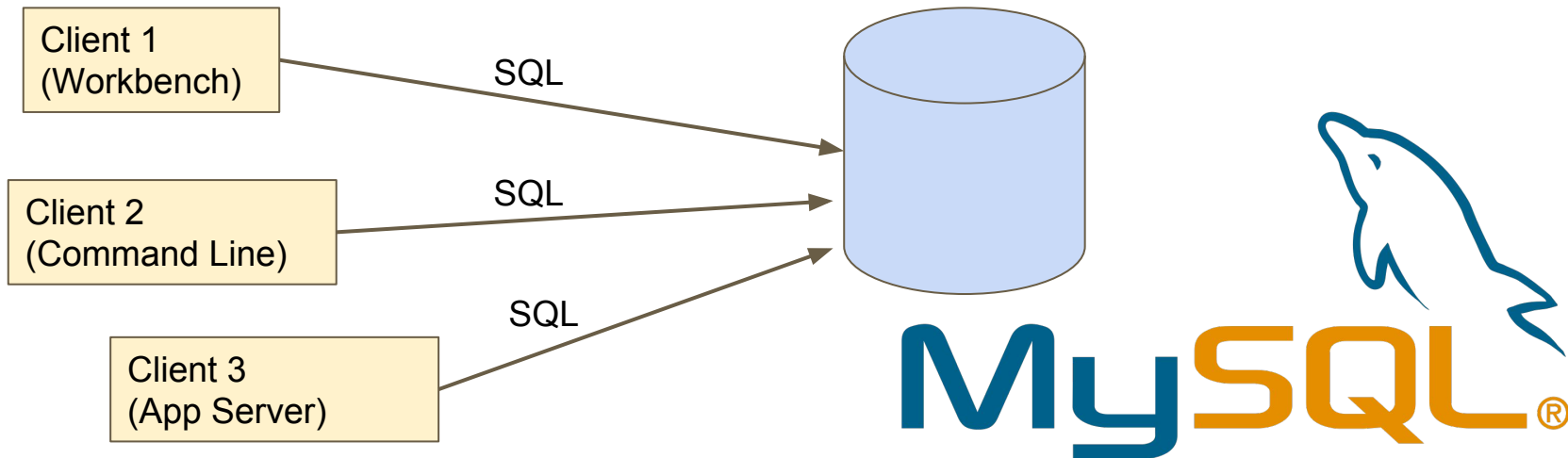

SQL

Database Systems

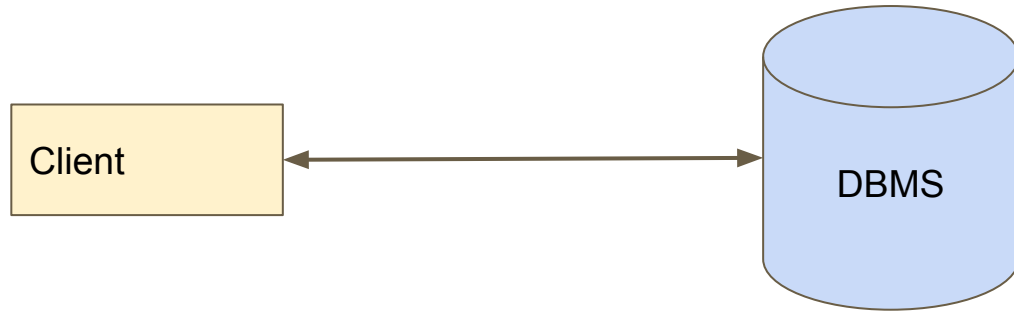
RDBMS: Relational DBMS

- Popular free relational DBMS
- Community Server <https://dev.mysql.com/downloads/mysql/>
- MySQL Workbench <https://dev.mysql.com/downloads/workbench/>



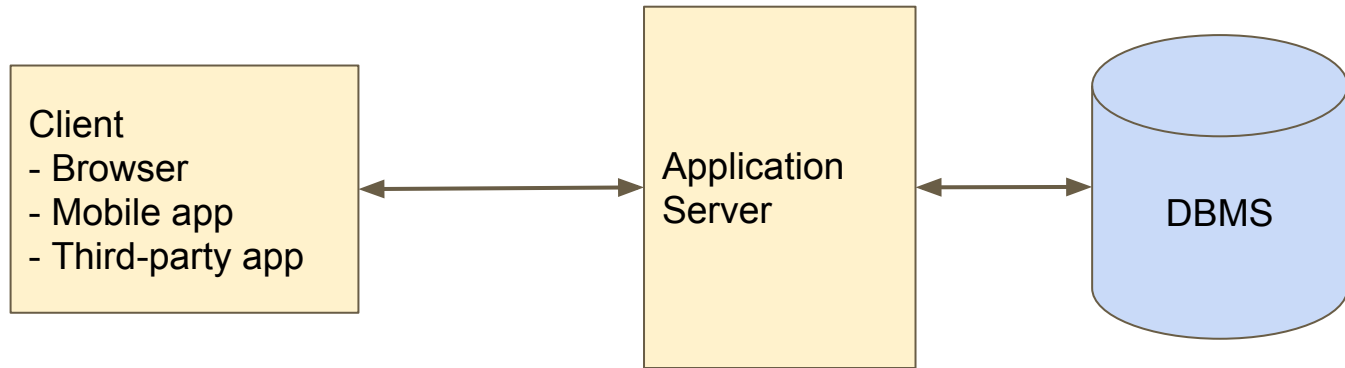
Two-tier architecture

- DBMS
- Client



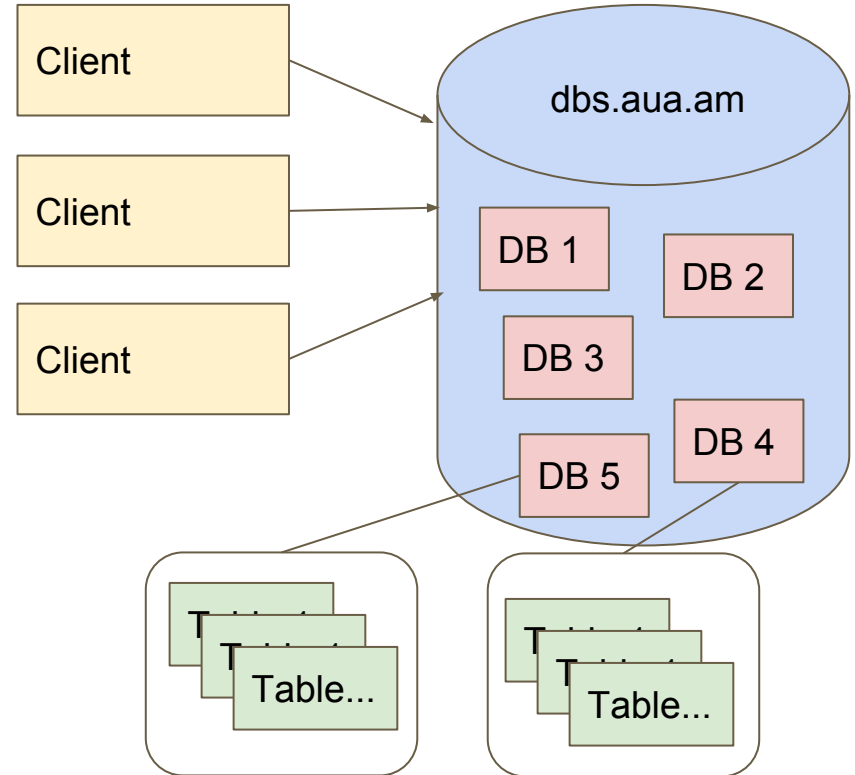
Three-tier architecture

- DBMS
- Application Server
- Client



AUA DBMS

- Host
 - Name: **db.s.aua.am**
 - IP: **95.140.195.69**
 - Default Port: **3306**
- Authentication
 - Username: **general**
 - Password:
- Each student will get a personal account
- *You don't have to install the server*



Connecting to AUA DBMS

- Install **MySQL Workbench** and **MySQL Utilities**
- Command Line

```
mysql -h dbs.aua.am -u general -p
```

```
mysql -h 95.140.195.69 -u general -p
```

- -h ⇒ Hostname or IP address
- -u ⇒ Provide the username
- -p ⇒ The program will ask for password
- Using the Workbench GUI
 - Create a connection with specified Host, Username and Password
 - Click & connect !!!

First commands...

- Viewing available schemas (databases)
- Navigating into schemas and tables
- Describing tables
- Viewing contents of tables

Creating Databases...

- Create Schema, or, Use Schema
- Create tables...
- SQL Data types
- SQL Constraints
- Adding data into tables

```
CREATE TABLE TableName (  
    field1          datatype,  
    field2          datatype,  
    PRIMARY KEY (field1),  
    FOREIGN KEY field2 REFERENCES AnotherTable (field)  
);
```


MySQL Data Types

- These are just the commonly used data types
- For full reference, visit ⇒ <https://dev.mysql.com/doc/refman/5.7/en/data-types.html>
- Data Types
 - Numeric types
 - Integer
 - Exact fractional numbers
 - Floating point fractional numbers
 - String types
 - Date and time types

MySQL Data Types: Numeric values

TINYINT SMALLINT MEDIUMINT INT BIGINT	1 byte 2 bytes 3 bytes 4 bytes 8 bytes	All types store negative & positive values. INT(1), INT(5), INT(n) UNSIGNED: Only positive numbers ZEROFILL AUTO_INCREMENT $-2^{32} \leq \text{INT} \leq 2^{32} - 1$ $0 \leq \text{INT UNSIGNED} \leq 2^{33} - 1$
DECIMAL(N,M)	Max 64 digits	$-999.99 \leq \text{DECIMAL}(5, 2) \leq 999.99$
FLOAT DOUBLE	4 bytes precision 8 bytes precision	Values outside the range will be rounded.
BIT(N)	$N = [1, 64]$	$b'111' = 7$, $b'011' = 3$

MySQL Data Types: String values

CHAR(n)	n = Exact length	$0 \leq n \leq 255$
VARCHAR(n)	n = Maximum length	$0 \leq n \leq 65535$
TINYTEXT TEXT MEDIUMTEXT LONGTEXT	256 bytes ~ 64 KB ~ 16 MB ~ 4 GB	Contain characters
TINYBLOB BLOB MEDIUMBLOB LOB	256 bytes ~ 64 KB ~ 16 MB ~ 4 GB	Contain binary information (Such as contents of a non-text file)

CHAR vs. VARCHAR

CHAR(30)
apple
orange
watermelon

$3 \times 30 = 90$ bytes

VARCHAR(30)	
5	apple
6	orange
10	watermelon

$3 + (5 + 6 + 10) = 24$ bytes

VARCHAR (255)	
1 byte	value

VARCHAR (1000)	
2 bytes	value

CHAR (100)	
value	

MySQL Data Types: More types

DATE DATETIME TIMESTAMP TIME YEAR	Date but no time Date and also time Date and time, in seconds from Jan 1, 1970, timezone-free Only time Only year
ENUM	ENUM('M', 'F') ENUM('Spring', 'Summer', 'Autumn', 'Winter') Only one value is allowed. Values are physically stored as INT.
SET	SET('a', 'b', 'c') All <i>mathematically valid</i> subsets are allowed { 'a', 'b' }, { 'a' }, { 'a', 'b', 'c' }, { 'a', 'c' }
More data types	https://dev.mysql.com/doc/refman/5.7/en/data-types.html

Choosing proper data types

- NEVER CHOOSE MORE THAN REQUIRED
- Analyze the requirement
- Keep it simple
 - `ENUM ('TRUE', 'FALSE')` vs. `BIT(1)`
 - Storing image URL as `VARCHAR` vs. storing image content as `BLOB`
- Keep it simple but wise

Choosing proper data types

- Mind the performance
 - Updates ⇒ When updating results in larger data values
 - Searching ⇒ Some data types are not efficiently indexed (TEXT, BLOB)
- Mind the precision (esp. for financial data)
- **Question:** When to use CHAR(n) and when to use VARCHAR(n)

ALTER TABLE

- Add, Delete, Modify columns after table is created
- Change data types after table is created and populated with data
- Add or drop constraints
 - `ADD CONSTRAINT PK_id PRIMARY KEY (id)`
 - `ADD CONSTRAINT PK_name PRIMARY KEY (firstName, lastName)`
 - `ADD CONSTRAINT U_email UNIQUE (email)`
 - `ADD CONSTRAINT FK_dept FOREIGN KEY dept REFERENCES Department.id`
 - **Constraint Name**
 - **Constraint Type**
 - **Constraint Properties**

Constraints consume storage and time



MySQL engine also uses special tables in **INFORMATION_SCHEMA** to track the structure of your tables and their respective constraints and index data.

- Storing them takes storage
- Processing them takes time

Modifying data: Insert, Delete, Update

- Insert: Adding rows to table

```
INSERT INTO Book(bookId, title, numAvailable, categoryId)
VALUES
(1, 'Java How to Program', 10, 1),
(2, 'C++ How to Program', 0, 2);
```

- Skipping the column names

```
INSERT INTO Book
VALUES
(1, 'Java How to Program', 10, 1),
(2, 'C++ How to Program', 0, 2);
```

Insert

- When using auto-increment primary key

```
INSERT INTO Book(title, numAvailable, categoryId)
VALUES
('Java How to Program', 10, 1),
('C++ How to Program', 0, 2);
```

- Changing the order of columns

```
INSERT INTO Book(categoryId, numAvailable, title)
VALUES
(1, 10, 'Java How to Program'),
(2, 0, 'C++ How to Program');
```

Insert

- Using default values

```
INSERT Book(title) VALUES ('Java Performance');
```

- INSERT → INSERT INTO
- What will be the values of **bookId**, **categoryId** and **numAvailable**?

Delete

Removing rows from table

- **DELETE FROM *tableName* [WHERE *condition*]**
- WHERE is optional: DELETE FROM *tableName* → Deletes all rows

```
DELETE FROM Book WHERE bookId = 1;
```

```
DELETE FROM Book WHERE bookId > 1;
```

```
DELETE FROM Category WHERE categoryName = 'Java'
```

Update

- Modify an existing row
- **UPDATE *tableName***
SET *column1* = *value1*, *column2* = *value2*
WHERE *condition*
- Change category of a book

```
UPDATE Book SET categoryId = 2 WHERE bookId = 1;
```

```
UPDATE Book SET categoryId = NULL WHERE categoryId = 1;
```

Update

- Change multiple rows

```
UPDATE Book SET categoryId = NULL WHERE categoryId = 1;
```

- Rename a book title

```
UPDATE Book  
SET title = 'Java how to program'  
WHERE title = 'Java How to Program' ;
```

Query types

- CRUD Operations: Create, Read, Update, Delete
 - Also an OOP term
 - Used also out of database scope
- INSERT – Create
- SELECT – Read
- UPDATE – Update
- DELETE – Delete

Select: Get rows from one or more tables

- `SELECT columns FROM tableNames [WHERE condition]`
- Select all columns from all rows

```
SELECT * FROM Book;
```

- Select some columns from some rows

```
SELECT categoryId, title FROM Book WHERE bookId > 0 AND bookId < 10
SELECT bookId, title FROM Book WHERE categoryId = 1
SELECT bookId, title FROM Book WHERE categoryId != 1
SELECT bookId, title FROM Book WHERE categoryId <> 1
SELECT DISTINCT categoryId FROM Book WHERE bookId > 2 AND bookId < 10
```

Select

- Books that belong to some categories (categoryId = { 1, 2, 3 })

```
SELECT title FROM Book  
WHERE categoryId = 1 OR categoryId = 2 OR categoryId = 3;
```

```
SELECT title FROM Book WHERE categoryId >= 1 AND categoryId <= 3;  
SELECT title FROM Book WHERE categoryId BETWEEN 1 AND 3;
```

- IN Statement

```
SELECT title FROM Book WHERE categoryId IN (1, 2, 3);  
SELECT title FROM Book WHERE categoryId IN (1, 3, 6);
```

```
SELECT bookId, title FROM Book WHERE title IN  
('Java How to Program', 'C++ How to Program,', 'Database Systems')
```

Select

- Tuple comparison

```
SELECT * FROM Book WHERE (numAvailable, categoryId) = (0, 2);
```

```
SELECT * FROM Book WHERE numAvailable = 0 AND categoryId = 2;
```

- Applying select and tuple comparison on two tables

```
SELECT * FROM Book, Category  
WHERE (Book.categoryId, categoryName) = (Category.categoryId, 'Java');
```

Select: Tuple Comparison in MySQL

- In Book: $(a, b) \leq (x, y) : a \leq x \text{ AND } b \leq y$
- In MySQL
 - $(a, b) = (x, y) \Rightarrow a = x \text{ AND } b = y$
 - $(a, b) < (x, y) \Rightarrow (a < x) \text{ OR } ((a = x) \text{ AND } (b < y))$
 - $(a, b) \leq (x, y) \Rightarrow (a < x) \text{ OR } ((a = x) \text{ AND } (b \leq y))$
- See: <http://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html>

Select and sort

- ORDER BY attribute [ASC, DESC]

```
SELECT * FROM Book ORDER BY numAvailable;
```

```
SELECT * FROM Book ORDER BY numAvailable DESC;
```

Sort using two columns:

```
SELECT * FROM Book ORDER BY numAvailable DESC, categoryId;
```

```
SELECT * FROM Book ORDER BY numAvailable DESC, categoryId DESC;
```

Handling NULLs

- NULL = We don't know or don't have the value
- $10 > \text{NULL}$? $10 = \text{NULL}$? 'Hello' = NULL ?
- Try : `SELECT * From Book WHERE categoryId = NULL`
- Try : `SELECT * From Book WHERE categoryId IS NULL`
- How to compare NULLs?
- Three-valued logic
 - True, False, Unknown
 - **NULL vs. Value = Unknown** such as, $5 = \text{NULL}$, $5 < \text{NULL}$, $\text{NULL} > \text{NULL}$

Three valued logic

- OR
 - Unknown OR True = True
 - Unknown OR False = Unknown
 - Unknown OR Unknown = Unknown
- AND
 - Unknown AND True = Unknown
 - Unknown AND False = False
 - Unknown AND Unknown = Unknown
- NOT
 - NOT Unknown = Unknown

Three valued logic

- **IS** and **IS NOT** keywords are used for comparing NULL and UNKNOWN.

Try: `SELECT * From Book WHERE (categoryId = NULL) IS UNKNOWN;`

Try: `SELECT * From Book WHERE (categoryId = 1) IS UNKNOWN;`

- Predicate evaluation
 - WHERE (predicate)
 - Rows are added to result if predicate is evaluated as TRUE
 - Rows where predicate is FALSE or UNKNOWN are excluded from result
 - categoryId = NULL
 - **categoryId IS NULL**
 - **categoryId IS NOT NULL**

Joins in MySQL (also YourSQL)

A query walks into a bar and asks two
tables: Can I join you?

Joins

- **Join types**

- INNER JOIN
- OUTER
 - Left
 - Right
 - Outer
- How to behave when rows from left and right tables do not have a match

- **Join conditions**

- NATURAL
- USING (...)
- ON < predicate >
- Defines the condition of matching left and right rows


General algorithm

```
select  $A_1, A_2, \dots, A_n$   
from  $r_1, r_2, \dots, r_m$   
where  $P$ ;
```

Types of join

- 1) How to construct t from t_1, t_2, \dots, t_m
- 2) P (Predicate): When to include t in result

```
for each tuple  $t_1$  in relation  $r_1$   
  for each tuple  $t_2$  in relation  $r_2$   
    ...  
    for each tuple  $t_m$  in relation  $r_m$ 
```



Concatenate t_1, t_2, \dots, t_m into a single tuple t
Add t into the result relation if P is true

Natural Join (\bowtie)

```
SELECT * FROM Book NATURAL JOIN Category;
```

```
Result = Book  $\bowtie$  Category
```

- Predicate is true if **all** common columns have same value
- Result row contains **union** of all columns. No duplicate columns

```
Result = ((Book  $\bowtie$  Borrow)  $\bowtie$  Student)
```

```
SELECT * FROM Book NATURAL JOIN Borrow NATURAL JOIN Student;
```

Natural Join

- (1) Which books belong to 'Database Systems' category? We don't know the ID of the category
- How to:
 - Get ID of 'Database Systems' category
 - Use the ID to find books
 - **In raw SQL there cannot be steps. Everything is executed as one statement**
- (2) Given a book, find its category name

Does it matter if you are searching from category to book, or from book to category?

Join: ON

- ON is used to explicitly impose the predicate

```
SELECT * FROM Book JOIN Category ON  
Book.categoryId = Category.categoryId
```
- Most useful when columns have different names

Department (id, deptName) Course(id, title, departmentId)

- List name of courses with the department name that offers them

Try:

```
SELECT * FROM Department NATURAL JOIN Course;
```

Join: ON

- Why? Common column names were irrelevant to the existing FK
 - Conclusion 1: DBMS considers column name not the FK constraint
 - Conclusion 2: You can have foreign keys without explicit FK constraints
- Provide the predicate: **Course.departmentId = Department.Id**

```
SELECT *  
FROM Department JOIN Course ON  
Department.id = Course.departmentId;
```

- Compare result columns of “ON” and “NATURAL JOIN”
- Get (Course ID, Course Title, Department Name)

Join: ON

- Try (1) `SELECT * FROM Department JOIN Course ON TRUE;`
- Try (2) `SELECT * FROM Department JOIN Course ON FALSE;`
- Try (3) `SELECT * FROM Department JOIN Course;`
- Try (4)

```
SELECT *  
FROM Student JOIN Department ON  
Student.fullname = 'Sheldon Cooper';
```

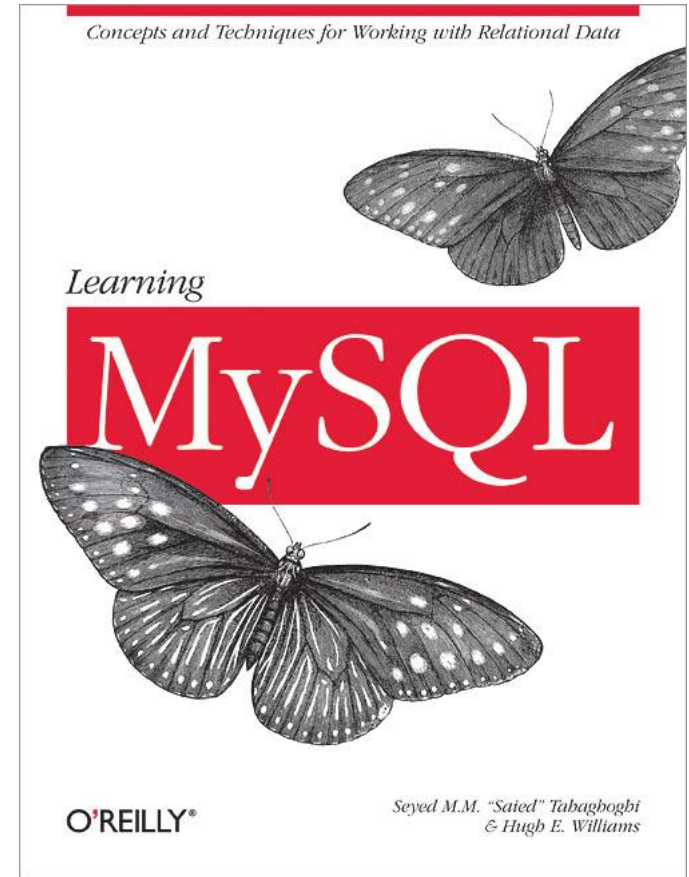
- Try: What is the department name of Sheldon Cooper?

Session 11

- Assignment 1 – Deadline: October 9
- Team for projects – team size = 4
- MySQL Workbench
- Foreign key constraints
 - On update, On delete...
 - Immediate, deferred
- IN (...)
- More Joins
 - Example: Numbers...
 - USING keyword
 - OUTER joins

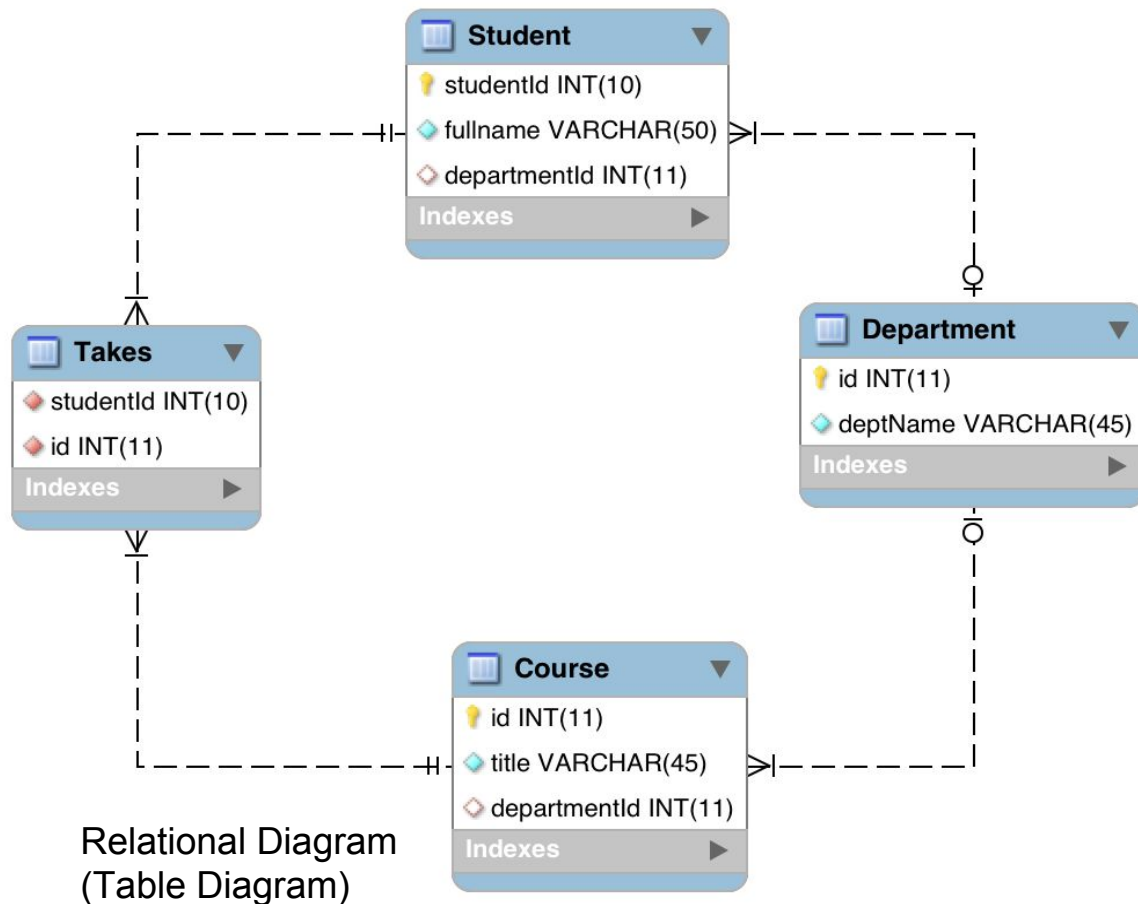
Book: Learning MySQL

- “New” book! Assisting book for MySQL
- **Learning MySQL – O’Reilly (2006)**
- Available on Moodle ([Right Here](#))
- SQL: Chapters 4 to 7



Join: USING

- Natural join compares all common columns, how to manage them?
- **Get List of students and their courses**



Join: USING

Student (**studentId**, fullname, **departmentId**)

Course (**id**, title, **departmentId**)

Takes (**studentId**, **id**)

```
SELECT * FROM Takes;      -- How many rows?
```

```
SELECT * FROM Student NATURAL JOIN Takes NATURAL JOIN Course;
```



Step 1




Step 2 = Final Result

Join: USING

- How to fix this “problem” ?

```
SELECT *  
FROM (Student NATURAL JOIN Takes) JOIN Course USING (id);
```

```
SELECT *  
FROM R JOIN S USING ( list of columns we are interested in )
```

```
SELECT *  
FROM (Student NATURAL JOIN Takes)  JOIN Course USING (id, departmentId);
```

Same as two Natural Joins

Inner Joins summary

NATURAL

Condition:

Equality of all common columns

Result row:

Union of all columns

```
SELECT *  
FROM R NATURAL JOIN S
```

USING (...)

Condition:

Equality of specified common columns

Result row:

Union of all columns

```
SELECT *  
FROM R JOIN S USING (a)
```

ON ...

Condition:

The provided predicate

Result row:

All columns

```
SELECT *  
FROM R JOIN S ON (condition)
```

➤ As an exercise, express all three inner joins with cartesian product

With SQL and relational algebra
SELECT * FROM R, S WHERE ...

Question

Try (1): `SELECT * FROM Book, Category`

Try (2): `SELECT * FROM BOOK JOIN Category`

Does it mean JOIN is same as Cartesian Product?

Why are the results of (1) and (2) the same?

Outer Joins

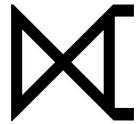
Works differently from INNER JOIN when the **predicate** is not true.

- Left Outer Join

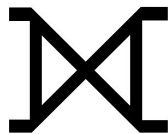


- Perform inner join
- Include rows from left table not present in the result and pair with NULLs
- Makes sure all rows from left table exist in the result

- Right Outer Join



- Perform inner join
- Include rows from right table not present in the result and pair with NULLs
- Makes sure all rows from right table exist in the result



Full Outer Join ⇒ Do left outer join , also do right outer join

Outer Joins

- NATURAL, ON and USING keywords can be applied
- In MySQL
 - JOIN = **INNER** JOIN (INNER keyword is optional)
 - LEFT JOIN = LEFT **OUTER** JOIN
 - RIGHT JOIN = RIGHT **OUTER** JOIN
 - FULL OUTER JOIN ⇒ Doesn't exist!
- Get students and their departments
- Are all departments present in the result?

Outer Joins

- Outer join with NATURAL condition
 - Get courses and registered student IDs
 - Which classes are empty?
 - Which students have no classes?
 - Which students have borrowed any book?
- Outer join with USING
 - Get all courses and name of participants
- Outer join with ON
 - Get all departments and their students

Column naming convention

Book (id, title, numAvailable, price, **categoryId**)

Category(id, title)

Student (id, fullName, **departmentId**)

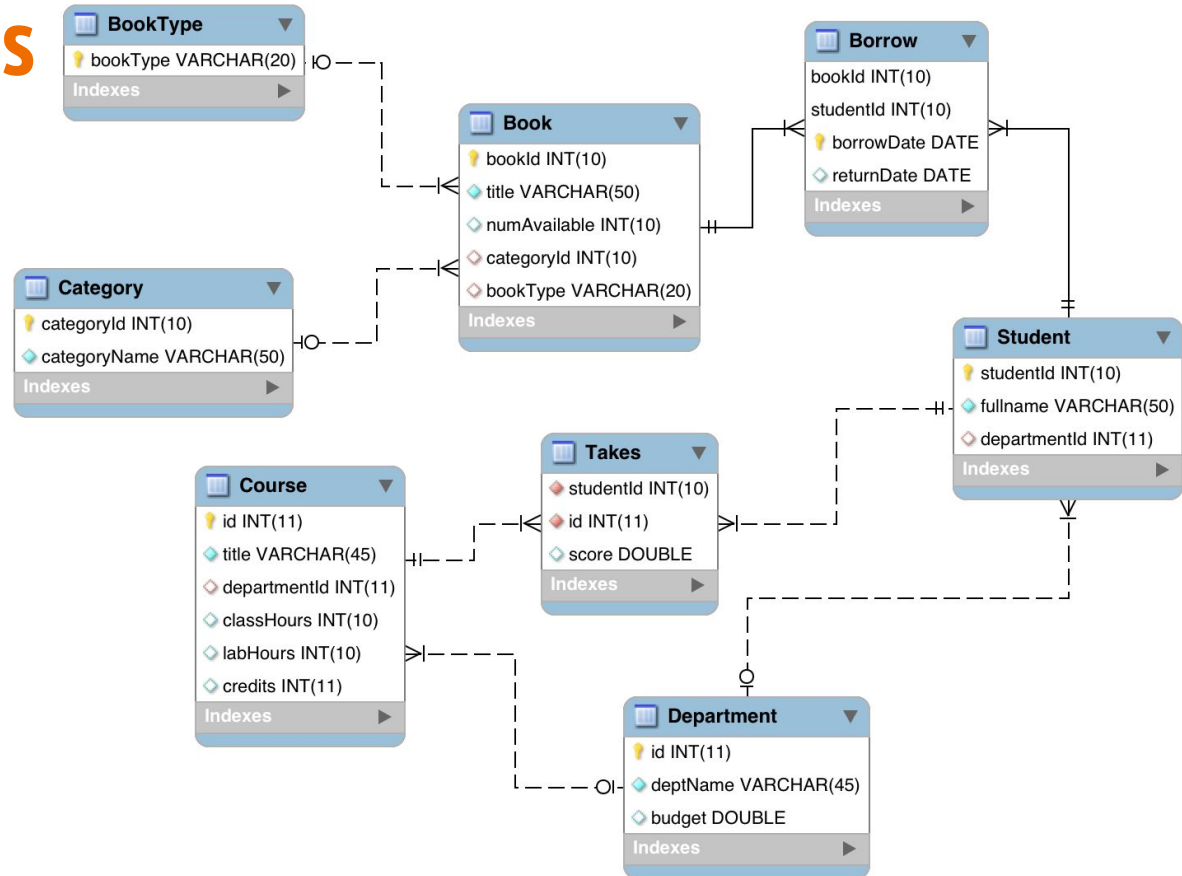
Borrow (bookId, studentId)

- Makes the column names easy to read and recognize
- categoryId suggests that it refers to category table's id column

SQL Syntax convention

- *Conventions are not rules, but improve teamwork. Kind of rules.*
- SQL is case insensitive, but it is preferred to write keywords UPPERCASE
- IF YOU OPEN A FILE AND FIND ALL LETTERS IN CAPITAL WILL IT BE EASY TO READ FOR YOU?
- **SELECT id, title FROM Book WHERE numAvailable > 0;**
- Tables: Start with Capital Letter
- Columns: Lowercase, composite names either
 - camelCase, such as numAvailable
 - or, underscored_names, such as num_available

Practice with JOINS



JOINS

- Find all courses that have at least one registered student

```
SELECT DISTINCT title  
FROM Course NATURAL JOIN Takes;
```

- Find all courses that have at least one “guest” student

```
SELECT DISTINCT title  
FROM Course NATURAL JOIN Takes  
JOIN Student USING (studentId);
```

JOINS

- Find “unpopular” courses

```
SELECT DISTINCT title
FROM Course NATURAL LEFT JOIN Takes
WHERE studentId IS NULL;
```

- Find the category of books that Computer Science students are interested in

```
SELECT categoryName, deptName
FROM Borrow NATURAL JOIN Book
JOIN Student ON Borrow.studentId = Student.studentId
NATURAL JOIN Category
JOIN Department ON Department.id = departmentId
WHERE deptName = 'Computer Science';
```

JOINS

- Students from which department are reading “History” books?

Exercise for students

- Find students that still have not returned a borrowed book

```
SELECT DISTINCT fullName FROM Borrow NATURAL JOIN Student
WHERE returnDate IS NULL;
```

- Find students and book titles that are not returned for more than 30 days
 - CURDATE(), DATEDIFF()

Exercise for students

JOINS

What is the meaning of this query?

Can you describe the result in plain English?

Department (id, deptName, budget)

Student (studentId, fullName, departmentId)

```
SELECT deptName
FROM Department LEFT OUTER JOIN Student
ON Department.id = Student.departmentId
WHERE studentId IS NULL;
```

Session 13, 14: Oct 10, Oct 12

- Alias
- Limit
- Aggregate Functions
- Grouping

Manipulating SELECT results

- Arithmetic operations

```
SELECT deptName, ROUND(budget * 480 / 12, 2) FROM Department;  
SELECT title, classHours + labHours FROM Course;
```

- String functions: CONCAT, SUBSTR, UPPER, LOWER etc.

```
SELECT CONCAT( UPPER(title), ' ', credits ) FROM Course;
```

- Adding / removing columns

```
SELECT Book.title, Borrow.borrowDate, Borrow.returnDate,  
       Student.fullName, '2016-10-10'  
FROM Book NATURAL JOIN Borrow NATURAL JOIN Student;
```

Rename (Alias) – AS

- Renaming columns

```
SELECT deptName, ROUND(budget / 12, 2) AS monthly  
FROM Department;
```

```
SELECT title, classHours + labHours AS totalHours  
FROM Course;
```

Book and Course both have “title” columns:

```
SELECT Book.title AS book, Course.title AS course  
FROM Book JOIN Course ON Book.numAvailable > 0;
```

Rename (Alias) – AS

- Renaming tables

Get book from same category

```
SELECT r.title AS book1, s.title AS book2, r.categoryId  
FROM Book r JOIN Book s USING (categoryId);
```

Renaming to shorter names

```
SELECT c.*, d.deptName FROM Course c JOIN Department d  
ON c.departmentId = d.id;
```

Limit

- Return a limited **number** of rows starting from an **offset**

```
SELECT * FROM Book LIMIT {offset}, {count};  
SELECT * FROM Book LIMIT {count}; -- offset = 0
```

- Example: Pagination

```
SELECT * FROM Book LIMIT 0, 3;    -- 0, 1, 2  
SELECT * FROM Book LIMIT 3, 3;    -- 3, 4, 5
```

Project Deliverable

1. Description and use cases
2. ERD
3. Conversion to tables: Diagram & DDL
4. Initialized data ~ 200 rows
5. Queries according to use cases
6. Source code and a running program, if there is any

Presentation: 3rd week of November, each team 15 mins

Office Hour until 2pm today

Aggregate Functions

- MAX
- MIN
- COUNT
- SUM
- AVG
- ...

The result is not a set. But a single **value.**

- Find maximum / minimum of budget of departments
- Find number of students registered for Data Structures course
- Find number of students in Physics department
- Find total number of copy of books in the library
- Find average score of Data Structures class
- Find GPA of Sheldon Cooper – considering course credits

Basic Aggregation

Returns a single value

```
SELECT AGGREGATE_FUNCTION ( column ) ...
```

```
SELECT MAX( budget ) FROM Department;
```

```
SELECT COUNT(*) AS num_registered From Takes WHERE id = 1;
```

```
SELECT AVG( score ) AS class_average From Takes WHERE id = 1;
```

All registered students. Not the number of registrations.

```
SELECT COUNT( DISTINCT studentId) FROM Takes;
```

Aggregation with grouping

Results a set

```
SELECT column1, AGGREGATE_FUNCTION(column2)
FROM table
GROUP BY column1
```

Number of books in a category

```
SELECT COUNT(*) FROM Book GROUP BY categoryId;

SELECT categoryId, COUNT(*) FROM Book GROUP BY categoryId;
```

GROUP BY

bookId	title	numAvailable	categoryId
1	C++ How to Program	0	2
2	Java How to Program	10	1
3	Database System Cocepts	5	3
4	Advanced Java	100	1
5	Data Structures with C++	3	2
6	NoSQL Databasee	5	3
7	Database Design	4	3
8	Republic of Armenia	3	5
9	Game of Thrones	12	6
10	A Song of Ice and Fire	20	NULL
11	Java OOP	10	1
12	C++ Data Structures	0	2
13	Java Performance	NULL	1

bookId	title	numAvailable	categoryId
10	A Song of Ice and Fire	20	NULL
2	Java How to Program	10	1
4	Advanced Java	100	1
11	Java OOP	10	1
13	Java Performance	NULL	1
1	C++ How to Program	0	2
5	Data Structures with C++	3	2
12	C++ Data Structures	0	2
3	Database System Cocepts	5	3
6	NoSQL Databasee	5	3
7	Database Design	4	3
8	Republic of Armenia	3	5
9	Game of Thrones	12	6

GROUP BY categoryId

GROUP BY

- Try: `SELECT * FROM Book GROUP BY categoryId;`
- Try: `SELECT bookId, categoryId FROM Book GROUP BY categoryId;`
- You should only use attributes that are related to groups in SELECT

Aggregation with grouping

Find category names and number of books in each category

```
SELECT categoryName, COUNT(*) FROM Book NATURAL JOIN Category  
GROUP BY categoryId;
```

Find category names and number of copies books in each category

```
SELECT categoryName, SUM(numAvailable)  
FROM Book NATURAL JOIN Category  
GROUP BY categoryId;
```

Aggregation with grouping

- Only columns participating in the grouping must be selected
 - The column in GROUP BY clause
 - The column in Aggregate Function

Wrong query →

```
SELECT credits, COUNT(*)  
FROM Course  
GROUP BY departmentId;
```

Sorting groups

```
SELECT deptName, COUNT(*) as num  
FROM Course c JOIN Department d ON c.departmentId = d.id  
GROUP BY departmentId  
ORDER BY num, deptName DESC;
```

Multiple groups

- GROUP BY multiple groups

```
SELECT COUNT(*), SUM(classHours + labHours), departmentId, credits
FROM Course
GROUP BY departmentId, credits;    -- All columns have equal values in groups
```

- Groups are formed if all specified columns are equal
- If GROUP BY ... is missing, there is only one group: All rows in

Multiple groups

	departmentId	credits	id	title	classHours	labHours
G1	1	2	9	Software Project Management	3	NULL
	1	2	11	Operating Systems	3	NULL
G2	1	3	1	Data Structures	3	1
	1	3	4	OOP	3	3
	1	3	5	Database Systems	3	2
G3	2	4	2	Calculus	3	NULL
	2	4	3	Discrete Math	2	NULL
G4	3	2	6	Thermodynamics	2	NULL
G5	3	3	7	Mechanics	2	NULL
	3	3	12	Electricity	3	1
G6	4	2	8	Financial Management	2	NULL
G7	4	3	10	Project Management	3	NULL

Apply Aggregate Function on each group

Selecting groups

- We want to choose groups WHERE some condition is true
- Keyword: **HAVING**
- Find departments that offer more than 2 courses

```
SELECT COUNT(*), departmentId FROM Course GROUP BY departmentId  
HAVING COUNT(*) > 2;
```

- WHERE and HAVING

```
SELECT COUNT(*) AS num, departmentId FROM Course  
WHERE credits = 3 —————→ Selects some rows  
GROUP BY departmentId  
HAVING num > 2; —————→ Selects some groups
```

Statement Evaluation Order

```
SELECT COUNT(*) AS num, departmentId  
FROM Course  
WHERE credits = 3  
GROUP BY departmentId  
HAVING num > 2;
```

5. SELECT

1. FROM

2. WHERE

4. HAVING

3. GROUP BY

1. Tables are fetched and joined if required
2. The WHERE predicate is applied, rows are filtered
3. Groups are formed with the specified columns
4. The HAVING predicate is applied, groups are filtered
5. Projection is applied, columns are filtered

Statement Evaluation Order

```
SELECT
    COUNT(*) AS num,
    departmentId
FROM Course
WHERE credits = 3
GROUP BY departmentId
HAVING num > 2;
```

1) A \Leftarrow SELECT * FROM Course

2) B \Leftarrow Filter A WHERE (credits = 3) IS TRUE

3) C \Leftarrow Group rows of B by departmentId

4) D \Leftarrow Remove groups of C when num \leq 2

5) E \Leftarrow Leave only num and departmentId columns

HAVING vs WHERE

- Order of evaluation
- Performance
- Unnecessary HAVING \Rightarrow Avoid this!

```
SELECT COUNT(*) AS num, departmentId
FROM Course
WHERE credits = 3
GROUP BY departmentId
HAVING departmentId = 1
```

NULLs in aggregate functions

- Try **SUM()**, **AVG()**, **MIN()**, **MAX()** on NULL values
 - Aggregate functions ignore NULL values
- Try **COUNT(*)** and **COUNT(column)** on NULL values
 - COUNT(*) includes NULLs
 - COUNT(column) ignores NULLs

Session 16: October 19

Midterm Exam

- When? Tuesday, October 24th, 7:00 pm
- Where? Large Auditorium
- How? Closed book
- Chapters
 - 1: Introduction
 - 2: Relational Model
 - 3 & 4: SQL
 - 7: E-R Model

Set Operations

- Union, Intersect, Except
- Set comparison
- Nested subqueries

Union

- Collect results from two queries

(SELECT something, somehow)

UNION

(SELECT same thing from elsewhere)

- Results of first and second query should have the same structure

```
(SELECT DISTINCT studentId FROM Takes)  
  UNION  
(SELECT DISTINCT studentId FROM Borrow) ;
```

$R = \{ \dots \}$

$S = \{ \dots \}$

Result = $R \cup S$

Union

- Can you replace ORs with UNION?

```
SELECT Book.*  
FROM Book NATURAL JOIN Category  
WHERE categoryName IN ( 'Java' , 'C++' );
```

- UNION vs. UNION ALL
 - UNION removes duplicates
 - UNION ALL keeps the duplicates

Nested Queries

- Using result of a query in another query
- Joining without JOIN

```
1) SET @myID = (SELECT categoryId FROM Category WHERE ...)  
2) SELECT * FROM Book WHERE categoryId = @myID
```

Combine

```
SELECT * FROM Book  
WHERE categoryId = (SELECT categoryId FROM Category  
WHERE ...)
```

Nested Query vs. Joins

- Which one is "better" ?
- Easy to read and understand
- Performance

Nested Queries

- Get book with maximum number of availability

```
SELECT *  
FROM Book  
WHERE numAvailable = ( The query returning maximum )
```

- Try this query using join
- Derived Table
 - The result of subquery is a “derived” table
 - A derived table is stored in memory, not on disk
 - It is accessible only within the current query context

Nested Queries

- Derived tables should have names


```
SELECT Student.fullname, T.average
FROM
    Student Natural JOIN
    ( SELECT studentId, AVG(score) AS average FROM
        Takes GROUP BY studentId ) AS T
```

Nested Queries

- Use database: **Practice**
- Get possible values for **orders.status**
- See number of orders in each status
- Compute % of shipped orders
- See customers who have any order with “In Process” status
- Add number of customers
- Get a list of customers with number of orders for each customer
- Show total number of orders per city

Customers with any order in process

```
SELECT customers.city, customers.phone,  
customers.customerName, A.c  
FROM customers NATURAL JOIN  
( SELECT count(*) AS c, customerNumber, status  
  FROM orders WHERE status = 'In Process'  
  GROUP BY customerNumber) AS A  
ORDER BY A.c DESC;
```



Exposed from nested query

Customers and # of total orders

```
SELECT customers.customerName, A.total
FROM customers NATURAL JOIN
( SELECT customerNumber, COUNT(*) AS total
  FROM orders GROUP BY customerNumber) AS A
ORDER BY A.total DESC;
```


Customers' cities with total orders

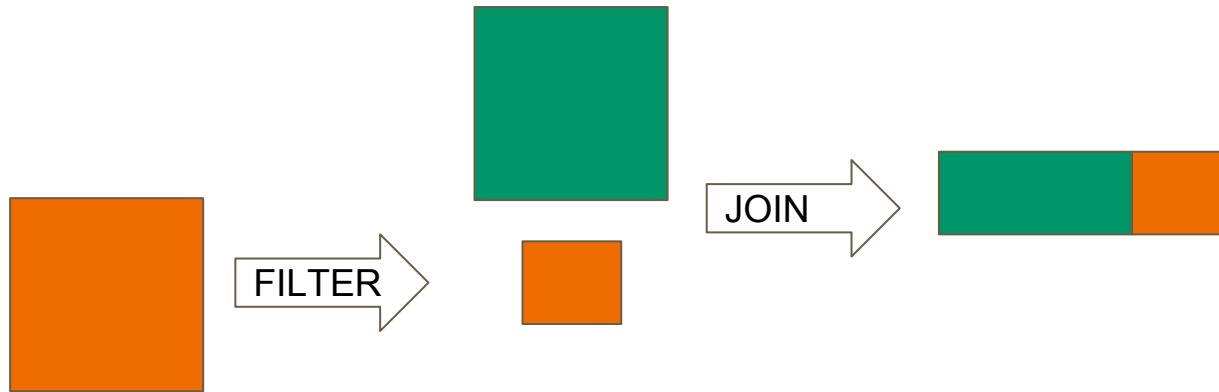
```
SELECT SUM(A.total) AS t, customers.city
FROM customers NATURAL JOIN
( SELECT customerNumber, COUNT(*) AS total
  FROM orders
  GROUP BY customerNumber) AS A
GROUP BY city ORDER BY t DESC
```

Again... compare with joins

- Exercise: Try to rewrite these queries using multiple JOINS
- JOINS produce larger tables
- We then filter most rows of the large tables



Nested queries: Smaller tables



Nested queries vs. JOINS

- (Sometimes) alternative ways to get to the same result
- You can think and write query in the most natural way that you find
- Also... think about the performance
- In general, getting to smaller tables: The sooner, the better
- 100,000 customers and 1,000,000 orders
 - Only 5% of them are “In Process”

IN

- Students who have taken a specific course and also borrowed a specific book
- Intersect: Not supported by MySQL
- Can be simulated with IN

```
SELECT *  
FROM Student WHERE studentId IN  
(SELECT studentId FROM Borrow WHERE bookId = 1)  
AND studentId IN  
(SELECT studentId FROM Takes WHERE id = 1);
```

NOT IN

- Students who have taken a specific course but not borrowed a specific book
- Except: Not supported by MySQL
- NOT IN can be used to simulate “Except”

```
SELECT *  
FROM Student WHERE studentId NOT IN  
(SELECT studentId FROM Borrow WHERE bookId = 1)  
AND studentId IN  
(SELECT studentId FROM Takes WHERE id = 1);
```

EXISTS, NOT EXISTS

- Test for empty results

```
SELECT *  
FROM Student S  
WHERE EXISTS  
    (SELECT * FROM Borrow WHERE studentId = S.studentId);
```

Correlated Subquery

- Try **NOT EXISTS**

ALL, ANY

ALL `SELECT *`
 `FROM Department`
 `WHERE budget >= ALL (SELECT budget FROM Department);`

ANY `SELECT *`
 `FROM Department`
 `WHERE Department.id NOT IN`
 `(SELECT D.id FROM Department D WHERE budget < ANY`
 `(SELECT budget FROM Department)) ;`

Modify data using nested queries

Move a course from one department to another

```
UPDATE Course
SET departmentId =
    (SELECT id FROM Department WHERE deptName = 'Business')
WHERE title = 'Financial Management';
```

```
INSERT Takes(id, studentId)                                -- Someone registers in a course
VALUE (
    (SELECT id FROM Course WHERE title = 'Financial Management'),
    (SELECT studentId FROM Student WHERE fullName = 'Leonard')
);
```

Modify data using nested queries

- All students should take the Financial Management course
 - **INSERT Takes(studentId, id) { Result of another query }**
 - Prepare the SELECT query
 - INSERT the result into **Takes** tables
 - Take care of duplicates

```
SELECT *                                -- What does this query do?
FROM Student S, Department D, Course C
WHERE D.deptName = 'Computer Science'
      AND C.title = 'Data Structures'
      AND (S.studentId, C.id) NOT IN
      (SELECT studentId, id FROM Takes);
```

Modify data using nested queries

- INSERT the result in **Takes** table

INSERT Takes (studentId, id) ...

Views

```
CREATE VIEW ViewName( fields ) AS  
    SQL_Query
```

```
CREATE VIEW StudentEntollment AS  
    SELECT ... FROM Takes NATURAL JOIN Student ... ;
```

Views

- Useful for
 - Query reuse
 - Getting more readable results
 - Access control
- Views are just a definition, they are not physically created
- But when a view is used frequently, they can be temporarily created
- Materialized views
 - Tracking updates: On database change vs. on view access
 - View modification: Adding rows into view... ?

Transactions

- Some operations should be *atomic*
- Hotel reservation
 - The room is marked as reserved
 - An invoice is issued to the guest
 - Payment
 - Guest pays: Room is reserved and number of available rooms is decreased
 - Payment rejected: Room is marked as available and invoice is cancelled
 - What if two guests from different locations are reserving the last available room?

Transactions*



- Guest 1 reserves but fails to pay
- Guest 1 reserves and pays while guest 2 pays faster
- Payments from both guests are accepted

**If anything can go wrong it will definitely go wrong*

Transactions

Avoid situations

- A room is reserved but not paid
- A guest pays but doesn't get the room
- Strangers ending up in the same room

Maintain the consistent state

- A reserved room must be paid and have a guest
- When a guest is denied, must get the payment back
- Transactions are **A**tomic, **C**onsistent, **I**solated and **D**urable

Transactions – Only 3 steps

SET AUTOCOMMIT = 1

- 1) BEGIN
- 2) Some SQL update queries
- 3) COMMIT or ROLLBACK

ROLLBACK

- Can be due to software or hardware failure
- Can be configured to happen after a certain time (timeout)