**Dr. Ammar Haider**
Assistant Professor
School of Computing

# CS3002 Information Security

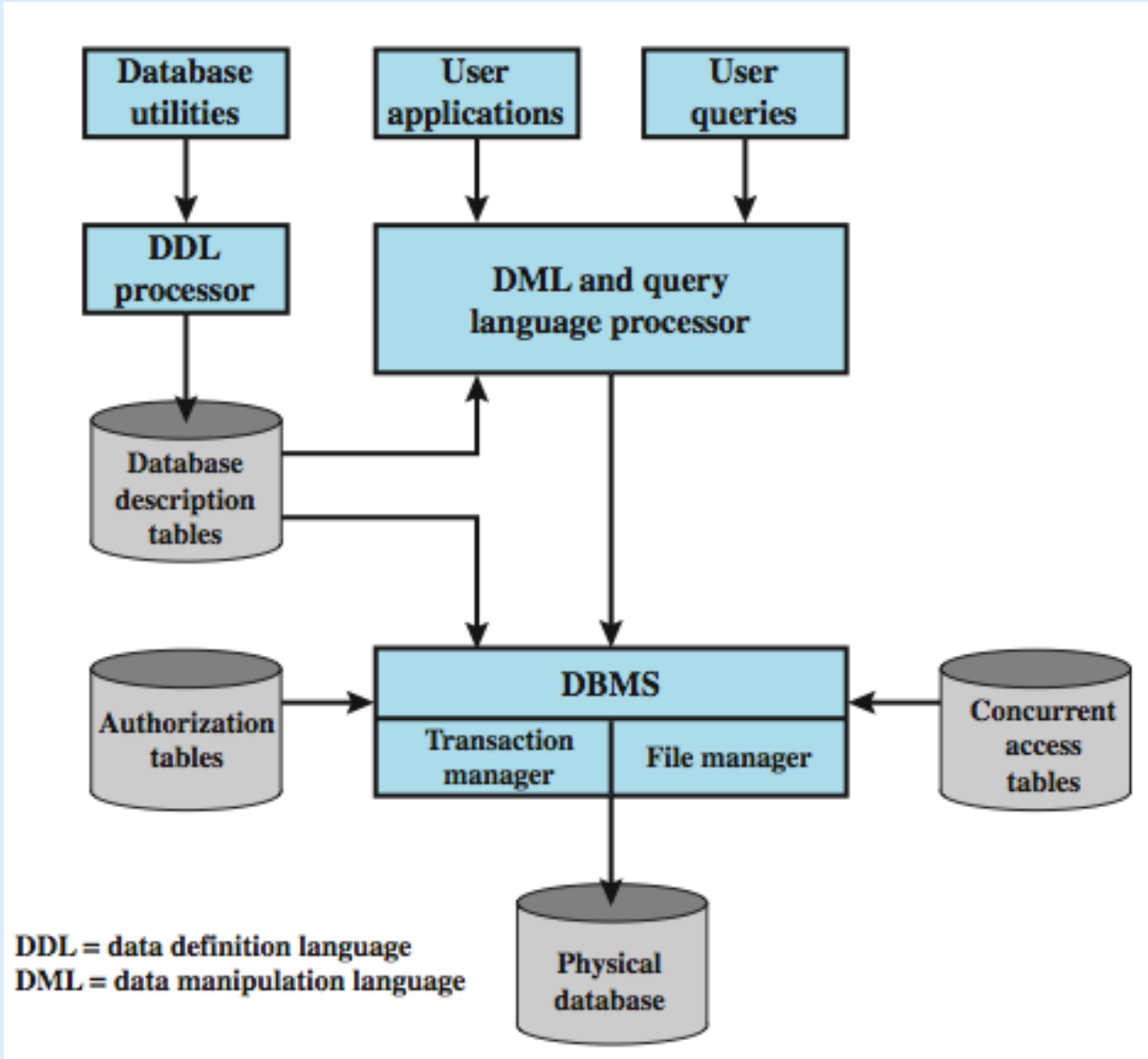
## Database Security

Reference: Stallings SPP chap 5

# Database Systems

- Collection of data stored for use by one or more applications
- Contains the relationships between data items and groups of data items
- Mostly contain sensitive data that needs to be secured
- Query language: Provides a uniform interface to the database

# Database Architecture

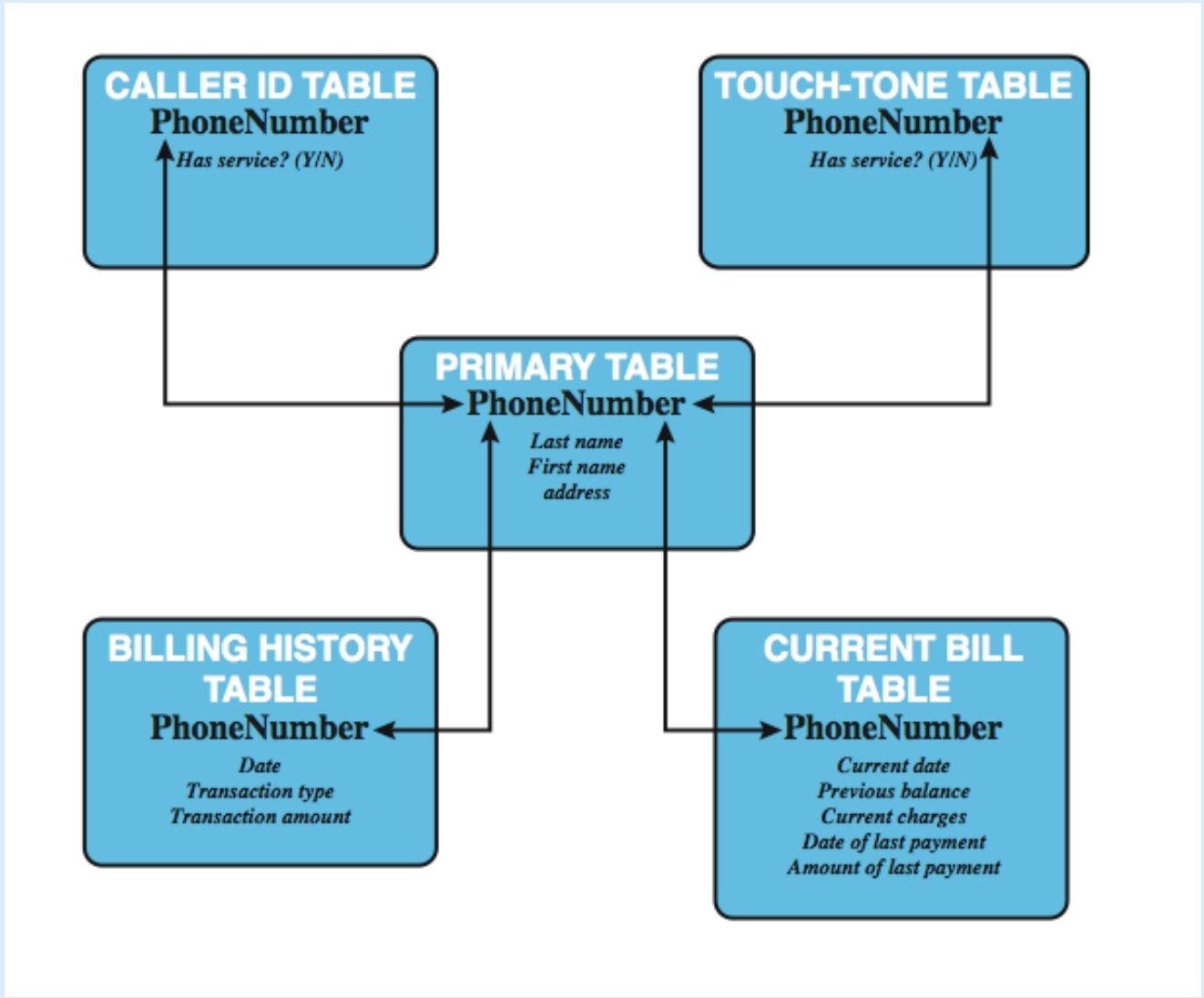



# Relational Databases

- Table of data consisting of rows and columns
- Each column holds a particular type of data
- Each row contains a specific value for each column
- Ideally has one column where all values are unique, forming an identifier/key for that row
  - Enables the creation of multiple tables linked together by a unique identifier that is present in all tables

- Use a relational query language to access the database
  - Allows the user to request data that fit a given set of criteria

# Relational Database Example




# Relational Database Elements

Two tables in a db

**Department Table**

| Did | Dname | Dacctno |
|---|---|---|
| 4 | human resources | 528221 |
| 8 | education | 202035 |
| 9 | accounts | 709257 |
| 13 | public relations | 755827 |
| 15 | services | 223945 |

primary key

**Employee Table**

| Ename | Did | SalaryCode | Eid | Ephone |
|---|---|---|---|---|
| Robin | 15 | 23 | 2345 | 6127092485 |
| Neil | 13 | 12 | 5088 | 6127092246 |
| Jasmine | 4 | 26 | 7712 | 6127099348 |
| Cody | 15 | 22 | 9664 | 6127093148 |
| Holly | 8 | 23 | 3054 | 6127092729 |
| Robin | 8 | 24 | 2976 | 6127091945 |
| Smith | 9 | 21 | 4490 | 6127099380 |

foreign key

primary key

A view derived from two tables

| Dname | Ename | Eid | Ephone |
|---|---|---|---|
| human resources | Jasmine | 7712 | 6127099348 |
| education | Holly | 3054 | 6127092729 |
| education | Robin | 2976 | 6127091945 |
| accounts | Smith | 4490 | 6127099380 |
| public relations | Neil | 5088 | 6127092246 |
| services | Robin | 2345 | 6127092485 |
| services | Cody | 9664 | 6127093148 |

# Structured Query Language

- SQL: originally developed by IBM in the mid-1970s
- standardized language to define, manipulate, and query data in a relational database
- several similar versions of ANSI/ISO standard

# SQL Examples

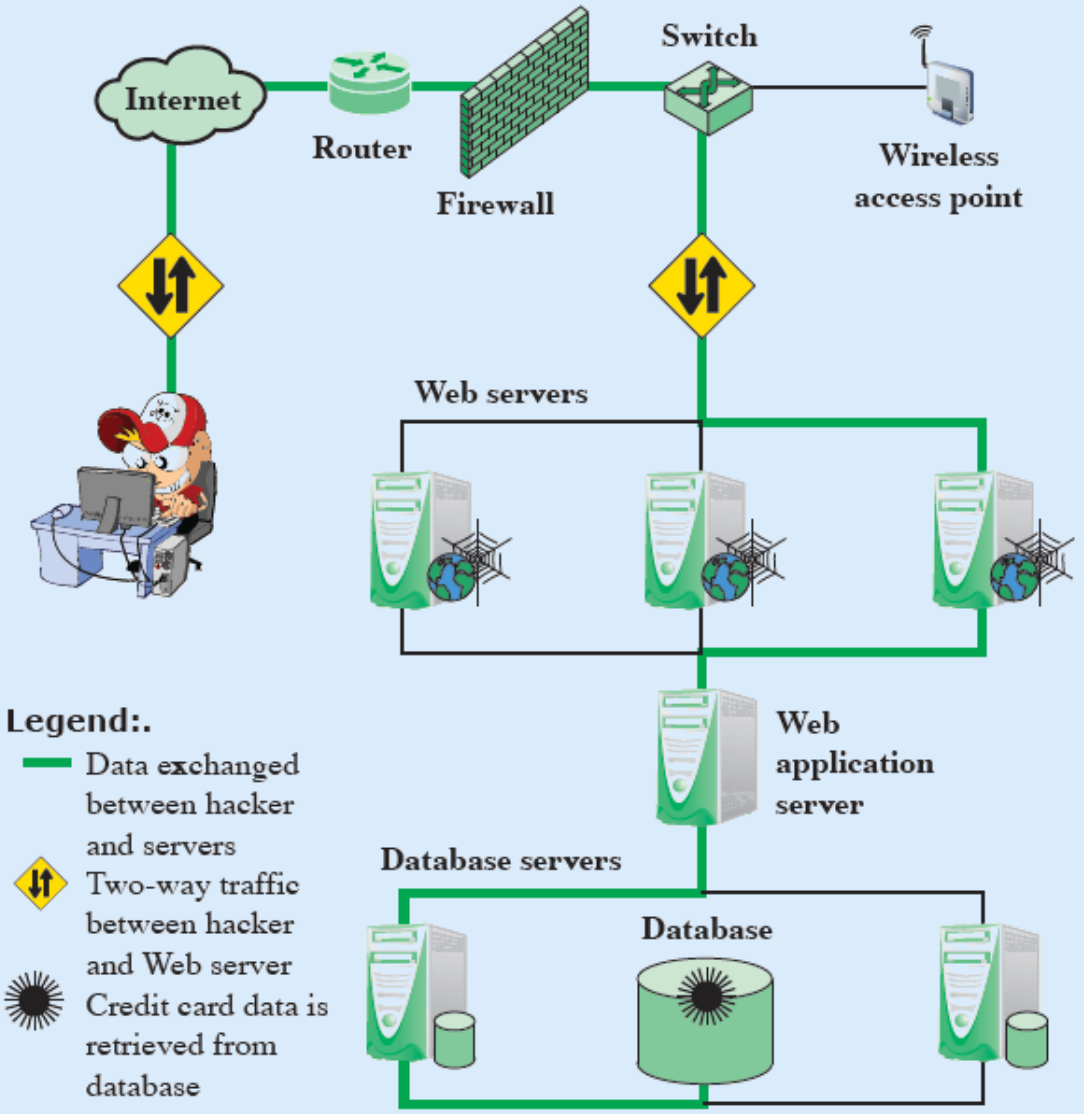
```
CREATE TABLE department (
    Did INTEGER PRIMARY KEY,
    Dname CHAR (30),
    Dacctno CHAR (6) )

CREATE TABLE employee (
    Ename CHAR (30),
    Did INTEGER,
    SalaryCode INTEGER,
    Eid INTEGER PRIMARY KEY,
    Ephone CHAR (10),
    FOREIGN KEY (Did) REFERENCES department (Did) )

CREATE VIEW newtable (Dname, Ename, Eid, Ephone)
AS SELECT D.Dname E.Ename, E.Eid, E.Ephone
FROM Department D JOIN Employee E ON E.Did = D.Did
```

# SQL Injection Attacks

# SQL Injection Attacks (SQLi)

- One of the most prevalent and dangerous network-based security threats
- Attacker sends malicious requests — containing SQL commands — to web (or application) server, which forward the commands to the database server
- SQL injection can be exploited to:
  - Steal data in bulk
  - Modify or delete data
  - Execute arbitrary operating system commands
  - Launch denial-of-service (DoS) attacks

# A typical injection attack




# A typical injection attack (steps)

1. Hacker finds a vulnerability in a custom Web application and injects a SQL command to a database by sending the command to the Web server. The command is injected into traffic that will be accepted by the firewall.
2. The Web server receives the malicious code and sends it to the Web application server.
3. The Web application server receives the malicious code from the Web server and sends it to the database server.
4. The database server executes the malicious code on the database. The database returns data from (e.g.) credit cards table.
5. The Web application server dynamically generates a page with data including credit card details from the database.
6. The Web server sends the credit card details to the hacker

# Injection Technique

- The SQLi attack typically works by prematurely terminating a text string and appending a new command.
  - Injected string is terminated with SQL comment mark (--)

```
SELECT fname, lname, phone
FROM student
WHERE fname = 'user_input';
```

What if user types the following input?

```
John'; DROP TABLE course; --
```

# Example SQL injection

Server-side C# code to get data from a web form

```
var ShipCity = Request.form("ShipCity");
var sql = "SELECT * FROM Orders WHERE ShipCity = '"
          + ShipCity + "';";
```

If user enters **Redmond** in the form, query becomes

```
SELECT * FROM Orders WHERE ShipCity = 'Redmond';
```

Suppose, however, the user enters the following in the form:

```
Redmond'; DROP table Orders; --
```

This results in the following SQL query:

```
SELECT * FROM Orders WHERE ShipCity = 'Redmond'; DROP
table Orders; --';
```

`Request.form()` returns the data submitted by user in HTTP POST request body.

# In-band attacks

**In-band:** same channel used for injecting SQL code and retrieving results. Such attacks can include one or more of the following elements:

- Tautology: This form of attack injects code in one or more conditional statements so that they always evaluate to true
- End-of-line comment: After injecting code into a particular field, legitimate code that follows are nullified through usage of end of line comments
- Piggybacked queries: The attacker adds additional queries beyond the intended query, piggy-backing the attack on top of a legitimate request

# Tautology Example

Server side PHP script

```
$query= "SELECT info FROM user WHERE name =
'`$_GET["name"]`' AND pwd = '`$_GET["pwd"]`'";
```

Attacker submits following for the **name** field:

```
' OR 1=1 --
```

Query becomes

```
SELECT info FROM users
WHERE name = '' OR 1=1 -- AND pwd = ''
```

`$_GET` provides the data submitted by user in HTTP GET request URL (aka query string).

# Using UNIONs

With the help of UNION operator, an attacker can retrieve data from other tables.

For example, if an application executes the following query (where `shirts` value is coming from client side)

```
SELECT name, description
FROM products
WHERE category = 'shirts';
```

An attacker can submit the input:

```
shirts' UNION SELECT username, password FROM users; --
```

which will make the server return all usernames and passwords along with the names and descriptions of products.

# Inferential Attack (gathering info)

There is no actual transfer of data, but the attacker is able to reconstruct the information by sending particular requests and observing the resulting behavior of the website/database server

**a) illegal/logically incorrect queries**

- lets an attacker gather important information about the type and structure of the backend database of a Web application
- Considered as a preliminary step for further attacks.
  - The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive.

# Inferential Attack (gathering info)

Example illegal query (missing whitespaces)

```
SELECT*FROM table_nameWHERE id=@id"
```

Possible web response, which reveals the server side script

notsosecureapp.com/orderitem?id=p11rutKSpNE%3d

Server Error in '/' Application.

*Unclosed quotation mark after the character string ''.*
*Incorrect syntax near ''.*

**Description:** An unhandled exception occurred during the execution of the current web request. Please review the stack

**Exception Details:** System.Data.SqlClient.SqlException: Unclosed quotation mark after the character string ''.
Incorrect syntax near ''.

**Source Error:**

```
Line 67:                SqlCommand cmd = new SqlCommand(sql, connection);
Line 68:                cmd.CommandType = CommandType.Text;
Line 69:                using (var reader = cmd.ExecuteReader())
Line 70:                {
Line 71:                    while (reader.Read())
```

https://notsosecure.com/anatomy-hack-sqli-crypto

# Inferential Attack (gathering info)

In other cases, database table and/or column names might be revealed.

Conversion failed when converting

ViewGallery.aspx?CatID=2' and 1=convert(int,(select top 1 ta 90%

## Server Error in '/' Application.

*Conversion failed when converting the nvarchar value 'Download_Dcoument' to data type int.*

**Description:** An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

**Exception Details:** System.Data.SqlClient.SqlException: Conversion failed when converting the nvarchar value 'Download_Dcoument' to data type int.

**Source Error:**

```
Line 29:         SqlDataAdapter da = new SqlDataAdapter(cmd);
Line 30:         DataSet ds = new DataSet();
Line 31:         da.Fill(ds);
Line 32:         DataList1.DataSource = ds;
Line 33:         DataList1.DataBind();
```

**Source File:** g:\pleskvhosts\angelhybrid.in\ViewGallery.aspx.cs **Line:** 31

**Stack Trace:**

```
[SqlException (0x80131904): Conversion failed when converting the nvarchar value 'Download_Dcoument' to data type int.]
   System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean breakConnection, Action`1 wrapCloseInAction) +2582782
   System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boolean breakConnection, Action`1 wrapCloseInAction) +6033430
   System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj, Boolean callerHasConnectionLock, Boolean asyncClose) +297
   System.Data.SqlClient.TdsParser.TryRun(RunBehavior runBehavior, SqlCommand cmdHandler, SqlDataReader dataStream, BulkCopySimpleResultSet bulkCopyHandler, TdsParserStat
   System.Data.SqlClient.SqlDataReader.TryHasMoreRows(Boolean& moreRows) +242
   System.Data.SqlClient.SqlDataReader.TryReadInternal(Boolean setTimeout, Boolean& more) +275
   System.Data.SqlClient.SqlDataReader.Read() +35
   System.Data.Common.DataAdapter.FillLoadDataRow(SchemaMapping mapping) +216
   System.Data.Common.DataAdapter.FillFromReader(DataSet dataset, DataTable datatable, String srcTable, DataReaderContainer dataReader, Int32 startRecord, Int32 maxRecord
   System.Data.Common.DataAdapter.Fill(DataSet dataSet, String srcTable, IDataReader dataReader, Int32 startRecord, Int32 maxRecords) +422
   System.Data.Common.DbDataAdapter.FillInternal(DataSet dataset, DataTable[] datatables, Int32 startRecord, Int32 maxRecords, String srcTable, IDbCommand command, Comman
   System.Data.Common.DbDataAdapter.Fill(DataSet dataSet, Int32 startRecord, Int32 maxRecords, String srcTable, IDbCommand command, CommandBehavior behavior) +138
   System.Data.Common.DbDataAdapter.Fill(DataSet dataSet) +89
```

https://infosecwriteups.com/exploiting-error-based-sql-injections-bypassing-restrictions-ed099623cd94

# Inferential Attack (gathering info)

**b) Blind SQL injection**

- Most systems have sufficient security in place to not display any erroneous information back to the attacker. They could display a generic “404 Not Found” or “500 Internal Server Error” error page without revealing exact the database error message.
- Blind injection is used in such cases. It allows attackers to **infer** the data present in a database system.
- For example, the attacker asks the server true/false questions to observe the functionality in each case.

# Inferential Attack (gathering info)

**Blind SQL injection Example**

Suppose the URL http://newspaper.com/items.php?id=2 sends the following query to the database:

```
SELECT title, descr, body FROM items WHERE ID = 2
```

The attacker may then try to inject a query that returns 'false':

http://newspaper.com/items.php?id=2 and 1=2

Now the SQL query should looks like this:

```
SELECT title, descr, body FROM items WHERE ID = 2 and 1=2
```

If the web application is vulnerable to SQLi, then it probably will not return anything (because WHERE clause is false). To make sure, the attacker will inject a query that will return 'true':

http://newspaper.com/items.php?id=2 and 1=1

If the content of this page is different from the previous one, then the attacker is able to verify that injection is working.

https://owasp.org/www-community/attacks/Blind_SQL_Injection

# Denial-of-Service using SQLi

Attackers can also inject queries that cause DBMS to perform long useless calculations, draining all CPU resources. e.g.

```
SELECT DECODE(
          ENCODE(
            CONVERT(COMPRESS(post) USING latin1),
            CONCAT(post, post, post, post)
          ),
          SHA1(CONCAT(post, post, post, post))
        ) AS column1
FROM table_1;
```

Note: `ENCODE(ptext, p)` encrypts `ptext` with password `p`. `DECODE` performs decryption.

Imagine running above query on a database containing thousands of post entries.

https://www.securityidiots.com/Web-Pentest/SQL-Injection/ddos-website-with-sqli-siddos.html

# SQLi attempts – Real example

Lee Penkman created a form on his website. It was hidden, so actual users could not see it, only bots (scanning for vulnerabilities) could see and submit it.

Here are some of the things submitted:

- `1 UNION ALL SELECT NULL#`
- `-1 OR 2+456-456-1=0+0+0+1`
- `wrBEIRqX'||DBMS_PIPE.RECEIVE_MESSAGE(CHR(98)||CHR(98)||CHR(98),15)||'`
- `1;WAITFOR DELAY '0:0:5'–`
- `(SELECT (CASE WHEN (2646=4989) THEN 2646 ELSE 2646*(SELECT 2646 FROM INFORMATION_SCHEMA.CHARACTER_SETS) END))`

https://github.com/lee101/hidden-form-on-the-internet

# SQLi countermeasures

**Defensive coding**

1) Using prepared statements (parameterized queries)
   - define all SQL code first at and pass in each parameter to the query later
   - This way database will always distinguish between code and data, regardless of what user input is supplied
   - When writing query, put placeholders for all parameters (typically ?s)
   - Next, **bind** the parameters to query

# SQLi countermeasures

## Prepared statements Java example

```
String query =
      "INSERT INTO products (name, price) VALUES (?, ?)";
PreparedStatement pstmt = connection.prepareStatement(query);

String name = request.getParameter("productName");
String price = request.getParameter("productPrice");

// (input validation code goes here)

pstmt.setString(1, name);  // bind 1st parameter to its value
pstmt.setDouble(2, Double.parseDouble(price)); // bind 2nd
ResultSet results = pstmt.executeQuery();
```

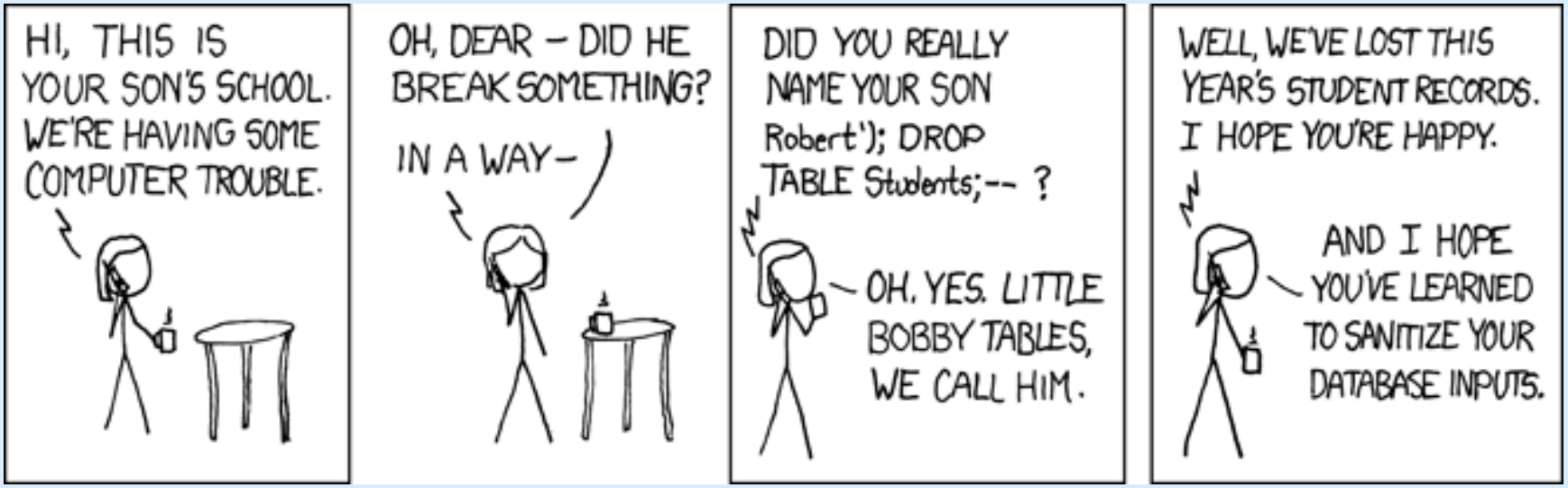
# SQLi countermeasures

**Defensive coding**

2) Stronger data validation
   - type checking, e.g. to check that inputs that are supposed to be numeric contain no characters other than digits
   - pattern matching to distinguish normal input from abnormal

3) Sanitizing (cleaning) inputs
   - Remove or escape all unsafe characters (quotes, semicolons, hyphens etc.) in inputs

# SQLi countermeasures




https://www.explainxkcd.com/wiki/index.php/327:_Exploits_of_a_Mom

# Database Access Control

# Database Access Control

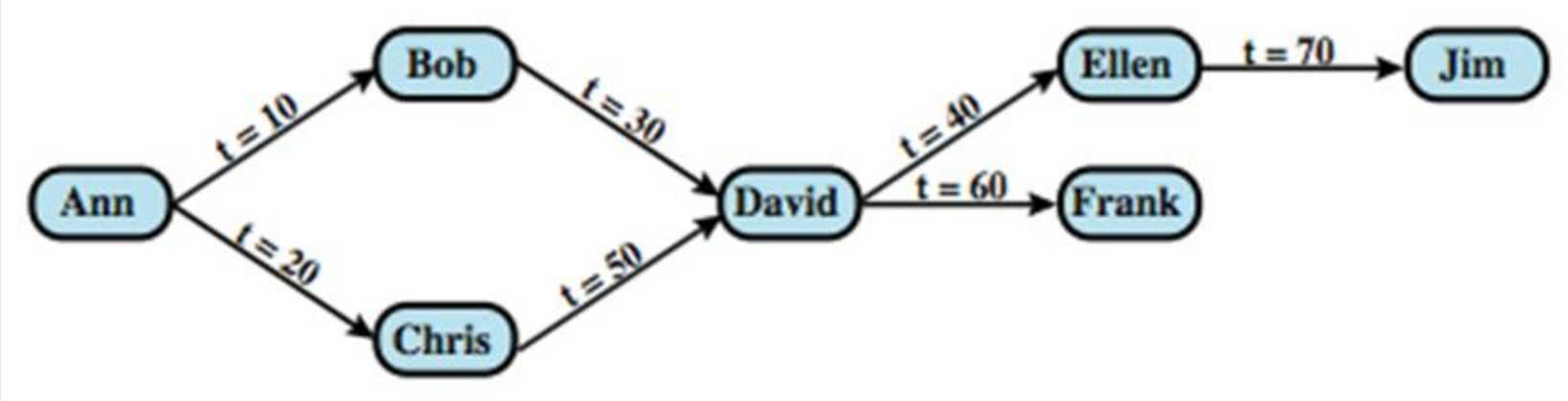
- DBMS can provide specific access rights to portions of the database
  - e.g. create, insert, delete, update, read, write
  - to entire database, tables, selected rows or columns
  - possibly dependent on contents of a table entry
    - e.g. A teacher can view marks of their own students only
- can support a range of policies:
  - centralized administration
  - ownership-based administration
  - decentralized administration

# SQL Access Control Commands

- To define if a user has access to the entire database or just portions of it, there are two commands:
- `GRANT {privileges} [ON table] TO {user | PUBLIC} [IDENTIFIED BY password] [WITH GRANT OPTION]`
  - e.g. `GRANT SELECT, INSERT ON ANY TABLE TO john`
- `REVOKE {privileges} [ON table] FROM {user | PUBLIC}`
  - e.g. `REVOKE SELECT ON ANY TABLE FROM john`
- Typical access rights (privileges) are: SELECT, INSERT, UPDATE, DELETE, REFERENCES. Column restriction can also be specified where applicable,
  - e.g. `GRANT UPDATE (col1, col2) ...`

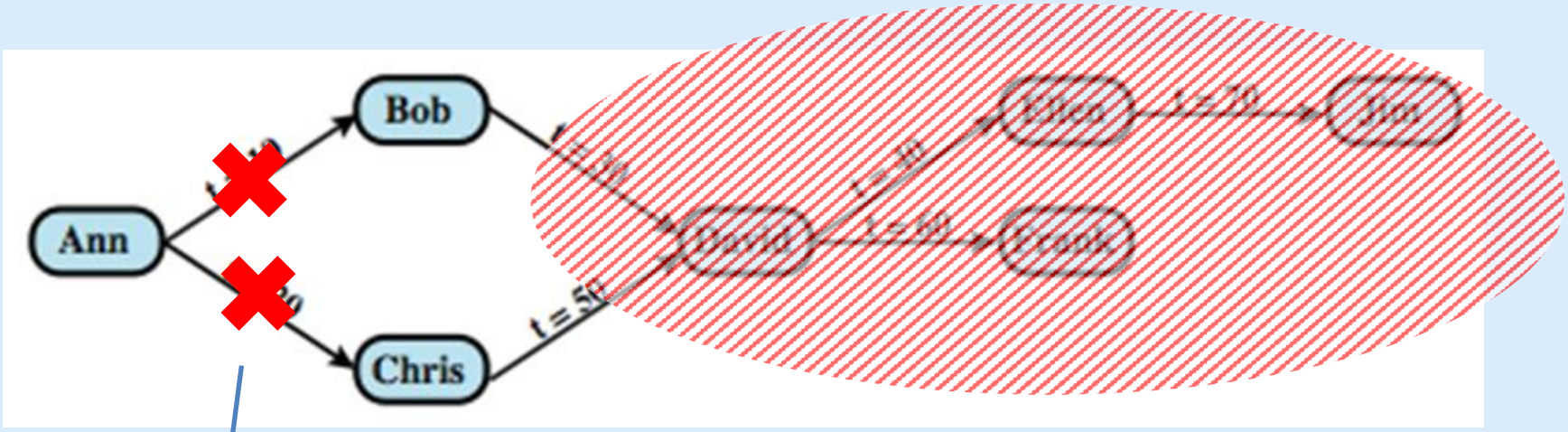
# Cascading Authorizations

- `WITH GRANT OPTION`: whether grantee can authorize (`GRANT`) other users with the same privileges
- When this option is included, access right will cascade through.



# Cascading Authorizations

It works the same way with **revocations**.

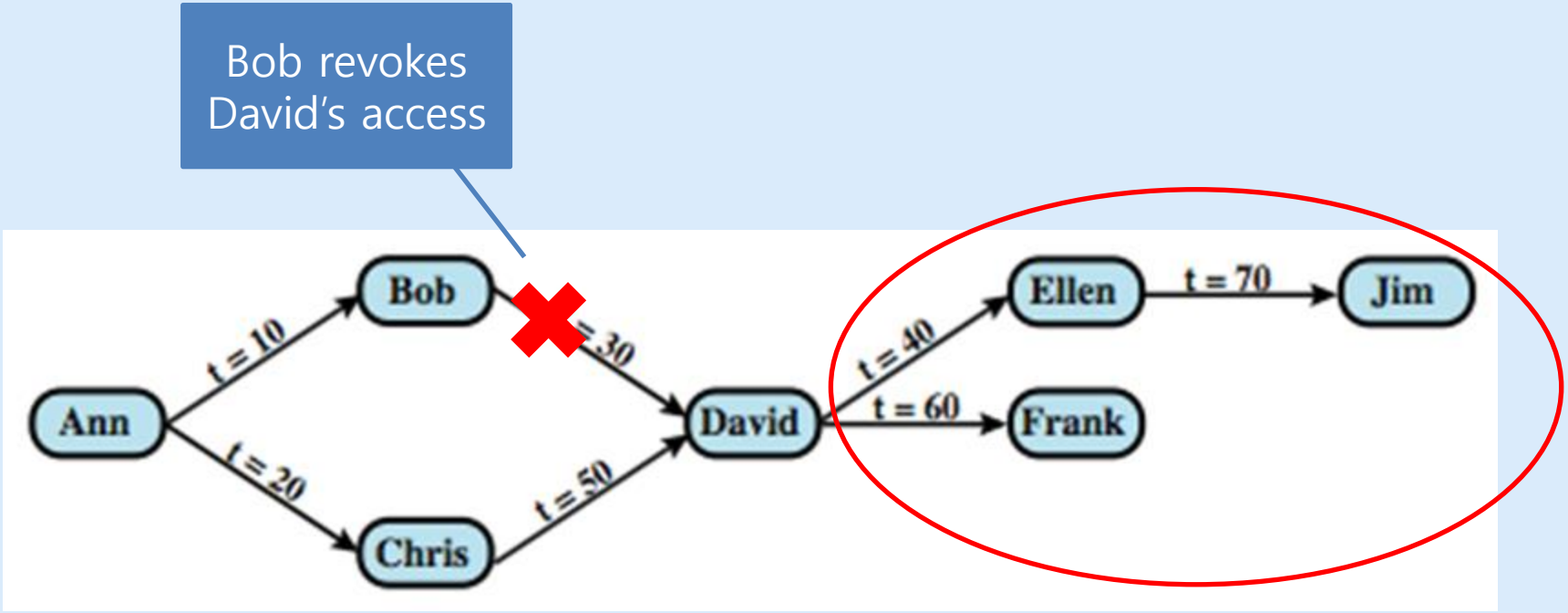



If Ann revokes Bob's and Chris' access

All cascaded access should also be revoked

# Cascading Authorizations

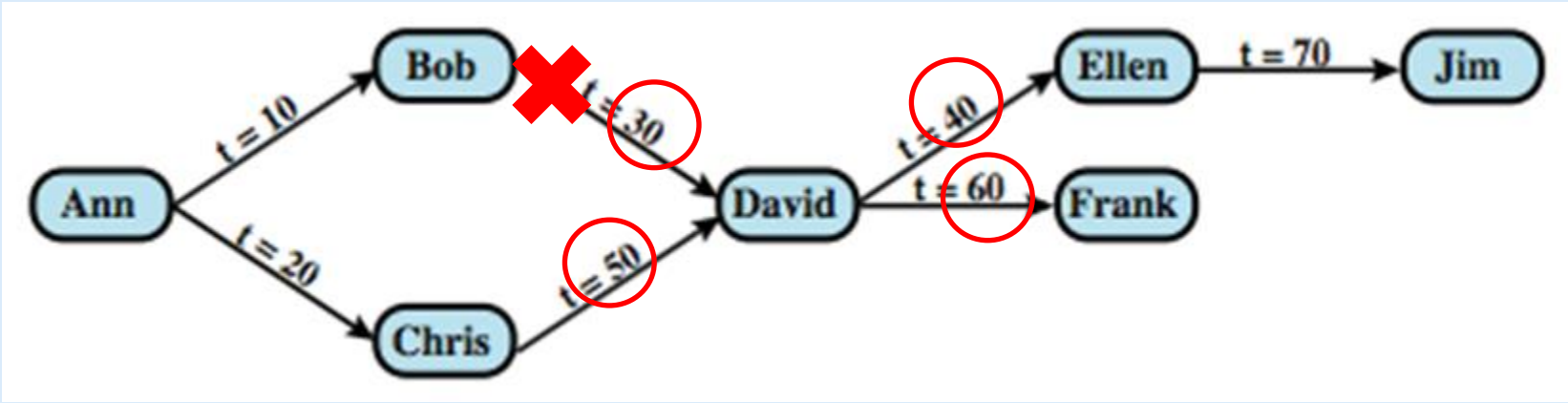
Complications can arise in some cases

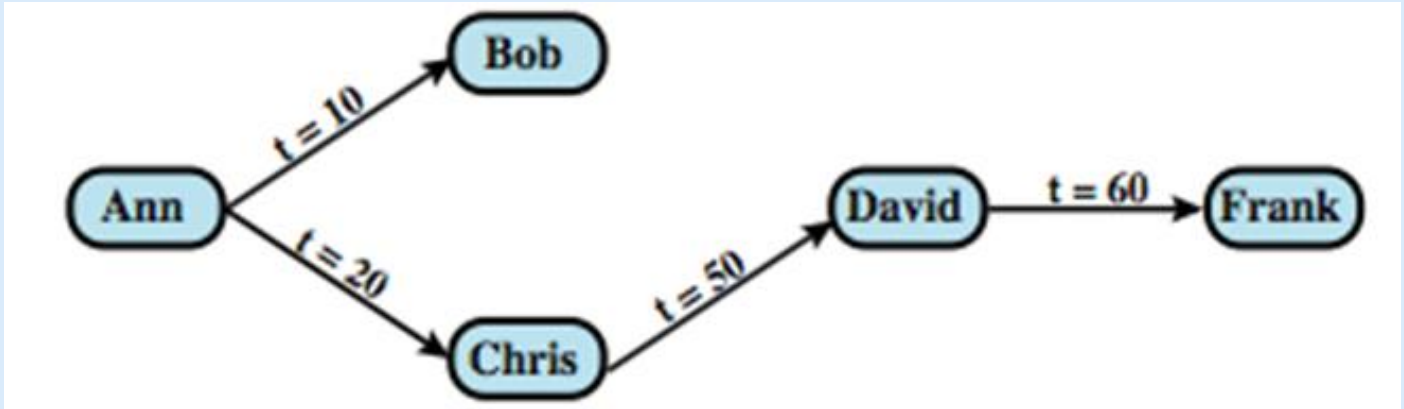



What to do with child accesses?
*David's access was also granted by Chris*

# Cascading Authorizations

One option: Consider the access grant times



# Role-Based Access Control (RBAC)

- Role-based access control work well for DBMS
  - eases admin burden, improves security
- Categories of database users:
  - application owner
  - end user
  - administrator
- Administrators are responsible for managing database as a whole (installation, upgrades, security, defining applications etc.)
- An application owner must assign roles to end users, e.g. software-engineer, marketing, finance, hr

# SQL commands for RBAC

- SQL GRANT and REVOKE commands can also be used for role assignments

Syntax

- `GRANT {role} TO {user}`
- `REVOKE {role} FROM {user}`

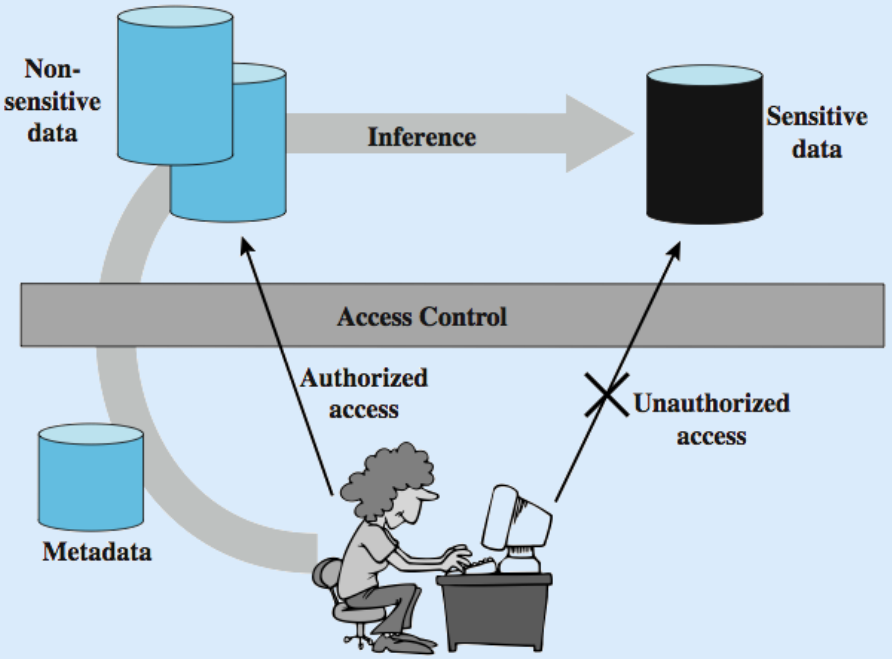
Examples

- `GRANT manager TO sarah` (role assignment)
- `GRANT INSERT, SELECT(col3) ON table4 TO manager` (privilege assignment to role)
- `REVOKE manager FROM sarah` (role removal)

# Database Inference

# Inference

- The process of performing authorized queries and deducing unauthorized information from the legitimate responses received
- A combination of data items can be used to infer data of a higher sensitivity



# Inference Example

| Name | Position | Salary ($) | Department | Dept. Manager |
|---|---|---|---|---|
| Andy | senior | 43,000 | strip | Cathy |
| Calvin | junior | 35,000 | strip | Cathy |
| Cathy | senior | 48,000 | strip | Cathy |
| Dennis | junior | 38,000 | panel | Herman |
| Herman | senior | 55,000 | panel | Herman |
| Ziggy | senior | 67,000 | panel | Herman |

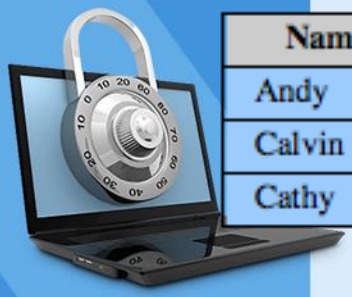
(a) Employee table

| Position | Salary ($) |
|---|---|
| senior | 43, 000 |
| junior | 35,000 |
| senior | 48,000 |

| Name | Department |
|---|---|
| Andy | strip |
| Calvin | strip |
| Cathy | strip |

(b) Two views

Suppose a user is not allowed to see the salary of a specific employee, i.e. (Name, Salary) combination can not be in a query



| Name | Position | Salary ($) | Department |
|---|---|---|---|
| Andy | senior | 43,000 | strip |
| Calvin | junior | 35,000 | strip |
| Cathy | senior | 48,000 | strip |

(c) Table derived from combining query answers

User can still derive the confidential information, because of same data ordering in two views

# Inference countermeasures

- Inference prevention during database design
  - Alter database structure or access controls
  - Splitting a table into multiple tables
  - More fine-grained access control roles
- Inference detection at query time
  - by monitoring and altering or rejecting queries

# Inference prevention during database design

To protect (Name, Salary) pair information, we can distribute data in three tables:

```
Employees (Emp#, Name, Department)
Salaries (S#, Salary)
Emp-Salary (Emp#, S#)
```

Protected Table

Now suppose we want to add a column of employee start dates. Which option is better for preventing inference?

```
Employees (Emp#, Name, Department)
Salaries (S#, Salary, Start-Date)
Emp-Salary (Emp#, S#)
```

OR

```
Employees (Emp#, Name, Department, Start-Date)
Salaries (S#, Salary)
Emp-Salary (Emp#, S#)
```

# Statistical Databases

- Specialized databased that only provide data of aggregate/summary or statistical nature
  - e.g. sum, counts, averages
  - So that privacy of individuals is maintained
- Two types:
  - pure statistical database
  - ordinary database with statistical access: some users have normal access, others statistical

- Security problem is one of inference

# Statistical Inference Examples

**Example 1**

- Count (MS.2024.EE.female) = 1
- Sum (MS.2024.EE.female, GPA) = 2.7

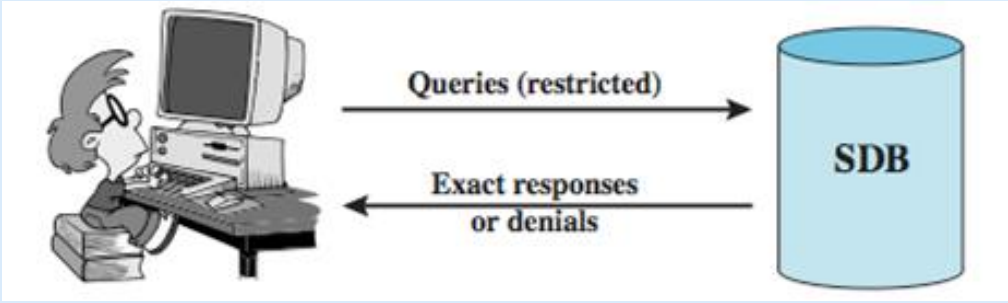
Privacy compromised (only one student in group)

**Example 2**

- Non secret information:
  - Salary range for system analyst with BS: (50K, 60K)
  - Salary range for system analyst with MS: (65K, 70K)
- An attacker obtains the sum of all employee salaries
- After a new employee joins, same query is run again.
- The difference will reveal the new employee's salary and qualification

# Protecting against Inference

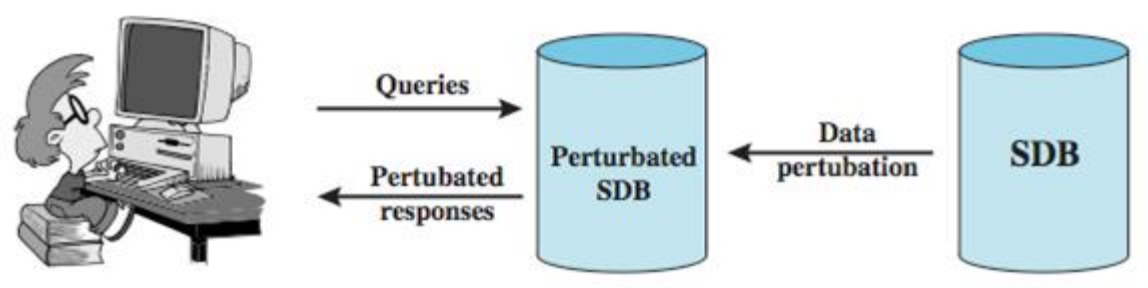
- Query restriction
  - Rejects a query that can lead to a compromise.
  - The answers provided are accurate.
- The simplest is query size restriction.
  - A query is permitted only if the number of records that match the query condition are more than a threshold, k (where k>1).
  - In practice, queries over the whole table are allowed, enabling users to easily access statistics calculated on the entire tables.



# Protecting against Inference

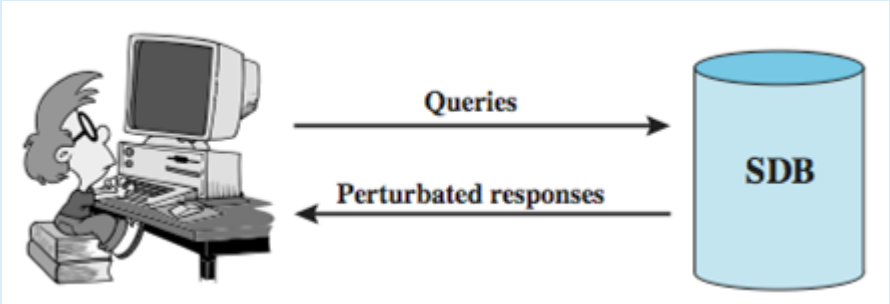
- Perturbation
  - Add noise to statistics generated from data
  - will result in differences in statistics

1. Data perturbation
   - Swap attribute data between rows so that individual records are not accurate, but overall statistics remain similar



# Protecting against Inference

2. Output perturbation
   - **Random-sample query**: Calculate the statistics on a properly selected sampled query subset
   - **Statistic adjustment**: Adjusting the answer up or down by a given amount in some systematic fashion
   - Must minimize loss of accuracy in results

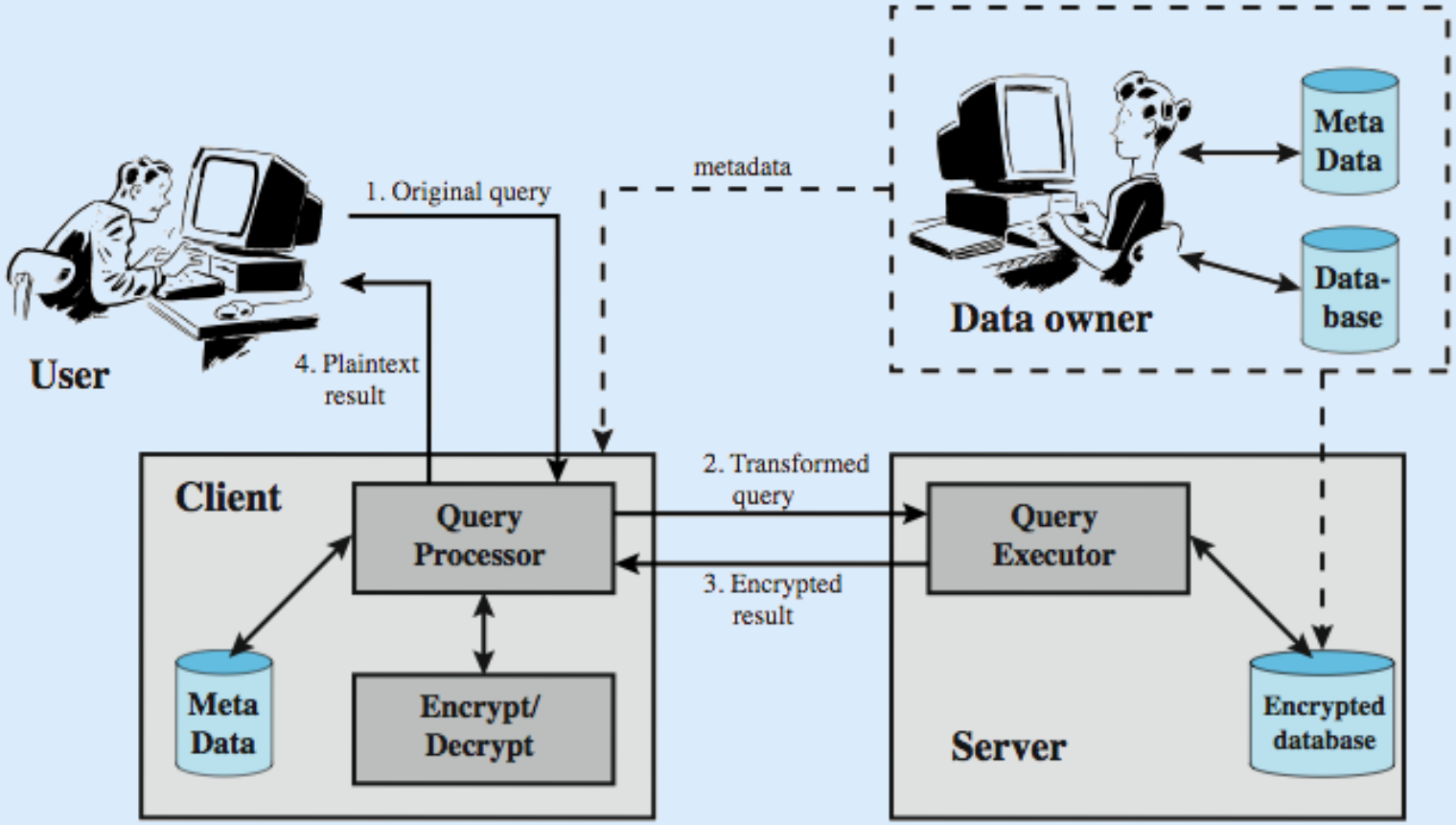


# Database Encryption

# Database Encryption

- Databases typically a valuable info resource
  - protected by multiple layers of security: firewalls, authentication, OS access control systems, DB access control systems
- For particularly sensitive data, DB encryption is the last line of defense
- DB Encryption creates challenges
  - How to distribute decryption keys
  - How to execute queries, i.e. search for records

# Querying an Encrypted DB




# How to Encrypt

- Can encrypt
  - entire database (tables)
    - very inflexible and inefficient (how to execute queries?)
  - individual fields
    - simpler but still inflexible
  - records (rows)
    - best if attribute indexes are created to help data retrieval
- Varying trade-offs

# Field Level Encryption

- Suppose that each individual item in the database is encrypted separately.
- The encrypted database is stored at the server, but the server does not have the key, so that the data is secure at the server. The client system does have a copy of the encryption key.

- A user at the client can retrieve a record from the database with the following sequence:

1. The user issues SQL query for fields from one or more records with a specific value of the attribute.

# Field Level Encryption

2. The query processor at the client encrypts the attribute parameters, modifies the SQL query accordingly, and transmits the query to the server.
3. The server processes the query using the encrypted value of the attributes and returns the (encrypted) records.
4. The query processor decrypts the data and returns the results to user.

# Field Level Encryption

**Example**

```
SELECT Ename, Eid, Ephone
FROM Employee
WHERE Did = 15
```

Assume that the encryption key **k** is used and that the encrypted value of the department id 15 is E(k, 15) = 1000110111001110.

The query processor at the client could transform the preceding query into

```
SELECT Ename, Eid, Ephone
FROM Employee
WHERE Did = 1000110111001110
```

# Field Level Encryption

**Issues**

- This method is certainly straightforward but lacks flexibility.
- For example, suppose the Employee table contains a salary attribute and the user wishes to retrieve all records for salaries less than 70K.
  - There is no obvious way to do this, because the attribute value for salary in each record is encrypted.
  - The set of encrypted values do not preserve the ordering of values in the original attribute.

# Record Level Encryption

- Each record (row) of a table in the database is encrypted as a whole, and treated as a contiguous block.
- For each row in the original database, there is one row in the encrypted database.
- To assist in data retrieval, **attribute indexes** are associated with each table. For some or all of the attributes an index value is created.
  - For any attribute, the range of attribute values is divided into a set of non-overlapping partitions that encompass all possible values, and an index value is assigned to each partition.

# Record Level Encryption

**Example**

- Suppose that employee ID (eid) values lie in the range [1, 1000]. We can divide these values into five partitions: [1, 200], [201, 400], [401, 600], [601, 800], and [801, 1000]; and then assign index values 1, 2, 3, 4, and 5, respectively.
- For a text field, we can derive an index from the first letter of the attribute value. For the attribute ename, let us assign index 1 to values starting with A or B, index 2 to values starting with C or D, and so on.
- Similar partitioning schemes can be used for each of the attributes.

# Record Level Encryption

**(a) Employee Table**

| eid | ename | salary | addr | did |
|---|---|---|---|---|
| 23 | Tom | 70K | Maple | 45 |
| 860 | Mary | 60K | Main | 83 |
| 320 | John | 50K | River | 50 |
| 875 | Jerry | 55K | Hopewell | 92 |

**(b) Encrypted Employee Table with Indexes**

| E($k$, $B$) | I(eid) | I(ename) | I(salary) | I(addr) | I(did) |
|---|---|---|---|---|---|
| 1100110011001011… | 1 | 10 | 3 | 7 | 4 |
| 0111000111001010… | 5 | 7 | 2 | 7 | 8 |
| 1100010010001101… | 2 | 5 | 1 | 9 | 5 |
| 0011010011111101… | 5 | 5 | 2 | 4 | 9 |

45K ≤ salary < 55K: 1
55K ≤ salary < 65K: 2
65K ≤ salary < 75K: 3

The clause
`WHERE salary > 60K`
gets transformed to
`WHERE I(salary) ≥ 2`