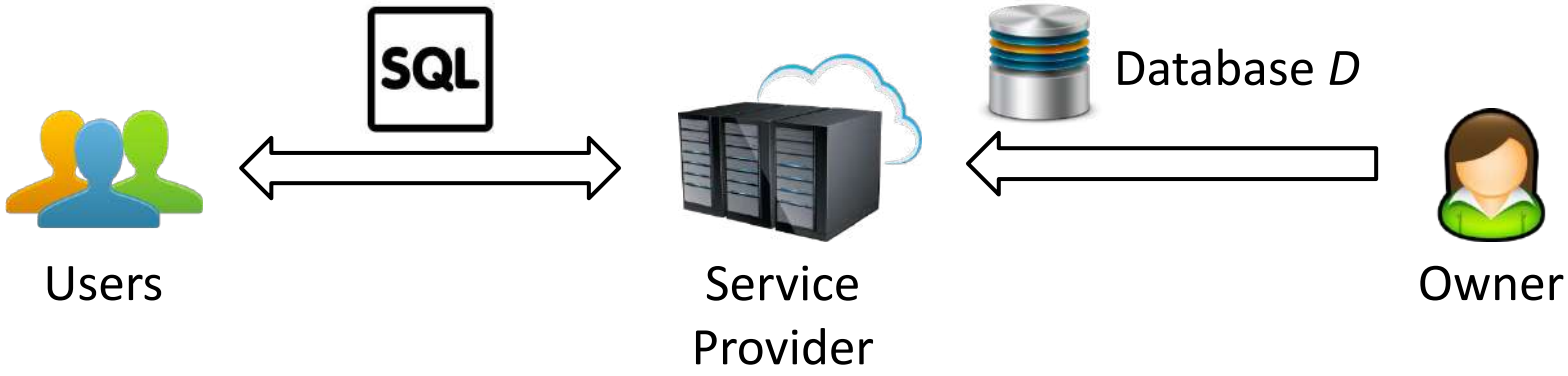
CS5322 Database Security

Encrypted Database: Motivation



- A data owner outsources her database to a service provider
- But she is concerned that the service provider may peek into her data (which could contain sensitive information)

Encrypted Database: Motivation

THE WALL STREET JOURNAL.

TECH

Tech's 'Dirty Secret': The App Developers Sifting Through Your Gmail

Software developers scan hundreds of millions of emails of users who sign up for email-based services

Encrypted Database: Motivation

THE ECONOMIC TIMES | tech

English Edition | E-Paper

Subscribe

Subs

Home ETPrime Markets News Industry RISE Politics Wealth MF Tech Jobs Opinion NRI Panache ET NOW More

ITES Tech & Internet Funding Startups Tech Bytes Newsletters Blogs & Opinion

Business News › Tech › Tech & Internet › How Facebook undermines privacy protections for its 2 billion WhatsApp users

How Facebook undermines privacy protections for its 2 billion WhatsApp users

By Peter Elkind, Jack Gillum & Craig Silverman | ProPublica, Last Updated: Sep 12, 2021, 02:17 PM IST

SHARE FONT SIZE SAVE PRINT

Synopsis

WhatsApp assures users that no one can see their messages, but the Facebook-owned messaging service—the world's largest—has an extensive monitoring operation and regularly shares personal information with prosecutors.

Ads by Google

Stop seeing this ad

Encrypted Database: Motivation

BUSINESS

Goldman Sachs employees concerned Bloomberg news reporters are using terminals to snoop

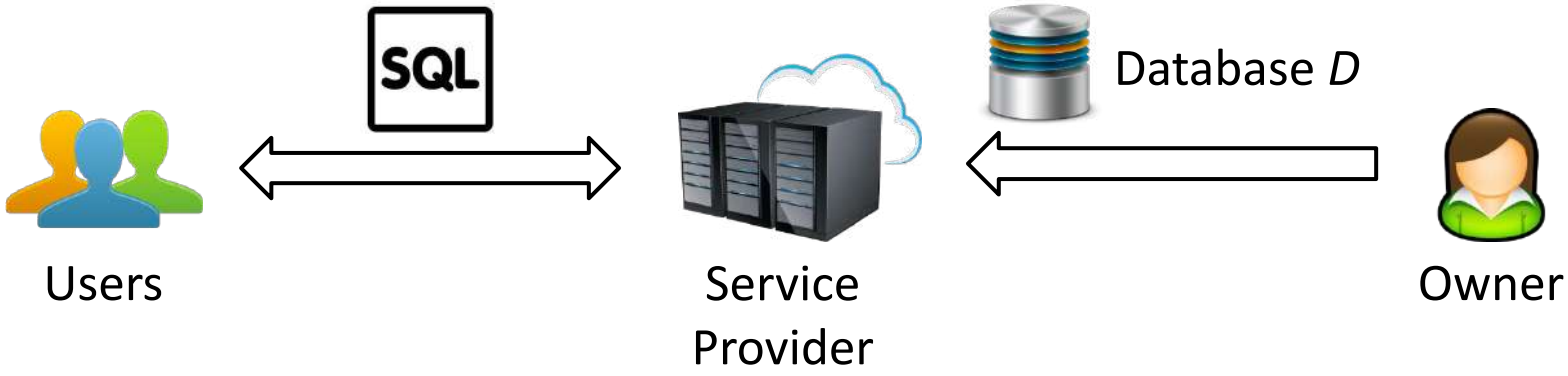
By Mark DeCambre

May 10, 2013 | 4:00am

In one instance, a Bloomberg reporter asked a Goldman executive if a partner at the bank had recently left the firm — noting casually that he hadn't logged into his Bloomberg terminal in some time, sources added.

Goldman later learned that Bloomberg staffers could determine not only which of its employees had logged into Bloomberg's proprietary terminals but also how many times they had used particular functions, insiders said.

Encrypted Database: Motivation



- Idea:
  - Let the data owner encrypt the database before sending it to the service provider
- Challenge:
  - How does the service provider answer queries without seeing the plaintext?

“Silver Bullets”

- Homomorphic encryption
- Hardware-assisted security

Homomorphic Encryption

- Encryption schemes that allow computation on encrypted data
- Basic idea:
  - The data owner encrypts each tuple *t* into E(*t*), and pass all tuples to the service provider
  - The user asks the service provider to perform a computation task on the tuples
  - The service provider looks at E(t1), E(t2), …, and derives an encrypted answer E(a), and then sends it to the user
  - The user decrypts E(a) to get the answer
- Key issue:
  - E(a) is derived from E(t1), E(t2), …, without letting the server know the encryption key or the plaintext of t1, t2, …

Homomorphic Encryption: Example

- Example: RSA (Rivest–Shamir–Adleman)
  - Public key: n *and* e; Private key: d
    - Such that for any integer m, we have $(m^e)^d \bmod n = m \bmod n$
  - Encryption:
    - $E(m) = m^e \bmod n$
  - Decryption:
    - $c = E(m)$
    - $D(c) = c^d \bmod n$
- Consider two integers m_1 and m_2
  - $E(m_1) \times E(m_2) = m_1^e \times m_2^e$
  - $= (m_1 \times m_2)^e = E(m_1 \times m_2)$
- In other words, multiplying the encrypted value gives an encrypted version of the multiplication result
- Therefore, RSA is multiplicative homomorphic

Homomorphic Encryption: Example

- There also exists encryption schemes that are additive homomorphic, i.e.,
  - $E(m_1) \square E(m_2) = E(m_1 + m_2)$
- But supporting only multiplication or addition is insufficient
- We need something that is *fully* homomorphic, i.e., supports both multiplication and addition

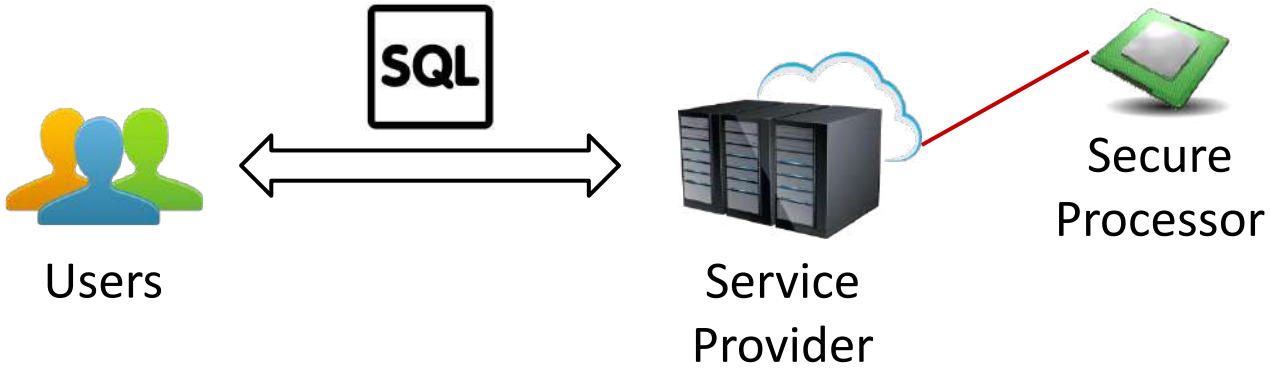
Fully Homomorphic Encryption

- First proposed by Craig Gentry in 2009
- Has been significantly improved in a series of follow-up work
- But still extremely slow and space-consuming
- Need some major breakthrough before it can be practical

“Silver Bullets”

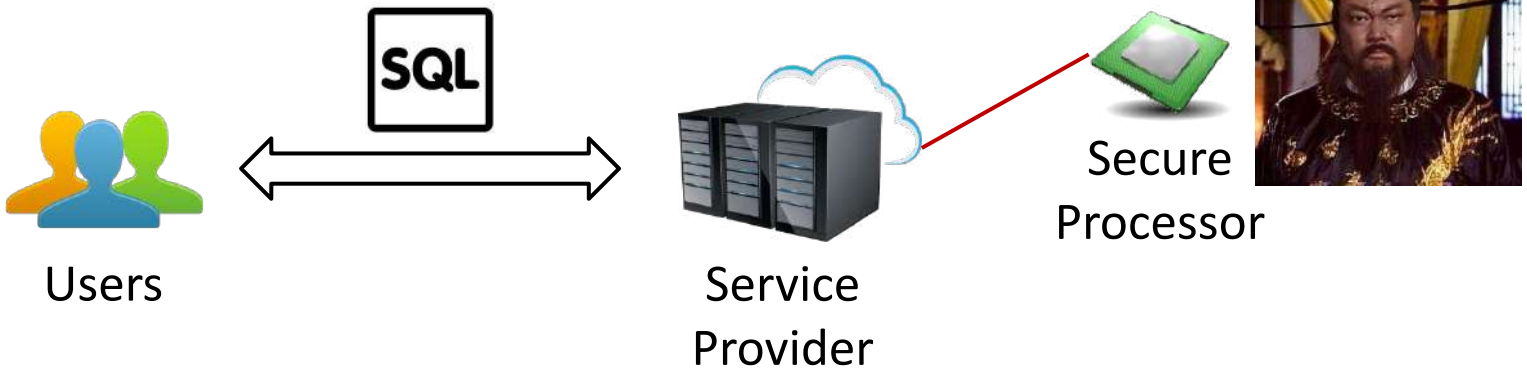
- Homomorphic encryption
  - Partial homomorphic encryption (e.g., RSA) is doable but only supports one type of operation
  - Fully homomorphic encryption is still far from being practical
- Hardware-assisted security

Hardware-Assisted Security



- Idea:
  - Install some *secure processors* on the service provider's machines
  - Whenever the user wants to compute something, she talks to the secure processors
  - The secure processors
    - Get encrypted data from the machines,
    - Decrypt them and compute the result
    - Send an encrypted version of the result back to user

Hardware-Assisted Security

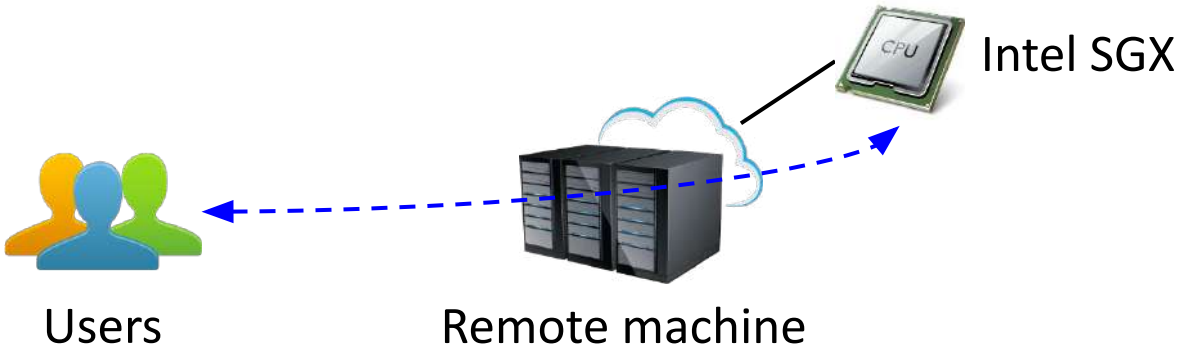


- Key issue:
  - The secure processors must be trusted
- Analogy:
  - You need to do business with an organization that might be corrupted
  - But there is one guy in the organization that is known to be extremely honest and fair
  - So you talk to that guy directly for your business

Secure Processor: IBM 4764

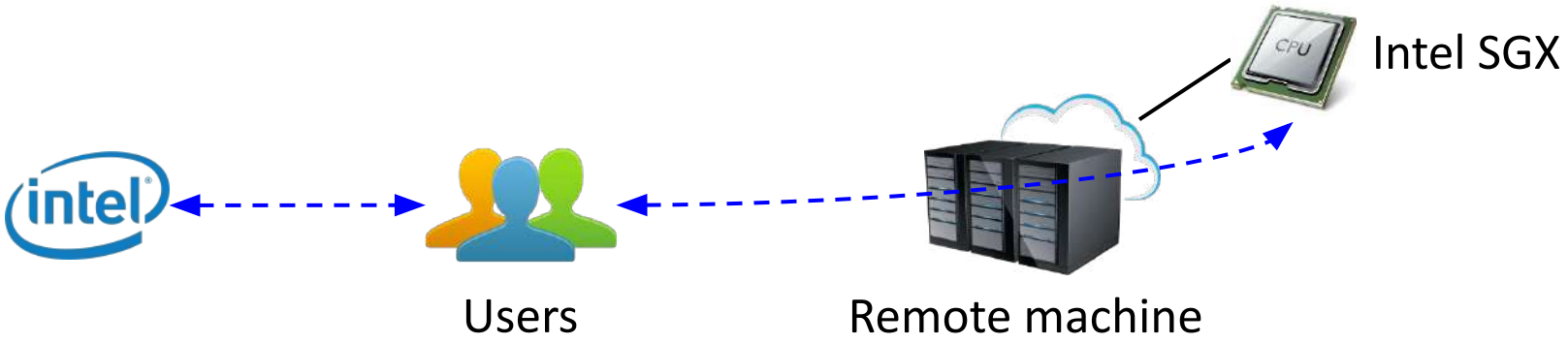
- A secure co-processor on a PCI board, built by IBM
- Temper-responsive, i.e.,
  - If someone wants to access the data stored in the processor via physical means, the co-processor would detect it and destroy everything stored
- Has a 233 MHz CPU and 32MB memory
- Discontinued in 2011

Secure Processor: Intel SGX



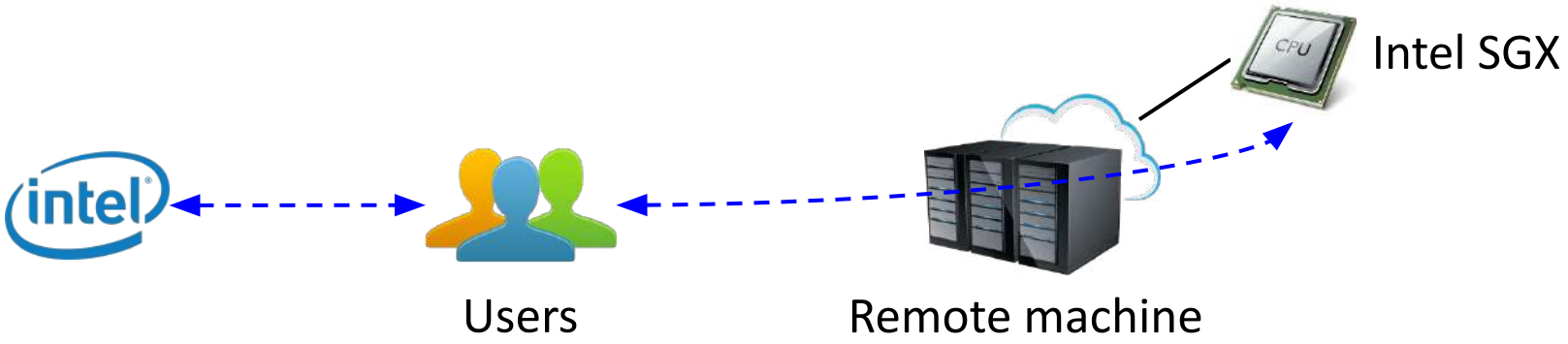
- Intel Software Guard Extensions (SGX)
  - A set of CPU instruction codes in the new generation of Intel CPUs
  - Allows a user to create a secure region (called an *enclave*) in the memory of a remote machine
  - Everything in the enclave is encrypted and cannot be seen by the machine owner

Secure Processor: Intel SGX



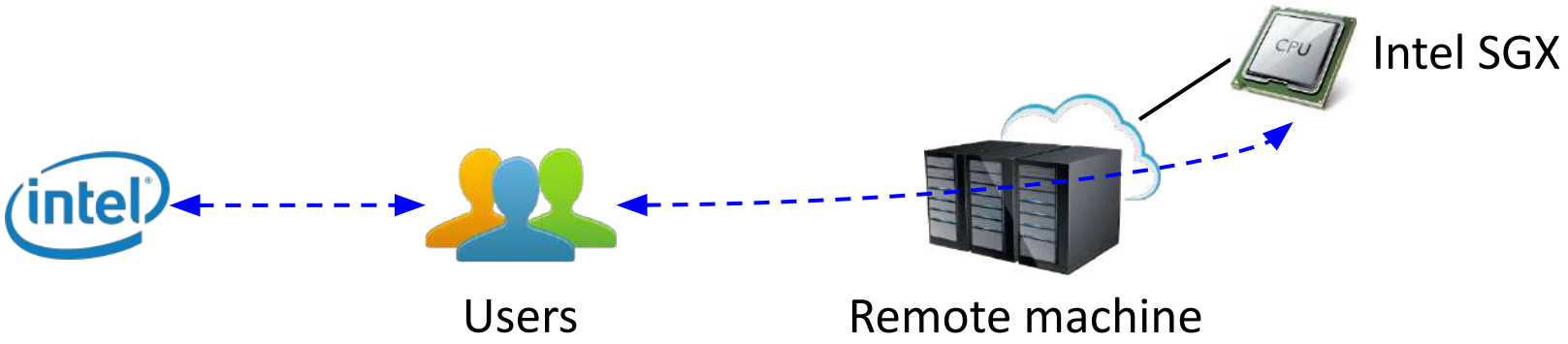
- Intel Software Guard Extensions (SGX)
  - Decryption is done only after the data is shipped to the CPU for computation
  - The computation is also shielded from the remote machine
  - Authentication of SGX requires communications with Intel

Issues with SGX



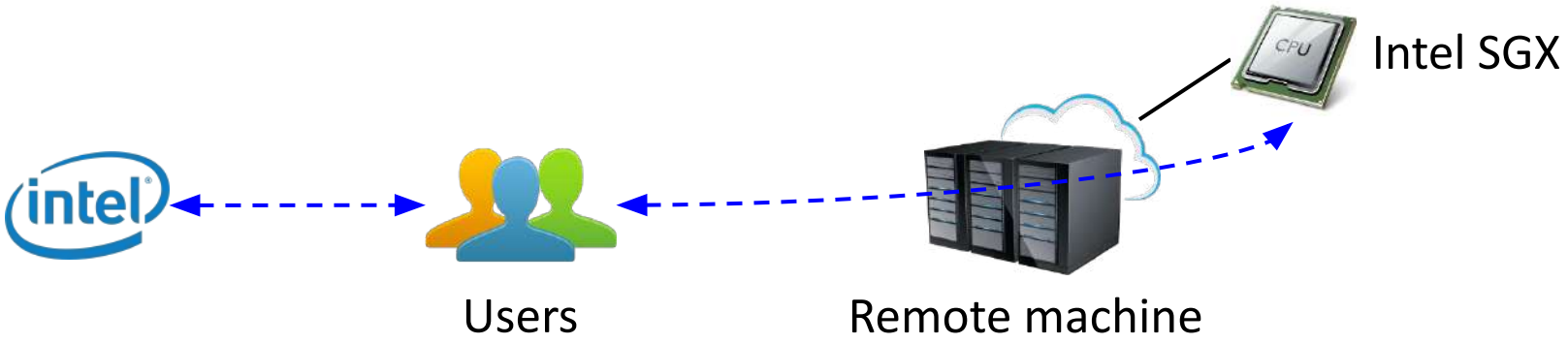
- The good
  - Using SGX can be much more efficient than homomorphic encryption
    - It is like using a normal CPU with encryption

Issues with SGX



- The bad
  - SGX is still very new
    - There is very limited offerings of software stacks with SGX
  - Early versions of SGX only allow the maximum enclave size to be 128 MB
    - So it is difficult to handle large programs

Issues with SGX



- The ugly
  - Not everyone trusts Intel
    - E.g., Chinese and Russian companies?
  - There exist attacks to Intel CPUs

“Silver Bullets”

- Homomorphic encryption
  - Partial homomorphic encryption (e.g., RSA) is doable but only supports one type of operation
  - Fully homomorphic encryption is still far from being practical
- Hardware-assisted security
  - Limited computation resource available (in most cases)
  - Attacks possible

“Silver Bullets”

- Homomorphic encryption
- Hardware-assisted security
- And a common problem for the above two: Leakage through *access patterns*, i.e.,
  - The service provider observes the frequency and order in which the encrypted data is accessed
  - Based on those, he infers information about the data and the users’ queries
- Analogy:
  - I monitor your phone calls
  - Your telco uses some crypto techniques to ensure that I am unable to eavesdrop your call or learn who you are calling
  - However, I can observe the starting and ending time of each call
  - In that case, I may still infer something
- Solution: Oblivious processing
  - We will not go into details here

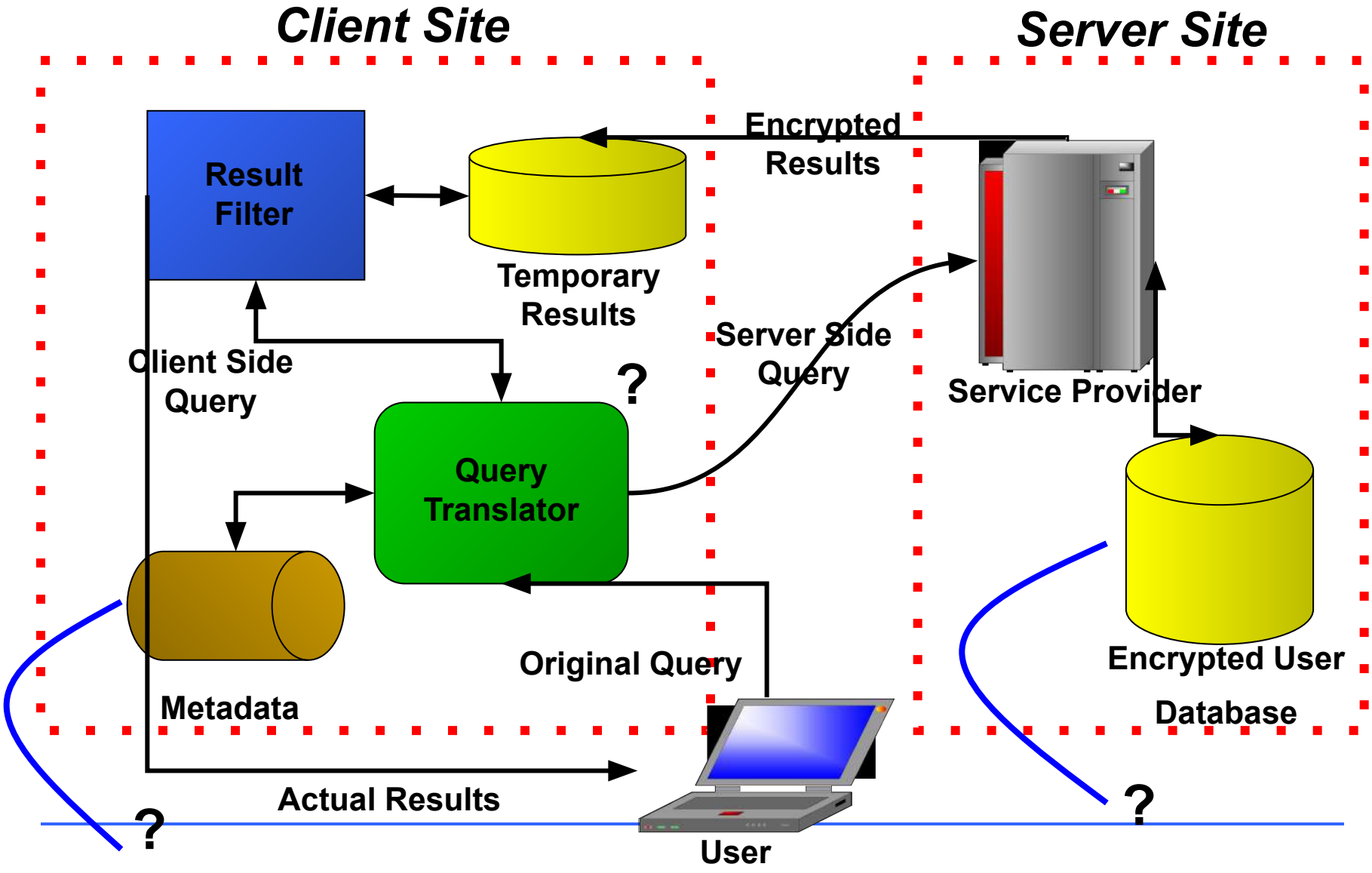
So there is no silver bullet (yet)…

- This motivates more practical approaches that
  - are not as secure as fully homomorphic encryption or hardware-assisted solutions
  - but could be implemented using conventional encryption methods
- We will discuss the solution (referred to as HILM) in the following paper
  - H. Hacigümüs, B. R. Iyer, C. Li, S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. SIGMOD 2002: 216-227

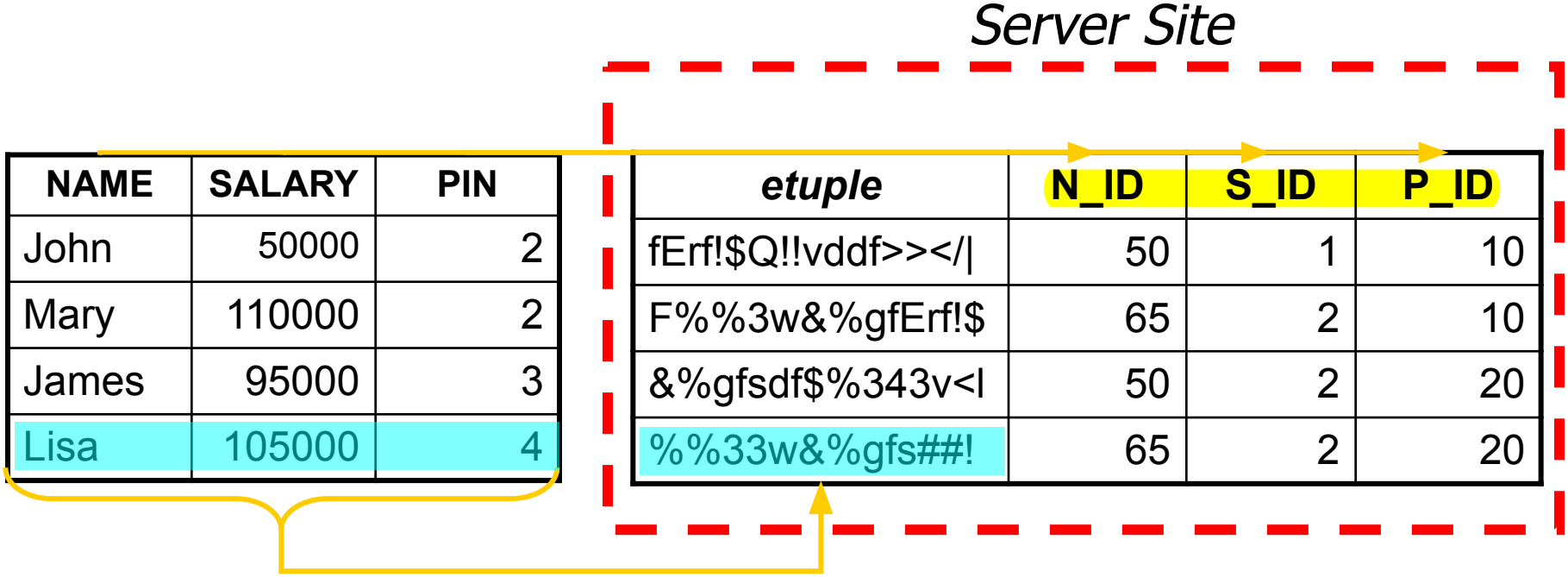
The HILM Approach: Basic Idea

- Divide encrypted tuples into groups, and attach some indices with each group
- The service provider processes encrypted tuples at the group level, using the indices
- Split the query processing task between the user and the service provider
  - The service provider returns some tuple groups relevant to the query
  - The user decrypts the tuple groups, and further processes them to produce the query result

The HILM Approach: Architecture



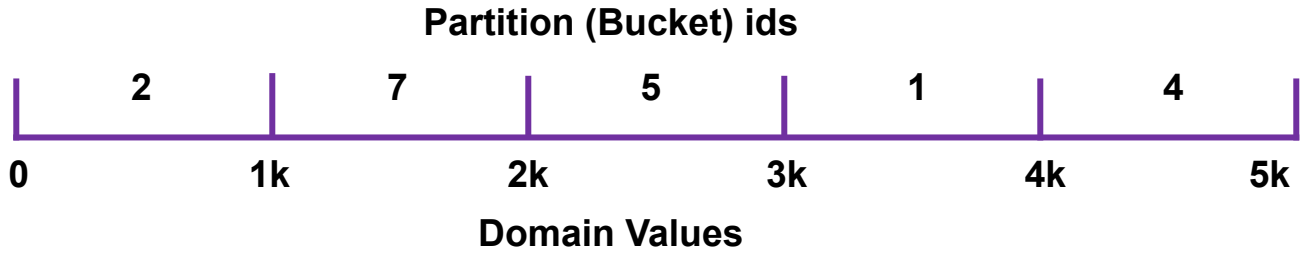
The HILM Approach: Encryption



- Store an encrypted string, *etuple*, for each tuple in the original table
  - This is called “row level encryption”
  - Any kind of encryption technique (e.g., AES, DES) can be used
- Create an index for each (or selected) attribute(s) in the original table

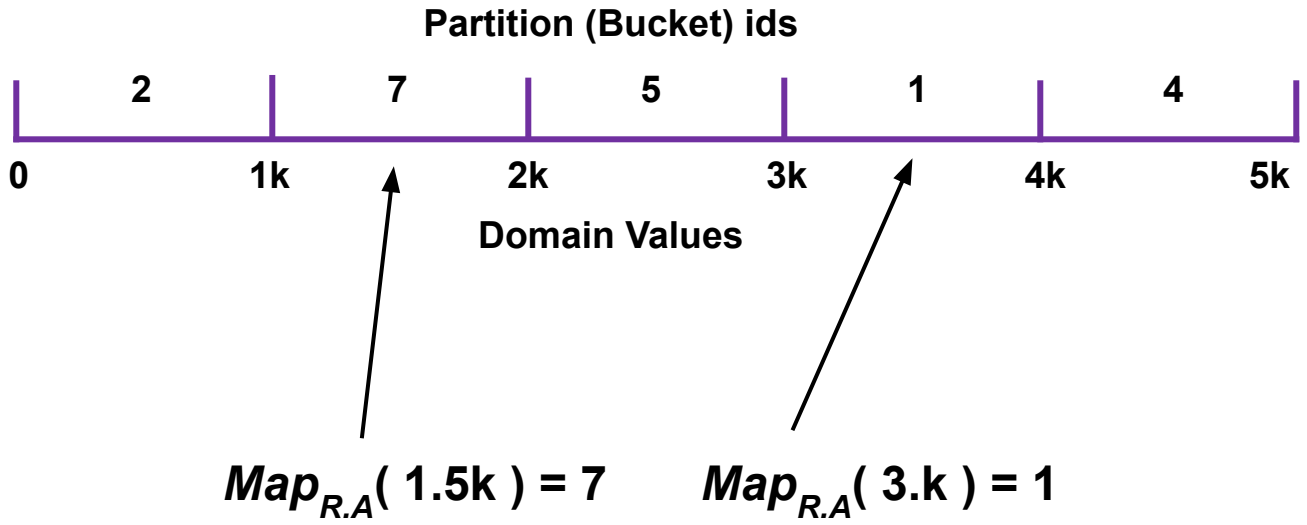
The HILM Approach: Indexing

- Given an attribute *A* in table *R* to be indexed, first divide the domain of *R.A* into a number of *buckets*
  - E.g., dividing the domain of Salary into [0, 1k), [1k, 2k), [2k, 3k), …,
- Then, assign an ID to each bucket



The HILM Approach: Indexing

- Define a mapping function $map_{R.A}$ that maps any *A* value to its partition ID



The HILM Approach: Indexing

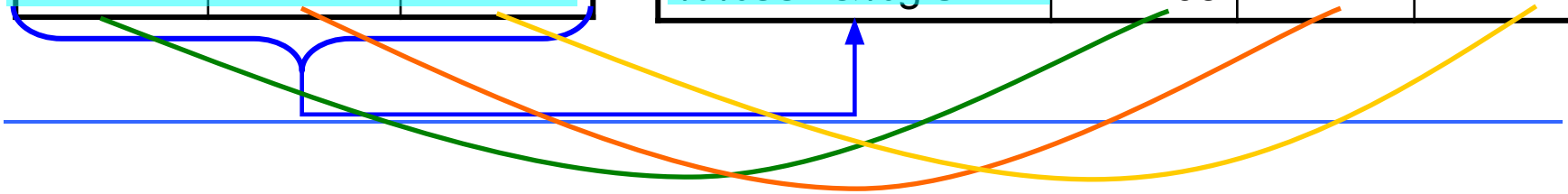
- The index on *A* is built by mapping each tuple's *A* value to its partition ID
  - R = < A, B, C > ⇒ R^S = < *etuple*, A_id, B_id, C_id >
  - A_id = $Map_{R.A}$(A), B_id = $Map_{R.B}$(B), C_id = $Map_{R.C}$(C)

Table: EMPLOYEE

NAME	SALARY	PIN
John	50000	2
Mary	110000	2
James	95000	3
Lisa	105000	4

Table: EMPLOYEE^S

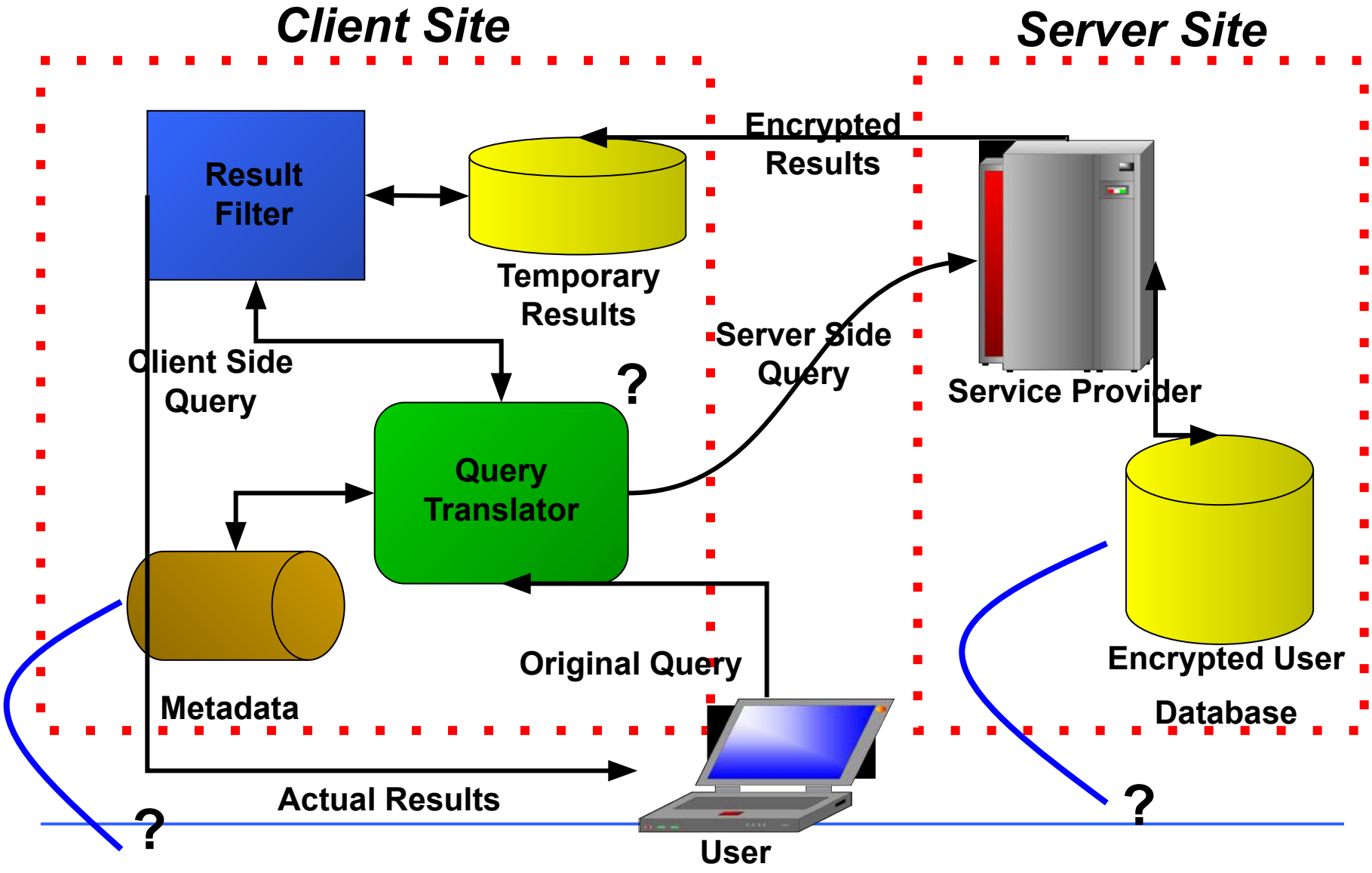
Etuple	N_ID	S_ID	P_ID
fErf!$Q!!vddf>><|/|	50	1	10
F%%3w&%gfErf!$	65	2	10
&%gfsdf$%343v<l	50	2	20
%%33w&%gfs##!	65	2	20



The HILM Approach

- Now we know how the HILM approach encrypts and indexes data
- Next, we will discuss how to implement the *query translator*
  - That is, how to convert a user's query into a query execution plan that splits the processing task between the user and the service provider

The HILM Approach: Architecture



The HILM Approach: Query Translation

- Consider the following query:
  - SELECT name, pname FROM emp, proj
    WHERE emp.pid=proj.pid AND salary > 100k
- The key issue here is that the service provider cannot see emp.pid, proj.pid, and salary
  - He can only see the bucket IDs
- So we need a way to map each query condition on original attributes into a condition on bucket IDs
- For this purpose, the HILM approach defines a mapping function Map_{cond} for mapping conditions

The HILM Approach: Mapping Condition

Table: EMPLOYEE

NAME	SALARY	PIN
John	50000	2
Mary	110000	2
James	95000	3
Lisa	105000	4

Table: EMPLOYEE^{S}

Etuple	N_ID	S_ID	P_ID
fErf!$Q!!vddf>><||	50	1	10
F%%3w&%gfErf!$	65	2	10
&%gfsdf$%343v<l	50	2	20
%%33w&%gfs##!	65	2	20

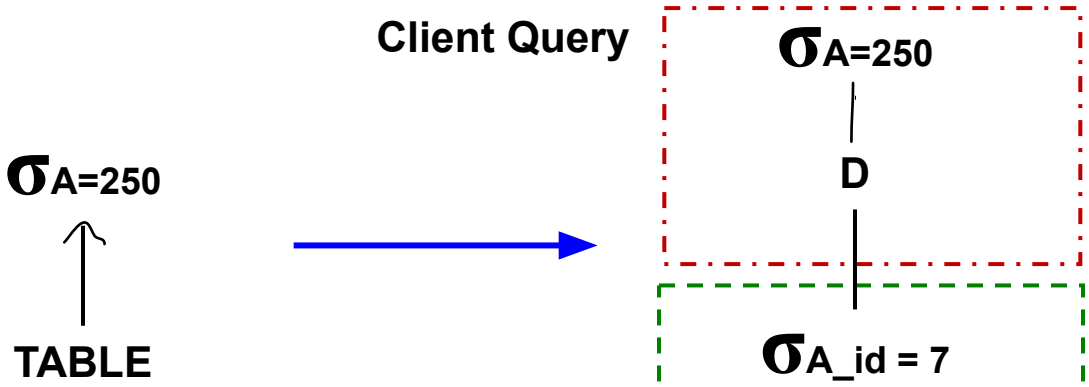
- Example:
  - If the condition is “Salary = 95000”
  - Then it is mapped to “S_ID = 2”
- That is, Map_{cond}(“Salary = 95000”) = “S_ID = 2”

The HILM Approach: Mapping Condition

- Once we have the mapping for the condition on each attribute, we can use them to transform users' queries
- We just need to follow some transformation rule for each query operator
- We will discuss two operators for example:
  - Selection σ
  - Join $\bowtie$

Selection

$$\sigma_c(R) = \sigma_c(D(\sigma^S_{Mapcond(c)}(R^S)))$$

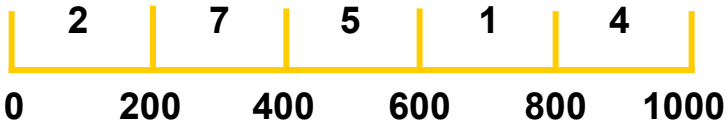


Null Answer

D(6*&^%) = 240
D(8***^%) = 300

E#	A#
2	120
6	240
8	300
12	500

etuple	A#
2%$&*	2
6*&^%	7
8***^%	7
12#@!	5



The HILM Approach: Mapping Condition

Table: E1

NAME	SALARY
John	50000
Mary	110000
James	95000
Lisa	105000

N1_ID
50
65
50
65

Table: E2

N2_ID
10
25
30
40

NAME	PIN
Alice	10
Bob	10
John	20
Lisa	20

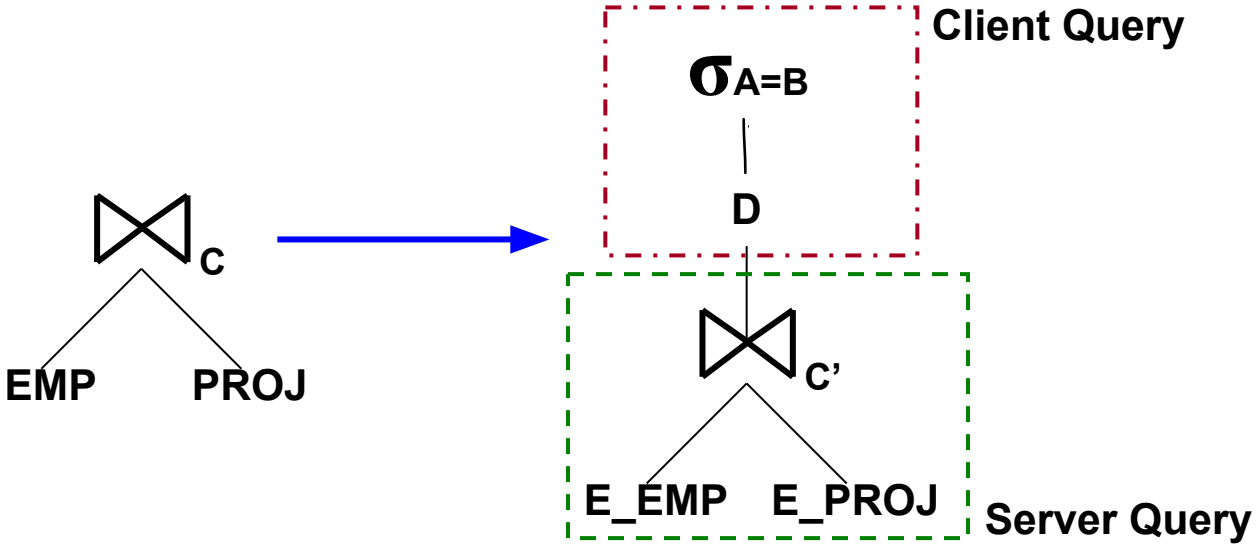
- Example:
  - If the condition is “E1.Name = E2.Name”
  - Then it is mapped to
    “(N1_ID = 50 AND N2_ID = 30) OR (N1_ID = 65 AND N2_ID = 40)”
- That is,
  - Map_{cond}(“E1.Name = E2.Name”)
    = “(N1_ID = 50 AND N2_ID = 30) OR (N1_ID = 65 AND N2_ID = 40)”

cannot simply be "N1_ID = N2_ID"

because buckets do not have the same value
- created randomly

Join

$$R \bowtie_c T = \sigma_c(D (R^S \bowtie^S_{Mapcond(c)} T^S))$$



Partitions	A_id
[0,100]	2
(100,200]	4
(200,300]	3

Partitions	B_id
[0,200]	9
(200,400]	8

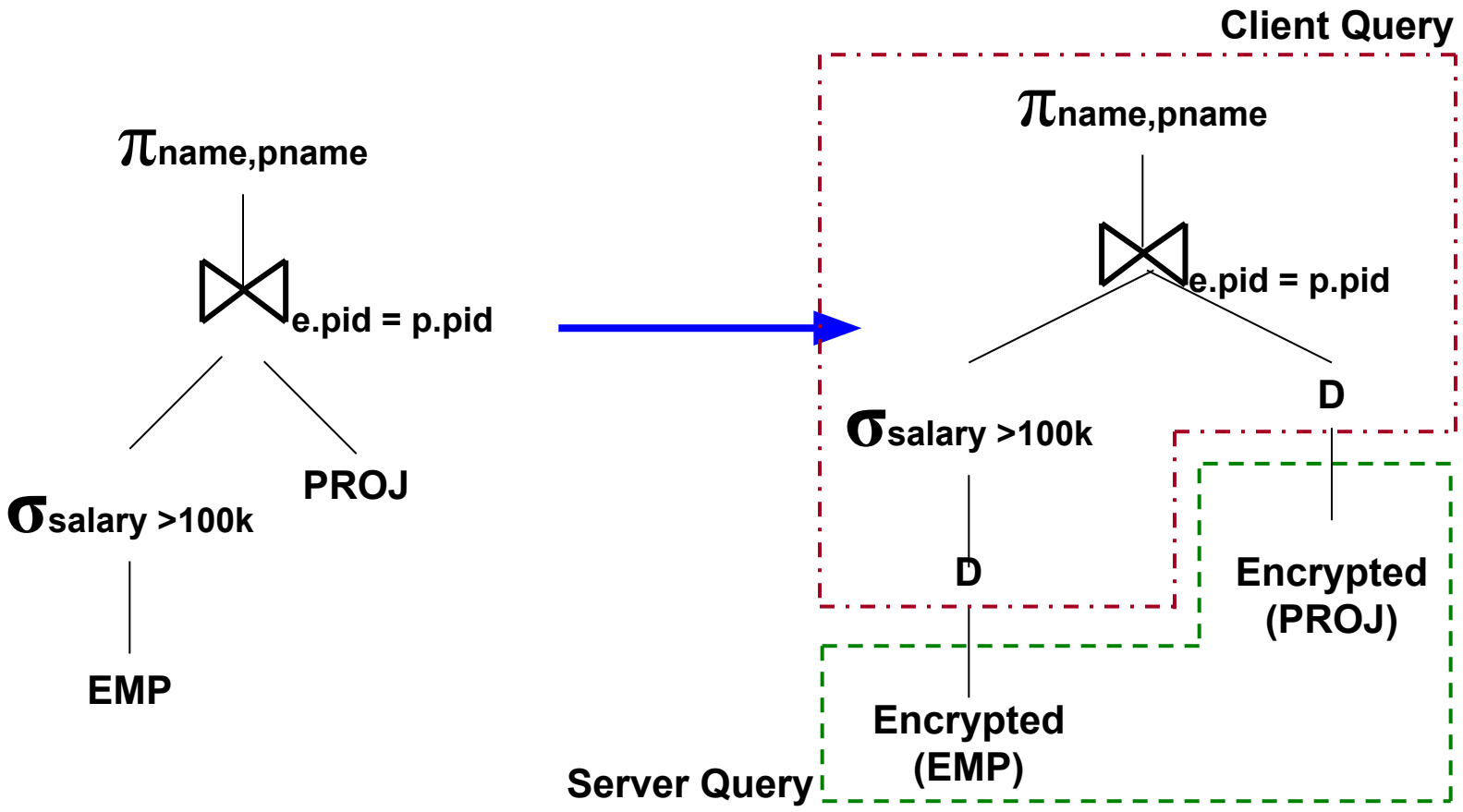
C : A = B ⇒ C' :(A_id = 2 ∧ B_id = 9)
∨ (A_id = 4 ∧ B_id = 9) ∨
(A_id = 3 ∧ B_id = 8)

The HILM Approach: Query Translation

- Now we know how to map query conditions
- We will proceed to discuss how to generate a query execution plan
- Objective: Minimize the work done by the client side
- Basic Idea:
  - Start from a simple query plan
  - Iteratively improve the plan by pushing operations to the server site
  - This is somewhat similar to heuristic optimization in conventional query plan generation

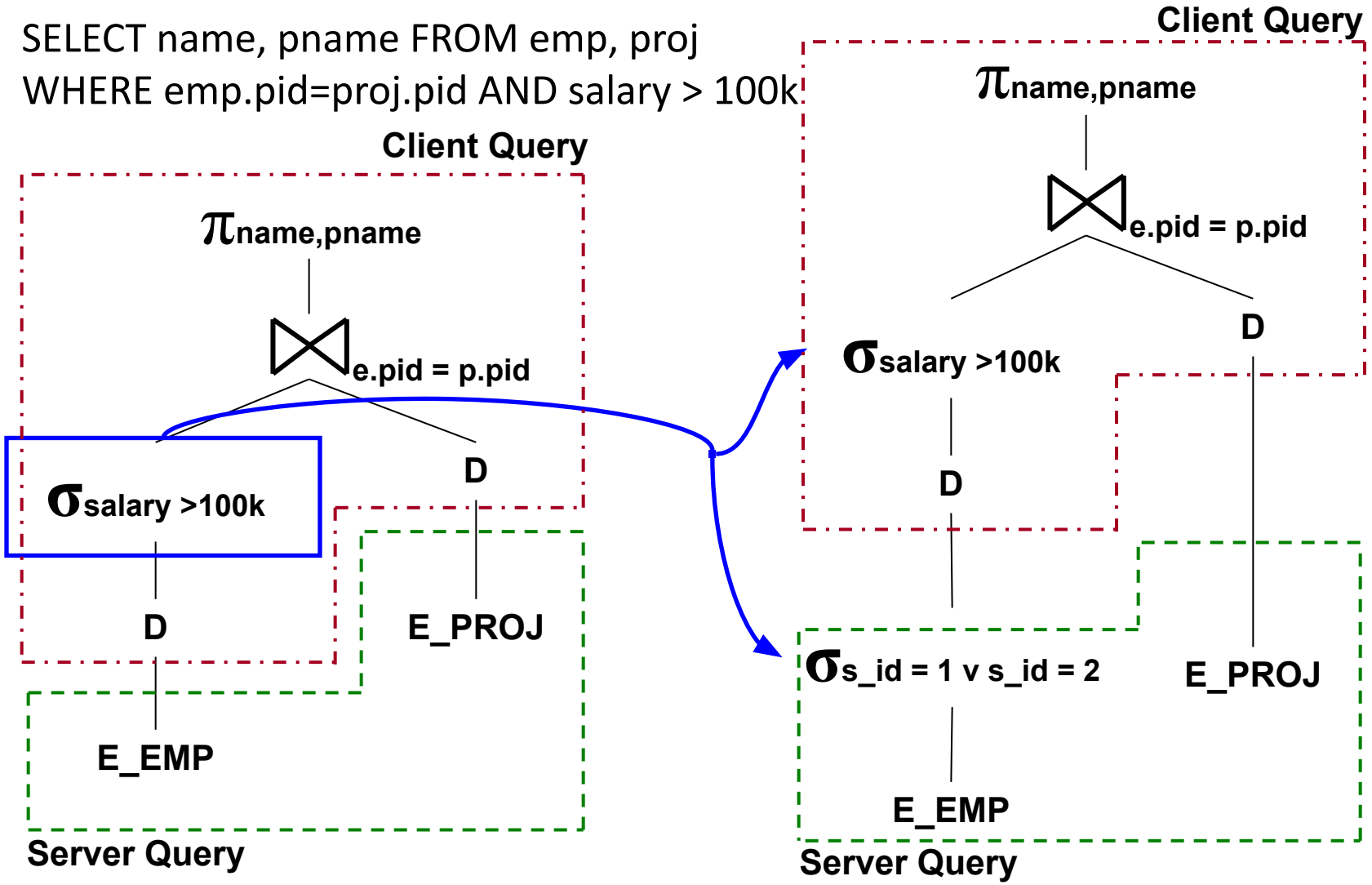
The HILM Approach: Query Translation

SELECT name, pname FROM emp, proj
WHERE emp.pid=proj.pid AND salary > 100k



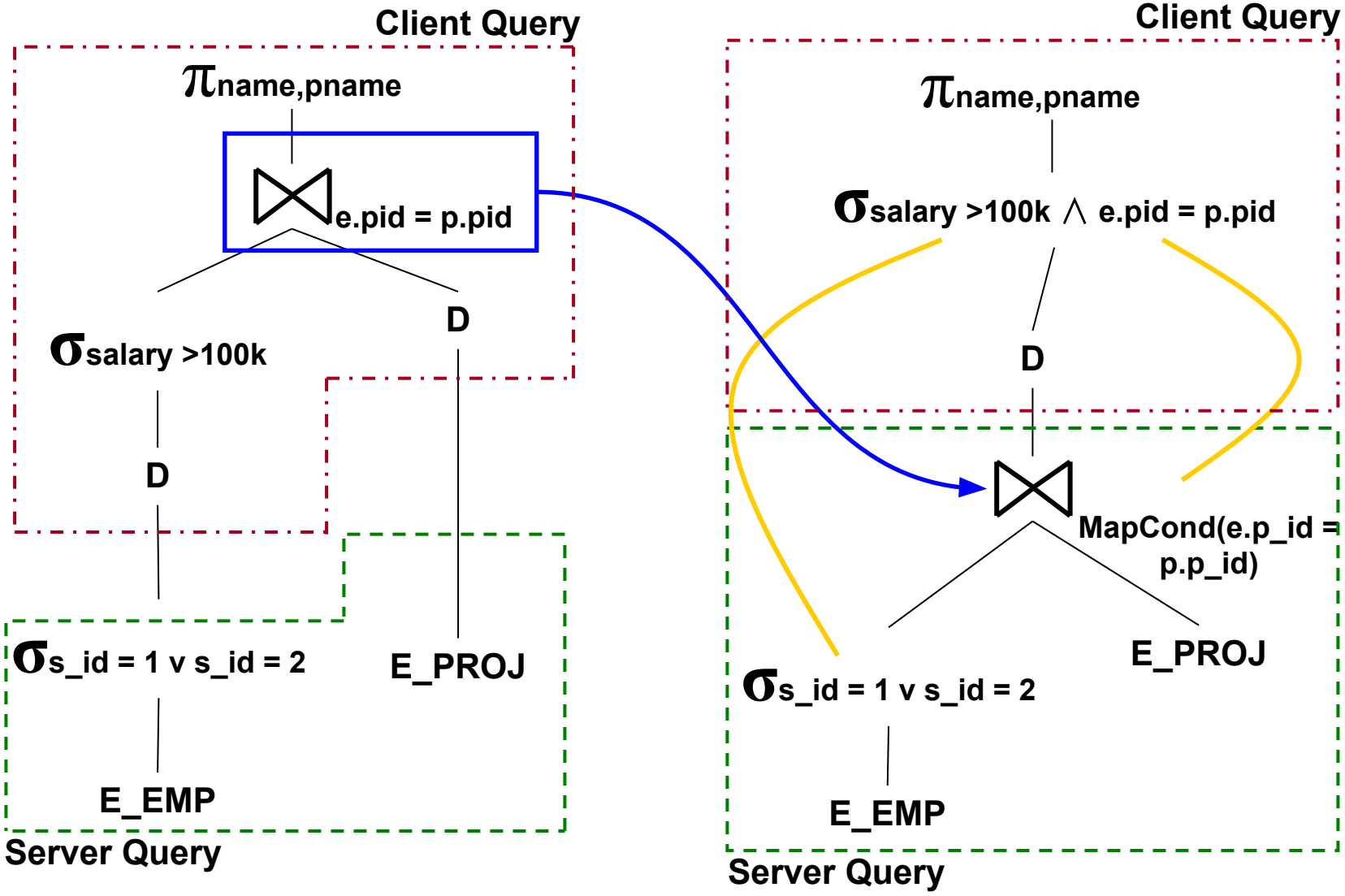
The HILM Approach: Query Translation

SELECT name, pname FROM emp, proj
WHERE emp.pid=proj.pid AND salary > 100k

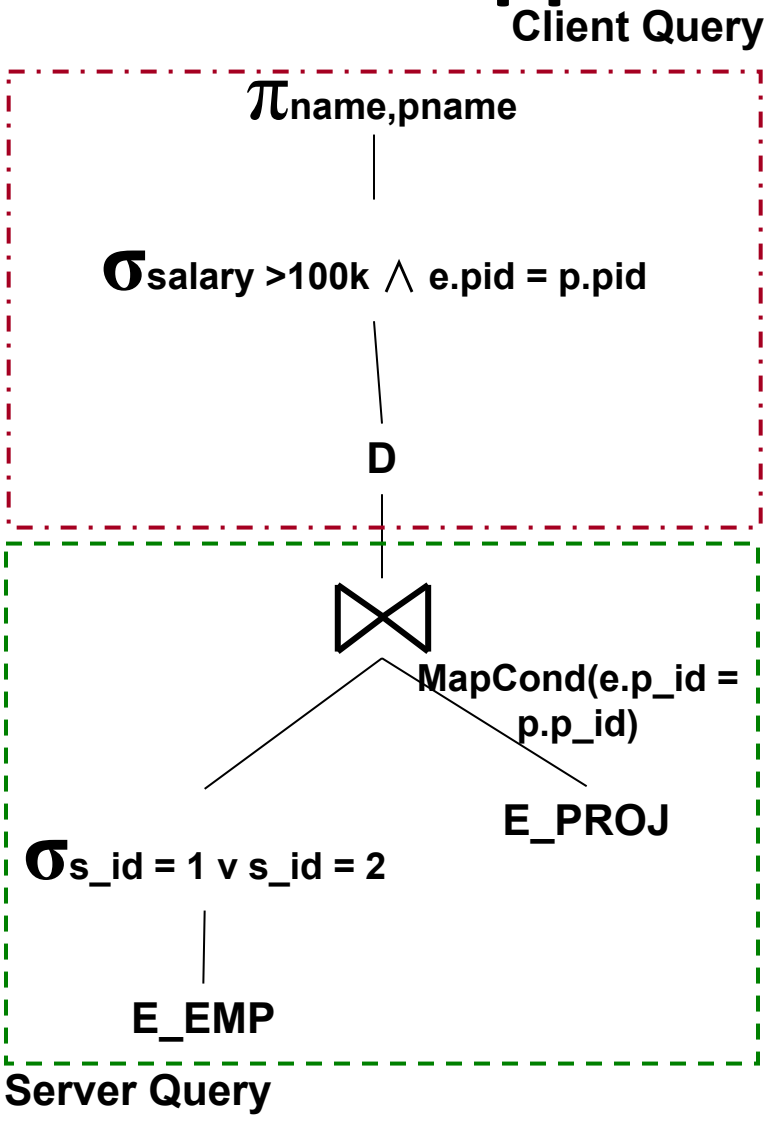


pushing selection to sever side

The HILM Approach: Query Translation



The HILM Approach: Query Translation



Q:

```
SELECT name, pname
FROM emp, proj
 WHERE emp.pid=proj.pid
 AND salary > 100k
```

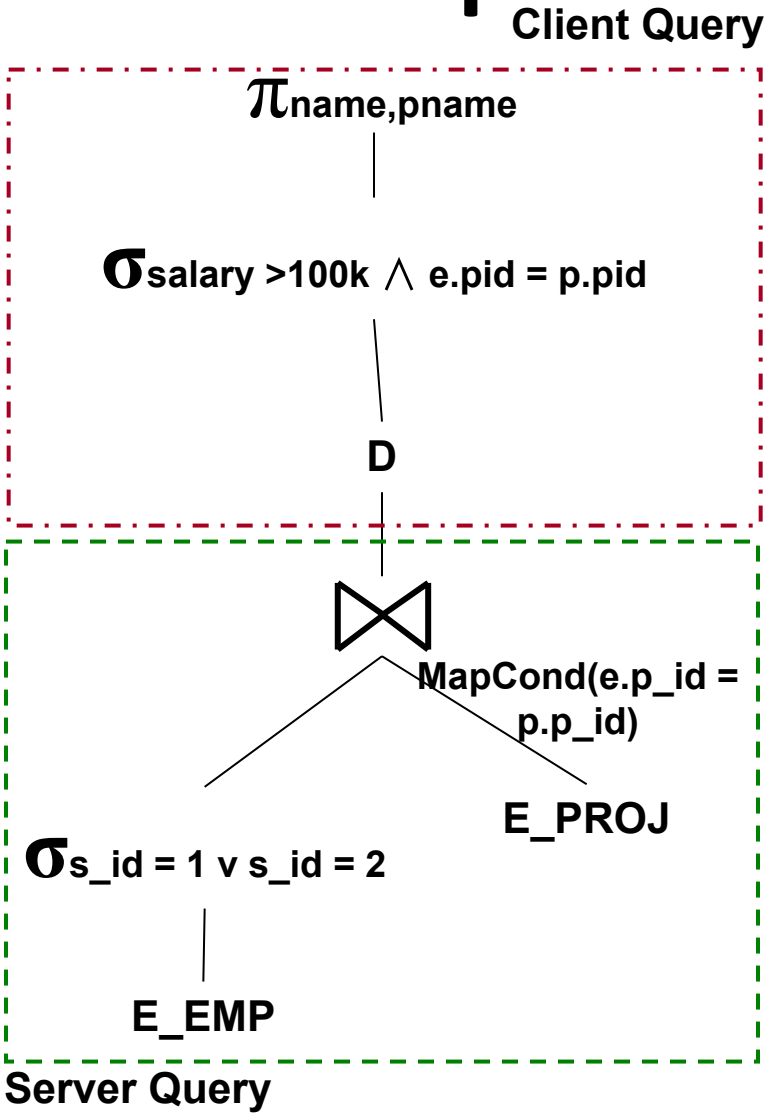
Q^S:

```
SELECT    e_emp.etuple, e_proj.etuple
FROM      e_emp, e_proj
 WHERE    e.p_id=p.p_id AND
(s_id = 1 OR s_id = 2)
```

Q^C:

```
SELECT    name, pname
FROM      temp
 WHERE    emp.pid=proj.pid AND
salary > 100k
```

Further Optimizations



- Observation:
  - No need to materialize the join result on the server
  - It suffices to send the "joinable" partitions to the clients

Q^S: SELECT e_emp.etuple, e_proj.etuple
FROM e_emp, e_proj
WHERE e.p_id=p.p_id AND
(s_id = 1 OR s_id = 2)

Q^C: SELECT name, pname
FROM temp
WHERE emp.pid=proj.pid AND
salary > 100k

Further Optimizations

- The HILM approach encrypts each tuple as a whole
- Therefore, even if the user's query is asking for one attribute, the HILM has to return whole encrypted tuples
- An improved approach: Encrypt each attribute separately

Table: EMPLOYEE

NAME	SAL	COM
John	50000	5000
Mary	110000	11000
James	95000	9500
Lisa	105000	10500

Table: EMPLOYEE**S**

E_NAME	E_SAL	E_COM
45ewt*(&	3t45f33	*&%*kk
(*#hKJ(0	Ek%98*	!DE#$F
()&%^JK	H^F(j^7	%^g%6
324(&^hj	(86&&h$	887^%$

Further Optimizations

- Some attributes that are frequently used for aggregation
  - E.g., Salary
- For these attributes, we may use partial homomorphic encryptions
  - So that some aggregations might be performed by the service provider

Table: EMPLOYEE

NAME	SAL	COM
John	50000	5000
Mary	110000	11000
James	95000	9500
Lisa	105000	10500

Table: EMPLOYEE**S**

E_NAME	E_SAL	E_COM
45ewt*(&	3t45f33	*&%*kk
(*#hKJ(0	Ek%98*	!DE#$F
()&%^JK	H^F(j^7	%^g%6
324(&^hj	(86&&h$	887^%$

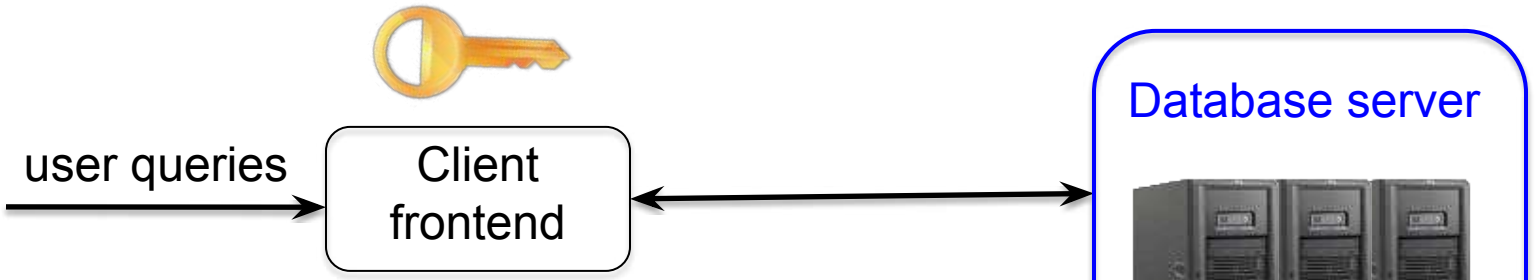
The HILM Approach: Limitations

- The user needs to do a lot of work for query processing
- Query translation is complicated sometimes
  - Especially for joins
- The partition approach does not have any formal security guarantee
  - It is unclear how much the service provider may infer from the partitions of attributes

We will discuss

- An improved approach that
  - Process queries *completely* at the server, asking the user only to decrypt results
    - By using less secure encryptions schemes whenever necessary
  - Have clearer security guarantees
- Reference:
  - R. A. Popa, C. M. S. Redfield, N. Zeldovich, H. Balakrishnan. “CryptDB: Protecting Confidentiality with Encrypted Query Processing”. SOSP 2011.

CryptDB: System Setup



- Trusted
- Stores schema, master key
- No query execution

- Stores the database and processes SQL queries
- Not trusted to keep data private

1. Support standard SQL queries on encrypted data
2. Process queries *completely* at the DB server
3. *No change* to existing DBMS

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- Encrypt each value of each tuple separately
- Each user query is directly translated into a server query on the encrypted data
  - Without using complex translation rules

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- Key observation:
  - There is no efficient encryption scheme that let the service provider handle all queries on encrypted data
  - So let's use *multiple* encryption schemes
    - Each scheme can support one type of queries
  - Some schemes would be more secure than the other
    - Accept lower security guarantees for the sake of query processing

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- SELECT * FROM Employee WHERE Rank = 3

  ==>
- SELECT * FROM E WHERE Col2 = xe81d17
- Note:
  - For this to work, the encryption scheme for Rank has to be *deterministic*
  - i.e., if two plaintexts are the same, then their ciphertexts are also the same
- Implication:
  - The service provider can know which Col2 values are the same

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- SELECT * FROM Employee WHERE Salary >= 30k

  ==>
- SELECT * FROM E WHERE Col3 >= x7108bd
- Note:
  - For this to work, the encryption scheme for Salary has to be *order preserving*
  - i.e., if salary1 < salary2, then encrypt(salary1) < encrypt(salary2)
- Implication:
  - The service provider can know the order of Col3 values

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- SELECT * FROM Employee WHERE Name LIKE '%Alice%'

  ==>
- SELECT * FROM E WHERE F(Col1, *token*) = true
- Note:
  - *token* is an encrypted value generated from "Alice"
  - F is a user defined function used to check whether "Alice" appears in a ciphertext
- Implication:
  - The encryption scheme for Name has to support this kind of keyword search

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- Summary: Different encryptions are used for different attributes:
  - To support equality search (e.g., A = x), use deterministic encryption
  - To support inequality search (e.g., A <= x), use order preserving encryption
  - To support keyword search (e.g., A LIKE '%X%'), use searchable encryption
  - If no search needs to be supported, use randomized encryption (i.e., no two ciphertext would be the same)

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- Note that different encryption schemes have different level of security
  - randomized > deterministic > order-preserving
- Idea:
  - Among the schemes that can support the queries needed, choose the most secure one
- Example:
  - If we are to support equality search on Rank, then choose deterministic encryption instead of order-preserving

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- Question:
  - What if we want to support multiple types of encryption?
  - e.g., we want to support both equality search and inequality search on Rank
- Straightforward solution:
  - Generate two encrypted versions of Rank
  - One using deterministic encryption, the other using order-preserving encryption

CryptDB: Basic Idea

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2-DET	Col2-OPE	Col3
xad4de6	xe81d17	x5e55e2	x7108bd
x7b7abb	xced8f1	x373c84	x15b212
x15fc58	x97d758	x8a0714	xa821fd
xddd463	xe81d17	x7a472d	xaca042

- Problem:
  - The service provider can see Col2-OPE (and learn the order of Col2) even if the user only uses equality search
  - i.e., the service provider learns more than what the user's query needs to reveal
- Solution:
  - Apply multiple encryption schemes *in an onion*

CryptDB: Onion

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

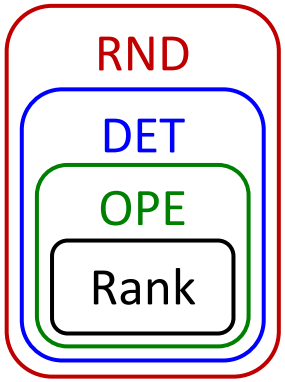
enc Rank first

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

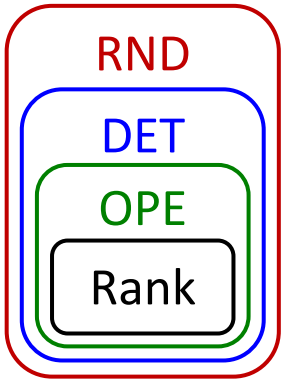
- Encrypt Rank three times
  - First, encrypt using an order-preserving scheme to get c1 = OPE(Rank)
  - Then, encrypt c1 using a deterministic scheme to get c2 = DET(c1)
  - Finally, encrypt c2 using a randomized scheme to get c3 = RND(c2)

CryptDB: Onion



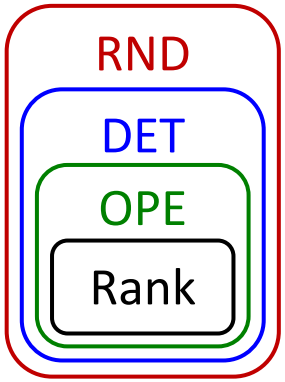
- Example: the user searches for “Rank = 2”
  - Provide the database with the key to decrypt the RND layer
  - After that, process the equality query at the DET layer
- In other words, we peel off the RND layer of the onion
- But the other layers remains

CryptDB: Onion



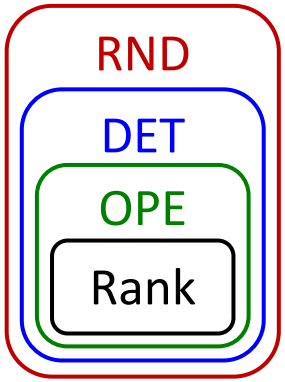
- Some time later, the user searches for “Rank >= 3”
  - Provide the database with they key to decrypt the DET layer
  - After that, process the query at the OPE layer
- In other words, we peel off the DET layer
- But the OPE layer remains

CryptDB: Onion



- In general, each column is initially encrypted into one (or more) onions
- After users start issuing queries, some layers are peeled off to enable query processing
- In other words, the security guarantee is reduced for the sake of query processing
- However, the plaintexts are never revealed to the service provider
  - Since the last layer of the onion is never peeled off

CryptDB: Onion



When do we need "more"?

- In general, each column is initially encrypted into one (or more) onions
- After users start issuing queries, some layers are peeled off to enable query processing
- In other words, the security guarantee is reduced for the sake of query processing
- However, the plaintexts are never revealed to the service provider
  - Since the last layer of the onion is never peeled off

CryptDB: Onion

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- Consider the Salary attribute in Employee
- How can we answer the following query?
  - Select SUM(Salary) FROM Employee WHERE Rank = 3
- "Rank = 3" can be handled with deterministic encryption on Rank
- But… What should we do with "SUM(Salary)"?
- We need additive homomorphic encryption on Salary

CryptDB: Onion

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- Assume that we apply additive homomorphic encryption on Salary
- Then how do we handle the following query?
  - SELECT Name FROM Employee WHERE Salary > 30k
- We need order-preserving encryption on Salary
- Is there an encryption scheme that is both order-preserving and additive homomorphic?
- No

CryptDB: Onion

Employee

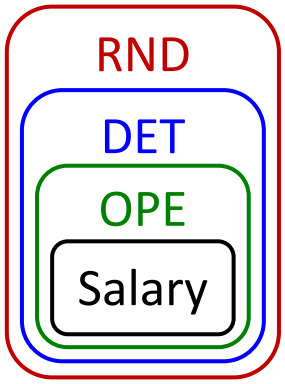
Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k
Dan C	3	50k

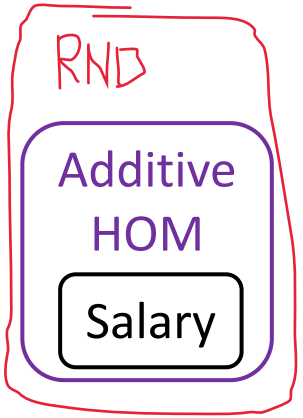
E

Col1	Col2	Col3
xad4de6	xe81d17	x7108bd
x7b7abb	xced8f1	x15b212
x15fc58	x97d758	xa821fd
xddd463	xe81d17	xaca042

- So what do we do?
- Solution: have two onions on Salary
  - One to handle SUM(Salary)
  - Another one to handle equality and inequality searches on Salary

CryptDB: Onion





- So what do we do?
- Solution: have two onions on Salary
  - One to handle SUM(Salary)
  - Another one to handle equality and inequality searches on Salary

CryptDB: Equi-Join

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k

Department

Name	Head
Finance	Cath Y
HR	Alice Z
Legal	Bob X

- Suppose that we need to join the above two tables
  - SELECT * FROM Employee E, Department D WHERE E.Name = D.Head
- How do we handle this query?
- Option 1:
  - Apply deterministic encryption on both Name and Head, using the same encryption key
  - So that “Alice Z” from Employee.Name has the same cipher text with “Alice Z” from Department.Head

CryptDB: Equi-Join

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k

Department

Name	Head
Finance	Cath Y
HR	Alice Z
Legal	Bob X

- Option 1:
  - Apply deterministic encryption on both Name and Head, using the same encryption key
- Any problem?
  - By looking at the ciphertexts, the service provider can already figure out which Employee.Name can be joined with which Department.Head
  - This can happen even if the user does not issue a join on Employee.Name = Department.Head

CryptDB: Equi-Join

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k

Department

Name	Head
Finance	Cath Y
HR	Alice Z
Legal	Bob X

- Objective:
  - If no user issues a join query on “Employee.Name = Department.Head”, the service provider should not be able to figure out which Employee.Name can be joined with which Department.Head

CryptDB: Equi-Join

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k

Department

Name	Head
Finance	Cath Y
HR	Alice Z
Legal	Bob X

- CryptDB's solution:
  - Encrypt the two columns using different keys
    - Results: DET_{k1}(Name), DET_{k2}(Head)
  - Generate a "tag" for each column, using different keys
    - Results: Tag_{k3}(Name), Tag_{k4}(Head)
  - Concatenate the tags with the encrypted values
    - Results: Tag_{k3}(Name)||DET_{k1}(Name), Tag_{k4}(Head)||DET_{k2}(Head)

CryptDB: Equi-Join

Employee

Name	Rank	Salary
Alice Z	3	30k
Bob X	1	10k
Cath Y	4	40k

Department

Name	Head
Finance	Cath Y
HR	Alice Z
Legal	Bob X

- CryptDB's solution:
  - Concatenate the tags with the encrypted values
    - Results: $Tag_{k3}(Name) \| DET_{k1}(Name)$, $Tag_{k4}(Head) \| DET_{k2}(Head)$
- When a user asks to join on E.Name = D.Head
  - *Adjust* the tag of Head to change the key to k3
    - $Tag_{k4}(Head) \| DET_{k2}(Head)$ --> $Tag_{k3}(Head) \| DET_{k2}(Head)$
    - This adjusting is done without revealing the plaintext
  - After that, do the join based on $Tag_{k3}(Name)$ and $Tag_{k3}(Head)$

Tag Generation

- Generation via encryption:
  - Use elliptic-curve cryptography (ECC)
  - public parameter *P,* secret key *K*, pseudo-random function *PRF* with key K_0
  - $\text{Tag}_K(\text{Name}) = P^{K*PRF(\text{Name})}$
- Do the same for Department.Head
  - Using the same *P* and PRF with key K_0, but different secret key *K'*
  - $\text{Tag}_{K'}(\text{Head}) = P^{K'*PRF(\text{Head})}$

Tag Adjustment

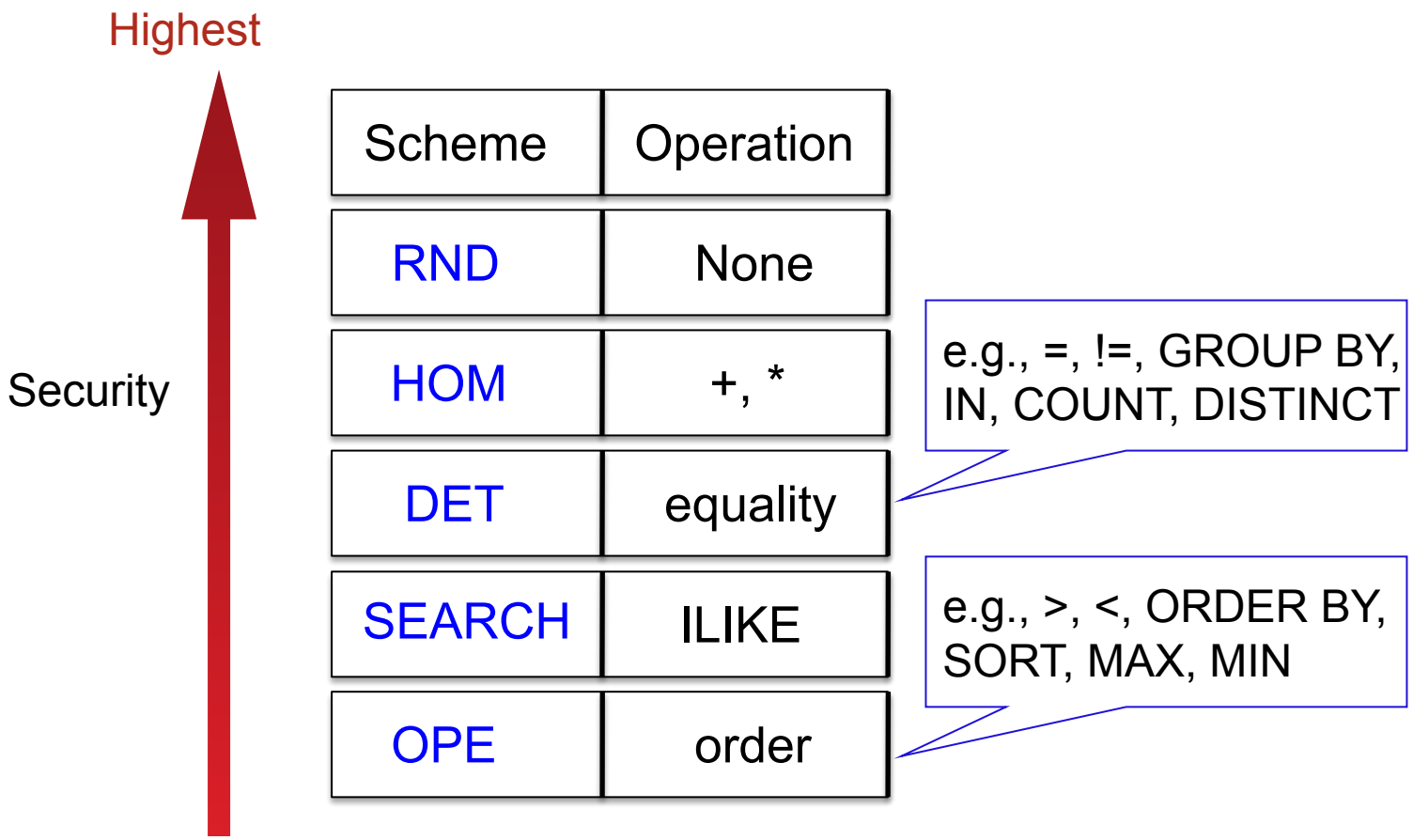
- Tags
  - $\text{Tag}_K(\text{Name}) = P^{K*PRF(\text{Name})}$
  - $\text{Tag}_{K'}(\text{Head}) = P^{K'*PRF(\text{Head})}$
- Initially, the service provider cannot figure out which tags represent the same value
- But when a user issues a join on Name = Head
  - We give the service provider the value of K/K'
  - Then, the service provider adjusts $\text{Tag}_{K'}(\text{Head})$ to be

    $(\text{Tag}_{K'}(\text{Head}))^{K/K'}$

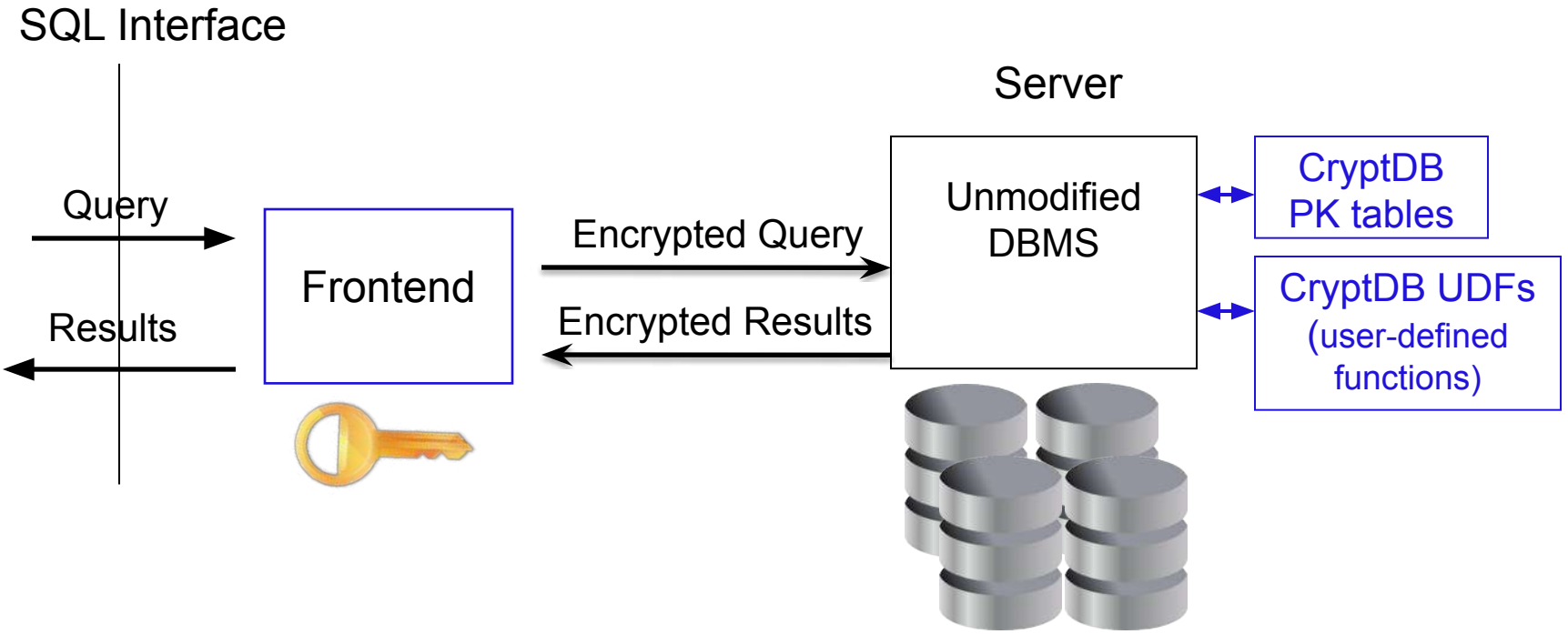
    $= (P^{K'*PRF(\text{Head})})^{K/K'} = P^{K*PRF(\text{Head})}$

    $= \text{Tag}_K(\text{Head})$
  - Now if Name = Head, then $\text{Tag}_K(\text{Name}) = \text{Tag}_K(\text{Head})$
- In other words, the security guarantees on Name and Head are lowered to reveal which of them can be joined
- But the plaintext of Name and Head remains secret

CryptDB: Summary



CryptDB: Implementation



- No change to the DBMS

CryptDB: Limitations

- Not all operations can be performed on encrypted data, e.g.,
  - WHERE T1.a + T1.b > T2.c
  - SUM(100*a + 2*b)

CryptDB: Conclusion

- A practical DBMS for running most standard queries on encrypted data
- Runs queries completely at server
- Provides provable privacy guarantees
- Modest overhead
- Does not change the DBMS or client applications