Lecture 4

Structured Query Language (SQL I)

COMP3278B

Introduction to Database Management Systems

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

Department of Computer Science, The University of Hong Kong

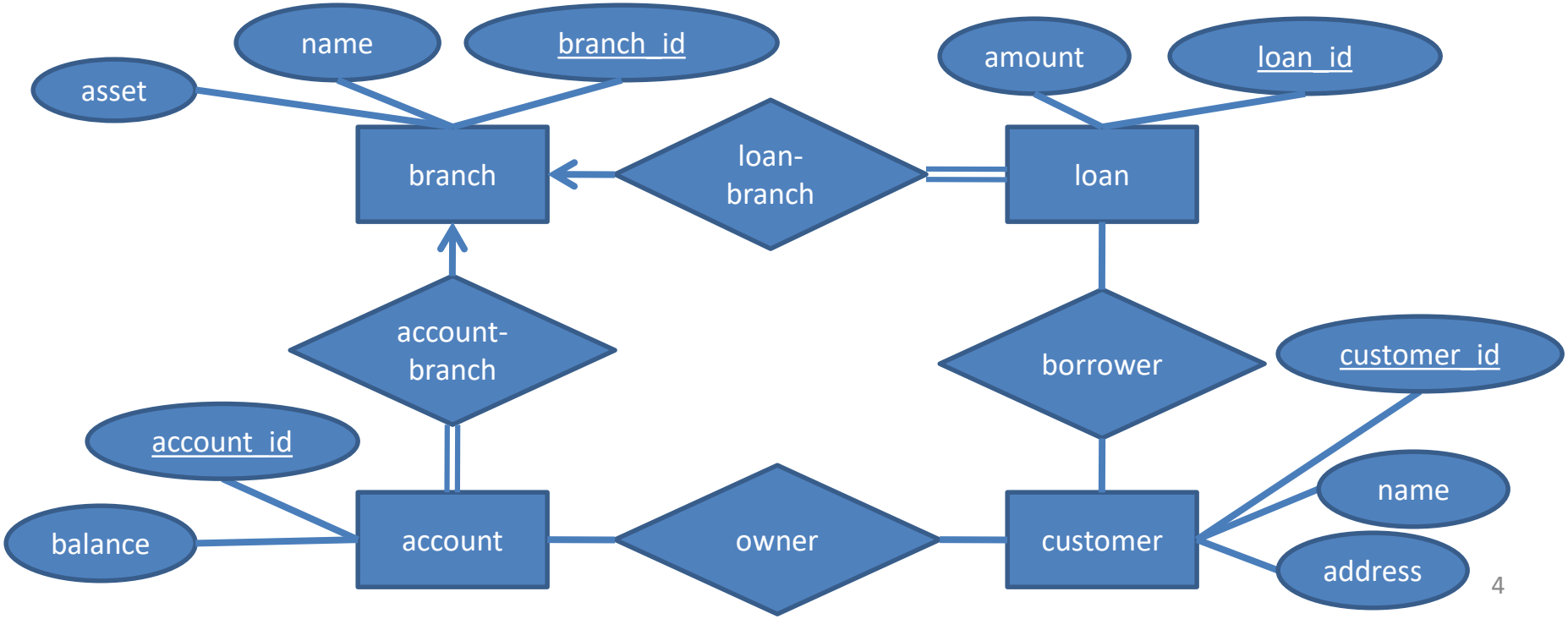
Acknowledgement: Dr. Chui Chun Kit

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



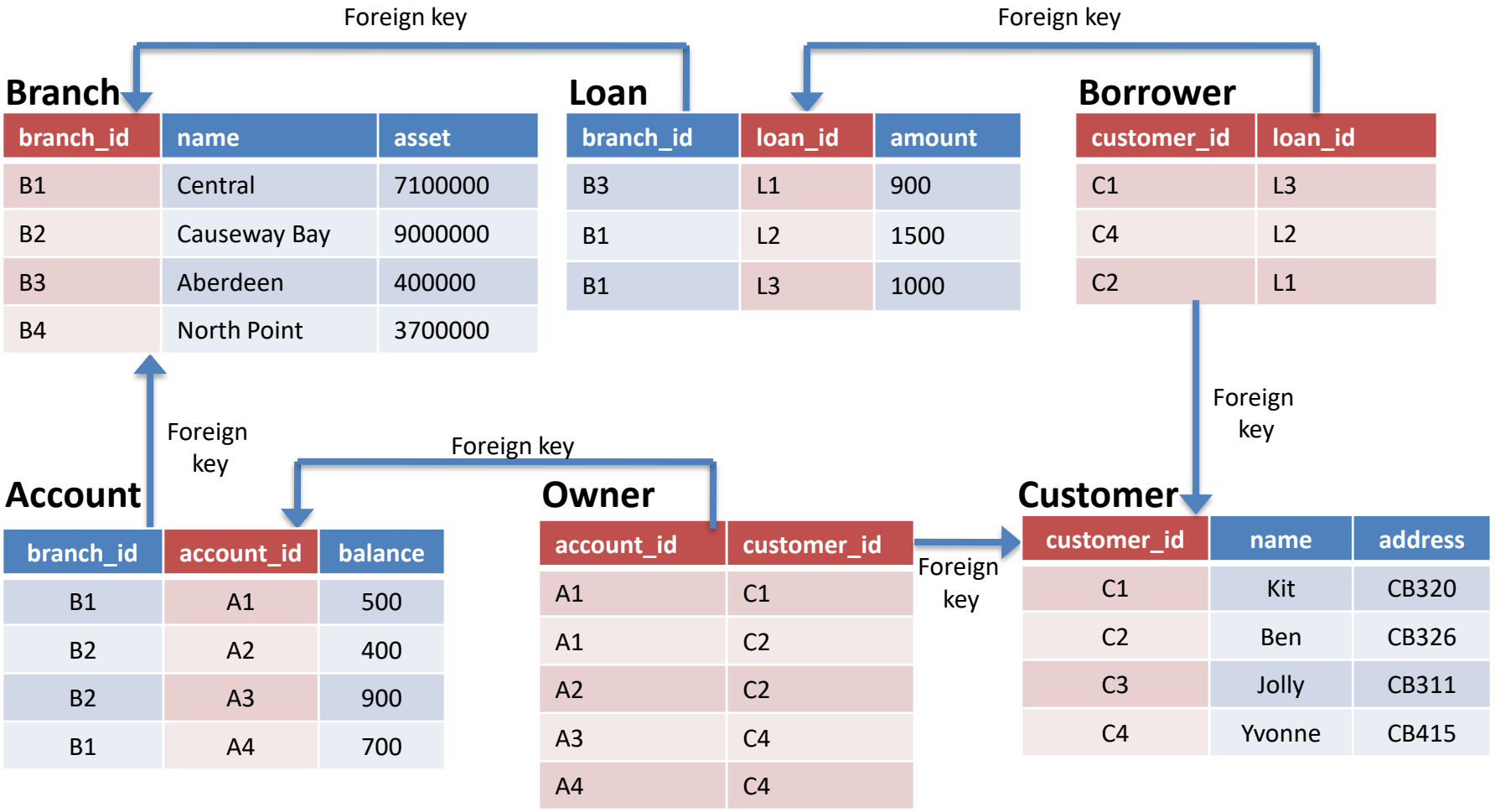
Recap

- **Step 2. Reduce to database table definitions**
  - **Branch** (branch_id, name, asset)
    Foreign key : none.
  - **Loan**(loan_id, amount, branch_id)
    Foreign key : branch_id **REFERENCES** Branch (branch_id).
  - **Customer**(customer_id, name, address)
    Foreign key : none.
  - **Account**(account_id, balance, branch_id)
    Foreign key : branch_id **REFERENCES** Branch (branch_id).
  - **Borrower**(loan_id, customer_id)
    Foreign key : loan_id **REFERENCES** Loan (loan_id).
    customer_id **REFERENCES** Customer (customer_id).
  - **Owner**(account_id, customer_id)
    Foreign key : account_id **REFERENCES** Account (account_id).
    customer_id **REFERENCES** Customer (customer_id).

Recap

- **Step 3.** Create the database and tables
- **Step 4.** Design the SQL to access data for the application

Running example



What is SQL?

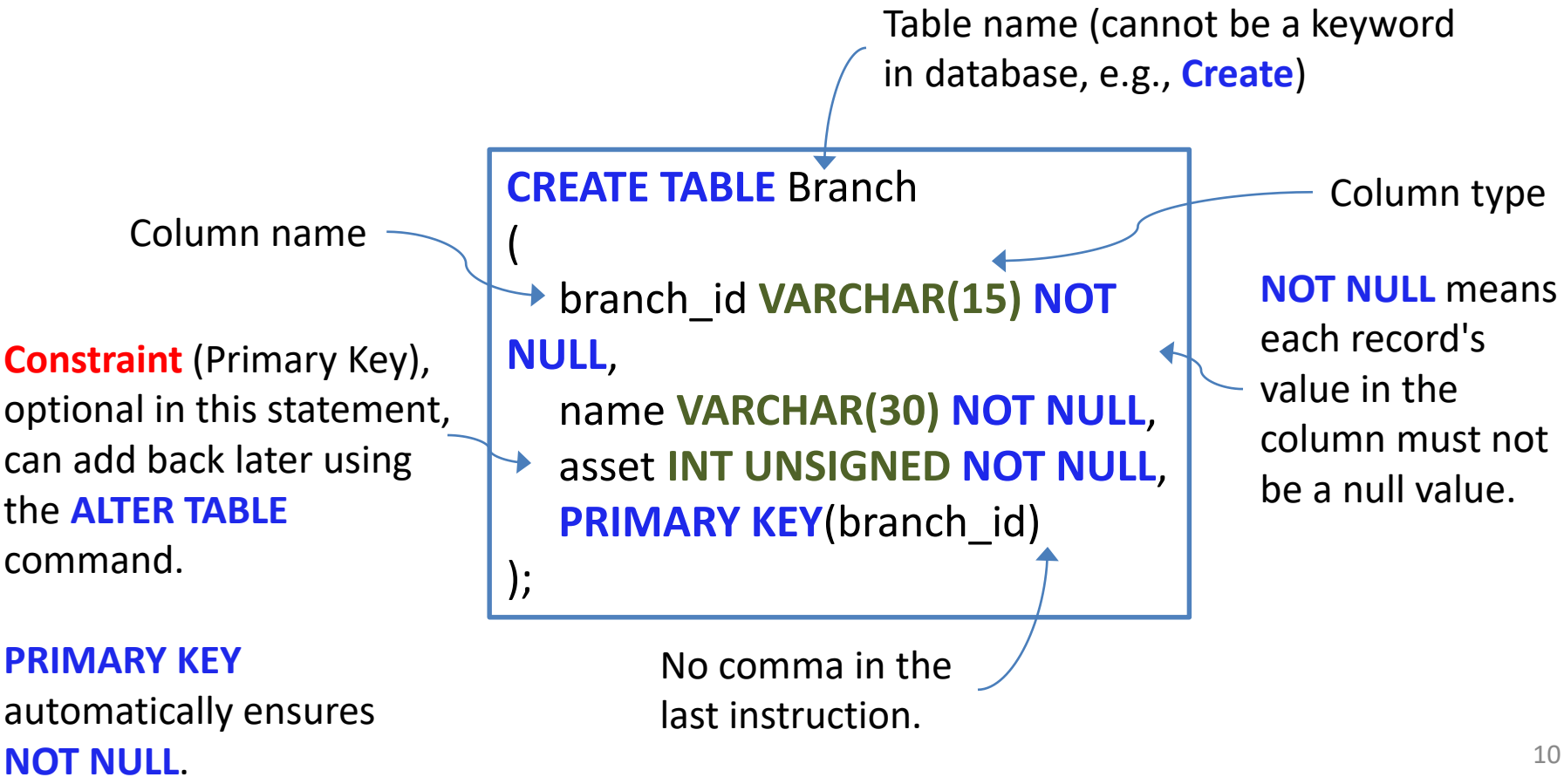
- **Structured Query Language (pronounced as “sequel”)**
- **Language for defining, modifying and querying data in an RDBMS.**
- **SQL is declarative**
  - Concerns about the task we want to accomplish, without specifying how.
- **SQL has many standards and implementations**
  - Read the documentation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

Section 1

Create and Drop Table

Create table

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



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.

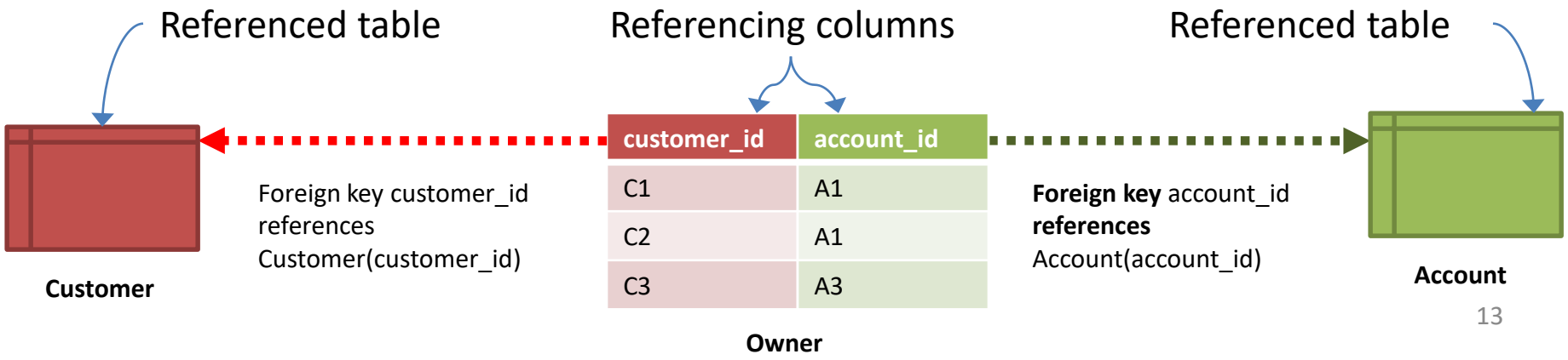
```
ALTER TABLE Branch DROP branch_phone;
```

  - 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.**



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);
```

Foreign key constraints

- In MySQL 5.5, the tables using the InnoDB storage engine (but not MyISAM) supports foreign key constraints.
  - Specify the storage engine while creating the table

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

  - Or we can change the storage engine after creating a table.

```
ALTER TABLE Branch ENGINE = INNODB;
```

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
```

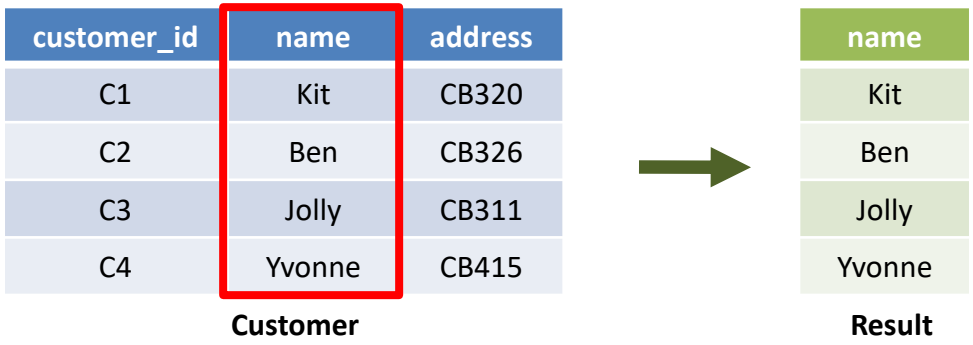
Note: When there are multiple **WHEN … THEN** in the query, only the first true statement (from top to bottom) will be executed.

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	customer_id	loan_id

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

Customer

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

customer_id	loan_id
C1	L3
C4	L2
C2	L1

Borrower

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

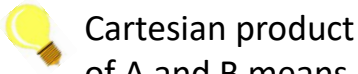
Cartesian product of Customer and Borrower

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

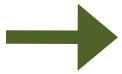
```
SELECT *
FROM Customer, Borrower;
```

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

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;
```

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.

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.
  - **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.

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

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

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

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

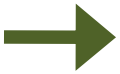
The WHERE clause

- 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.
  - **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

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

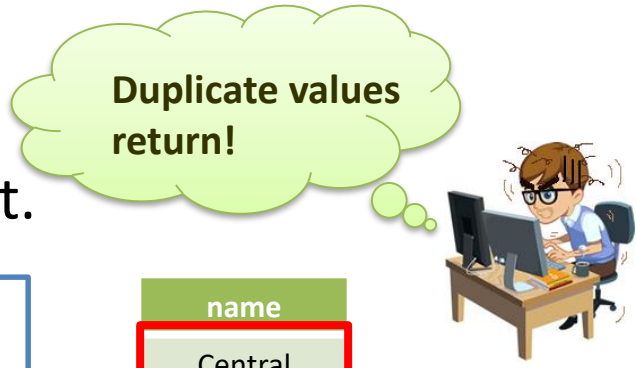
- **Query:** Find the names of all branches that have a loan.
  - **Step 1.** Identify the tables that contain the necessary information to answer the query.

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



  - **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.

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

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

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

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

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

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;
```

Step1. Grouping
GROUP BY branch_id

Step2. Aggregation
AVG(balance)

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

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;
```

Result

branch_id	AVG (balance)
B2	650.0000

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

Lecture 4

END

COMP3278B

Introduction to Database Management Systems

Dr. Ping Luo

Email : pluo@cs.hku.hk

Department of Computer Science, The University of Hong Kong