

# CMPT 354: Database System I

Lecture 4. SQL Advanced

# Announcements!

- A1 is due today
- A2 is released (due in 2 weeks)

# Outline

- **Joins**
  - Inner Join
  - Outer Join
- **Aggregation Queries**
  - Simple Aggregations
  - Group By
  - Having
- **Discussion**

# Joins: Recap

Student

name	gpa
Mary	3.8
Tom	3.6
Jack	3.7

Enroll

stdName	course
Mary	354
Tom	354
Tom	454
Bob	354

```
SELECT name, course  
FROM Student, Enroll  
WHERE name = stdName
```

name	gpa
------	-----

Mary	354
------	-----

Tom	354
-----	-----

Tom	454
-----	-----

# Two equivalent ways to write joins

```
SELECT name, course  
FROM Student, Enroll  
WHERE name = stdName
```



```
SELECT name, course  
FROM Student JOIN Enroll ON  
name = stdName
```

# Join Types

```
SELECT name, course  
FROM Student INNER JOIN Enroll ON  
      name = stdName
```

```
SELECT name, course  
FROM Student FULL OUTER JOIN Enroll ON  
      name = stdName
```

```
SELECT name  
FROM Student LEFT OUTER JOIN Enroll ON  
      name = stdName
```

```
SELECT name  
FROM Student RIGHT OUTER JOIN Enroll ON  
      name = stdName
```

# Join Types

```
SELECT name, course  
FROM Student           JOIN Enroll ON  
      name = stdName
```

```
SELECT name, course  
FROM Student FULL        JOIN Enroll ON  
      name = stdName
```

```
SELECT name  
FROM Student LEFT         JOIN Enroll ON  
      name = stdName
```

```
SELECT name  
FROM Student RIGHT        JOIN Enroll ON  
      name = stdName
```

# Left Join

Student

name	gpa
Mary	3.8
Tom	3.6
Jack	3.7

Enroll

stdName	course
Mary	354
Tom	354
Tom	454
Bob	354

```
SELECT name, course  
FROM Student LEFT JOIN Enroll ON  
name = stdName
```

We want to include all students no matter whether they enroll a course or not. How?

```
SELECT name, course  
FROM Student LEFT JOIN Enroll ON  
name = stdName
```

**Student**

<b>name</b>	<b>gpa</b>
Mary	3.8
Tom	3.6
Jack	3.7

**Enroll**

<b>stdName</b>	<b>course</b>
Mary	354
Tom	354
Tom	454
Bob	354

**Output**

<b>name</b>	<b>course</b>
Mary	354
Tom	354
Tom	454
Jack	NULL

```
SELECT name, course  
FROM Student RIGHT JOIN Enroll ON  
name = stdName
```

**Student**

<b>name</b>	<b>gpa</b>
Mary	3.8
Tom	3.6
Jack	3.7

**Enroll**

<b>stdName</b>	<b>course</b>
Mary	354
Tom	354
Tom	454
Bob	354

**Output**

<b>name</b>	<b>course</b>
Mary	354
Tom	354
Tom	454
NULL	354

```
SELECT name, course  
FROM Enroll FULL JOIN Student ON  
name = stdName
```

Enroll

stdName	course
Mary	354
Tom	354
Tom	454
Bob	354

Student

name	gpa
Mary	3.8
Tom	3.6
Jack	3.7

Output

name	course
Mary	354
Tom	354
Tom	454
Jack	NULL
NULL	354

# Outer Join

TableA (LEFT/RIGHT/FULL) JOIN TableB

- Left outer join:
  - Include tuples from tableA even if no match
- Right outer join:
  - Include tuples from tableA even if no match
- Full outer join:
  - Include tuples from both even if no match

# Exercise - 1

```
SELECT name, course  
FROM Student LEFT JOIN Enroll ON  
name = stdName AND course = 354
```

Student

name	gpa
Mary	3.8
Tom	3.6
Jack	3.7

Enroll

stdName	course
Mary	354
Tom	354
Tom	454
Bob	354

name	course
Mary	354
Tom	354
Jack	NULL

(A)

name	course
Mary	354
Tom	354

(B)

# Exercise - 2

```
SELECT name, course  
FROM Student LEFT JOIN Enroll ON  
name = stdName  
WHERE course = 354
```

Student

name	gpa
Mary	3.8
Tom	3.6
Jack	3.7

Enroll

stdName	course
Mary	354
Tom	354
Tom	454
Bob	354

name course

Mary	354
Tom	354
Jack	NULL

(A)

name course

Mary	354
Tom	354

(B)

# Outline

- **Joins**
  - Inner Join
  - Outer Join
- **Aggregation Queries**
  - Simple Aggregations
  - Group By
  - Having
- **Discussion**

# Simple Aggregation

```
SELECT agg(column)
FROM   <table name>
WHERE  <conditions>
```

**agg** = COUNT, SUM, AVG, MAX, MIN, etc.

Except count, all aggregations apply to a single attribute

# Examples

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	4

```
SELECT COUNT(*) FROM Student
```

5

```
SELECT SUM(gpa) FROM Student
```

17

```
SELECT AVG(gpa) FROM Student
```

3.4

```
SELECT MIN(gpa) FROM Student
```

3

```
SELECT MAX(gpa) FROM Student
```

4

# Examples

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	4

```
SELECT COUNT(DISTINCT gpa) FROM Student
```

2

```
SELECT SUM(DISTINCT gpa) FROM Student
```

7

```
SELECT AVG(gpa) FROM Student  
WHERE gender = 'F'
```

3.5

# The need for Group By

- How to get AVG(gpa) for each gender?

```
SELECT AVG(gpa) FROM Student WHERE gender = 'M'
```

```
SELECT AVG(gpa) FROM Student WHERE gender = 'F'
```

- How to get AVG(gpa) for each age?

```
SELECT AVG(gpa) FROM Student WHERE age = 18
```

```
SELECT AVG(gpa) FROM Student WHERE age = 19
```

```
SELECT AVG(gpa) FROM Student WHERE age = 20
```

•  
•  
•

# Grouping and Aggregation

```
SELECT agg(column)
FROM   <table name>
WHERE  <conditions>
GROUP BY <columns>
```

- How to get AVG(gpa) for each gender?

```
SELECT AVG(gpa) FROM Student GROUP BY gender
```

- How to get AVG(gpa) for each age?

```
SELECT AVG(gpa) FROM Student GROUP BY age
```

# Grouping and Aggregation

- How is the following query processed?

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 2.5
GROUP BY gender
```

- Semantics of the query
  1. Compute the **FROM** and **WHERE** clauses
  2. Group by the attributes in the **GROUP BY**
  3. Compute the **SELECT** clause: grouped attributes and aggregates

# 1. Compute the **FROM** and **WHERE** clauses

```
SELECT gender, AVG(gpa)  
FROM Student  
WHERE gpa > 2.5  
GROUP BY gender
```

name	gender	gpa
Bob	M	2
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	3



name	gender	gpa
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	3

## 2. Group by the attributes in the GROUP BY

```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 2.5
GROUP BY gender
```

name	gender	gpa
Mike	M	3
Alice	F	3
Mary	F	4
Tom	M	3



gender	name	gpa
M	Mike	3
	Tom	3
F	Alice	3
	Mary	4

### 3. Compute the **SELECT** clause: grouped attributes and aggregates

```
SELECT gender, AVG(gpa)
```

```
FROM Student
```

```
WHERE gpa > 2.5
```

```
GROUP BY gender
```

gender	name	gpa
M	Mike	3
	Tom	3
F	Alice	3
	Mary	4



gender	AVG(gpa)
M	3
F	3.5

# Exercise: Empty Group

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa)  
FROM Student  
WHERE gpa > 3.5  
GROUP BY gender
```

gender	AVG(gpa)
F	4

VS

gender	AVG(gpa)
F	4
M	NULL

(A)

(B)

# Exercise: Empty Group

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3



```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4

# Exercise: Empty Group

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3



```
SELECT gender, AVG(gpa)
FROM Student
WHERE gpa > 3.5
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4



gender	name	gpa
F	Alice	4
	Mary	4

# Exercise: Empty Group

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa)  
FROM Student  
WHERE gpa > 3.5  
GROUP BY gender
```



name	gender	gpa
Alice	F	4
Mary	F	4



gender	name	gpa
F	Alice	4
	Mary	4



gender	AVG(gpa)
F	4

# Exercise: Invalid Selection

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa), name  
FROM Student  
WHERE gpa > 3.5  
GROUP BY gender
```

gender	AVG(gpa)	name
F	4	Alice

VS

gender	AVG(gpa)	name
F	4	Mary

(A)

(B)

# Exercise: Invalid Selection

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3



```
SELECT gender, AVG(gpa), name  
FROM Student  
WHERE gpa > 3.5  
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4

# Exercise: Invalid Selection

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3



```
SELECT gender, AVG(gpa), name  
FROM Student  
WHERE gpa > 3.5  
GROUP BY gender
```

name	gender	gpa
Alice	F	4
Mary	F	4



gender	name	gpa
F	Alice	4
	Mary	4

# Exercise: Invalid Selection

Everything in SELECT must be either a GROUP-BY attribute, or an aggregate

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa), name  
FROM Student  
WHERE gpa > 3.5  
GROUP BY gender
```



name	gender	gpa
Alice	F	4
Mary	F	4



gender	name	gpa
F	Alice	4
	Mary	4



gender	AVG(gpa)	name
F	4	???

# HAVING Clause

- Specify which groups you are interested in

```
SELECT agg(column)
FROM   <table name>
WHERE  <conditions>
GROUP BY <columns>
HAVING <columns>
```

# HAVING Clause

- Same query as before, except that we require each group has more than 10 students

```
SELECT AVG(gpa), gender  
FROM Student  
WHERE gpa > 2.5  
GROUP BY gender  
HAVING COUNT(*) > 10
```

HAVING clause contains conditions on aggregates.

# Order of Evaluation

SELECT	S
FROM	$R_1, \dots, R_n$
WHERE	$C_1$
GROUP BY	$a_1, \dots, a_k$
HAVING	$C_2$

- Create the cross product of the tables in the **FROM** clause
- Remove rows not meeting the **WHERE** condition
- Divide records into groups by the **GROUP BY** clause
- Remove groups not meeting the **HAVING** clause
- Create one row for each group and remove columns not in the **SELECT** clause

# Exercise

StudentInfo

name	gender	gpa
Bob	M	3
Mike	M	3
Alice	F	4
Mary	F	4
Tom	M	3

```
SELECT gender, AVG(gpa)
FROM StudentInfo
WHERE gpa > 2.5
GROUP BY gender
HAVING COUNT(*) > 2
```

```
SELECT gender, AVG(gpa)
FROM StudentInfo
WHERE gpa > 2.5
GROUP BY gender
HAVING SUM(gpa) < 9
```

gender	AVG(gpa)
M	3

(A)

gender	AVG(gpa)
F	4

(B)

gender	AVG(gpa)
M	3
F	4

(C)

Imagine you are a **data scientist**  
at a Bank

# Computer Science vs. Data Science

What	When	Who	Goal
Computer Science	1950-	Software Engineer	Write software to make computers work

Plan → Design → Develop → Test → Deploy → Maintain

What	When	Who	Goal
Data Science	2010-	Data Scientist	Extract insights from data to answer questions

Collect → Clean → Integrate → Analyze → Visualize → Communicate

# Discussion

Customer = {customerID, firstName, lastName, income, birthDate}

Account = {accNumber, type, balance, branchNumber<sup>FK-Branch</sup>}

Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

Q1. Who is the richest customer?

Q2. Which customers have ONLY one account?

# Discussion

Employee = {sin, firstName, lastName, salary, branchNumber<sup>FK-Branch</sup>}

Branch = {branchNumber, branchName, managerSIN<sup>FK-Employee</sup>, budget}

- Q3. How many customers does each branch have?
- Q4. Which branch have a higher pay?

# Outline

- **Joins**
  - Inner Join
  - Outer Join
  - Self Join
- **Aggregation Queries**
  - Simple Aggregations
  - Group By
  - Having
- **Subqueries**
  - In the FROM clause
  - In the WHERE clause

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Subqueries

- A subquery is a SQL query nested inside a larger query
- Such inner-outer queries are called nested queries

```
SELECT C.customerID, C.birthDate, C.income
```

```
FROM Customer C
```

```
WHERE C.customerID IN
```

```
(
```

```
    SELECT O.customerID
```

```
    FROM Account A, Owns A
```

```
    WHERE A.accNumber = O.accNumber
```

```
        AND A.branchName = 'Lonsdale'
```

```
)
```

Outer Query

Inner Query

# Subqueries

- Subqueries may appear in
  - A **FROM** clause,
  - A **WHERE** clause, and
  - A **HAVING** clause

```
SELECT <columns>
FROM   <table name>
WHERE  <conditions>
GROUP BY <columns>
HAVING <columns>
```

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Subqueries in FROM

- Sometimes we need to compute an intermediate table only to use it later in a SELECT-FROM-WHERE
- Who is the richest customer?

```
SELECT firstName, lastName, MAX(sumBalance)
FROM (SELECT firstName, lastName, sum(balance) AS sumBalance
      FROM Customer C, Account A, Owns O
      WHERE C.customerID = O.customerID
            AND O.accNumber = A.accNumber
      GROUP BY C.customerID )
```

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Subqueries in FROM

- Sometimes we need to compute an intermediate table only to use it later in a SELECT-FROM-WHERE
- Which customers have a total balance equal to 0?

```
SELECT firstName, lastName, sumBalance  
FROM (SELECT firstName, lastName, sum(balance) AS sumBalance  
      FROM Customer C, Account A, Owns O  
      WHERE C.customerID = O.customerID  
            AND O.accNumber = A.accNumber  
      GROUP BY C.customerID) AS T  
WHERE T.sumBalance = 0
```

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Subqueries in FROM

- Sometimes we need to compute an intermediate table only to use it later in a SELECT-FROM-WHERE
- Which customers have a total balance equal to 0?

```
SELECT firstName, lastName, sum(balance) AS sumBalance  
FROM Customer C, Account A, Owns O  
WHERE C.customerID = O.customerID AND O.accNumber = A.accNumber  
GROUP BY C.customerID  
HAVING sumBalance = 0
```

Rule of thumb: avoid nested queries when possible

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Subqueries in WHERE

- Subqueries return a single constant
  - >, <, =, <>, >=, <=
- Find the customerIDs of customers whose income is larger than avg(income)

```
SELECT C.customerID  
FROM Customer C1  
WHERE C1.income > (SELECT avg(C2.income)  
                      FROM Customer C2)
```

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Subqueries in WHERE

- Subqueries return a relation
  - IN
  - NOT IN
  - EXISTS
  - NOT EXISTS
  - ANY
  - ALL

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Accounts IN Burnaby

- Find the customerIDs of customers with an account at the Burnaby branch

```
SELECT C.customerID  
FROM Customer C  
WHERE C.customerID IN (SELECT O.customerID  
                        FROM Account A, Owns O  
                        WHERE A.accNumber = O.accNumber  
                          AND A.branchName = 'Burnaby')
```

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Accounts NOT IN Burnaby

- Find the customerIDs of customers who do *not* have an account at the Burnaby branch

```
SELECT C.customerID  
FROM Customer C  
WHERE C.customerID NOT IN(  
    SELECT O.customerID  
    FROM Account A, Owns O  
    WHERE A.accNumber = O.accNumber AND  
        A.branchName = 'Burnaby')
```

# Uncorrelated Queries

- The query shown previously contains an *uncorrelated*, or *independent*, sub-query
  - The sub-query does not contain references to attributes of the outer query
- An independent sub-query can be evaluated before evaluation of the outer query
  - And needs to be evaluated only once
    - The sub-query result can be checked for each row of the outer query
  - The cost is the cost for performing the sub-query (once) and the cost of scanning the outer relation

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# EXISTing BurnabyAccounts

- Find the customerIDs of customers with an account at the Burnaby branch

```
SELECT C.customerID  
FROM Customer C  
WHERE EXISTS ( SELECT *  
    FROM Account A, Owns O  
    WHERE C.customerID = O.customerID  
        AND A.accNumber = O.accNumber  
        AND A.branchName = 'Burnaby')
```

EXISTS and NOT EXISTS test whether the associated sub-query is non-empty or empty

# Correlated Queries

- The previous query contained a *correlated* sub-query
  - With references to attributes of the outer query
    - ... **WHERE** C.customerID = O.customerID ...
  - It is evaluated once *for each row* in the outer query
    - i.e. for each row in the Customer table
- Correlated queries are often inefficient

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# EXISTing BurnabyAccounts

- Find the customerIDs of customers with an account at the Burnaby branch

```
SELECT DISTINCT C.customerID  
FROM Customer C, Account A, Owns O  
WHERE C.customerID = A.accNumber  
    AND A.accNumber = O.customerID  
    AND A.branchName = 'Burnaby'
```

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}  
Branch = {branchNumber, branchName, managerSIN<sup>FK-Employee</sup>, budget}

# Have an account in all branches

- Find the customerIDs of customers who have an account in all branches

SQ1 – A list of all branch names

SQ2 – A list of branch names that a customer has an account at

EXCEPT



If the customer has an account at every branch then this result is empty

# Have an account in all branches

- Putting it all together we have

```
SELECT C.customerID  
FROM Customer C  
WHERE NOT EXISTS ( (SELECT B.branchName  
                    FROM Branch B)  
                  EXCEPT  
                  (SELECT A.branchName  
                   FROM Account A, Owns O  
                   WHERE O.customerID = C.customerID  
                         AND O.accNumber = A.accNumber))
```

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# ANYone Richer Than Bruce

- Find the customerIDs of customers who earn more than *some* customer called Bruce

```
SELECT C.customerID  
FROM Customer C  
WHERE C.income > ANY  
(SELECT Bruce.income  
FROM Customer Bruce  
WHERE Bruce.firstName = 'Bruce')
```

Customers in the result table must have incomes greater than at least one of the rows in the sub-query result

Customer = {customerID, firstName, lastName, income, birthDate}  
Account = {accNumber, type, balance, branchName}  
Owns = {customerID<sup>FK-Customer</sup>, accNumber<sup>FK-Account</sup>}

# Richer Than ALL the Bruces

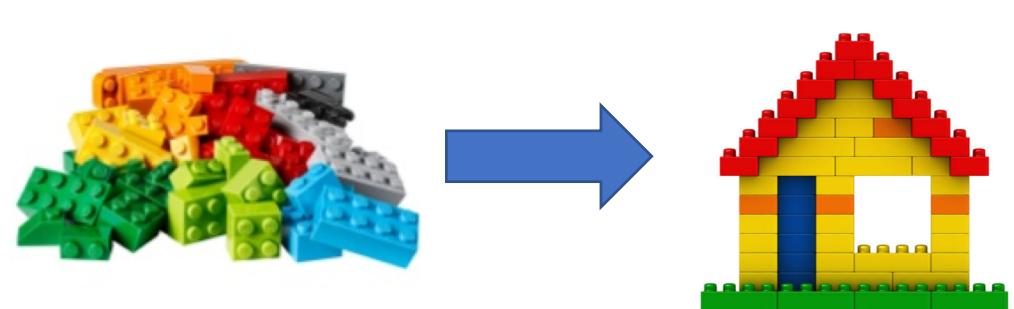
- Find the customerIDs of customers who earn more than *all* customer called Bruce

```
SELECT C.customerID  
FROM Customer C  
WHERE C.income > ALL  
    (SELECT Bruce.income  
     FROM Customer Bruce  
     WHERE Bruce.firstName = 'Bruce')
```

If there were no customers called Bruce this query would return all customers

# Summary

- Selection
- Projection
- Set Operators (UNION, INTERSECT, EXCEPT)
- Joins (INNER, OUTER)
- Aggregation
- Group By
- Having
- Order By
- Distinct
- Subqueries



SQL operators can be composed just like building LEGO buildings

# Acknowledge

- Some lecture slides were copied from or inspired by the following course materials
  - “W4111: Introduction to databases” by Eugene Wu at Columbia University
  - “CSE344: Introduction to Data Management” by Dan Suciu at University of Washington
  - “CMPT354: Database System I” by John Edgar at Simon Fraser University
  - “CS186: Introduction to Database Systems” by Joe Hellerstein at UC Berkeley
  - “CS145: Introduction to Databases” by Peter Bailis at Stanford
  - “CS 348: Introduction to Database Management” by Grant Weddell at University of Waterloo