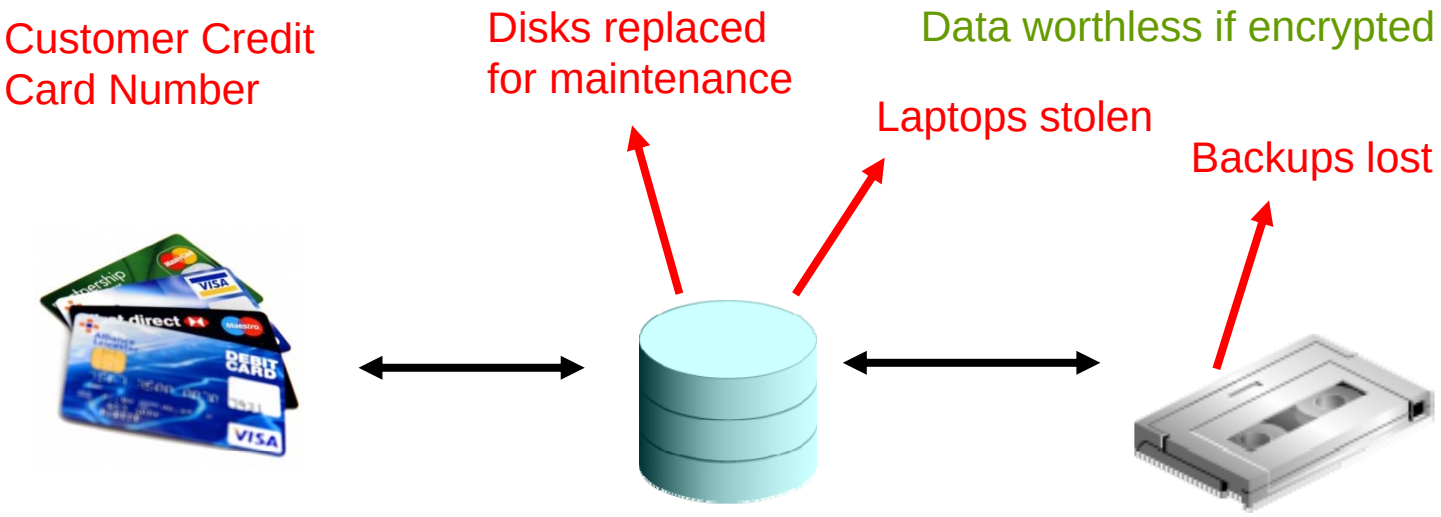
Security in Outsourced Databases II (Query Processing on Encrypted Data)



Why Encrypt Data?

- We have already discussed authentication and access control as means to allow access to the data to authorized persons only
- However, authentication & access control may not be enough (DB administrators can still access and see the data; intrusion/sql injection, etc)
- If data are sensitive it is also possible to encrypt them
  - Data encryption is the last barrier to protect sensitive data confidentiality

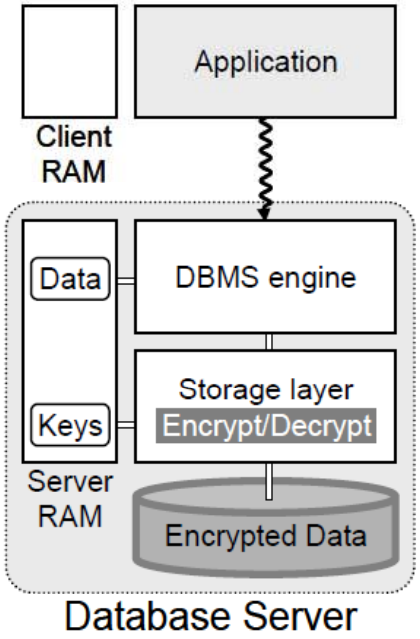
Why Encypt Data? - External requirements

- Health Insurance Portability & Accountability Act (HIPPA):
  - Requires data safeguards that protect against “intentional or unintentional use or disclosure of protected health information”
  - It *mandates* “to ensure the confidentiality, integrity and availability of all electronic protected health information the covered entity creates, receives, maintains, or transmits”
  - It *mandates* “to implement a mechanism to encrypt and decrypt electronic protected health information”

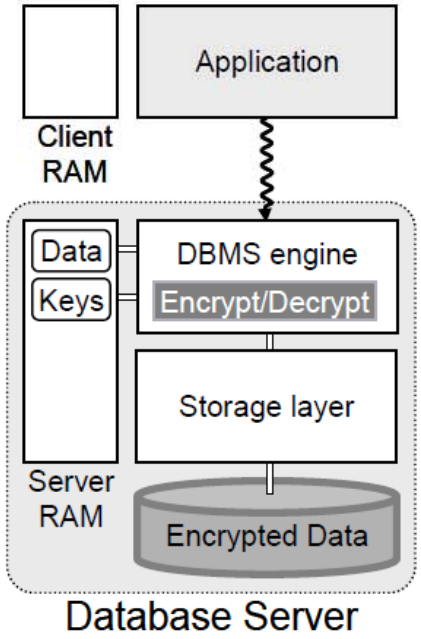
Why Encypt Data? - Business Compliance

- Payment Card Industry (PCI) Data Security Standard
  - Stored cardholder *data must be rendered unreadable*, and it includes cryptographic methods in the recommended controls
  - Adopted by American Express, Visa, MasterCard and several other payment card companies

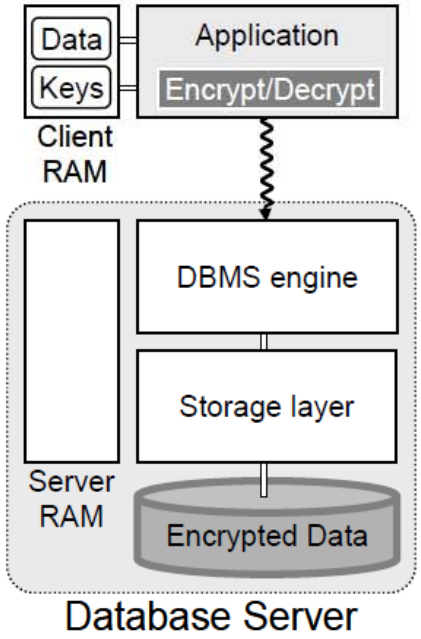
Three options for database encryption



a. Storage-level encryption



b. Database-level encryption



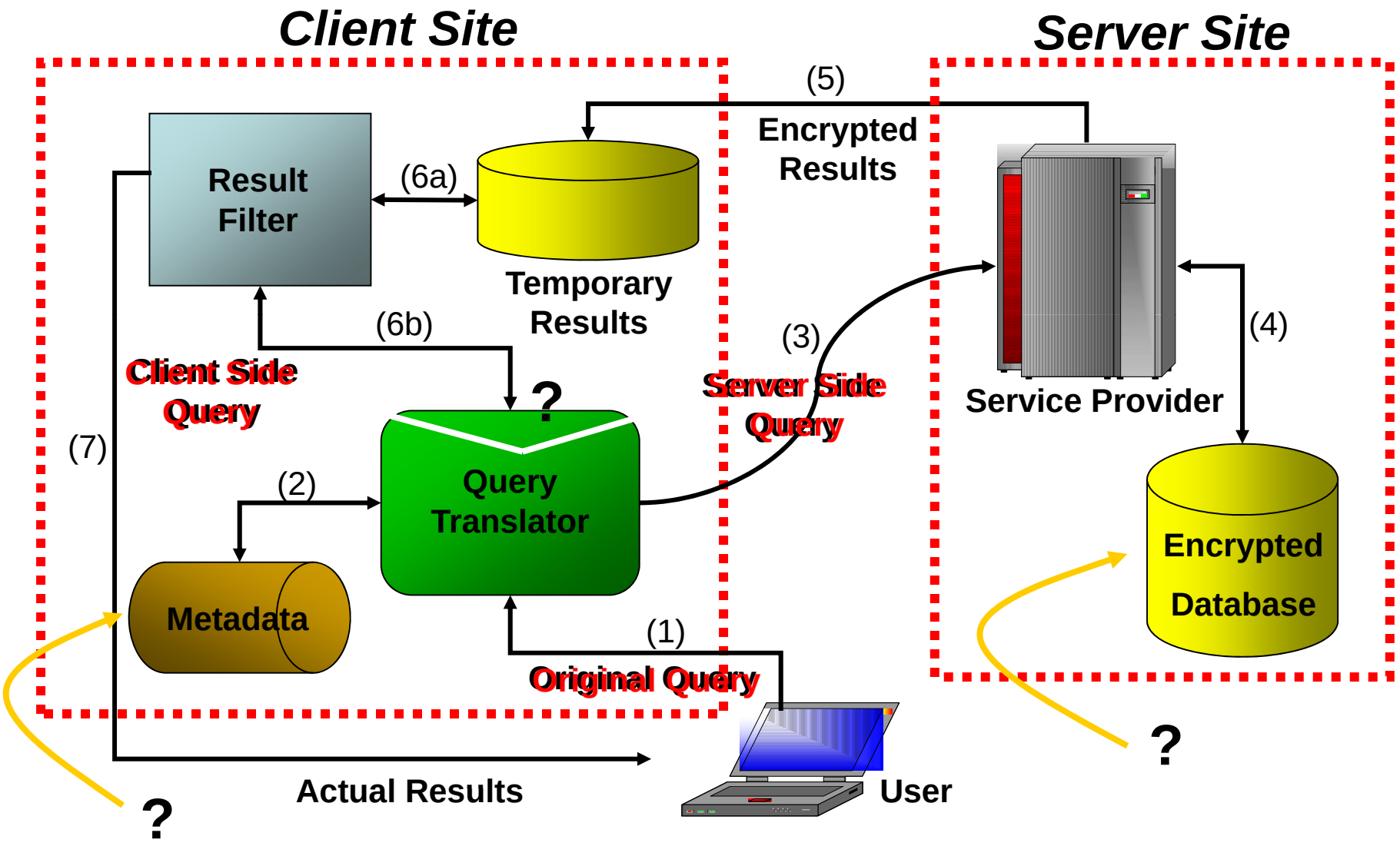
c. Application-level encryption

SQL Server TDE (Transparent Data Encryption)
Oracle 10g/11g TDE

Can we offer better performance?

- We DO NOT fully trust the service provider with sensitive information
  - Encrypt client's data and store at server
  - Client:
    - runs queries over encrypted remote data
    - verifies integrity/authenticity of results (covered in the last lecture)
- Most of the processing work to be done by the server
- Consider passive adversary
  - A malicious individual who has access to data but only tries to learn sensitive information about the data without actively modifying it or disrupting any kind of services

Service Provider Architecture



H. Hacigumus, B. R. Iyer, C. Li, S. Mehrotra: Executing SQL over encrypted data in the database-service-provider model. 2002 International Conference on Management of Data (SIGMOD'2002), 216-227

Query Processing 101…

- At its core, query processing consists of:
  - Logical comparisons (> , <, = , <=, >=)
  - Pattern based queries (e.g., *Arnold*egger*)
  - Simple arithmetic (+, *, /, ^, log)
- Higher level operators implemented using the above
  - Joins
  - Selections
  - Unions
  - Set difference
  - …
- To support any of the above over encrypted data, need to have mechanisms to support **basic** operations over encrypted data

Searching over Encrypted Data

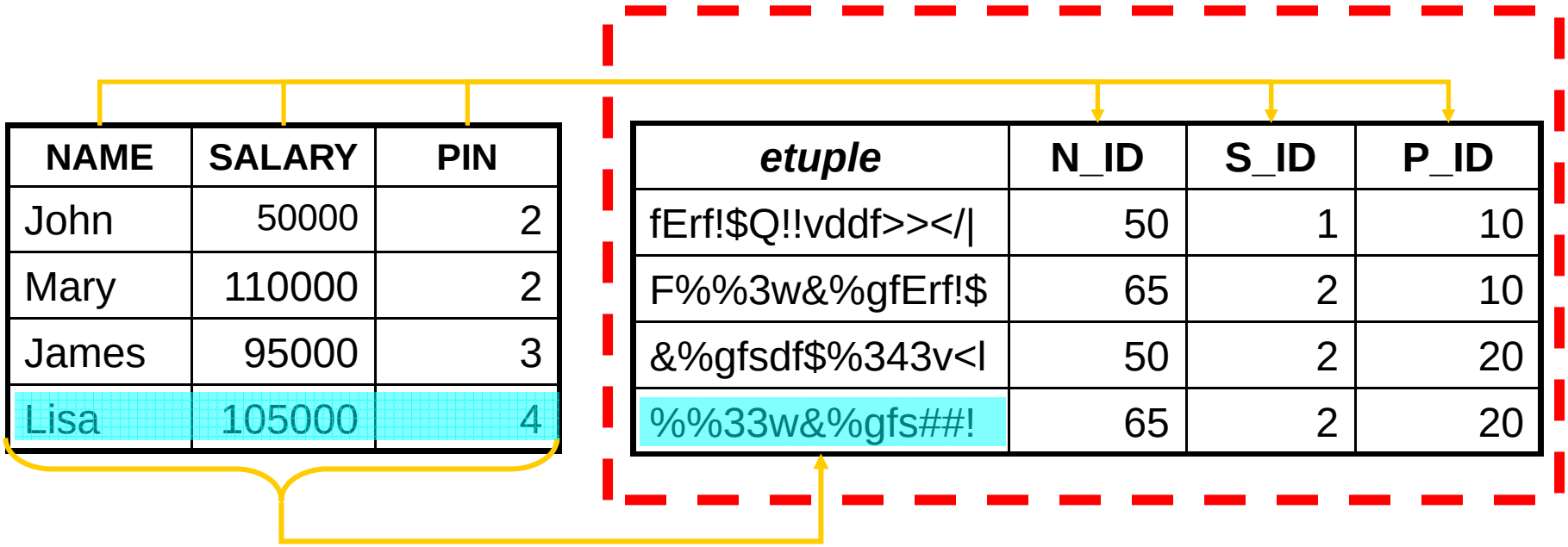
- Want to be able to perform operations over encrypted data (for efficiency)

```
SELECT AVG(E.salary)
FROM EMP
WHERE age > 55
```

- Fundamental observations
  - Basic operations do not need to be fully implemented over encrypted data
  - To test (AGE > 55), it might suffice to devise a strategy that allows the test to succeed in most cases (might not work in all cases)
  - If test does not result in a clear positive or negative over encrypted representation, resolve later at client-side, after decryption.

Relational Encryption

Server Site



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

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

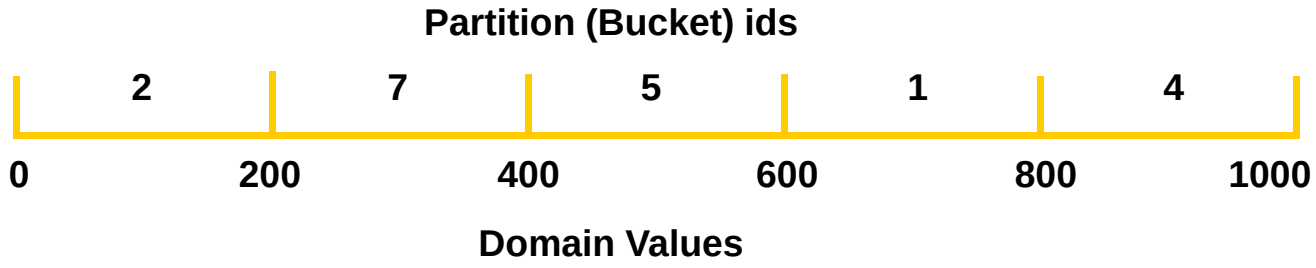
- Store an encrypted string – *etuple* – for each tuple in the original table
  - This is called “row level encryption”
  - Any kind of encryption technique (e.g., AES, DES) can be used
- Create an index for each (or *selected*) attribute(s) in the original table

Building the Index

- *Partition function* divides domain values into partitions (buckets)

  Partition (*R.A*) = { [0,200], (200,400], (400,600], (600,800], (800,1000] }
  - partition function has impact on performance as well as privacy
  - very much domain/attribute dependent
  - equi-width vs. equi-depth partitioning

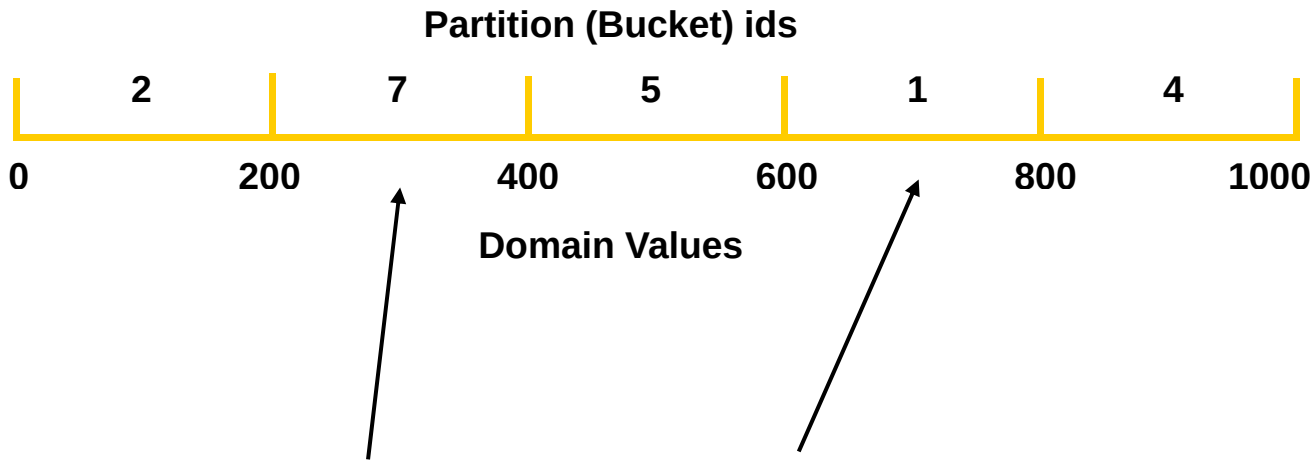
- *Identification function* assigns a partition id to each partition of attribute *A*



  - e.g. $ident_{R.A}$((200,400]) = 7
  - Any function can be use as identification function, e.g., hash functions
  - Client keeps partition and identification functions secret (as **metadata**)

Building the Index

- *Mapping function* maps a value ***v*** in the domain of attribute *A* to partition id



– **e.g.,** $Map_{R.A}(250) = 7$ $Map_{R.A}(620) = 1$

Storing Encrypted Data

$R = < A, B, C > \Rightarrow R^S = <$ etuple, A_id, B_id, C_id $>$

etuple = *encrypt* (A | B | C)

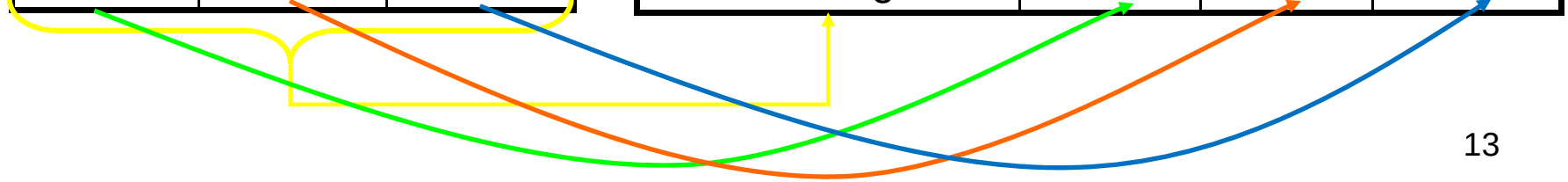
A_id = $Map_{R.A}$(A), B_id = $Map_{R.B}$(B), C_id = $Map_{R.C}$(C)

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



Referring back to our example

```
SELECT AVG(E.salary)
FROM EMP
WHERE age > 55
```

- Suppose the partitions on age are as follows: P1 - [20,30); P2 - [30,40); P3 - [40,50); P4 - [50,60); P5 - [60,100)
- To test (AGE > 55), it suffices to retrieve all data that falls into partitions that contain at least one employee with age > 55
  - P4 and P5
  - These partitions (e.g., P4) may contain records with age $\leq$ 55; they can be examined at the client-side after records are decrypted.
- Records belonging to partitions that contain only employees with age $\leq$ 55 (e.g., P1, P2 and P3) will not need to be returned.

Mapping Conditions

Q: SELECT name, pname FROM employee, project
WHERE employee.pin=project.pin AND salary>100k

- Server stores attribute indices determined by mapping functions
- Client stores metadata and uses it to translate the query

Conditions:

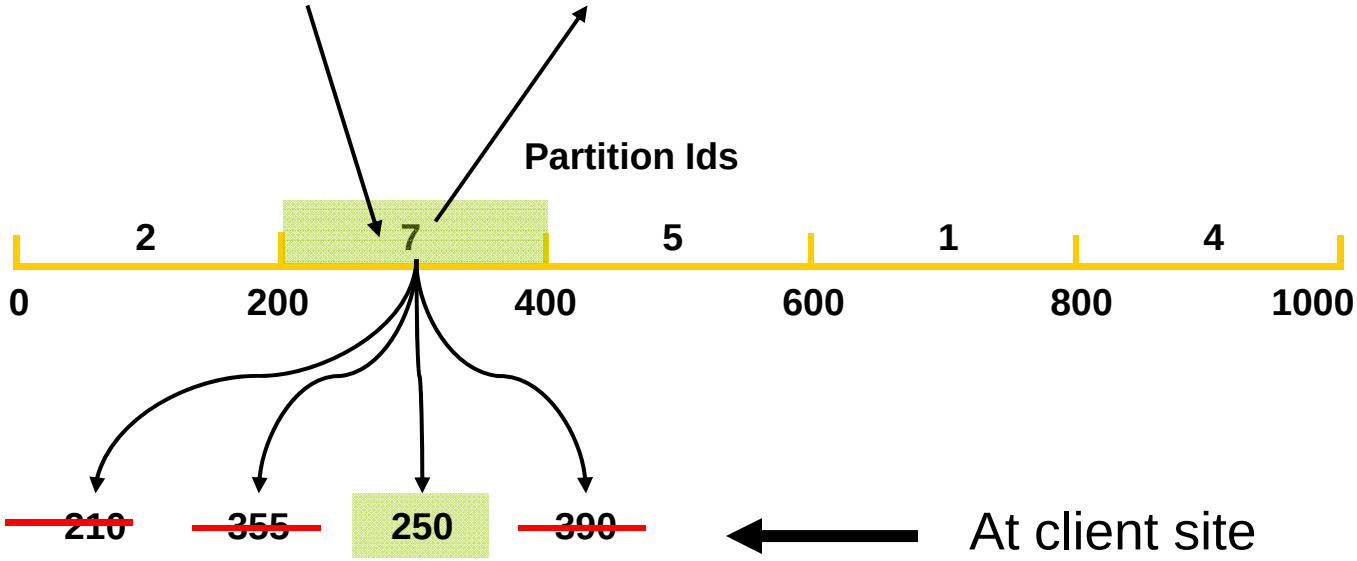
- Condition ← Attribute *op* Value
- Condition ← Attribute *op* Attribute
- Condition ← (Condition ∨ Condition) | (Condition ∧ Condition)
  | (not Condition)

Where *op* = { = , >, ≥, <, ≤ }

Mapping Conditions (2)

Example: Equality

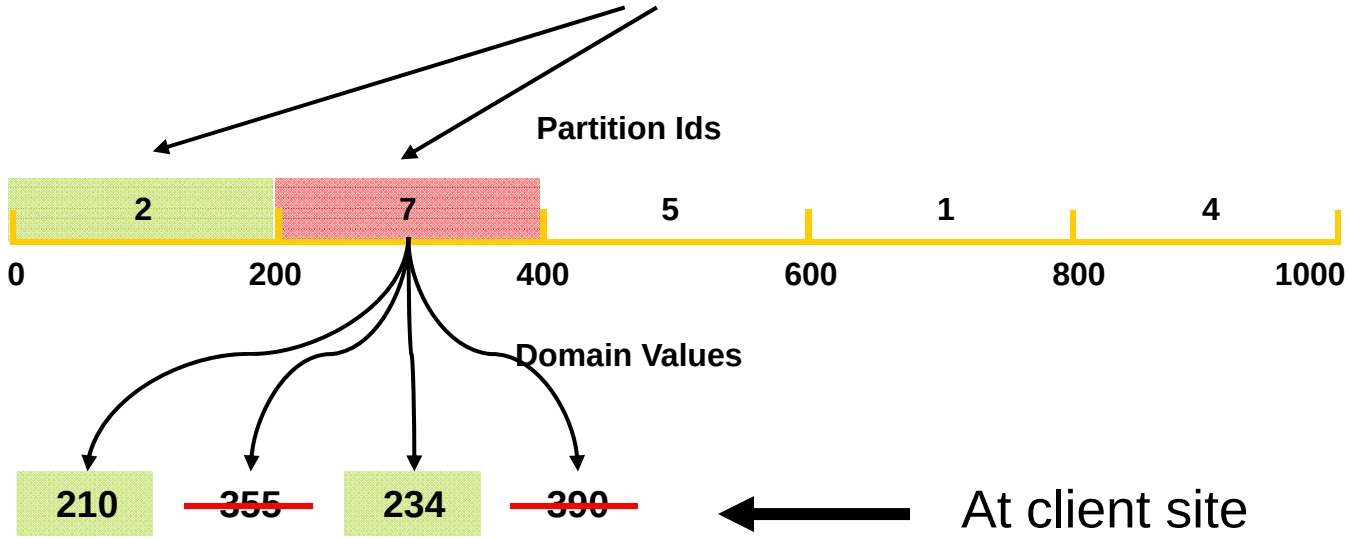
- Attribute = Value
  - $Map_{cond}(A = \boldsymbol{v}) \Rightarrow A^{S} = Map_{A}(\boldsymbol{v})$
  - $Map_{cond}(A = 250) \Rightarrow A^{S} = 7$



Mapping Conditions (3)

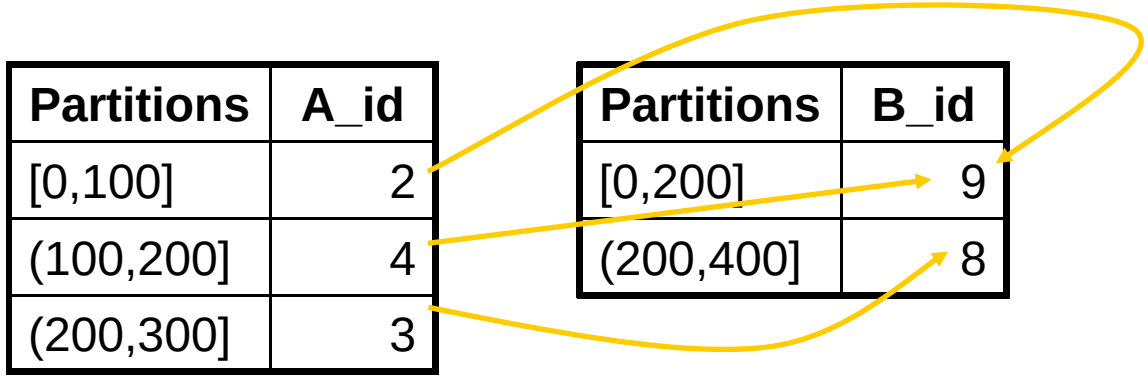
Example: Inequality (<, >, etc.)

- Attribute < Value
  - $Map_{cond}(A < v) \Rightarrow A^S \in \{ ident_A(p_j) \mid p_j.low \leq v) \}$
  - $Map_{cond}(A < 250) \Rightarrow A^S \in \{2,7\}$



Mapping Conditions (4)

- Attribute1 = Attribute2 (useful for JOIN-type queries)
  - $Map_{cond}(A = B) \Rightarrow \bigvee_N (A^S = ident_A(\boldsymbol{p_k}) \wedge B^S = ident_B(\boldsymbol{p_l}))$
    where N is $\boldsymbol{p_k} \in$ *partition* (A), $\boldsymbol{p_l} \in$ *partition* (B), $\boldsymbol{p_k} \cap \boldsymbol{p_l} \neq \varnothing$



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

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

C : A = B $\Rightarrow$ C' : (A_id = 2 $\wedge$ B_id = 9)
$\vee$ (A_id = 4 $\wedge$ B_id = 9)
$\vee$ (A_id = 3 $\wedge$ B_id = 8)

Relational Operators over Encrypted Relations

- Partition the computation of the operators across client and server
- Compute (possibly) superset of answers at the server
- Filter the answers at the client
- *Objective* : *minimize the work at the client* and process the answers as soon as they arrive *requiring minimal storage* at the client

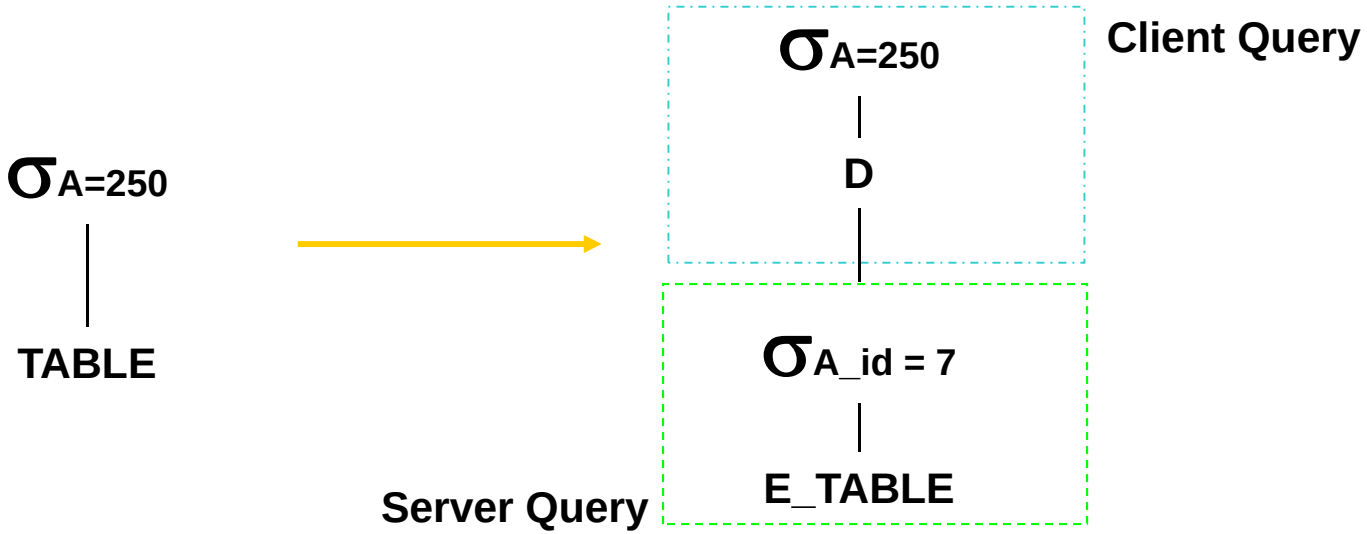
Operators:

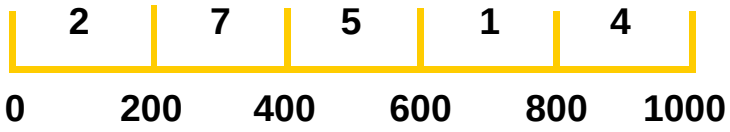
- **Selection**
- **Join**
- **Grouping and Aggregation**
- Others: Sort, duplicate elimination, set difference, union, projection

Selection Operator

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

Example:



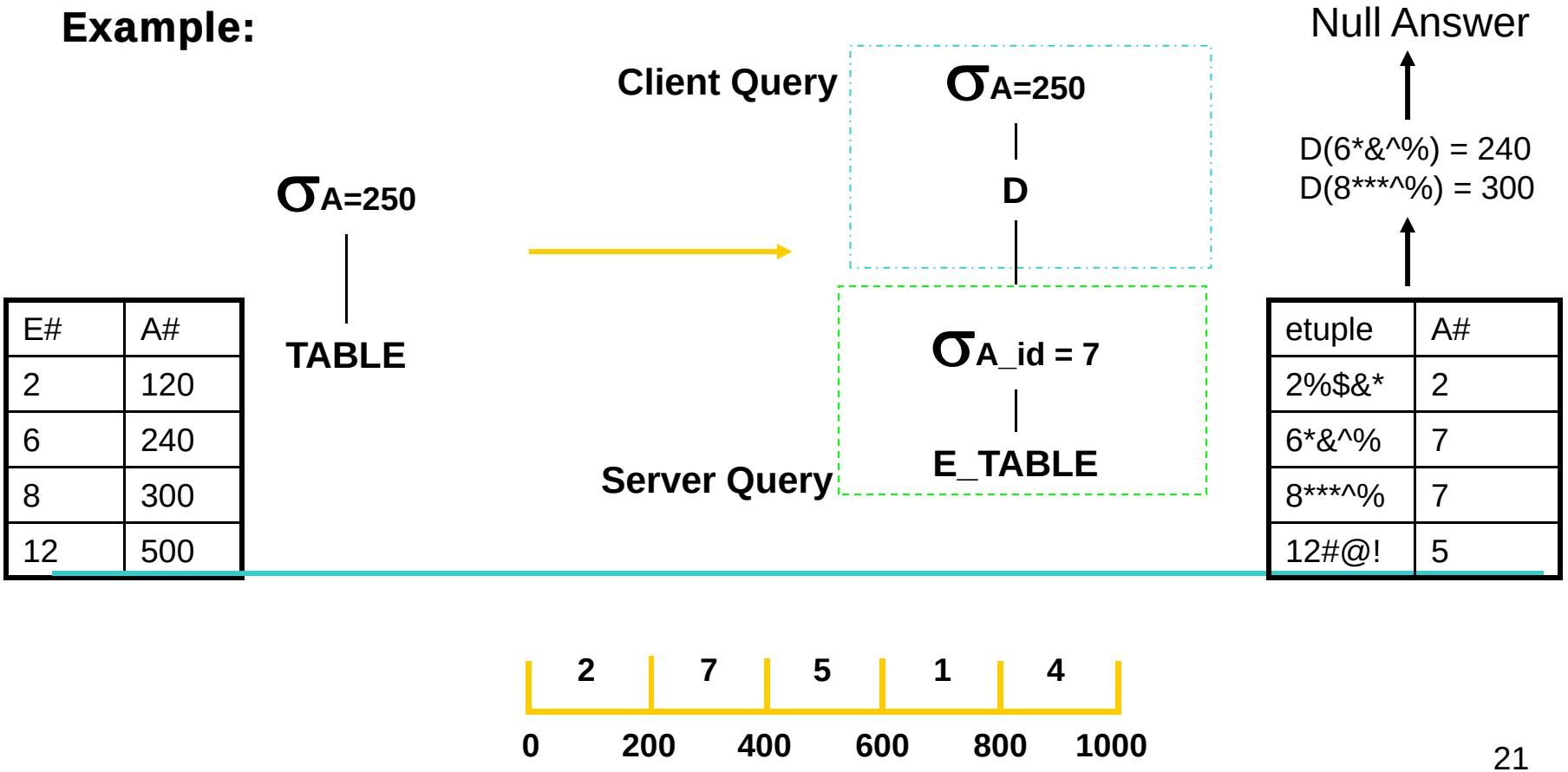


Selection Operator

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

D = Decrypt

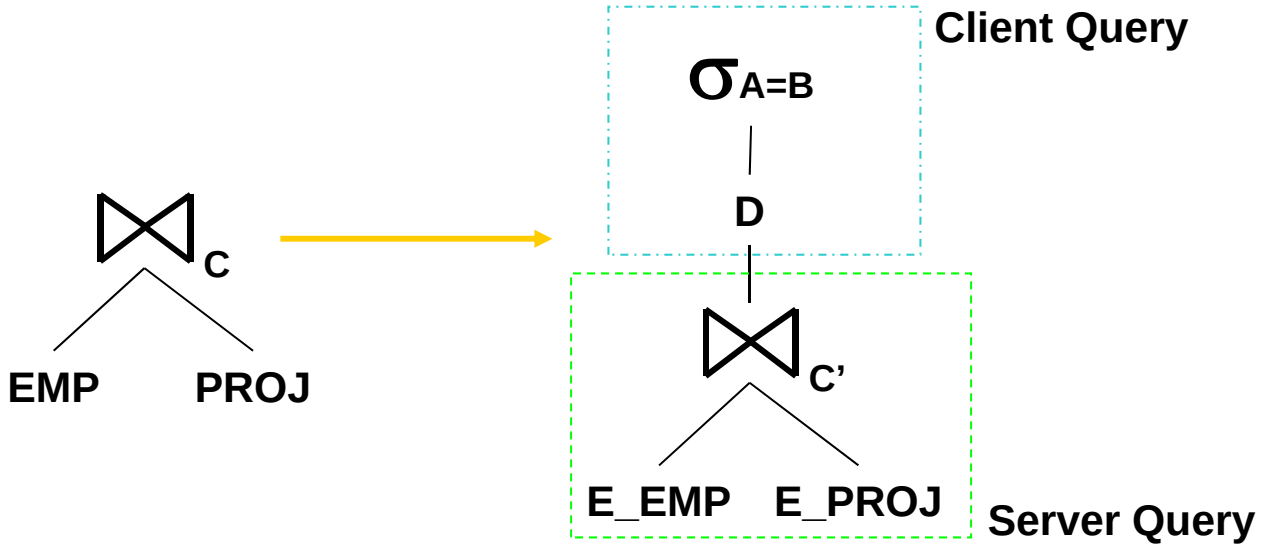
Example:



Join Operator

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

Example:



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

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

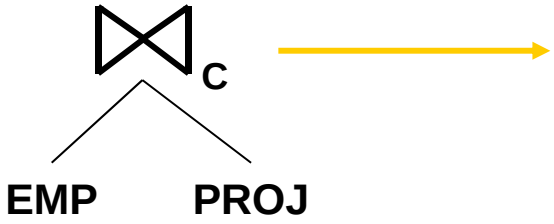
$C : A = B \Rightarrow C' : (A\_id = 2 \wedge B\_id = 9)$
$\vee (A\_id = 4 \wedge B\_id = 9)$
$\vee (A\_id = 3 \wedge B\_id = 8)$

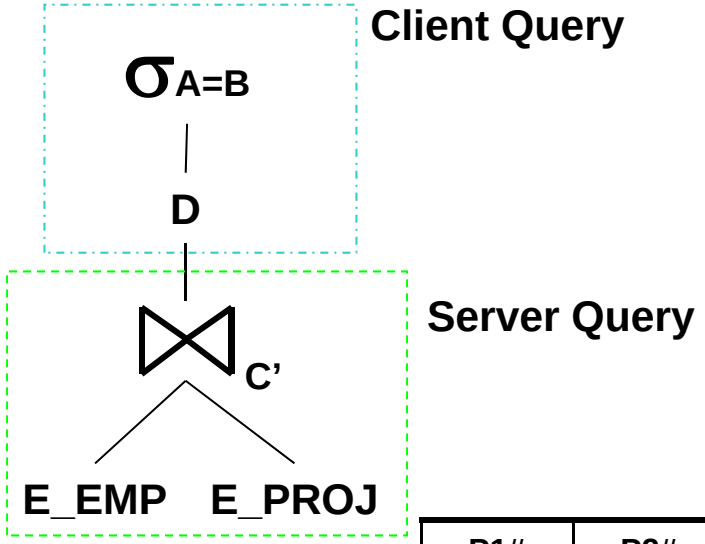
$C : A = B \Rightarrow C' : (A\_id = 2 \wedge B\_id = 9) \vee (A\_id = 4 \wedge B\_id = 9) \vee (A\_id = 3 \wedge B\_id = 8)$

Join Operator

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

Example:





P1#	Partition	A_id
10	(0,100]	2
30	(0,100]	2
120	(100,200]	4
250	(200,300]	3

P2#	Partition	B_id
70	(0,200]	9
120	[0,200]	9
220	(200,400]	8

Condition:
P1# = P2#

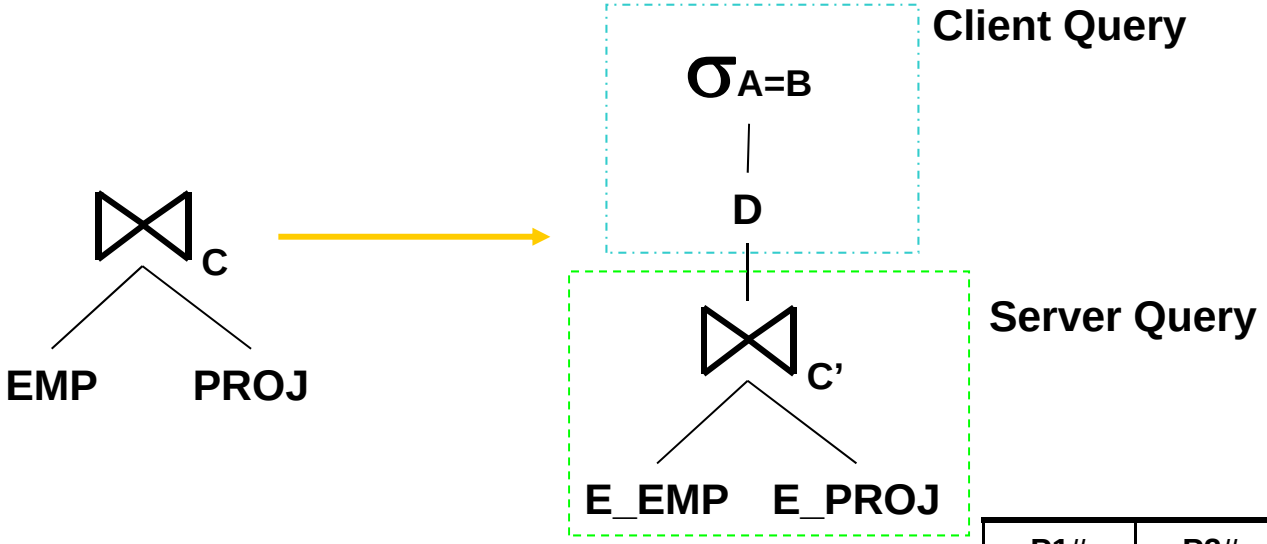
P1#	P2#
10	70
10	120
30	70
30	120
120	70
120	120
250	220

$C : A = B \Rightarrow C' : (A\_id = 2 \wedge B\_id = 9) \vee (A\_id = 4 \wedge B\_id = 9) \vee (A\_id = 3 \wedge B\_id = 8)$

Join Operator

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

Example:



P1#	Partition	A_id
10	(0,100]	2
30	(0,100]	2
120	(100,200]	4
250	(200,300]	3

P2#	Partition	B_id
70	(0,200]	9
120	[0,200]	9
220	(200,400]	8

Condition:
P1# = P2#

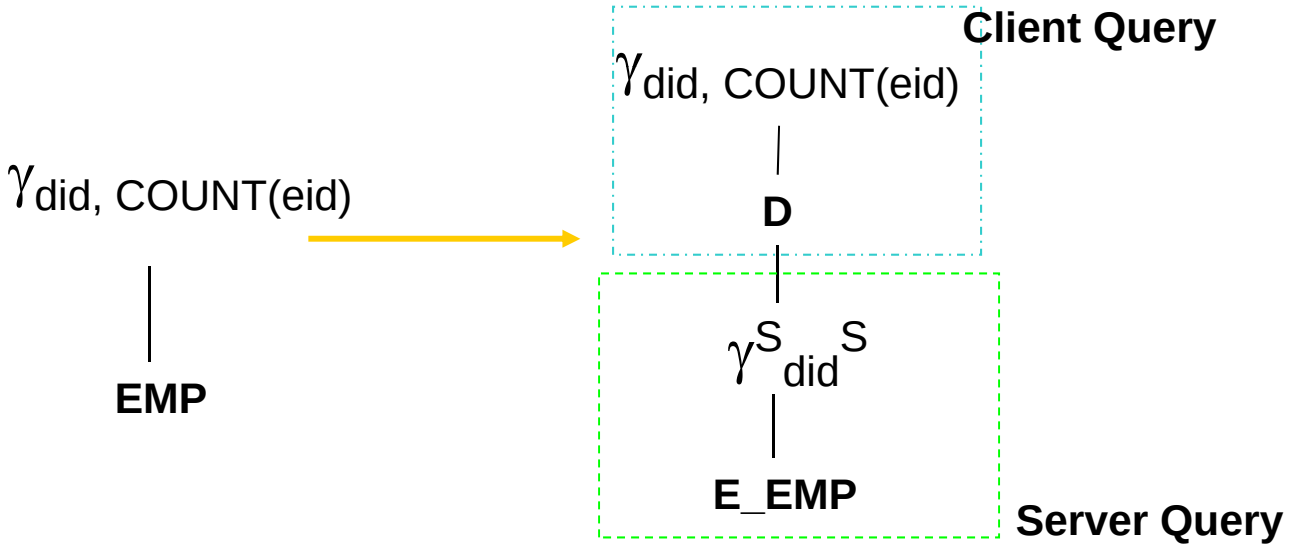
P1#	P2#
10	70
10	120
30	70
30	120
120	70
120	**120**
250	220

Grouping & Aggregation Operator

$$\gamma_L(R) = \gamma_L(\mathbf{D} (\gamma_{L'} (R^S)))$$

where L = {grouping attributes} ∪ {aggregate operations}

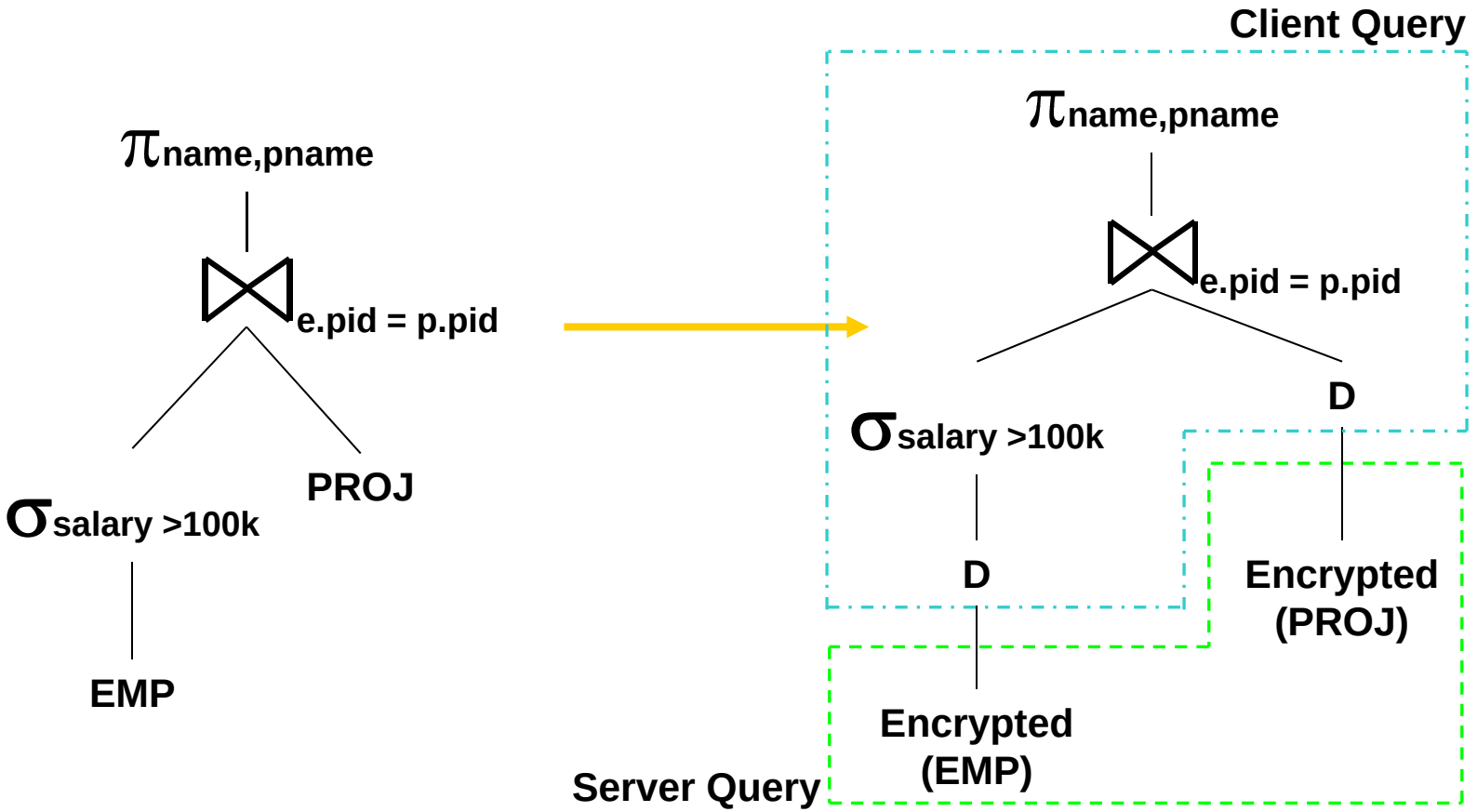
Example:



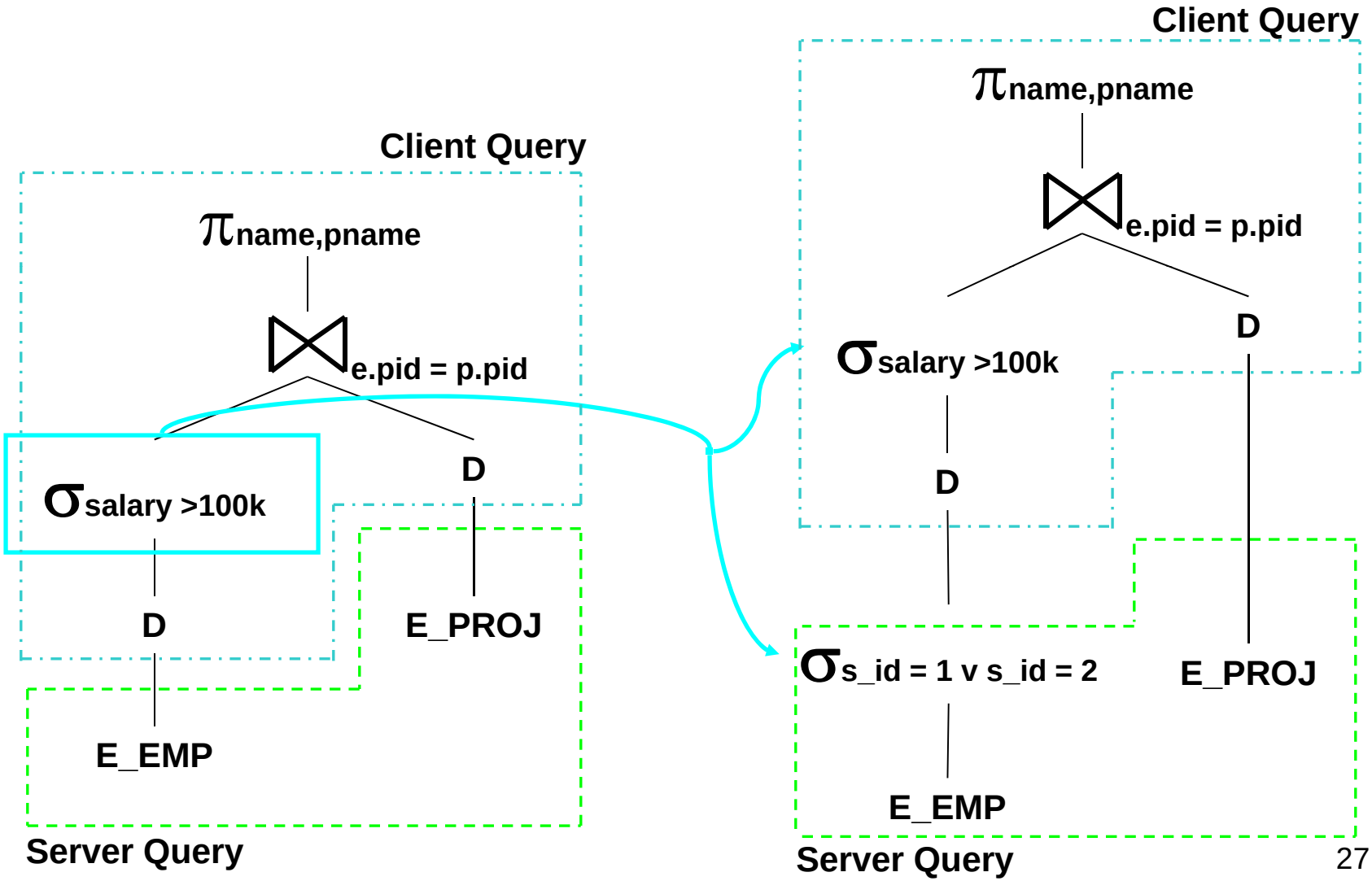
a) Partial sorting done at server
b) No gain in terms of communication, but client side saves up on sorting

Query Decomposition

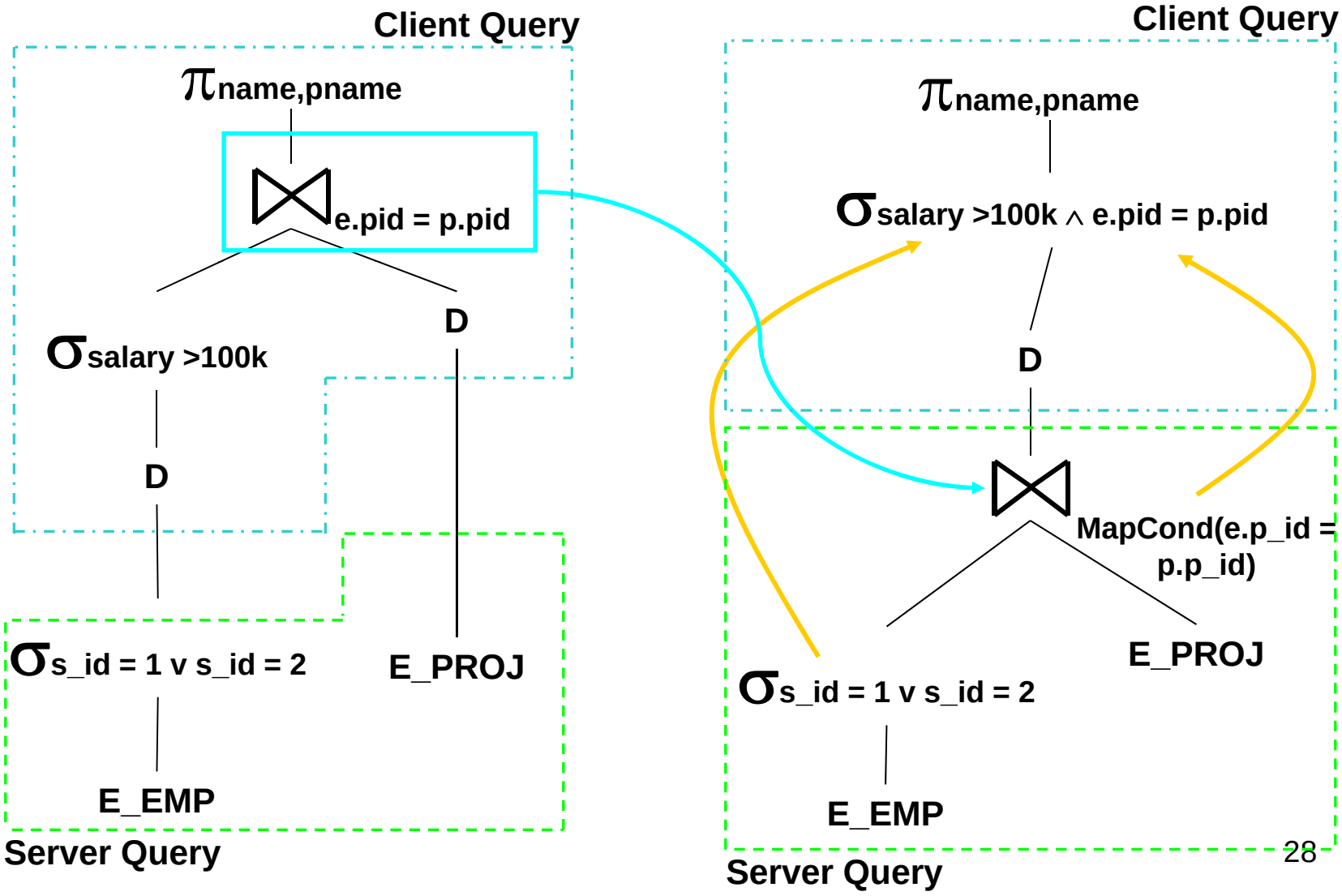
Q: SELECT name, pname FROM emp, proj
WHERE emp.pid=proj.pid AND salary > 100k



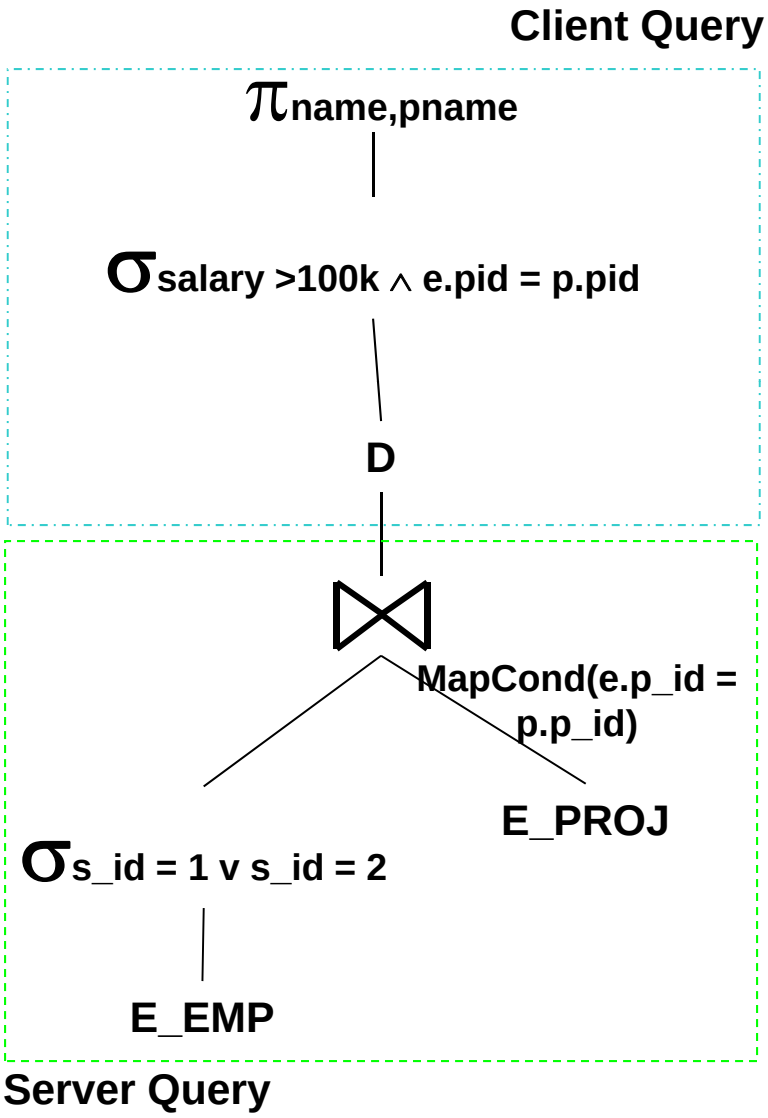
Query Decomposition (2)



Query Decomposition (3)



Query Decomposition (4)



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

Temp = Decrypted intermediate result

Query Precision vs. Privacy

- Observation:

Allocating a large number of buckets to crypto-indices increases query precision but reduces privacy. On the other hand, a small number of buckets increases privacy but adversely affects performance.

The goal of the client is thus twofold:

Server Efficiency: maximize the server-side accuracy of range query evaluation. Higher efficiency results in lower server-client communication overhead and lower post-processing costs for the client.

Maximum Privacy: minimize the information revealed to the server through the crypto-indices. In other words, maximize data privacy.

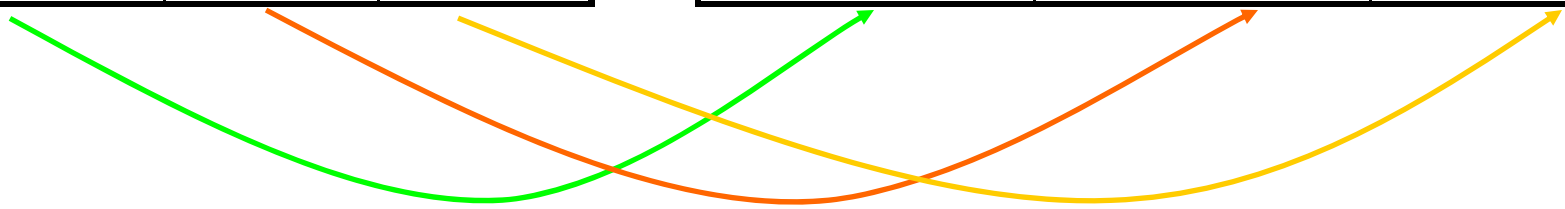
Fine Encryption Granularity

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^%$



H. Hacigumus, B. R. Iyer, C. Li, S. Mehrotra: Efficient Execution of Aggregation Queries over Encrypted Relational Databases. DASFAA04, 125-136

Can we do better with aggregation?

- Use homomorphic encryption functions
  - E (encryption function), D (decryption function)
  - $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ (functions on plaintext),
  - $\beta = \{\beta_1, \beta_2, \ldots \beta_n\}$ (functions on encrypted data)
  - (E, D, α, β) is a privacy homomorphism if
    - $D(\beta_i(E(a_1),E(a_2),\ldots, E(a_m))) = \alpha_i(a_1, a_2, \ldots, a_m)$

Aggregation over encrypted data

- One such scheme
  - Key k = (p,q), p & q are prime numbers chosen by client used for encryption/decryption (hidden from server)
  - N = p.q, revealed to server
  - $E_k(a) = (a \bmod p, a \bmod q)$, $a \in Z_N$
  - $D_k(d_1, d_2) = d_1 q q^{-1} + d_2 p p^{-1} \pmod N$
    - $q q^{-1} = 1 \pmod p$, $p p^{-1} = 1 \pmod q$
  - $\alpha = \{-_n, +_n, \times_n\}$
  - $\beta = \{-, +, \times\}$

Aggregation over encrypted data

- Example
  - p = 5, q = 7, so N = pq = 35, k = (5, 7)
  - Suppose we want to add a1=5 and a2=6
  - $E_k(5) = (0,5)$, $E_k(6) = (1,6)$ (stored in server)
  - At server
    - Compute $E_k(5) + E_k(6) = (1,11)$
  - At client
    - Decrypts (1,11) = (1.7.3 + 11.5.3) (mod 35) = 11 = 5 + 6!

In relational DBMS

- For each attribute A that will be used in aggregation, create two fields to encode E(a), a $\in$ domain(A), e.g., for salary, we create Sp = salary mod p and Sq = salary mod q
- Now SUM(salary + commission) can be processed at the server as
  - SELECT SUM(Sp+Cp) as s1, SUM(Sq+Cq) as s2 FROM EMP^S
- Client decrypts result as
  - $S1*q*q^{-1} + s2*p*p^{-1} \pmod{p*q}$

Table: EMPLOYEE

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

Table: $EMPLOYEE^S$

Etuple	Sp	Sq	Cp	Cq
fErf!$Q!!vddf	34	24	63	23
F%%3w&%g	56	26	34	22
&%gfsdf$%3	25	55	47	44
%%33w&%gf	86	33	42	92

Complete example

eid	name	salary	city	did
23	Tom	70K	Maple	10
860	John	60K	Maple	55
320	Jim	23K	River	35
875	Tim	45K	Maple	58
870	Mary	40K	Maple	10
200	Susan	45K	Ruver	10

Salary	
Paritions	**ID**
0-25K	59
25K-50K	49
50K-75K	81
75K-100K	7

Salary^{PH}

etuple	S_id	City_id	Did_id	E_city	E_did	Sal_p	Sal_q
fErf!$Q!	81	18	2	**^((	@R@*	7	27
F%%3g	81	18	3	**^((	(&%^4	18	17
&%gfsd	59	22	4	ll23^	$(7%$	2	23
^#$#%^	49	18	3	**^((	#%&*9	3	2
%%33w	49	18	2	**^((	@R@*	8	7
fErf!Q!!	49	22	2	ll23^	@R@*	13	12

Select SUM(Salary) FROM emp, mgr
WHERE city=Maple AND salary< 65K and emp.did = mgr.did

- For city=Maple, we use E_city
- For salary < 65K, we use
  - S_id = 49 OR S_id = 59 (no false drop)
  - S_id = 81 (false drop exists)
- For emp.did = mgr.did, we use E_did

- So, we can have TWO subqueries at the server (why?)
  - SELECT $SUM^{PH}(Salary^{PH})$ FROM emp^{S}, mgr^{S} WHERE E_city=E(Maple) AND (S_id=49 OR S_id=59) AND emp^{S}.E_did=mgr^{S}.E_did
  - SELECT emp^{S}.etuple FROM emp^{S}, mgr^{S} WHERE E_city=E(Maple) AND S_id=81 AND emp^{S}.E_did=mgr^{S}.E_did
- Client?

Summary

- Store encrypted data at server
- Process as much at server as possible, and postprocess at client
- Storage cost is higher (hash values can be as large as the original values)
- Leak some information
  - number of distinct values, which records have the same values in certain attribute, which records are join-able,
  - violate access control
- Effectiveness depends on the partitioning/index granularity