Lecture 4

Structured Query Language (basic)

COMP3278A

Introduction to Database Management Systems

Dr. Ping Luo
Email : pluo@cs.hku.hk

Department of Computer Science, The University of Hong Kong

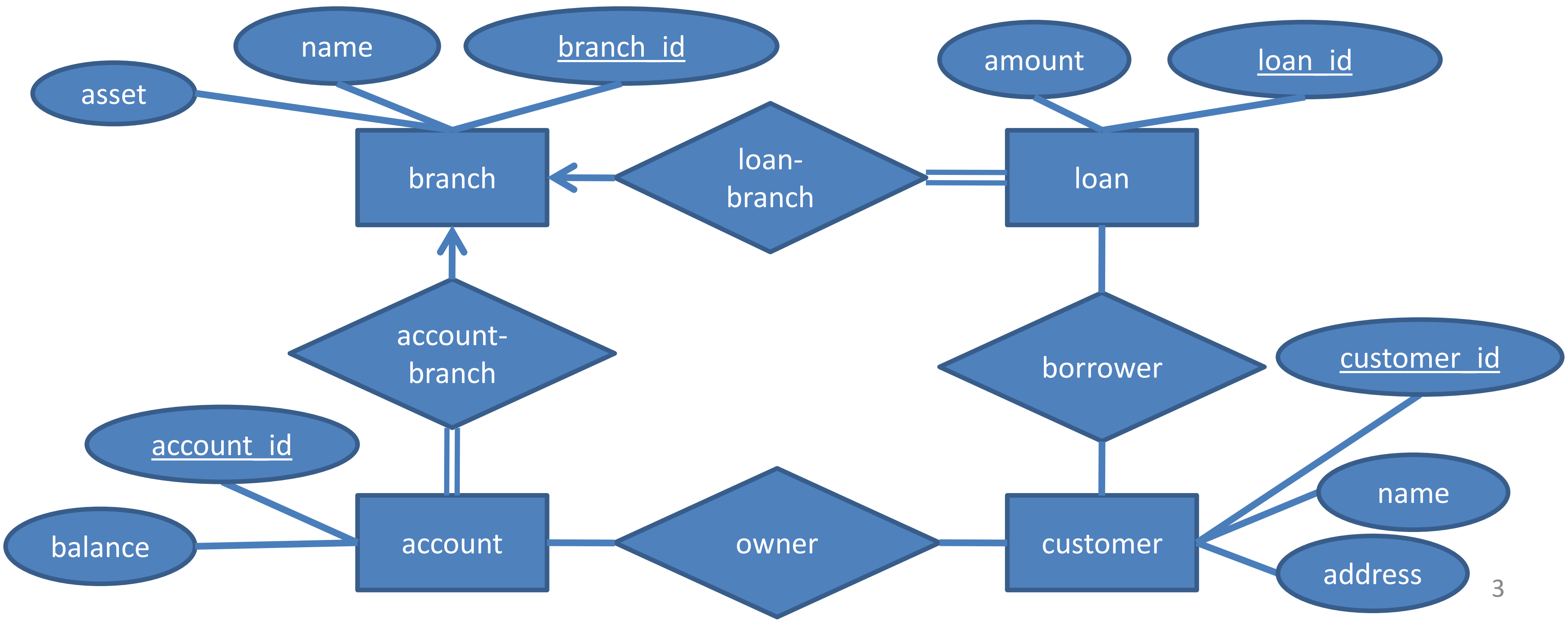
Acknowledgement: Dr Chui Chun Kit

Outcome based learning (OBL)

- Outcome 1. **Information Modeling**
  - Able to understand the modeling of real life information in a database system.
- Outcome 2. **Query Languages**
  - Able to understand and use the languages designed for data access.
- Outcome 3. **System Design**
  - Able to understand the design of an efficient and reliable database system.
- Outcome 4. **Application Development**
  - Able to implement a practical application on a real database.

Recap

- **Let's consider the following steps in developing a database application in a banking enterprise.**
- **Step 1. Information modeling**



Recap

- Step 2. Reduce to database table definitions

Recap

- **Step 3.** Create the database and tables
- **Step 4.** Design the SQL to access data for the application
- **Step 5.** Relational Algebra optimizes SQL (DBMS does this automatically). We'll learn its basic concepts.

Running example

Foreign key

Foreign key

Branch

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

Borrower

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Foreign key

Foreign key

Foreign key

Account

branch_id	account_id	balance
B1	A1	500
B2	A2	400
B2	A3	900
B1	A4	700

Owner

account_id	customer_id
A1	C1
A1	C2
A2	C2
A3	C4
A4	C4

Foreign key

Customer

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

What is SQL?

- **Structured Query Language (pronounced as "sequel")**
- **Language for defining, modifying and querying data in an RDBMS.**
- **SQL is declarative**
  - Concerns about the task we want to accomplish, without specifying how.
- **SQL has many standards and implementations**
  - Read the documentation on which features are supported exactly.

Section 1

Create and Drop Table

Create table

- A database table is defined using the **CREATE TABLE** command.

Table name (cannot be a keyword in database, e.g., **Create**)

Column name

Column type

```
CREATE TABLE Branch
(
  branch_id VARCHAR(15) NOT NULL,
  name VARCHAR(30) NOT NULL,
  asset INT UNSIGNED NOT NULL,
  PRIMARY KEY(branch_id)
);
```

NOT NULL means each record's value in the column must not be a null value.

Constraint (Primary Key), optional in this statement, can add back later using the **ALTER TABLE** command.

No comma in the last instruction.

PRIMARY KEY automatically ensures **NOT NULL**.

Drop table

- **DROP TABLE** deletes all information about the dropped table from the database.

```
DROP TABLE Branch;
```

- The DBMS may **reject** the **DROP TABLE** instruction when the table is referenced by another table via some constraints (e.g., **referential constraints**).

Foreign key

Customer

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Borrower

customer_id	loan_id
C1	L3
C4	L2
C2	L1

After the foreign key is established, if we drop the Customer table, the records in the Borrower table will lost their references .
(i.e., **Cannot find out who borrow the loan anymore**.)

Alter table

- **ALTER TABLE** can be used to
  - Add columns to an existing table.

```
ALTER TABLE Branch ADD branch_phone INT (12);
```

  - Remove a column from a table.

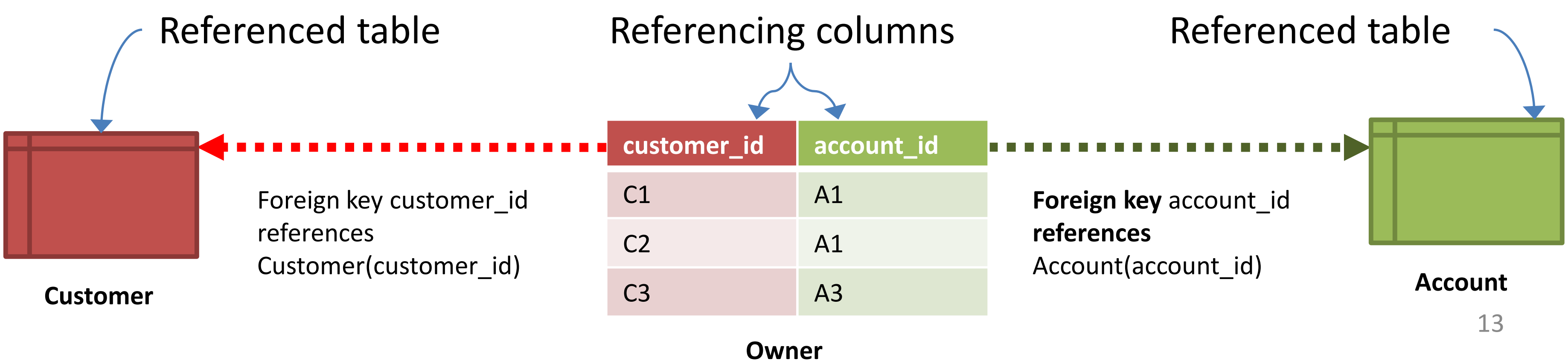
```
ALTER TABLE Branch DROP branch_phone;
```

  - Add constraints (e.g., PRIMARY KEY) to a table.

```
ALTER TABLE Branch ADD PRIMARY KEY (branch_id);
```

Foreign key constraints

- A **foreign key** is a referential constraint between two tables.
- The columns in the referencing table must reference the columns of the **primary key** or other **superkey** in the referenced table.
  - i.e., The value in one row of the **referencing columns** must occur in a single row in the **referenced table. The referencing columns must be primary/candidate key of another table. The referencing table cannot contain record that doesn't exist in the referenced table.**



customer_id	account_id
C1	A1
C2	A1
C3	A3

Foreign key constraints

- The foreign key can be established in the **CREATE TABLE** command.

```
CREATE TABLE Owner
(
    customer_id VARCHAR(15),
    account_id VARCHAR(15),
    PRIMARY KEY(customer_id, account_id),
    FOREIGN KEY(customer_id) REFERENCES Customer(customer_id),
    FOREIGN KEY(account_id) REFERENCES Account(account_id)
);
```

- The foreign key can also be defined using the **ALTER TABLE** command.

```
ALTER TABLE Owner
ADD FOREIGN KEY (customer_id) REFERENCES Customer(customer_id);
```

Section 2

Insert, Delete and Update

The INSERT clause

- The **INSERT INTO** command is used to insert records (tuples) into the database table.

branch_id	name	asset
Empty		

Branch

→

branch_id	name	asset
B1	Central	7100000

Branch

Table name (Case sensitive)

```
INSERT INTO Branch VALUES ( 'B1' , 'Central', 7100000);
```

Value in the **first** column — Value in the **second** column — Value in the **third** column

- Inserting multiple records

```
INSERT INTO Branch VALUES
( 'B2' , 'Causeway Bay', 9000000),
( 'B3' , 'Aberdeen', 400000);
```

The **INSERT** clause

- Most DBMS provide an alternative way to insert large amount of records into a table.
  - E.g., **LOAD DATA LOCAL INFILE** in MySQL.

```
LOAD DATA LOCAL INFILE  'text.txt'
INTO TABLE Branch
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\n';
```

```
B1;Central;7100000
B2;Causeway Bay;9000000
B3;Aberdeen;400000
B4; North Point;3700000
…
```

text.txt

The DELETE clause

- The **DELETE FROM** command is used to delete records (tuples) from a database table.
  - **Query:** Delete all records from the Branch table.

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Branch

→

branch_id	name	asset
Empty		

Branch

```
DELETE FROM Branch;
```

The DELETE clause

- The **DELETE FROM** command is used to delete records (tuples) from a database table.
  - **Query:** Delete the branch "Central" from the Branch table.

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Branch

→

branch_id	name	asset
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Branch

```
DELETE FROM Branch WHERE name = 'Central';
```

The tuples that satisfy the conditions specified here are deleted.

The UPDATE clause

- The **UPDATE** command is used to update records (tuples) from a database table.
  - **Query:** Update the asset of branch with branch_id 'B1' to $0.

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Branch

→

branch_id	name	asset
B1	Central	0
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Branch

```
UPDATE Branch
SET  asset = 0
WHERE branch_id = 'B1';
```

The UPDATE clause

- The UPDATE command can also be used with **arithmetic expressions**.
  - **Query:** Increase all accounts with balances over $500 by 6%.

account_id	branch_id	balance
A1	B1	500
A2	B2	400
A3	B2	900
A4	B1	700

Account

→

account_id	branch_id	balance
A1	B1	500
A2	B2	400
A3	B2	954
A4	B1	742

Account

```
UPDATE Account
SET balance = balance * 1.06
WHERE balance > 500;
```

The UPDATE clause

- The **UPDATE** command can also be used with **arithmetic expressions**.
  - **Query:** Increase all accounts with balances under $500 by 5% and all other accounts by 6%.

account_id	branch_id	balance
A1	B1	500
A2	B2	400
A3	B2	900
A4	B1	700

Account

account_id	branch_id	balance
A1	B1	530
A2	B2	420
A3	B2	954
A4	B1	742

Account

```
UPDATE Account
SET balance = balance * 1.05
WHERE balance < 500;
```

```
UPDATE Account
SET balance = balance * 1.06
WHERE balance >= 500;
```

The order of executing these two is important!

The UPDATE clause

- The CASE command can be used to perform conditional update.

```
UPDATE Account
SET balance = CASE
WHEN balance <=500 THEN balance *1.05
ELSE balance * 1.06
END
```

Note: When there are multiple **WHEN … THEN** in the query, only the first true statement (from top to bottom) will be executed.

Section 3

Querying

The SELECT clause

- The **SELECT** clause lists the attributes desired in the result of a query.
  - **Query:** Find the names of all customers.

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Customer

name
Kit
Ben
Jolly
Yvonne

Result

```
SELECT name FROM Customer;
```

- An asterisk in the select clause denotes "all attributes"
  - **Query:** List all column values of all customer records.

```
SELECT * FROM Customer;
```

The SELECT clause

- The **SELECT** clause can contain **arithmetic expressions** (+, –, *, /) operating on constants or attributes of tuples.
  - **Query:** List the loan_id and amount of each loan record, display the amount in USD (originally stored in HKD).

loan_id	branch_id	amount
L1	B3	900
L2	B2	1500
L3	B1	1000

Loan

→

loan_id	amount /7.8
L1	115.385
L2	192.308
L3	128.205

Loan

```
SELECT loan_id, amount/7.8
FROM Loan;
```

The FROM clause

- The **FROM** clause lists the relations (tables) involved in the query.
  - **Query:** Find the **Cartesian product** of Customer and Borrower

```
SELECT *
FROM Customer, Borrower;
```

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Customer

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Borrower

→

Cartesian product of A and B means generate **all possible pairs** of records from A and B.

customer_id	name	address	customer_id	loan_id

Cartesian product of Customer and Borrower

The **FROM** clause

- The **FROM** clause lists the relations (tables) involved in the query.
  - **Query:** Find the **Cartesian product** of Customer and Borrower

```
SELECT *
FROM Customer, Borrower;
```

customer_id	name	address	customer_id	loan_id
C1	Kit	CB320	C1	L3

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Customer

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Borrower

Cartesian product of A and B means generate **all possible pairs** of records from A and B.

Cartesian product of Customer and Borrower

The FROM clause

- The **FROM** clause lists the relations (tables) involved in the query.
  - **Query:** Find the **Cartesian product** of Customer and Borrower

```
SELECT *
FROM Customer, Borrower;
```

customer_id	name	address	customer_id	loan_id
C1	Kit	CB320	C1	L3
C2	Ben	CB326	C1	L3

Cartesian product of Customer and Borrower

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Customer

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Borrower

Cartesian product of A and B means generate **all possible pairs** of records from A and B.

The **FROM** clause

- The **FROM** clause lists the relations (tables) involved in the query.
  - **Query:** Find the **Cartesian product** of Customer and Borrower

```
SELECT *
FROM Customer, Borrower;
```

customer_id	name	address	customer_id	loan_id
C1	Kit	CB320	C1	L3
C2	Ben	CB326	C1	L3
C3	Jolly	CB311	C1	L3
C4	Yvonne	CB415	C1	L3
C1	Kit	CB320	C4	L2
C2	Ben	CB326	C4	L2
C3	Jolly	CB311	C4	L2
C4	Yvonne	CB415	C4	L2
C1	Kit	CB320	C2	L1
C2	Ben	CB326	C2	L1
C3	Jolly	CB311	C2	L1
C4	Yvonne	CB415	C2	L1

Cartesian product of Customer and Borrower

Cartesian product is the most primitive way of joining two tables. However, many resulting tuples are not very useful. Therefore, we often need to specify the joining condition to filter out the non-meaningful results.

The WHERE clause

- The **WHERE** clause specifies **conditions** that the result must satisfy.
  - **Query:** For each loan, find out the name of the customer who borrow the loan.

Let us learn the process of constructing the SQL for this query.

The WHERE clause

- The **WHERE** clause specifies **conditions** that the result must satisfy.
  - **Query:** For each loan, find out the name of the customer who borrow the loan.

Step 1. What are the table(s) that contain the information to answer this query?

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Borrower

Observation 1.
First, the information of customers (customer_id) who borrow loan is in the **Borrower** table.

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Customer

Observation 2.
Second, we need to find out the name of the customer, the name is in the **Customer** table.

The **WHERE** clause

SELECT Borrower.loan_id, Customer.name
FROM Customer, Borrower

Step 2. Now we want to relate two tables, if no conditions is specified, Cartesian product will be returned. What is the joining condition?

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Borrower

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Customer

→

customer_id	name	address	customer_id	loan_id
C1	Kit	CB320	C1	L3
C2	Ben	CB326	C1	L3
C3	Jolly	CB311	C1	L3
C4	Yvonne	CB415	C1	L3
C1	Kit	CB320	C4	L2
C2	Ben	CB326	C4	L2
C3	Jolly	CB311	C4	L2
C4	Yvonne	CB415	C4	L2
C1	Kit	CB320	C2	L1
C2	Ben	CB326	C2	L1
C3	Jolly	CB311	C2	L1
C4	Yvonne	CB415	C2	L1

Cartesian product of Customer and Borrower

The WHERE clause

SELECT Borrower.loan_id, Customer.name
FROM Customer, Borrower
WHERE **Customer.customer_id = Borrower.customer_id**

loan_id	name
L3	Kit
L2	Yvonne
L1	Ben

Result

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Borrower

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Customer

customer_id	name	address	customer_id	loan_id
C1	Kit	CB320	C1	L3
C2	Ben	CB326	C1	L3
C3	Jolly	CB311	C1	L3
C4	Yvonne	CB415	C1	L3
C1	Kit	CB320	C4	L2
C2	Ben	CB326	C4	L2
C3	Jolly	CB311	C4	L2
C4	Yvonne	CB415	C4	L2
C1	Kit	CB320	C2	L1
C2	Ben	CB326	C2	L1
C3	Jolly	CB311	C2	L1
C4	Yvonne	CB415	C2	L1

Cartesian product of Customer and Borrower

The WHERE clause

- The **WHERE** clause specifies **conditions** that the result must satisfy.
- Comparison results can be combined using logical connectives **AND**, **OR**, and **NOT**.
  - **Query:** Find all loan ID of loans made at branch_id B1 with loan amounts >$1200.

There are two conditions in the query!

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

Loan

loan_id
L2

Result

```
SELECT loan_id
FROM Loan
WHERE branch_id = 'B1' AND
        amount > 1200;
```

Exercise

- **Query:** Find the names of all branches that have a loan.

Branch

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

Borrower

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Account

branch_id	account_id	balance
B1	A1	500
B2	A2	400
B2	A3	900
B1	A4	700

Owner

account_id	customer_id
A1	C1
A1	C2
A3	C3
A4	C4
A5	C4

Customer

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

Exercise

- **Query:** Find the names of all branches that have a loan.
  - **Step 1.** Identify the tables that contain the necessary information to answer the query.

Branch

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

  - **Step 2.** Construct the **SELECT** statement.

```
SELECT ?
FROM Branch, Loan
WHERE ?
;
```

Exercise

- **Query:** Find the names of all branches that have a loan.
  - **Step 1.** Identify the tables that contain the necessary information to answer the query.

Branch

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

  - **Step 2.** Construct the **SELECT** statement.

```
SELECT Branch.name
FROM Branch, Loan
WHERE ?
;
```

Exercise

- **Query:** Find the names of all branches that have a loan.
  - Step 1. Identify the tables that contain the necessary information to answer the query.

Branch

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

Usually, when **linking the information of two tables**, we need to specify **the joining condition.** Often we need to join the columns that participate in the referential constraint between the two tables.

  - Step 2. Construct the **SELECT** statement.

```
SELECT Branch.name
FROM Branch, Loan
WHERE Branch.branch_id = Loan.branch_id
;
```

Joining condition

Exercise

- **Query:** Find the names of all branches that have a loan.
  - **Step 1.** Identify the tables that contain the necessary information to answer the query.

Branch

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

Duplicate values return!

  - **Step 2.** Construct the **SELECT** statement.

```
SELECT Branch.name
FROM Branch, Loan
WHERE Branch.branch_id = Loan.branch_id
;
```

name
Central
Central
Aberdeen

Result

Exercise

- **Query:** Find the names of all branches that have a loan.
  - **Step 1.** Identify the tables that contain the necessary information to answer the query.

You can eliminate duplicate values in the results by using the **DISTINCT** keyword.

Branch

branch_id	name	asset
B1	Central	7100000
B2	Causeway Bay	9000000
B3	Aberdeen	400000
B4	North Point	3700000

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L2	1500
B1	L3	1000

Duplicate values return!

  - **Step 2.** Construct the **SELECT** statement.

```
SELECT DISTINCT Branch.name
FROM Branch, Loan
WHERE Branch.branch_id = Loan.branch_id
;
```

name
Central
Aberdeen

Result

Section 4

Renaming

Renaming

```
SELECT DISTINCT Branch.name
FROM Branch, Loan
WHERE Branch.branch_id = Loan.branch_id
;
```

name
Central
Aberdeen

Result

- Rename can be operated on both tables and **attributes**.
  - Rename on attribute.

I want to **rename** the column “name” in the result into “Branch name”.

We use the keyword **AS** to signify renaming.

```
SELECT DISTINCT Branch.name AS 'Branch name'
FROM Branch, Loan
WHERE Branch.branch_id = Loan.branch_id
;
```

Branch name
Central
Aberdeen

Result

Renaming

```
SELECT DISTINCT Branch.name
FROM Branch, Loan
WHERE Branch.branch_id = Loan.branch_id
;
```

name
Central
Aberdeen

Result

- Rename can be operated on both **tables** and attributes.
  - Rename on tables.

The two SQLs are equivalent to each other.

```
SELECT DISTINCT B.name
FROM Branch B, Loan L
WHERE B.branch_id = L.branch_id
;
```

name
Central
Aberdeen

Result

Section 5

String operations

The LIKE clause

- The most commonly used operation on strings is pattern matching using **LIKE**.
  - Percent(%):matches any substring.
  - Underscore(_): matches any character.
    - 'Perry%' matches any string beginning with “Perry”.
    - '_ _ _ %' matches any string of at least 3 characters.
- Note: Patterns are **case sensitive**.

The LIKE clause

- **Query:** Find the names of all customers whose address includes the substring '**320**'.

Customer

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

→

name
Kit

Result

https://dev.mysql.com/doc/refman/8.0/en/pattern-matching.html

```
SELECT name
FROM Customer
WHERE address LIKE '%320%';
```

Question: How about matching using regular expression? ☺

The LIKE clause

WHERE name LIKE 'a%'	Finds any values that start with "a"
WHERE name LIKE '%a'	Finds any values that end with "a"
WHERE name LIKE '%or%'	Finds any values that have "or" in any position
WHERE name LIKE '_r%'	Finds any values that have "r" in the second position
WHERE name LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE name LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

Section 6

Ordering results

The ORDER BY clause

- The **ORDER BY** clause list the result in sorted order.
  - **Query:** List the names of all customers **in alphabetic order.**

Customer

customer_id	name	address
C1	Kit	CB320
C2	Ben	CB326
C3	Jolly	CB311
C4	Yvonne	CB415

name
Ben
Jolly
Kit
Yvonne

Result

```
SELECT name
FROM Customer
ORDER BY name ASC;
```

  - Use **DESC** for descending order, and **ASC** for ascending order. Default: ascending

name
Yvonne
Kit
Jolly
Ben

Result

```
SELECT name
FROM Customer
ORDER BY name DESC;
```

The ORDER BY clause

- The **ORDER BY** clause list the result in sorted order.
  - **Query:** List the loan records in **ascending order of the branch_id**, if two tuples having the same branch_id, order by their **loan amount in descending order.**

Loan

branch_id	loan_id	amount
B3	L1	900
B1	L3	1000
B1	L5	1500

branch_id	loan_id	Amount
B1	L3	1000
B1	L5	1500
B3	L1	900

Intermediate Result

branch_id	loan_id	Amount
B1	L2	1500
B1	L3	1000
B3	L1	900

Final Result

```
SELECT *
FROM Loan
ORDER BY branch_id ASC,
              amount DESC;
```

Section 7

Simple Nested Query

The IN clause

- The **IN** clause allows you to specify discrete values in the **WHERE** search criteria.
  - **Query:** Find the customer_id of all customers who have both an account and a loan.

Borrower

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Owner

account_id	customer_id
A1	C1
A1	C2
A2	C2

→

customer_id
C1
C2

Result

```
SELECT DISTINCT customer_id
FROM Borrower
WHERE customer_id IN
        (SELECT customer_id FROM Owner);
```

The result of this sub-query is {C1,C2,C2}.

The IN clause

- The **IN** clause allows you to specify discrete values in the **WHERE** search criteria.
  - **Query:** Find the customer_id of all customers who have a lone but **not having an account**.

Borrower

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Owner

account_id	customer_id
A1	C1
A1	C2
A2	C2

→

customer_id
C4

Result

```
SELECT DISTINCT customer_id
FROM Borrower
WHERE customer_id NOT IN
        (SELECT customer_id FROM Owner);
```

The result of this sub-query is {C1,C2,C2}.

Section 8

Aggregation

Aggregate functions

- Aggregation functions take a collection of values as input and return a **single value**.
  - **Query:** Find the **average** balance of all accounts at the branch with branch_id 'B2'.

Account

branch_id	account_id	balance
B1	A1	500
B2	A2	400
B2	A3	900
B1	A4	700

AVG (balance)
650.0000

Result

```
SELECT AVG(balance)
FROM Account
WHERE branch_id = 'B2';
```

Aggregate functions

- Aggregation functions.
  - AVG
  - MIN
  - MAX
  - SUM
  - COUNT

The **GROUP BY** clause

- Aggregation function can be applied to **a group of sets of tuples** by using **GROUP BY** clause.
  - **Query:** Find the **average** balance at each branch.

Step1. Grouping

GROUP BY branch_id

Account

branch_id	account_id	balance
B1	A1	500
B2	A2	400
B2	A3	900
B1	A4	700

branch_id	account_id	balance
B1	A1	500
	A4	700
B2	A2	400
	A3	900

The **GROUP BY** clause

- Aggregation function can be applied to **a group of sets of tuples** by using **GROUP BY** clause.
  - **Query:** Find the **average** balance at each branch.

```
SELECT branch_id, AVG(balance)
FROM Account
GROUP BY branch_id;
```

Account

branch_id	account_id	balance
B1	A1	500
B2	A2	400
B2	A3	900
B1	A4	700

Step1. Grouping
GROUP BY branch_id

branch_id	account_id	balance
B1	A1	500
	A4	700
B2	A2	400
	A3	900

Step2. Aggregation
AVG(balance)

branch_id	AVG (balance)
B1	600.0000
B2	650.0000

Result

The **HAVING** clause

- It is useful to state a condition that applies to **groups** rather than to tuples.
  - **Query:** Find the branches where the average account balance is no less than $650.

```
SELECT branch_id, AVG(balance)
FROM Account
GROUP BY branch_id
HAVING AVG(balance) >= 650;
```

branch_id	AVG (balance)
B2	650.0000

Result

Step3. Filtering
(Having AVG(balance) >= 650)

Account

branch_id	account_id	balance
B1	A1	500
B2	A2	400
B2	A3	900
B1	A4	700

branch_id	account_id	balance
B1	A1	500
	A4	700
B2	A2	400
	A3	900

branch_id	AVG (balance)
B1	600.0000
B2	650.0000

Section 9

Join

Join

- A join takes 2 tables as input and returns a table.

Employee

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	**NULL**

Department

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

Cartesian product, then
E.department_id = D.department_id

```
SELECT *
FROM Employee E, Department D
WHERE E.department_id =
        D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME

Result

The OUTER JOIN clause

- An **outer join** does not require each record in the two joined tables to have a matching record.

Employee

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

Department

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

Even if the **LEFT table record does not have matching records in the RIGHT table**, we still output the tuple in the LEFT table (with null values for the columns of the RIGHT table).

```
SELECT *
FROM Employee E LEFT OUTER JOIN
Department D
ON E.department_id =
D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
David	NULL	NULL	NULL

Result

The OUTER JOIN clause

- An **outer join** does not require each record in the two joined tables to have a matching record.

Employee

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

Department

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

Even if the **RIGHT table record does not have matching records in the LEFT table**, we still output the tuple in the RIGHT table (with null values for the columns of the LEFT table).

```
SELECT *
FROM Employee E RIGHT OUTER JOIN
Department D
ON E.department_id =
D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
NULL	NULL	35	EEE

Result

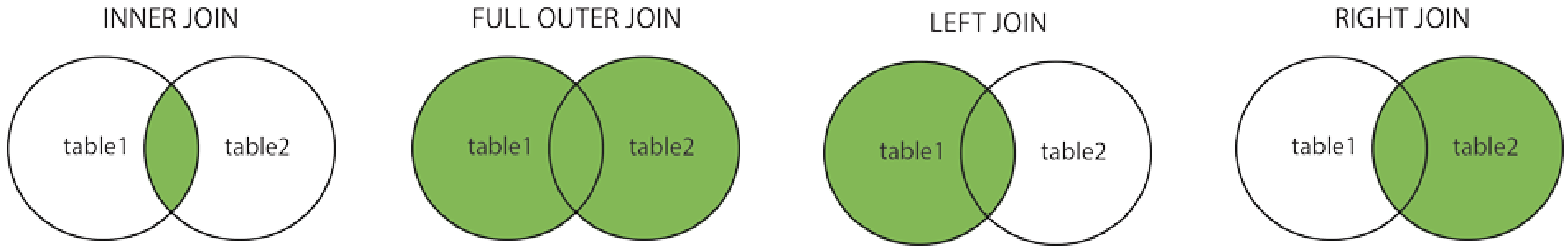
Here are the different types of the JOINs in SQL:

•**(INNER) JOIN**: Returns records that have matching values in both tables

•**LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table

•**RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table

•**FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table

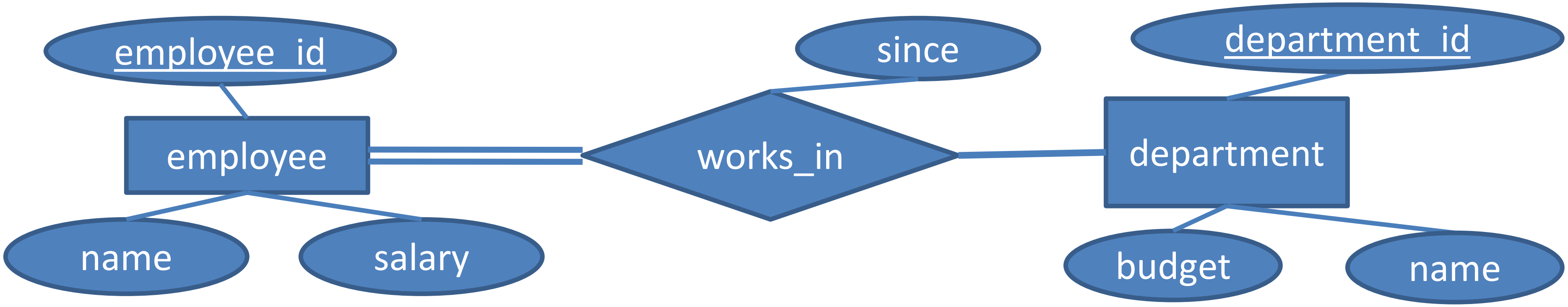


Section 10 More on SQL

- An Example
- Set operations
- More on nested queries
- Null values
- Views
- Authorization
- Assertion
- Other SQL constructs

Example

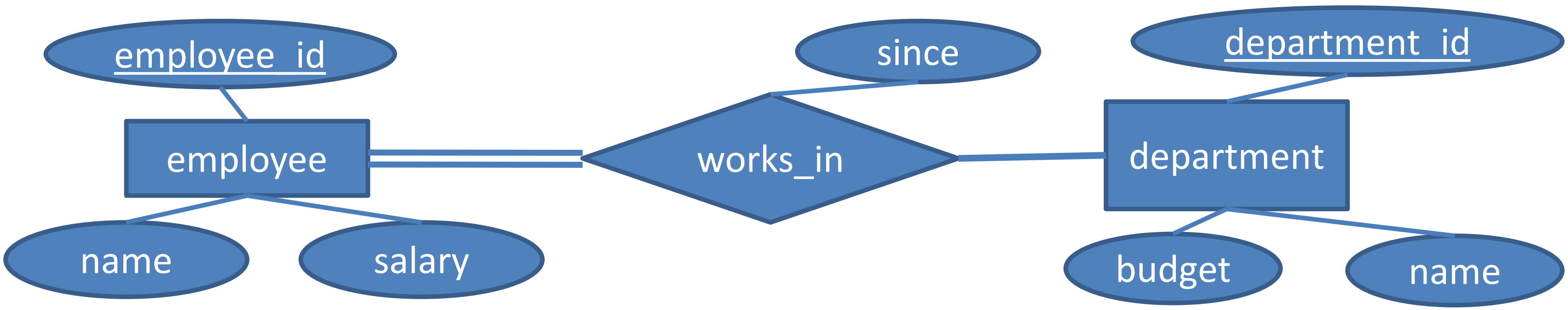
- **Step 1.** **Information modeling.**



Before we proceed to learning more SQL constructs, lets have a revision on what we have learned up to this chapter.

Example

- **Step 1. Information modeling**



- **Step 2. Reduce to database tables**
  - **Employees** (<u>employee_id</u>, name, salary)
    Foreign key : none.
  - **Departments** (<u>department_id</u>, name, budget)
    Foreign key : none.
  - **Works_in**(<u>employee_id</u>, <u>department_id</u>, since)
    Foreign key : employee_id **REFERENCES** Employee (employee_id).
    department_id **REFERENCES** Department (department_id).

Example

- **Step 3.** **Create the database**

```
CREATE TABLE Employees (
    employee_id INT(12),
    name VARCHAR(30) NOT NULL,
    salary INT UNSIGNED NOT NULL,
    PRIMARY KEY(employee_id)
)ENGINE = INNODB;
```

INNODB storage engine, just for MySQL to support foreign key constraints.

```
CREATE TABLE Departments (
    department_id INT(12),
    name VARCHAR(30) NOT NULL,
    budget INT UNSIGNED NOT NULL,
    PRIMARY KEY(department_id)
) ENGINE = INNODB;
```

```
CREATE TABLE Works_in(
    employee_id INT(12),
    department_id INT(12),
    since DATE NOT NULL,
    PRIMARY KEY(employee_id, department_id),
    FOREIGN KEY (employee_id) REFERENCES Employees (employee_id),
    FOREIGN KEY (department_id) REFERENCES Departments (department_id)
) ENGINE = INNODB;
```

Example

- **Step 3.** **Create the database**

```
INSERT INTO Employees VALUES ( 1, 'Jones', 26000);
INSERT INTO Employees VALUES ( 2, 'Smith', 28000);
INSERT INTO Employees VALUES ( 3, 'Parker', 35000);
INSERT INTO Employees VALUES ( 4, 'Smith', 24000);
```

```
INSERT INTO Departments VALUES ( 1, 'Toys', 122000), ( 2, 'Tools', 239000), ( 3, 'Food', 100000);
```

```
INSERT INTO Works_in VALUES ( 1, 1, '2001-1-1'), ( 2, 1, '2002-4-1'), ( 2, 2, '2005-2-2'), ( 3, 3,
'2003-1-1'), ( 4, 3, '2005-1-1');
```

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Example

- **Step 4. Design the SQL to access data for the application**
  - **Query 1:** Find the names of all employees and remove duplicates.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 2:** Find the employee_ids and names of employees who work in department with department_id=2.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 3:** Find the dept. names where employee with employee_id = 2 works.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 4:** Find the dept. ids where employees named Smith work.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 5:** Find the dept. names where employees named Smith work.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 6:** Find the names of departments which have an employee named Smith and their budget is greater than 100000.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 7:** Find the budgets of departments, who employ an employee called 'Smith' .

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 8:** For each department, find the total number of employees it employs.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 9:** Find the dept. names with **at least 2** employee.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 10:** Find the employee_id of all employees whose name includes the substring “one”.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 11:** Find the employee_id and name of the employees who worked in the departments with budget more than 100,000.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 12:** Find the name and budget of the department with the greatest budget.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 13:** Find the names of employees who work in at least 2 departments.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Exercises

- **Query 14:** In each department, find the highest salary of the employee in that department.

Employees

employee_id	name	salary
1	Jones	26000
2	Smith	28000
3	Parker	35000
4	Smith	24000

Works_in

employee_id	department_id	since
1	1	2001-1-1
2	1	2002-4-1
2	2	2005-2-2
3	3	2003-1-1
4	3	2005-1-1

Departments

department_id	name	budget
1	Toys	122000
2	Tools	239000
3	Food	100000

Q&A Game

Please read the below table schemas and answer four SQL questions.

- **Restaurant** (*restaurant_id*, *name*)
- **RestaurantCategory** (*restaurant_id*, *category*)
  - Foreign key: {*restaurant_id*} references **Restaurant**
- **Branch** (*restaurant_id*, *branch_no*, *location*, *seats*)
  - Foreign key: {*restaurant_id*} references **Restaurant**
- **Member** (*member_id*, *name*, *birthday, joined, points*)
  - (*joined* is the year the member joined)
- **Visits** (*visit_id*, *member_id*, *restaurant_id*, *branch_no*, *date*, *score*)
  - Foreign keys:{restaurant_id, branch_no} references **Branch,**
    {*member_id*} references **Member**

Restaurant

restaurant_id	name
1	McRonalds
2	PizzaHub
3	DeliItaly
4	UltraSandwich
5	Starducks

RestaurantCategory

restaurant_id	category
1	Fast Food
1	Take Away
2	Italian
3	Cafe
3	Italian
4	Light meal
5	Cafe
5	Light meal
5	Take Away

Member

member_id	name	birthday	joined	points
1	Cleo	1983-09-25	2017	690
2	Evan	1988-03-28	1989	130
3	Todd	1966-06-22	1967	190
4	Lonny	1973-04-05	2016	10
5	Keith	1991-06-19	1992	380
6	Royce	1965-06-14	2006	300
7	Mavis	1977-08-21	1981	840
8	Alvin	1998-04-17	2008	900
9	Ira	1993-07-17	2015	100
10	Dino	1968-12-26	2012	150
11	A. Bella	1989-11-26	2016	20

Visits

visit_id	member_id	restaurant_id	branch_no	date	score
1	1	4	3	2015-12-30	5
2	8	4	3	2016-01-15	4
3	1	2	1	2018-06-16	5
4	2	3	2	2018-06-14	5
5	1	2	1	2018-12-09	3
6	3	3	2	2018-01-16	6
7	2	4	1	2018-03-26	3
8	4	3	2	2018-08-02	7
9	5	3	1	2018-04-22	8
10	4	5	2	2018-05-10	0
11	5	5	2	2018-05-08	6
12	3	5	1	2018-06-21	1
13	6	5	1	2018-03-06	2
14	4	5	1	2018-02-12	3
15	7	4	2	2018-06-07	1
16	5	4	2	2018-12-04	2
17	3	4	2	2018-01-19	3
18	5	3	3	2018-01-27	5

Branch

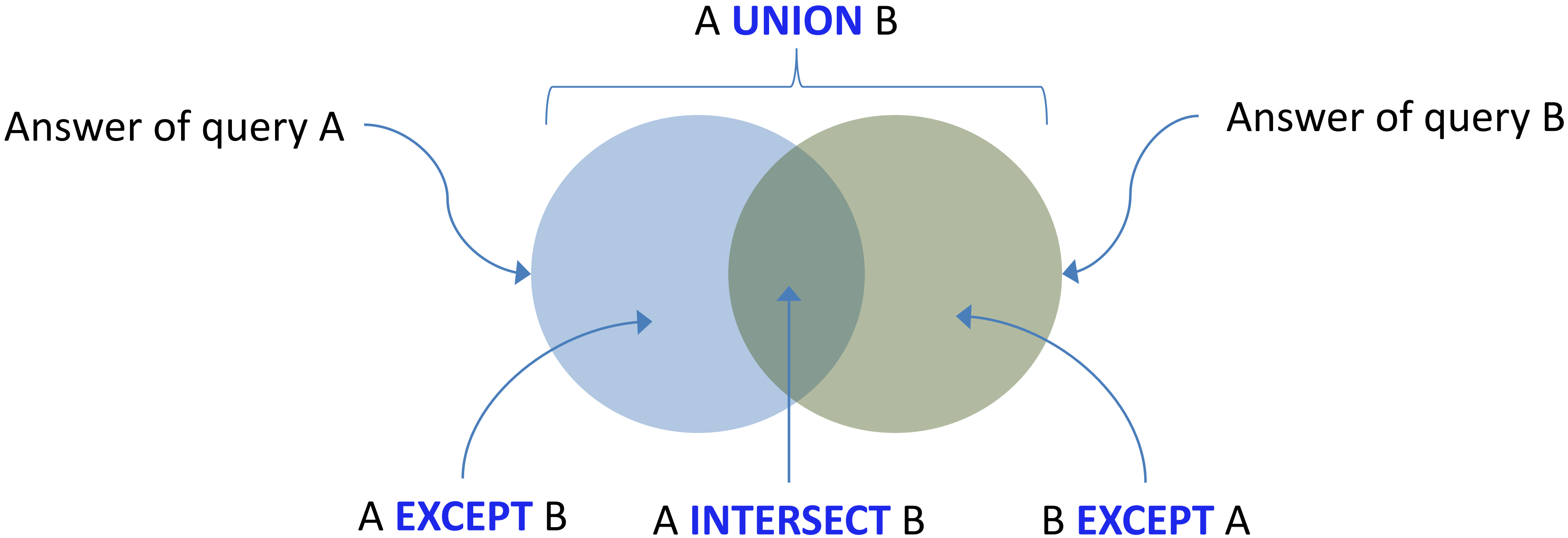
restaurant_id	branch_no	location	seats
1	1	Admiralty	10
1	2	Central	20
2	1	Causeway Bay	5
3	1	Admiralty	25
3	2	Wan Chai	45
3	3	Causeway Bay	35
4	1	Central	170
4	2	Admiralty	100
4	3	North Point	120
5	1	Central	80
5	2	Wan Chai	40

Section 11

Set Operations

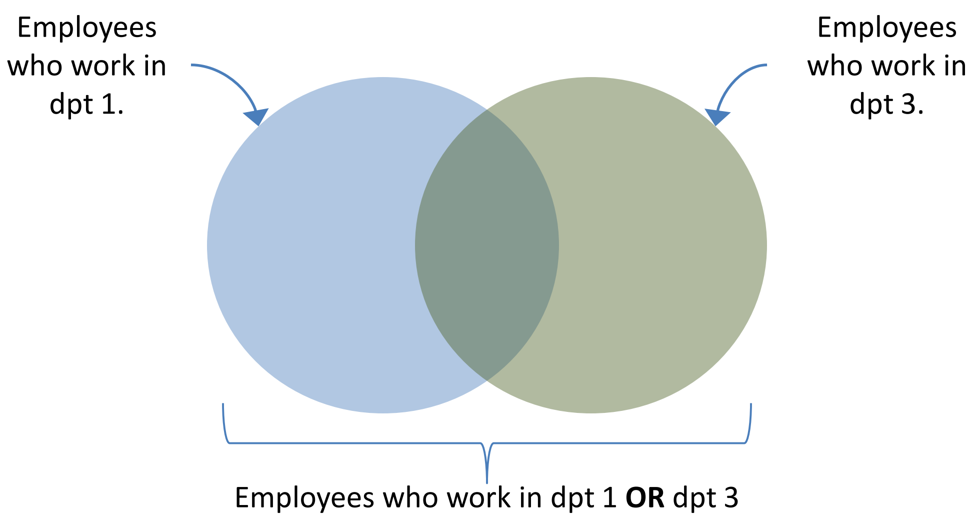
Set operations

- Set operations can be expressed in SQL using clauses **UNION**, **INTERSECT**, **EXCEPT**.
- Using the set operations can ease the design of SQL by breaking down a complex query to a number of simpler sub-queries.



The UNION clause

- **Query:** Find the names of employees who work in department 1 **or** department 3.




```
SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id= W.employee_id AND
        (W. department_id = 1 OR
         W. department_id = 3)
```

```
SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id = W.employee_id AND
        W.department_id = 1

UNION

SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id = W.employee_id AND
         W. department_id = 3
```

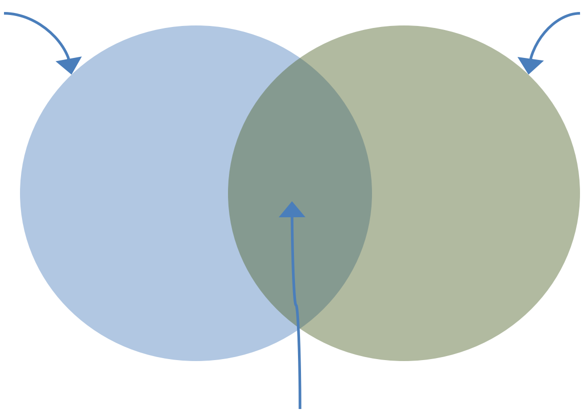
Note : The two SQLs are NOT equivalent to each other!
Duplicates are eliminated when two sets are unified.

The INTERSECT clause

- **Query:** Find the name of employees who work in department 1 **and** department 3.

Employees who work in dpt 1.

Employees who work in dpt 3.



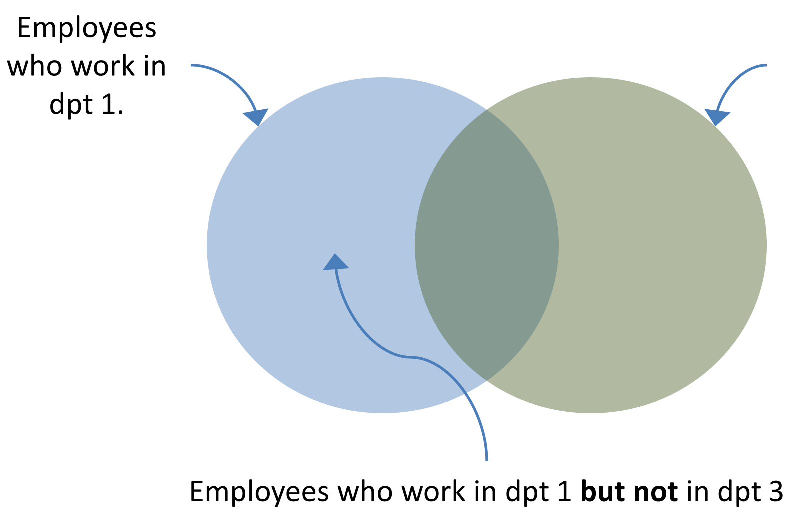
Employees who work in both dpt 1 **AND** dpt 3

```
SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id = W.employee_id AND
         W.department_id = 1

INTERSECT

SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id = W.employee_id AND
          W. department_id = 3
```

Note : MySQL doesn't support the keyword INTERSECT.
But we can replace INTERSECT by joining tables.

The **EXCEPT** clause

- **Query:** Find the name of employees who work in department 1 **but not** department 3.




```sql
SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id = W.employee_id AND
        W.department_id = 1

EXCEPT

SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id = W.employee_id AND
        W. department_id = 3
```

Note : MySQL doesn't support the keyword EXCEPT.
But we can replace EXCEPT by using NOT IN.

Section 12

More on Nested Queries

Nested queries

- Nested queries have other **subqueries** embedded in them.
- Used for the ease of expressing a natural language request in SQL.
- Subqueries are usually nested under **WHERE** clauses.
  - May also be enclosed under **FROM** or **HAVING** clauses

The IN clause

- **Query:** Find the names of the employees in department 1.

```
SELECT E.name
FROM Employees E, Works_in W
WHERE E.employee_id = W.employee_id AND
        W.department_id = 1;
```

- **In natural language :** Find employee names whose employee_id **appears in the set of** employee_ids working for department 1.

```
SELECT E.name
FROM Employees E
WHERE E.employee_id IN (
        SELECT W.employee_id
        FROM Works_in W
        WHERE W.department_id = 1);
```

Just like searching the employee_id in the **result** of the nested query.

The SOME clause

- **Query:** Find department names that have greater budget than **some** department where employee 4 works.

```
SELECT W.department_id
FROM Works_in W
WHERE W.employee_id = 4
```

Find the department ID where employee 4 works.

The SOME clause

- **Query:** Find department names that have greater budget than **some** department where employee 4 works.

```
SELECT D2.budget
FROM Departments D2
WHERE D2.department_id IN (
        SELECT W.department_id
        FROM Works_in W
        WHERE W.employee_id = 4
 )
```

Find the budget of those departments.

Find the department ID where employee 4 works.

The SOME clause

- **Query:** Find department names that have greater budget than **some** department where employee 4 works.

```
SELECT D.name
FROM Departments D
WHERE D.budget > SOME(
        SELECT D2.budget
        FROM Departments D2
        WHERE D2.department_id IN (
              SELECT W.department_id
              FROM Works_in W
              WHERE W.employee_id = 4
        )
);
```

Find the name of the department with budget > some budgets (those returned by inner query).

Find the budget of those departments.

Find the department ID where employee 4 works.

- **IMPORTANT NOTE:** If nested query result is empty, then **> SOME** will return **false** for every D.budget!

The ALL clause

- **Query:** Find department names that have greater budget than **all** departments where employee 4 works.

```
SELECT W.department_id
FROM Works_in W
WHERE W.employee_id = 4
```

Find the department ID where employee 4 works.

The ALL clause

- **Query:** Find department names that have greater budget than **all** departments where employee 4 works.

```
SELECT D2.budget
FROM Departments D2
WHERE D2.department_id IN (
        SELECT W.department_id
        FROM Works_in W
        WHERE W.employee_id = 4
 )
```

Find the budget of those departments.

The ALL clause

- **Query:** Find department names that have greater budget than **all** departments where employee 4 works.

```
SELECT D.name
FROM Departments D
WHERE D.budget > ALL (
        SELECT D2.budget
        FROM Departments D2
        WHERE D2.department_id IN (
                SELECT W.department_id
                FROM Works_in W
                WHERE W.employee_id = 4
        )
);
```

Find the name of the department with budget > ALL budgets (those returned by inner query).

- **IMPORTANT NOTE:** If nested query result is empty, then **> ALL** will return **true** for every D.budget!

The ALL clause

- **Query:** Find department names that have the greatest budget than **all** departments.

```
SELECT D.name
FROM Departments D
WHERE D.budget >= ALL (
        SELECT D2.budget
        FROM Departments D2
);
```

- **Question:** What would the result be if **>ALL** is used?

Can you rewrite the above query using **Aggregate function MAX** in a **nested query**?

The EXISTS clause

- The inner subquery could depend on the row **currently examined** in the outer query.
  - **Query:** Find the names of employees who work in department with department_id=1.

```
SELECT E.name
FROM Employees E
WHERE EXISTS (
        SELECT *
        FROM Works_in W
        WHERE W.department_id = 1 AND
              E.employee_id = W.employee_id);
```

For each employee record ***r***. If the inner query can return some records, we will return the record ***r***.

- **EXISTS** is a boolean set-comparison operator that returns **false** if the input set is empty and **true** otherwise.

Section 13

Null values

NULL value

- Handling null values is a non-trivial topic in database research.
- **null**: **unknown** value or value **does not exist**.
- Use predicate **IS NULL** to check for **null** values.
  - **Query:** Find all employee names for which the salary is unknown or undetermined.

Employees

employee_id	name	salary
1	Jones	
2	Smith	28000
3	Parker	
4	Smith	24000

```
SELECT name
FROM Employees
WHERE salary IS NULL
```

NULL value

- The result of any arithmetic expression involving **null** is **null**.
  - **5** + **null** returns **null**.
- Any comparison with **null** returns **UNKNOWN**.
  - Both **5** < **null** , **null** = **null** return **UNKNOWN**.
- Use ***P*** **IS UNKNOWN** to check if a predicate ***P*** is unknown or not.
- For the result of **WHERE** or **HAVING** clause, predicate is **false** if it evaluates to **UNKNOWN**.

Three valued logic

OR

	T	Un	F
T	T	T	T
Un	T	Un	Un
F	T	Un	F

AND

	T	Un	F
T	T	Un	F
Un	Un	Un	F
F	F	F	F

NOT

T	F
Un	Un
F	T

NULL value and aggregates

```
SELECT SUM (budget)
FROM Departments
```

- The statement above ignores **null** amounts.
- All aggregate operations except **COUNT(*)** ignore tuples with **null** values on the aggregated attributes.
  - **COUNT** counts not **null** values only.

Departments

department_id	name	budget
1	Toys	
2	Tools	
3	Food	100000

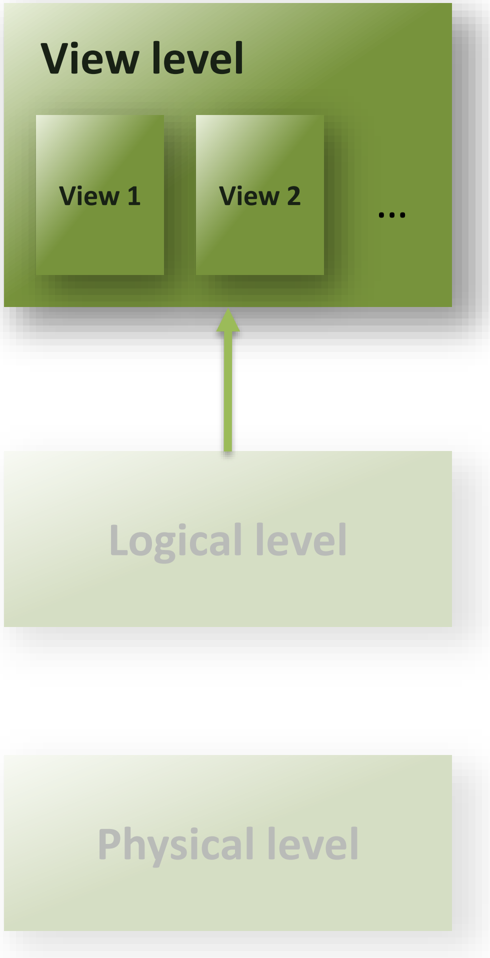
- **SUM**(budget) returns **100000**.
- **COUNT**(*) returns **3**.
- **COUNT**(budget) returns **1**.

Section 14

Views

The CREATE VIEW clause

- Views provide a mechanism to hide certain data from the view of certain users.



- **Syntax:**

  CREATE VIEW *view_name* **AS** *<expression>*

```
CREATE VIEW Employee_hide_salary AS (
    SELECT employee_id, name
    FROM Employees);
```

```
CREATE VIEW Dpt_size(name,num_of_employee) AS (
    SELECT D.name, COUNT(*)
    FROM Departments D, Works_in W
    WHERE D.department_id = W.department_id
    GROUP BY W.department_id );
```

Section 15

Authorization

Authorization

- The DBA can grant access/update authorization to users.
- **Syntax:**

GRANT <priviledge list>
ON <table name or view name>
TO <user/role list>

Johnson, Brown are usernames.

```
GRANT SELECT ON Departments TO Johnson, Brown
```

```
GRANT UPDATE(budget) ON Departments TO Johnson
```

```
GRANT UPDATE(budget) ON Departments TO manager
```

manager is not a username, it is a **role**.

Authorization

- Rights can be revoked.

```
REVOKE SELECT ON Departments FROM Johnson, Brown
```

- Create a role.

```
CREATE ROLE manager;
```

- Grant a role to a user.

```
GRANT manager TO Brown;
```

Section 16

Assertion

Assertions

- An **assertion** ensures a certain condition will always exist in the database.
  - Assume that we want to enforce that the number of departments cannot exceed the number of employees at any valid instance of our database.

```
CREATE ASSERTION EmpsNoLessThanDepts
CHECK (
              (SELECT COUNT(*) FROM Departments)
               <=
              (SELECT COUNT(*) FROM Employees)
          );
```

- **Assertions are checked every time the involved tables are updated** and they could be very expensive.

Section 17 With clause

A common table expression (CTE) is a named temporary result set that exists within the scope of a single statement and that can be referred to later within that statement, possibly multiple times.

```
WITH cte1 AS (SELECT a, b FROM table1),
cte2 AS (SELECT c, d FROM table2)
SELECT b, d FROM cte1
INNER JOIN cte2
ON cte1.a = cte2.c;
```

Section 17 With clause

```
WITH cte (col1, col2) AS
( SELECT 1, 2
UNION ALL
SELECT 3, 4 )
SELECT col1, col2 FROM cte;
```

Section 17 With clause

```sql
WITH cte (col1, col2) AS
( SELECT 1, 2
UNION ALL
SELECT 3, 4 )
SELECT col1, col2 FROM cte;
```

col1	col2
1	2
3	4

Section 17 With clause

A recursive common table expression is one having a subquery that refers to its own name.

```
WITH RECURSIVE cte (n) AS
( SELECT 1
UNION ALL
SELECT n + 1 FROM cte
WHERE n < 5 )
SELECT * FROM cte;
```

```
+------+
| n    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
+------+
```

The first SELECT produces the initial row or rows for the CTE and does not refer to the CTE name. The second SELECT produces additional rows and recurses by referring to the CTE name in its FROM clause. Recursion ends when this part produces no new rows. Thus, a recursive CTE consists of a nonrecursive SELECT part followed by a recursive SELECT part.

The recursive CTE subquery shown earlier has this nonrecursive part that retrieves a single row to produce the initial row set:

```
SELECT 1
```

The CTE subquery also has this recursive part:

```
SELECT n + 1 FROM cte WHERE n < 5
```

At each iteration, that `SELECT` produces a row with a new value one greater than the value of `n` from the previous row set. The first iteration operates on the initial row set (`1`) and produces `1+1=2`; the second iteration operates on the first iteration's row set (`2`) and produces `2+1=3`; and so forth. This continues until recursion ends, which occurs when `n` is no longer less than 5.

Section 17 With clause

A Fibonacci series begins with the two numbers 0 and 1 (or 1 and 1) and each number after that is the sum of the previous two numbers.

```
WITH RECURSIVE fibonacci (n,
fib_n, next_fib_n) AS
( SELECT 1, 0, 1
UNION ALL
SELECT n + 1, next_fib_n, fib_n +
next_fib_n FROM fibonacci
WHERE n < 10 )
SELECT * FROM fibonacci;
```

```
+------+-------+------------+
| n    | fib_n | next_fib_n |
+------+-------+------------+
|    1 |     0 |          1 |
|    2 |     1 |          1 |
|    3 |     1 |          2 |
|    4 |     2 |          3 |
|    5 |     3 |          5 |
|    6 |     5 |          8 |
|    7 |     8 |         13 |
|    8 |    13 |         21 |
|    9 |    21 |         34 |
|   10 |    34 |         55 |
+------+-------+------------+
```

Lecture 4

END

COMP3278A

Introduction to Database Management Systems

Dr. Ping Luo
Email : pluo@cs.hku.hk

Department of Computer Science, The University of Hong Kong