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 Encrypt 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 Encrypt 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

- **Storage-level encryption**
  - Client encrypts the data and stores it on the server.
  - Client can run queries over the encrypted remote data.
  - Server performs most of the processing work.

- **Database-level encryption**
  - Similar to storage-level encryption, but with a focus on the database layer.

- **Application-level encryption**
  - The client encrypts the data and sends it to the server.
  - The client verifies the integrity and authenticity of the results.

**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.
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)
  
  ```sql
  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

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>50000</td>
<td>2</td>
</tr>
<tr>
<td>Mary</td>
<td>110000</td>
<td>2</td>
</tr>
<tr>
<td>James</td>
<td>95000</td>
<td>3</td>
</tr>
<tr>
<td>Lisa</td>
<td>105000</td>
<td>4</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>etuple</th>
<th>N_ID</th>
<th>S_ID</th>
<th>P_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>fErfl$Q!vddf&gt;&gt;&amp;/</td>
<td>50</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>F%3w&amp;gfErfl$</td>
<td>65</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>&amp;gfdf$%343v&lt;l</td>
<td>50</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>%33w%gf##!</td>
<td>65</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>
```

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

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

  - e.g., \(\text{Map}_{R.A}(250) = 7\) \(\text{Map}_{R.A}(620) = 1\)
Storing Encrypted Data

\[ R = \langle A, B, C \rangle \implies R^s = \langle \text{etuple}, A_{id}, B_{id}, C_{id} \rangle \]

\[ \text{etuple} = encrypt\ (A \mid B \mid C) \]

\[ A_{id} = Map_{R,A}(A), B_{id} = Map_{R,B}(B), C_{id} = Map_{R,C}(C) \]

Table: EMPLOYEE

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>50000</td>
<td>2</td>
</tr>
<tr>
<td>Mary</td>
<td>110000</td>
<td>2</td>
</tr>
<tr>
<td>James</td>
<td>95000</td>
<td>3</td>
</tr>
<tr>
<td>Lisa</td>
<td>105000</td>
<td>4</td>
</tr>
</tbody>
</table>

Table: EMPLOYEE$

<table>
<thead>
<tr>
<th>Name</th>
<th>N_ID</th>
<th>S_ID</th>
<th>P_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erf6Qvddf!</td>
<td>50</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>F%-%3w&amp;%gfErf$</td>
<td>65</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>&amp;%gfsdf$%.343v&lt;l</td>
<td>50</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>%%33w&amp;%gfs##!</td>
<td>65</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

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 ≤ 55; they can be examined at the client-side after records are decrypted.
- Records belonging to partitions that contain only employees with age ≤ 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 $\leftarrow$ Attribute $op$ Value
- Condition $\leftarrow$ Attribute $op$ Attribute
- Condition $\leftarrow$ (Condition $\lor$ Condition) | (Condition $\land$ Condition)
  | (not Condition)

Where $op = \{ =, >, \geq, <, \leq \}$

Mapping Conditions (2)

Example: Equality

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

Partition Ids

At client site
Mapping Conditions (3)

Example: Inequality (<, >, etc.)

- Attribute < Value
  - $\text{Map}_{\text{cond}}( A < v ) \Rightarrow A^\delta \in \{ \text{ident}_A(p_j) \mid p_j.\text{low} \leq v \}$
  - $\text{Map}_{\text{cond}}( A < 250 ) \Rightarrow A^\delta \in \{2,7\}$

Partition Ids

Domain Values

At client site

Mapping Conditions (4)

- Attribute1 = Attribute2 (useful for JOIN-type queries)
  - $\text{Map}_{\text{cond}}( A = B ) \Rightarrow \bigvee_N (A^\delta = \text{ident}_A(p_k) \land B^\delta = \text{ident}_B(p_l))$
  - where $N$ is $p_k \in \text{partition}(A)$, $p_l \in \text{partition}(B)$, $p_k \cap p_l \neq \emptyset$

<table>
<thead>
<tr>
<th>Partitions</th>
<th>A_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,100]</td>
<td>2</td>
</tr>
<tr>
<td>(100,200)</td>
<td>4</td>
</tr>
<tr>
<td>(200,300]</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partitions</th>
<th>B_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,200]</td>
<td>9</td>
</tr>
<tr>
<td>(200,400]</td>
<td>8</td>
</tr>
</tbody>
</table>

$C : A = B \quad \Rightarrow \quad C' : (A_{id} = 2 \land B_{id} = 9)
\lor (A_{id} = 4 \land B_{id} = 9)
\lor (A_{id} = 3 \land B_{id} = 8)$
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**

\[ \sigma_c(R) = \sigma_c(D(\sigma_{\text{Mapcond}(c)}(R^S))) \]

\( D = \text{Decrypt} \)

**Example:**

- Client Query: \( \sigma_{A=250} \)
- Server Query: \( \sigma_{A_id = 7} \)

![Diagram of selection operator example]
Selection Operator

\[ \sigma_c(R) = \sigma_c(D(\sigma_{Mapcond(c)}(R^S))) \]

**Example:**

<table>
<thead>
<tr>
<th>E#</th>
<th>A#</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>240</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
</tr>
<tr>
<td>12</td>
<td>500</td>
</tr>
</tbody>
</table>

**Null Answer**

\[ D(6**8%) = 240 \]
\[ D(8****%) = 300 \]

**Client Query**

\[ \sigma_{A=250}(\sigma_{A_id=7}(E_TABLE)) \]

**Server Query**

\[ \sigma_{A_id=7}(E_TABLE) \]

**Join Operator**

\[ R \bowtie T = \sigma_c(D(R^S \bowtie_{Mapcond(c)} T^S)) \]

**Example:**

<table>
<thead>
<tr>
<th>Partitions A_id</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,100]</td>
<td>2</td>
</tr>
<tr>
<td>(100,200]</td>
<td>4</td>
</tr>
<tr>
<td>(200,300]</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partitions B_id</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,200]</td>
<td>9</td>
</tr>
<tr>
<td>(200,400]</td>
<td>8</td>
</tr>
</tbody>
</table>

**Client Query**

\[ \sigma_{A=B}(C') \]

**Server Query**

\[ E_EMP \bowtie E_PROJ \]

\[ C : A = B \Rightarrow C : \{ (A_id = 2 \land B_id = 9) \lor (A_id = 4 \land B_id = 9) \lor (A_id = 3 \land B_id = 8) \} \]
Join Operator

\[ R \bowtie T = \sigma_c^e ( \left( \pi^D ( R \bowtie^s T ) \right) \mapcond(c) ) \]

Example:

\[ \sigma_{A=B} \]

\[ EMP \quad PROJ \]

Server Query

\[ E_EMP \quad E_PROJ \]

Client Query

\[ EMP \quad PROJ \]

\[ \sigma_{A=B} \]

\[ D \]

\[ C' \]

\[ \sigma_{A=B} \]

\[ D \]

\[ C' \]

\[ \sigma_{A=B} \]

\[ D \]

\[ C' \]

Condition: \[ P1# = P2# \]

<table>
<thead>
<tr>
<th>P1#</th>
<th>Partition A_id</th>
<th>P2#</th>
<th>Partition B_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>(0,100]</td>
<td>2</td>
<td>(0,200]</td>
</tr>
<tr>
<td>30</td>
<td>(0,100]</td>
<td>2</td>
<td>(0,200]</td>
</tr>
<tr>
<td>120</td>
<td>(100,200]</td>
<td>4</td>
<td>(200,400]</td>
</tr>
<tr>
<td>250</td>
<td>(200,300]</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>(0,200]</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>(200,400]</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Join Operator

\[ R \bowtie T = \sigma_c^e ( \left( \pi^D ( R \bowtie^s T ) \right) \mapcond(c) ) \]

Example:

\[ \sigma_{A=B} \]

\[ EMP \quad PROJ \]

Server Query

\[ E_EMP \quad E_PROJ \]

Client Query

\[ EMP \quad PROJ \]

\[ \sigma_{A=B} \]

\[ D \]

\[ C' \]

\[ \sigma_{A=B} \]

\[ D \]

\[ C' \]

Condition: \[ P1# = P2# \]

<table>
<thead>
<tr>
<th>P1#</th>
<th>Partition A_id</th>
<th>P2#</th>
<th>Partition B_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>(0,100]</td>
<td>2</td>
<td>(0,200]</td>
</tr>
<tr>
<td>30</td>
<td>(0,100]</td>
<td>2</td>
<td>(0,200]</td>
</tr>
<tr>
<td>120</td>
<td>(100,200]</td>
<td>4</td>
<td>(200,400]</td>
</tr>
<tr>
<td>250</td>
<td>(200,300]</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>(0,200]</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>(200,400]</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

\[ C : A = B \Rightarrow C': (A_id = 2 \land B_id = 9) \]
\[ \lor (A_id = 4 \land B_id = 9) \]
\[ \lor (A_id = 3 \land B_id = 8) \]
Grouping & Aggregation Operator

\[ \gamma_L(R) = \gamma_L\left(\mathcal{D}\left(\gamma_L\left(R^1\right)\right)\right) \]

where \( L = \{\text{grouping attributes}\} \cup \{\text{aggregate operations}\} \)

Example:

\[ \gamma_{\text{did, COUNT(eid)}} \]

Client Query

\[ \mathcal{D} \]

Server Query

\[ \gamma_{\text{did, COUNT(eid)}} \]

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

Client Query

\[ \pi_{\text{name, pname}} \]

\[ \sigma_{\text{salary >100k}} \]

\[ e.pid = p.pfid \]

\[ \pi_{\text{name, pname}} \]

\[ \sigma_{\text{salary >100k}} \]

\[ e.pid = p.pfid \]

Server Query

Encrypted (EMP)

Encrypted (PROJ)
Query Decomposition (2)

Query Decomposition (3)
Query Decomposition (4)

Client Query

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

Qs: 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)

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

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 ac-
curacy 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 re-
vealed to the server through the crypto-indices. In
other words, maximize data privacy.
Fine Encryption Granularity

Table: EMPLOYEE

<table>
<thead>
<tr>
<th>NAME</th>
<th>SAL</th>
<th>COM</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>50000</td>
<td>5000</td>
</tr>
<tr>
<td>Mary</td>
<td>110000</td>
<td>11000</td>
</tr>
<tr>
<td>James</td>
<td>95000</td>
<td>9500</td>
</tr>
<tr>
<td>Lisa</td>
<td>105000</td>
<td>10500</td>
</tr>
</tbody>
</table>

Table: EMPLOYEE

<table>
<thead>
<tr>
<th>E_NAME</th>
<th>E_SAL</th>
<th>E_COM</th>
</tr>
</thead>
<tbody>
<tr>
<td>45ewt*(&amp;</td>
<td>3H45f33</td>
<td>*&amp;%*kk</td>
</tr>
<tr>
<td>(*#hJK(0</td>
<td>Ek%98*</td>
<td>IDE#$F</td>
</tr>
<tr>
<td>()&amp;%^JK</td>
<td>H^F(^7</td>
<td>%^g%6</td>
</tr>
<tr>
<td>324(&amp;^hj</td>
<td>(86&amp;&amp;h$</td>
<td>887%^$</td>
</tr>
</tbody>
</table>

Can we do better with aggregation?

- Use homomorphic encryption functions
  - $E$ (encryption function), $D$ (decryption function)
  - $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ (functions on plaintext),
  - $\beta = \{\beta_1, \beta_2, \ldots, \beta_n\}$ (functions on encrypted data)
  - $(E, D, \alpha, \beta)$ 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)
  – \( N = pq \), revealed to server
  – \( E_k(a) = (a \mod p, a \mod q), a \in \mathbb{Z}_N \)
  – \( D_k(d_1, d_2) = d_1qq^{-1} + d_2pp^{-1} \pmod{N} \)
    • \( qq^{-1} = 1 \pmod{p} \), \( pp^{-1} = 1 \pmod{q} \)
  – \( \alpha = \{-n, +n, \times n\} \)
  – \( \beta = \{-, +, \times\} \)

• Example
  – \( p = 5, q = 7 \), so \( N = pq = 35 \), \( k = (5, 7) \)
  – Suppose we want to add \( a_1=5 \) and \( a_2=6 \)
  – \( E_k(5) = (0, 5), E_k(6) = (1, 6) \) (stored in server)
  – At server
    • Compute \( E_k(5) + E_k(6) = (1, 11) \)
  – At client
    • Decrypts \( (1, 11) = (1.7.3 + 11.5.3) \pmod{35} = 11 = 5 + 6! \)
In relational DBMS

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>SAL</th>
<th>COM</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>50000</td>
<td>5000</td>
</tr>
<tr>
<td>Mary</td>
<td>110000</td>
<td>11000</td>
</tr>
<tr>
<td>James</td>
<td>95000</td>
<td>9500</td>
</tr>
<tr>
<td>Lisa</td>
<td>105000</td>
<td>10500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SAL</th>
<th>COM</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>50000</td>
<td>5000</td>
</tr>
<tr>
<td>Mary</td>
<td>110000</td>
<td>11000</td>
</tr>
<tr>
<td>James</td>
<td>95000</td>
<td>9500</td>
</tr>
<tr>
<td>Lisa</td>
<td>105000</td>
<td>10500</td>
</tr>
</tbody>
</table>

- Complete example

<table>
<thead>
<tr>
<th>eid</th>
<th>name</th>
<th>salary</th>
<th>city</th>
<th>did</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Tom</td>
<td>70K</td>
<td>Maple</td>
<td>10</td>
</tr>
<tr>
<td>860</td>
<td>John</td>
<td>60K</td>
<td>Maple</td>
<td>55</td>
</tr>
<tr>
<td>320</td>
<td>Jim</td>
<td>23K</td>
<td>River</td>
<td>35</td>
</tr>
<tr>
<td>875</td>
<td>Tim</td>
<td>45K</td>
<td>Maple</td>
<td>58</td>
</tr>
<tr>
<td>870</td>
<td>Mary</td>
<td>40K</td>
<td>Maple</td>
<td>10</td>
</tr>
<tr>
<td>200</td>
<td>Susan</td>
<td>45K</td>
<td>River</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SalaryPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paritions</td>
</tr>
<tr>
<td>0-25K</td>
</tr>
<tr>
<td>25K-50K</td>
</tr>
<tr>
<td>50K-75K</td>
</tr>
<tr>
<td>75K-100K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>etuple</th>
<th>S_id</th>
<th>City_id</th>
<th>Did_id</th>
<th>E_city</th>
<th>E_did</th>
<th>Sal_p</th>
<th>Sal_q</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Erfl$Q!</td>
<td>81</td>
<td>18</td>
<td>2</td>
<td>**^((</td>
<td>@R@*</td>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>F%%3g</td>
<td>81</td>
<td>18</td>
<td>3</td>
<td>**^((</td>
<td>&amp;%^4</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>&amp;%gsd</td>
<td>59</td>
<td>22</td>
<td>4</td>
<td>ll23^</td>
<td>$(%$</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>^#$%^</td>
<td>49</td>
<td>18</td>
<td>3</td>
<td>**^((</td>
<td>&amp;%&amp;9</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>%%%3w</td>
<td>49</td>
<td>18</td>
<td>2</td>
<td>**^((</td>
<td>@R@*</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>@ErflQ!</td>
<td>49</td>
<td>22</td>
<td>2</td>
<td>ll23^</td>
<td>@R@*</td>
<td>13</td>
<td>12</td>
</tr>
</tbody>
</table>
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(Salary) FROM empS, mgrS WHERE E_city=E(Maple)
    AND (S_id=49 OR S_id=59) AND empS.E_did=mgrS.E_did
  – SELECT empS.setuple FROM empS, mgrS WHERE E_city=E(Maple)
    AND S_id=81 AND empS.E_did=mgrS.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