Private Database Queries Using Somewhat Homomorphic Encryption

Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, David J. Wu

ACNS 2013

Fully Private Conjunctive Database Queries

user

SELECT * FROM db WHERE

dest = LAX AND age = 25

indices of matching records

database

Goals:

1. database learns nothing about query or response (not even # of matching records)
2. user learns nothing about non-matching records

Motivations

Law Enforcement

law enforcement officer

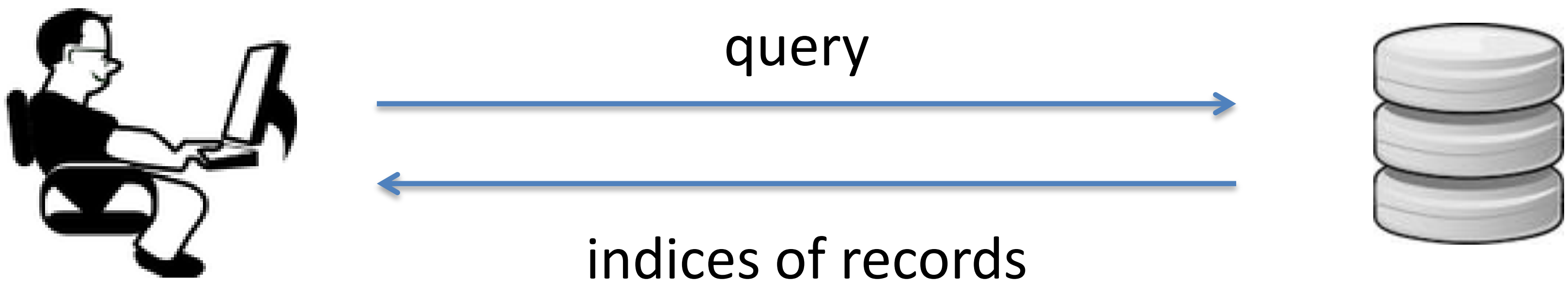
select records for Bob from the last six months

indices of records for Bob

local police department

- law enforcement officers should not learn information about other clients
- local police department should not learn who is currently under investigation

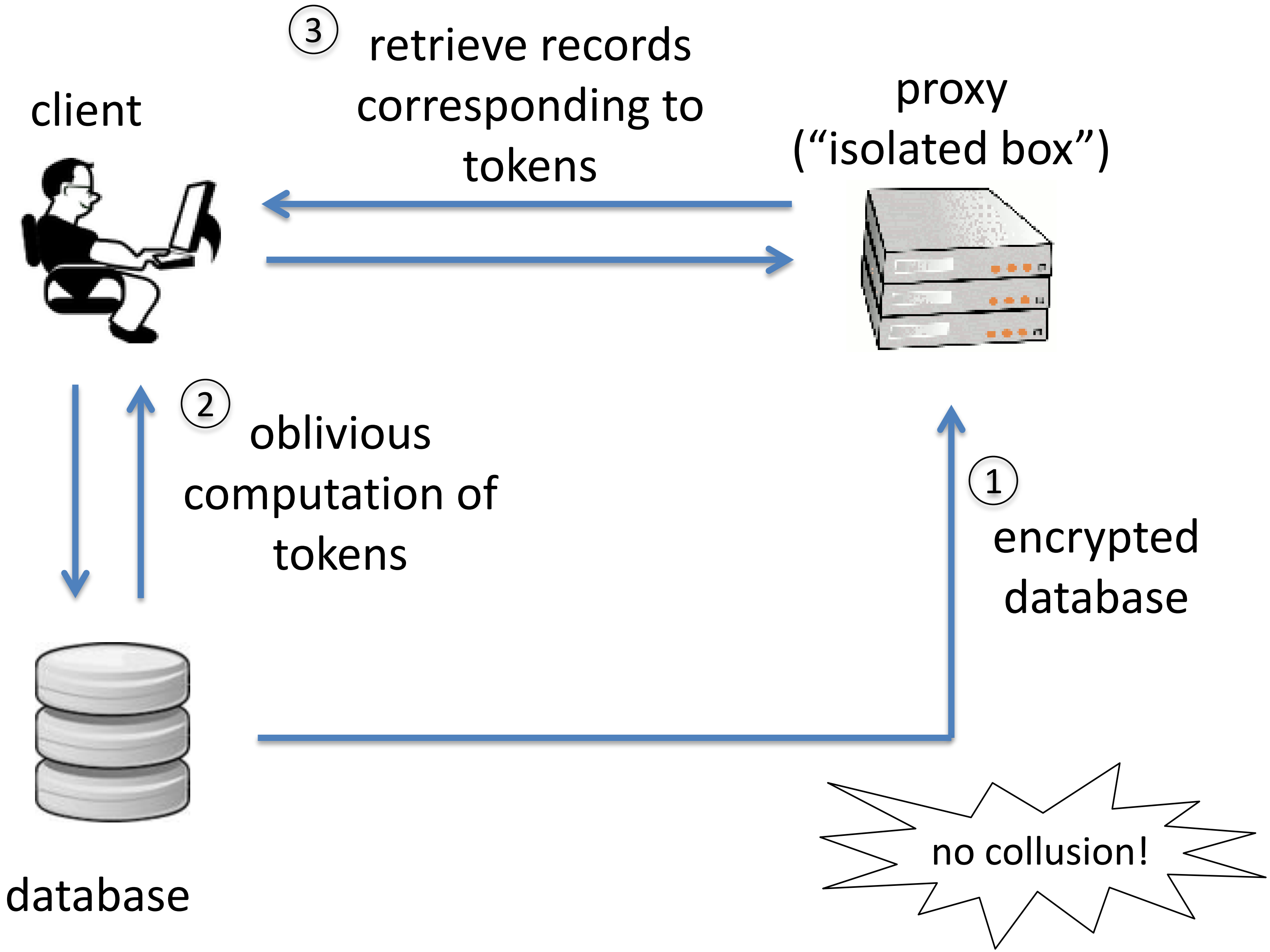
Limitations of the Two-Party Model



Computation Time: Linear in size of database

Otherwise, database learns something about query

3-Party Protocol (De Cristofaro et al.)



Related Work

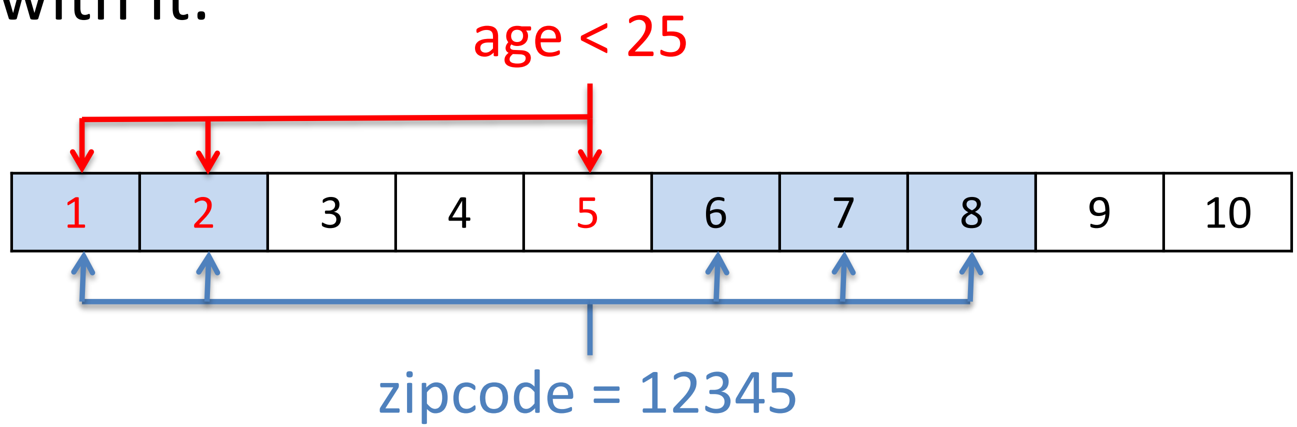
- Chor et al. (1998)
    - Private information retrieval (PIR) with sublinear communication complexity
    - *Not a private database query protocol*
- De Cristofaro et al. (2011)
    - 3-Party Protocol for fully private disjunctive queries
    - *Does not support conjunctive queries*
- Raykova et al. (2012)
    - Multi-party protocol using bloom filters and deterministic encryption to support private queries
    - *Query complexity linear in number of records*

Our contribution: <u>Efficient</u> support for fully private <u>conjunctive</u> queries

Representing the Database

For each attribute-value pair, there is a set of records associated with it:

Database:

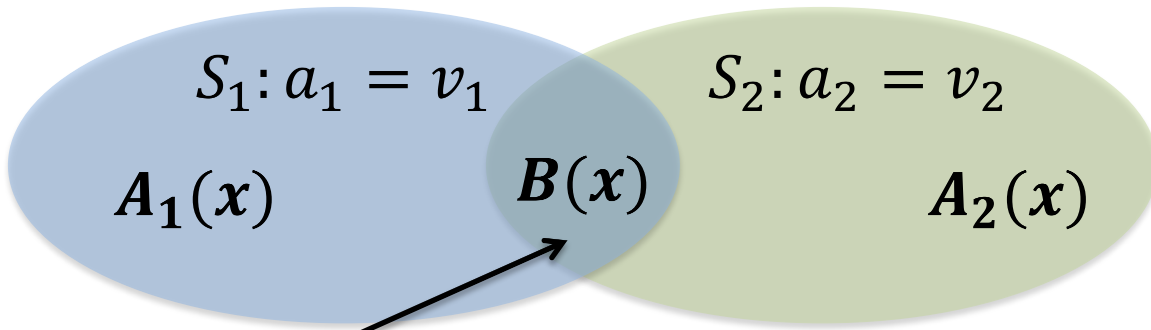


Represent each set as a **polynomial** with roots corresponding to matching records:

age < 25: $(x-1)(x-2)(x-5)$
zipcode = 12345: $(x-1)(x-2)(x-6)(x-7)(x-8)$

Conjunctive Queries

Query: SELECT * FROM db WHERE $a_1 = v_1$ and $a_2 = v_2$



Intersection

$A_1(x), A_2(x) \in \mathbb{F}_p[x]$

Kissner-Song Approach: Take $B \in \mathbb{F}_p[x]$ to be random linear combination of $A_1(x)$ and $A_2(x)$:

encoding of $\gcd(A_1, A_2)$

$$B(x) = A_1(x)R_1(x) + A_2(x)R_2(x)$$

for *random polynomials* $R_1(x), R_2(x) \in \mathbb{F}_p[x]$

Protocol Description: Setup

database

1. For each $a_i = v_i$ pair, construct tag $\mathrm{tg}_i = \mathrm{PRF}_s(a_i = v_i)$
2. Send $(\mathrm{tg}_i, \mathrm{Enc}(S_i))$

proxy

Each set S_i is a polynomial $A_i(x)$. We use a *somewhat homomorphic encryption scheme* (SWHE) to encrypt the *coefficients*.

Encrypting a Polynomial

$$\boxed{x^2} + \boxed{(-3)x} + \boxed{2}$$

$$\downarrow \qquad\qquad \downarrow \qquad\qquad \downarrow$$

$\mathrm{Enc}(1)$	$\mathrm{Enc}(-3)$	$\mathrm{Enc}(2)$

Polynomial addition: Additive homomorphism

Multiplying by plaintext polynomial: Possible if SWHE supports scalar multiplication

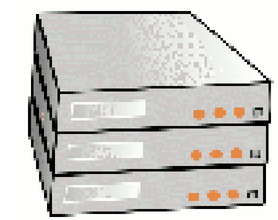
Protocol Description: Query

client

② $t_1, \dots, t_n$

$B(x)$

proxy



① *oblivious* PRF evaluation

$$\begin{aligned} t_1 &= \mathrm{PRF}_s(a_1 = v_1) \\ &\vdots \\ t_n &= \mathrm{PRF}_s(a_n = v_n) \end{aligned}$$

1. Gets $A_1(x), \dots, A_n(x)$ corresponding to tags
2. Compute $B(x) = \sum_i A_i R_i$ for random $R_1, \dots, R_n$

additive homomorphism

database

Query: SELECT * FROM db WHERE $a_1 = v_1$ AND $\cdots$ AND $a_n = v_n$

Protocol Description: Query

client

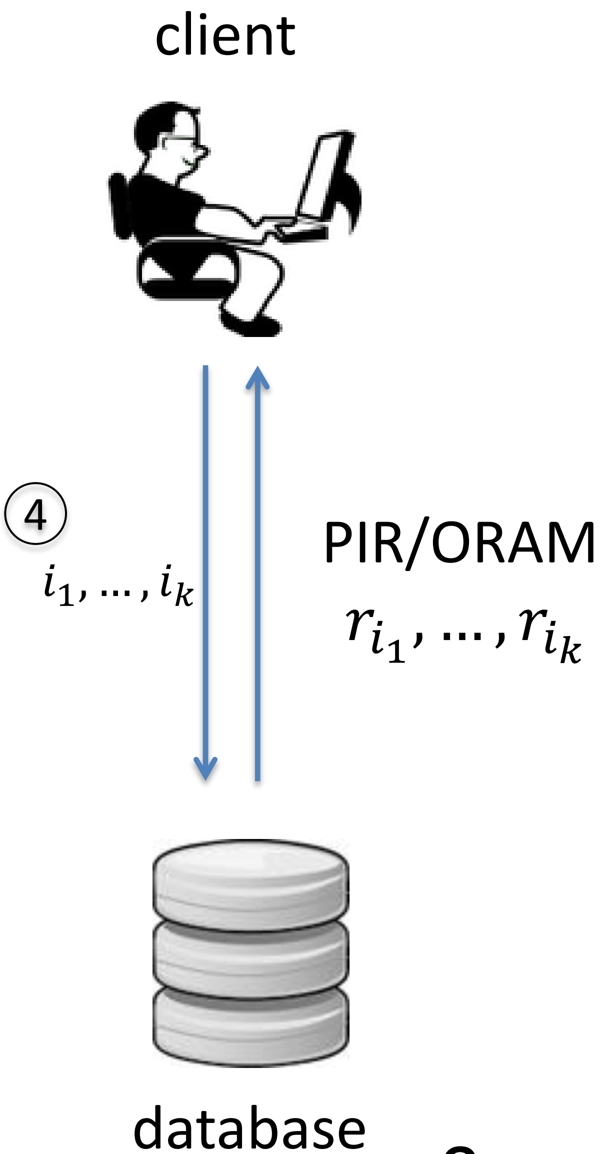
Factors polynomial to obtain roots (record indices) $i_1, \dots, i_k$

③ *oblivious* decryption of $B(x)$

database

Query: SELECT * FROM db WHERE $a_1 = v_1$ AND $\cdots$ AND $a_n = v_n$

Protocol Description: Query



Query: SELECT * FROM db WHERE $a_1 = v_1$ AND $\cdots$ AND $a_n = v_n$

Conserving Bandwidth

Recall computation performed by proxy:

proxy

$$t_1 \rightarrow A_1(x)$$
$$t_2 \rightarrow A_2(x)$$
$$\vdots$$
$$t_n \rightarrow A_n(x)$$

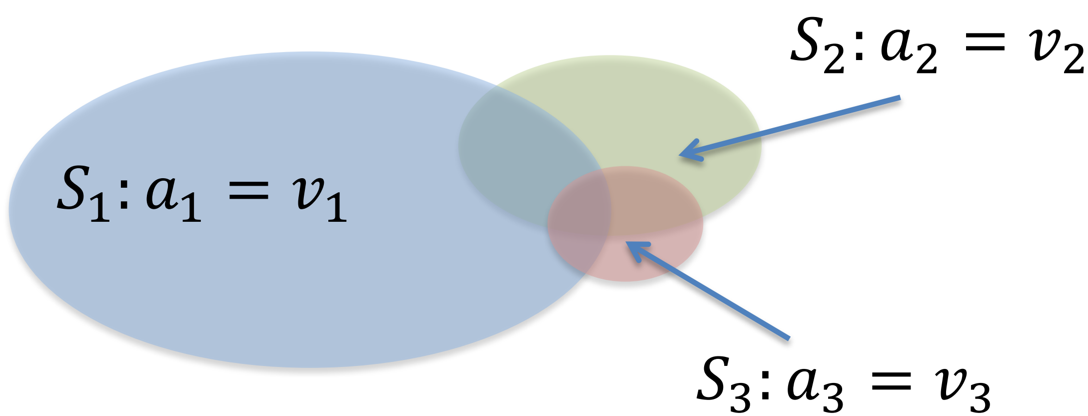
$$B(x) = \sum_{i=1}^{n} A_i(x) R_i(x)$$

$$\deg A_i(x) = |S_i| \qquad \deg B(x) \approx 2 \cdot \max_i \deg A_i(x)$$

Question: Can we do better?

Conserving Bandwidth

Unbalanced Query: large disparity between size of smallest set and size of largest set



Example:

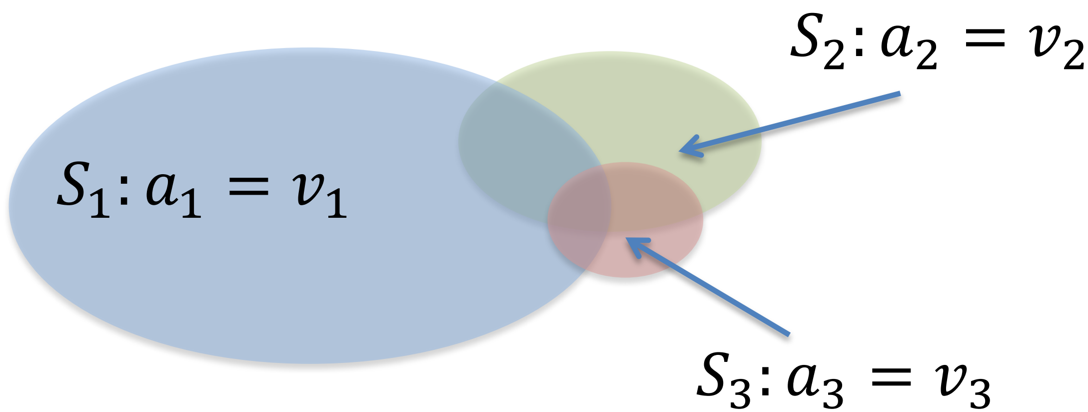
≈ 2,000,000 records

SELECT * FROM db WHERE location = "New York" AND name = "John Smith"

≈ 200 records

Conserving Bandwidth

Unbalanced Query: large disparity between size of smallest set and size of largest set



Desiderata: Bandwidth proportional to size of *smallest* set:

$$\min_i \deg A_i(x) \text{ rather than } \max_i \deg A_i(x)$$

Conserving Bandwidth

Easy to get $\min_i \deg A_i(x) + \max_i \deg A_i(x)$:

Suppose $A_1(x)$ has lowest degree. Construct *random* linear combination of the rest:

$$A'(x) = \sum_{i=2}^{n} \rho_i A_i(x)$$

and ρ_i are random *scalars.*

Then, proxy computes and sends

$$B(x) = A_1(x)\underset{\substack{\uparrow \\ \deg A'(x)}}{R_1(x)} + A'(x)\underset{\substack{\uparrow \\ \deg A_1(x)}}{R'(x)}$$

no extra homomorphism

$$\deg B(x) = \max_i \deg A_i(x) + \min_i \deg A_i(x)$$

Modular Reduction

Recall: intersection of $A_1(x), \dots, A_n(x)$ is given by

$$G = \gcd(A_1(x), \dots, A_n(x)).$$

Suppose $A_1(x)$ has smallest degree.

First step of Euclidean algorithm: reduce modulo $A_1(x)$:

$$G = \gcd\left(A_1(x), A_2(x) \left(\operatorname{mod} A_1(x)\right) \dots, A_n(x) \left(\operatorname{mod} A_1(x)\right)\right).$$

Modular Reduction

Instead of computing

$$A'(x) = \sum_{i=2}^{n} \rho_i A_i(x)\,,$$

compute

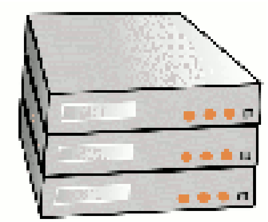
$$A''(x) = \sum_{i=2}^{n} \rho_i A_i(x) \pmod{A_1(x)}$$

$$\boxed{\deg(A''(x)) = \deg(A_1(x)) - 1}$$

Can be done with quadratic homomorphism. See paper.

Modular Reduction

proxy



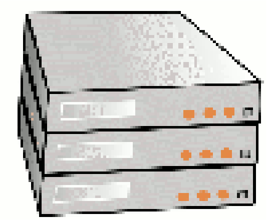
$$A'(x) = \sum_{i=2}^{n} \rho_i A_i(x)$$

$$B(x) = A_1(x)R_1(x) + A'(x)R'(x)$$

$$\deg\big(B(x)\big) = \min_i \deg A_i(x) + \max_i \deg A_i(x)$$

client

proxy



$$A''(x) = \sum_{i=2}^{n} \rho_i A_i(x) \pmod{A_1(x)}$$

$$B(x) = A_1(x)R_1(x) + A''(x)R''(x)$$

$$\deg\big(B(x)\big) = 2 \cdot \min_i \deg A_i(x) - 1$$

client

Big win if $\max_i \deg A_i(x) \gg \min_i \deg(A_i(x))$

Further Speedup via Batching

Recent fully homomorphic encryption schemes allow “batching” (encrypt + process array of values at no extra cost):

1	2	3	4

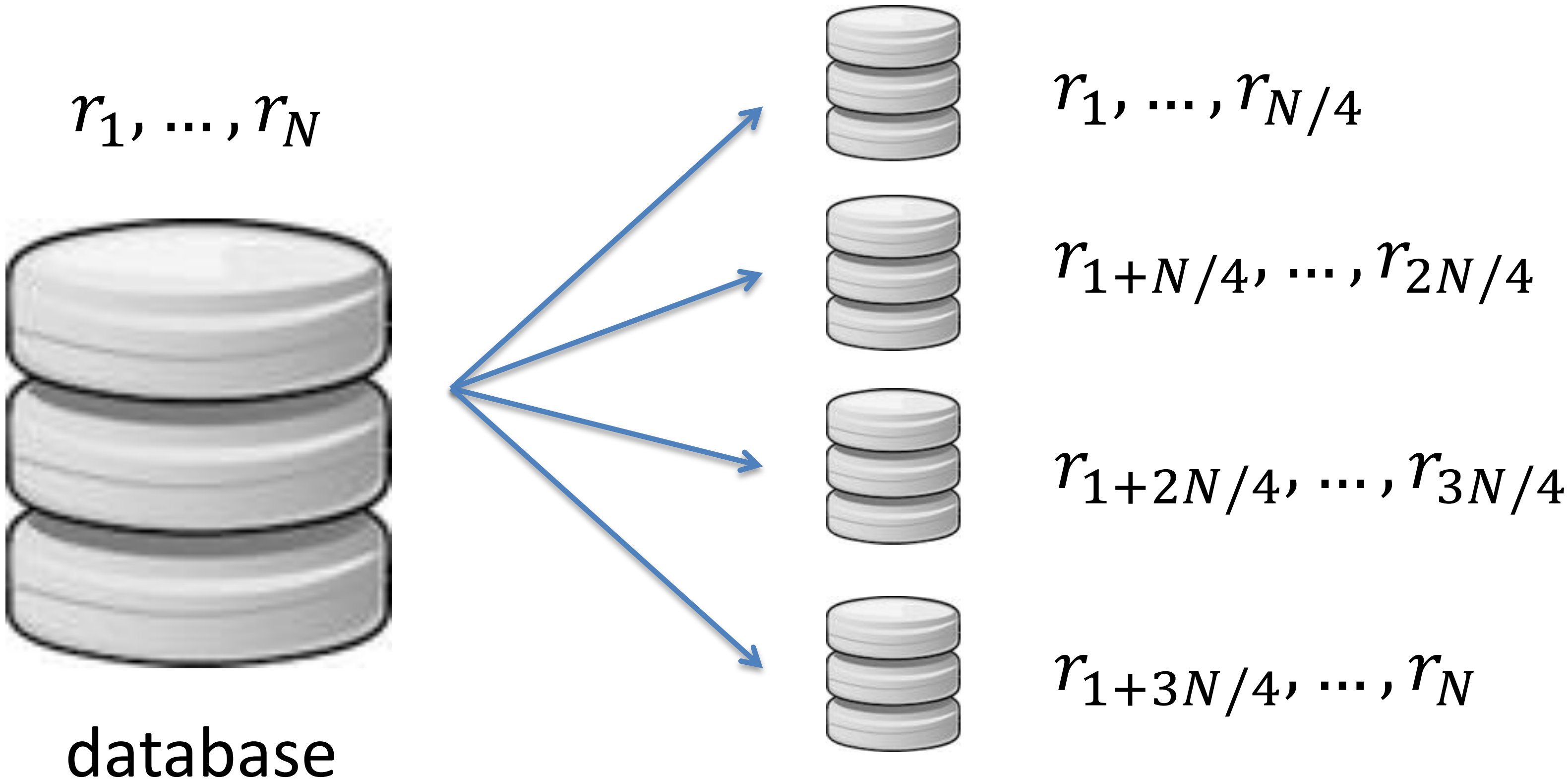
+

7	5	3	1

8	7	6	5

Further Speedup via Batching

Split database into many smaller databases and run query against all databases *in parallel*:



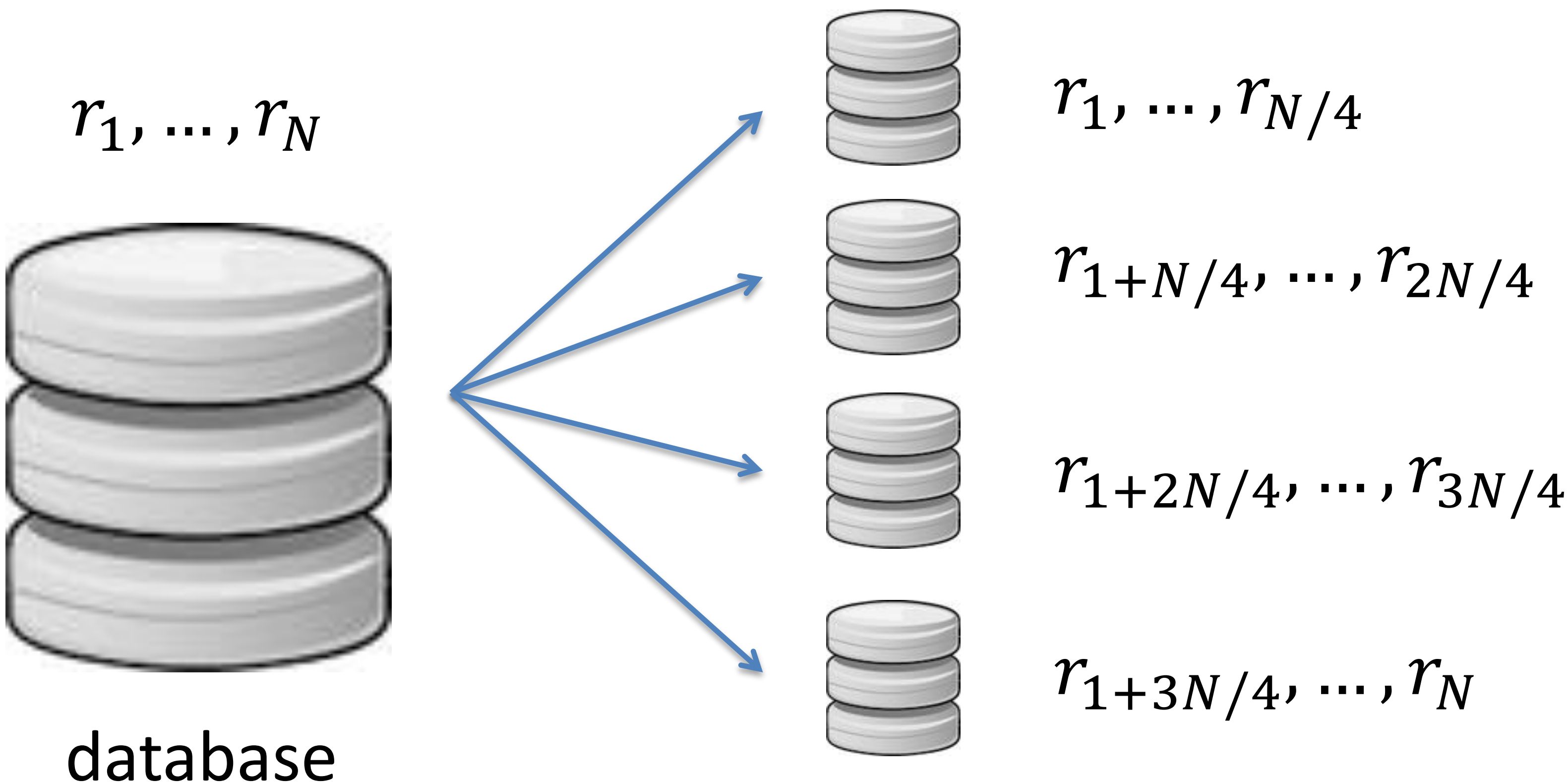
In practice, arrays have length 5000+, so split into **5000+** databases

Further Speedup via Batching

Runtime depends on size of small "database":

Faster computation, reduced bandwidth

Crucial for scalability



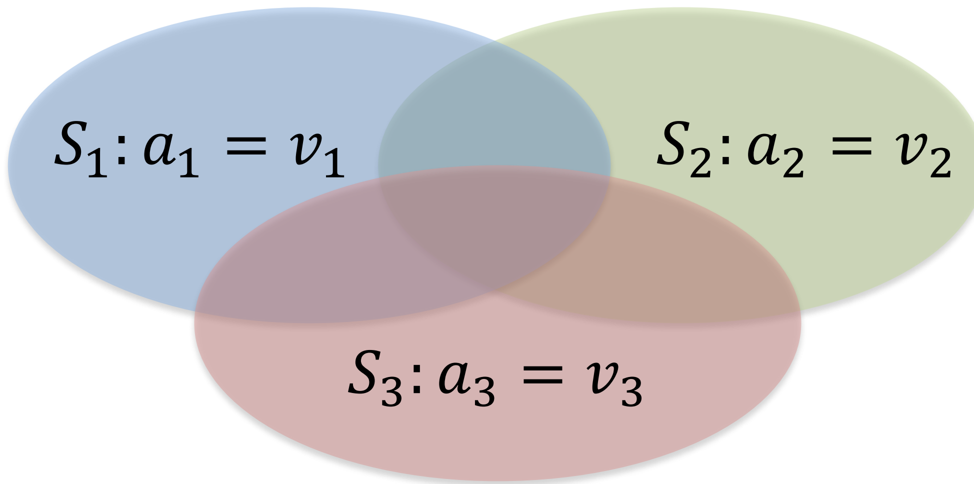
Implementations

Basic scheme (only requiring additive homomorphism) → Paillier cryptosystem

Modular reduction, batching (additive + multiplicative homomorphism) → Brakerski cryptosystem

Performance Characteristics

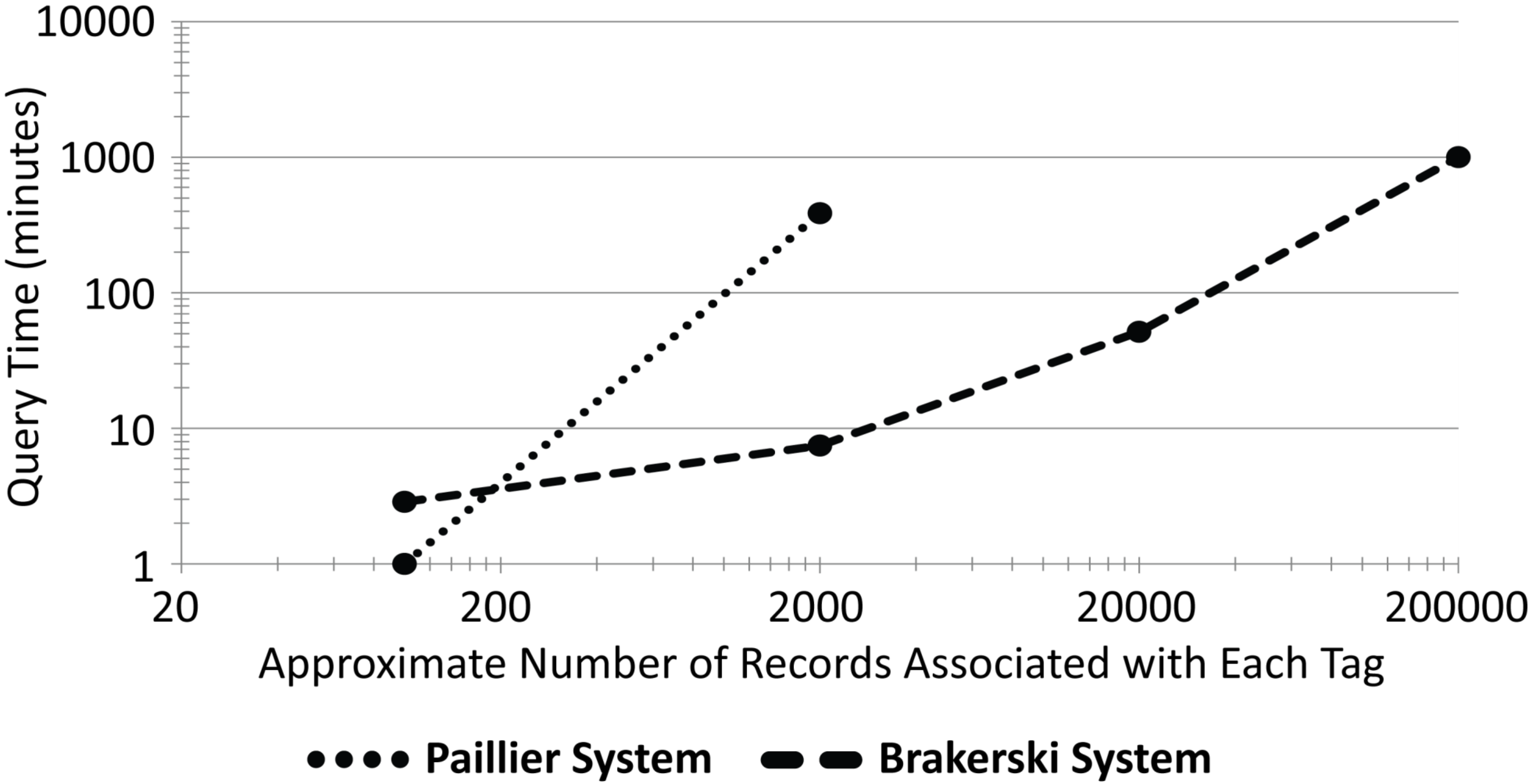
Balanced Query: number of records in each tag approximately equal



Experimental setup:

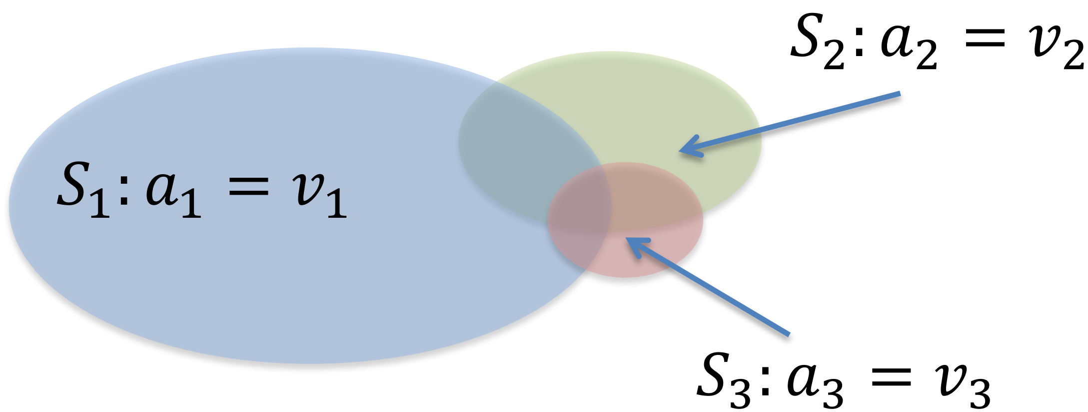
- Database of 1,000,000 records
- Queries consist of *five* tags
- Focus on time to perform set-intersection

Performance Characteristics



Performance Characteristics

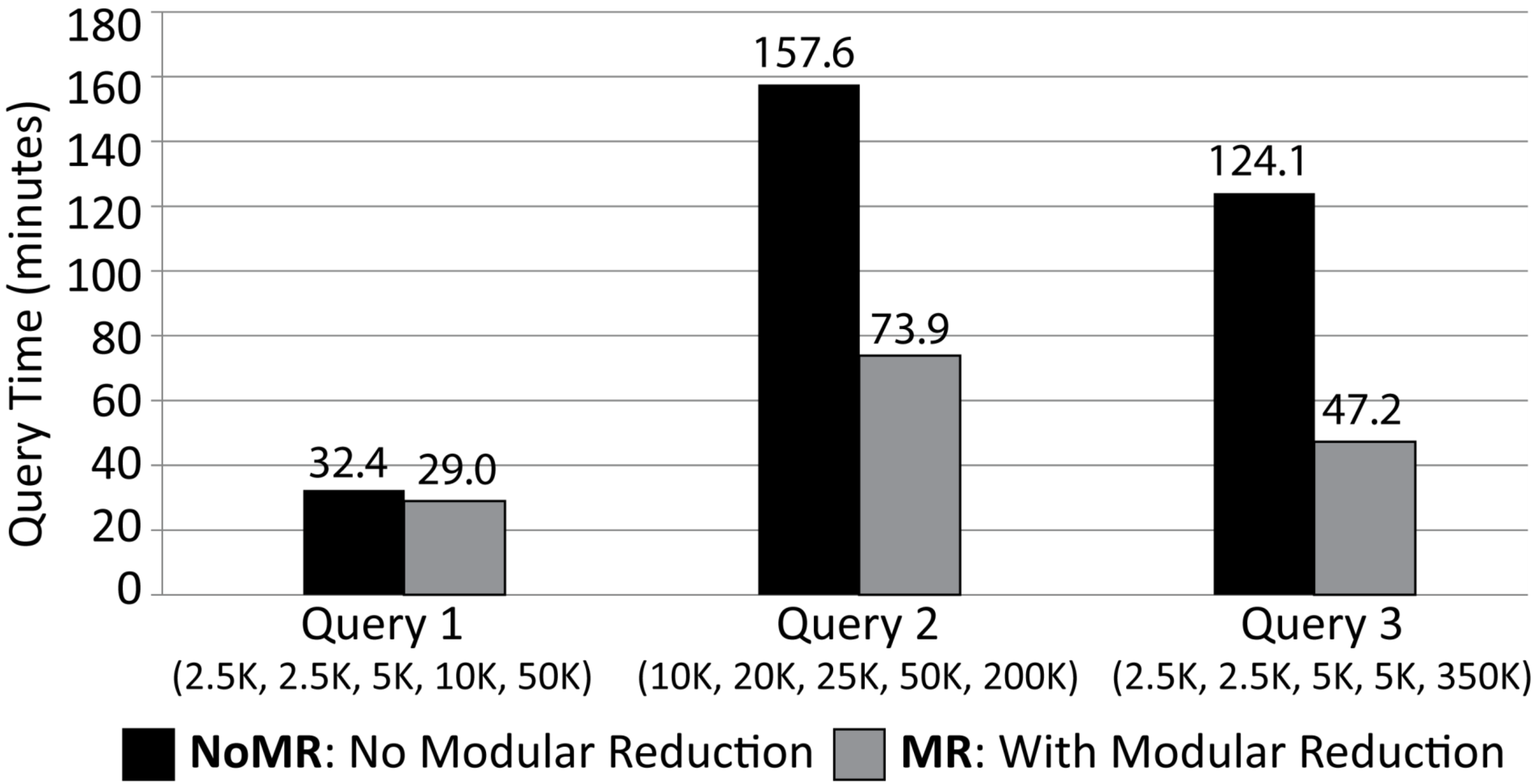
Unbalanced Query: large disparity between size of smallest set and size of largest set



Experimental setup:

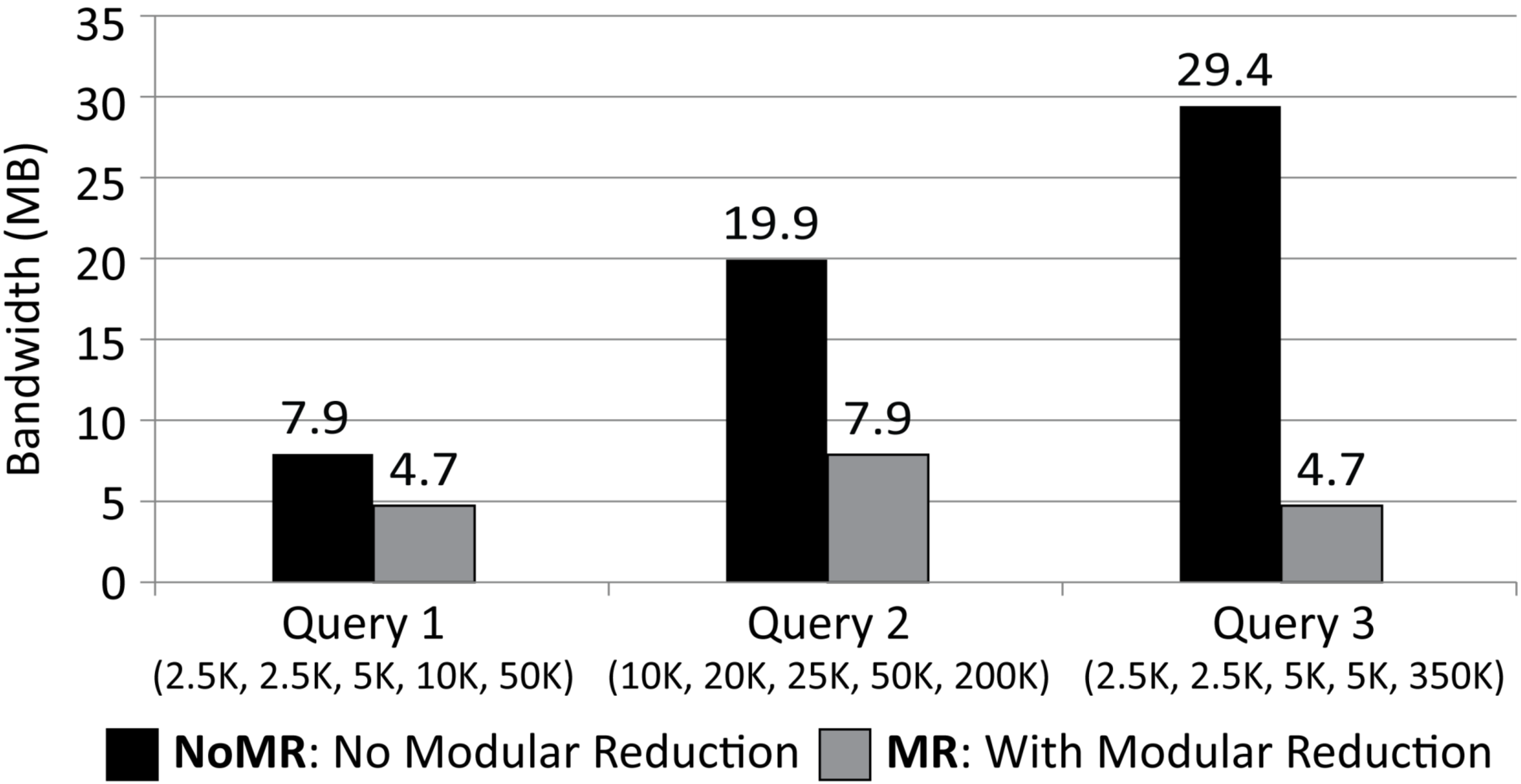
- Database of 1,000,000 records
- Intersection of *five* sets
- Size of smallest set at most 5% size of largest set

Performance Characteristics



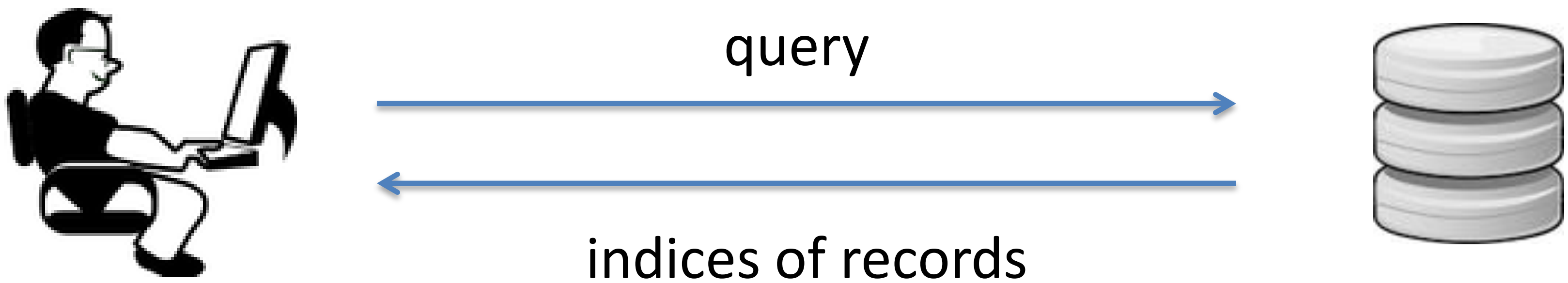
Intersection of five sets of varying size

Performance Characteristics



Intersection of five sets of varying size

Conclusion



- Fully private database query system for conjunction queries
- Query support via polynomial encoding of database, can be implemented via SWHE
- Modular reduction + batching optimizations crucial for scalability and performance (reduction in time *and* space for certain queries)

Thank you!