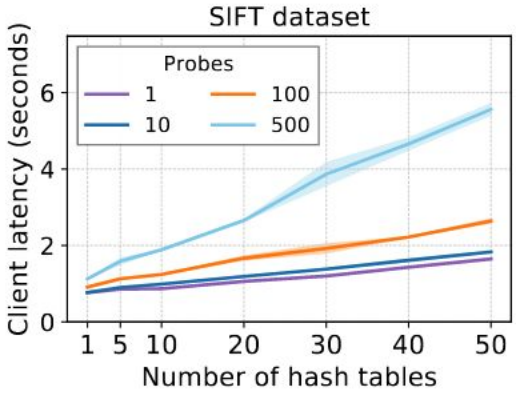
- **300 milliseconds per query.**

Communication: 800 KB between client and both servers.

Efficiency on large datasets (1M to 10M items)





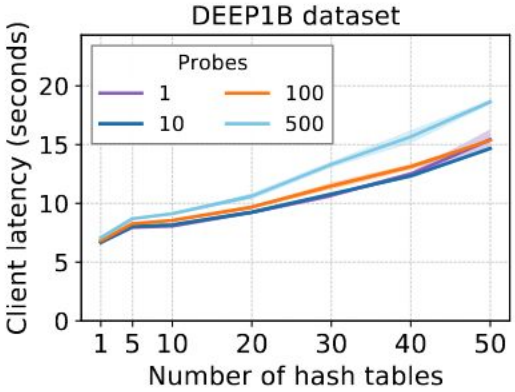


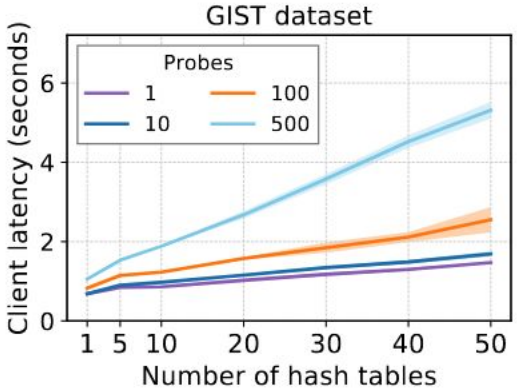
Server computation for > 95% accuracy (32 core servers)**:**

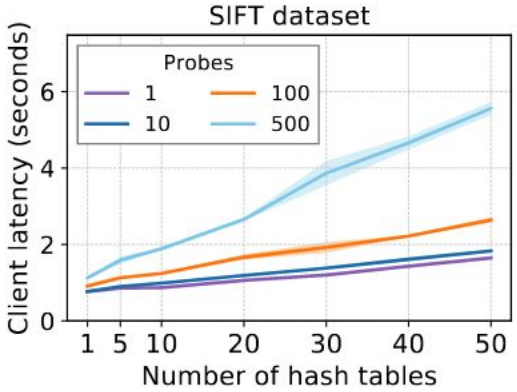
- **1.2 seconds per query** on 1M item datasets.
- **8 seconds per query** on 10M item datasets.

Communication: 1-2MB between client and both servers.

Efficiency on large datasets (1M to 10M items)



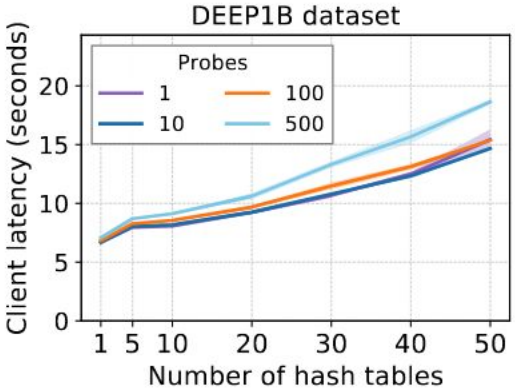


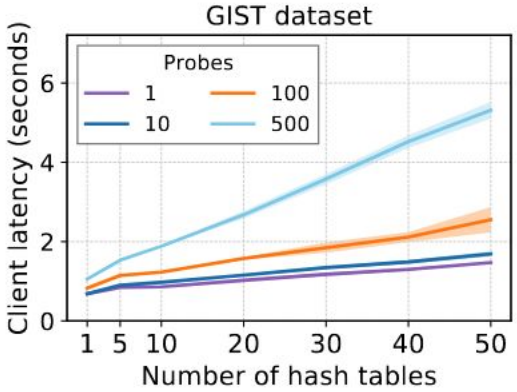


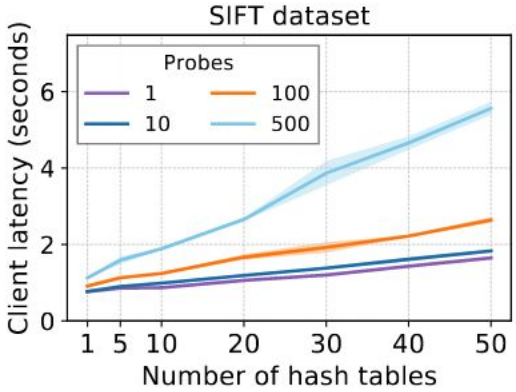
Server computation for > 95% accuracy (32 core servers)**:**

Five orders of magnitude less computation compared to FHE-based approaches.

Efficiency on large datasets (1M to 10M items)







1,000 to 3,000X less communication compared to two-party computation!

Communication: **1-2MB** between client and both servers.

Thank you!

Sacha Servan-Schreiber

with Simon Langowski and Srini Devadas

Full paper: ia.cr/2021/1157
Code: github.com/sachaservan/private-ann
Contact: 3s@mit.edu, slangows@mit.edu