Announcements

• Due dates approaching:
  – Project 1 Milestone 1
  – Term Paper

• Next:
  – Project 2 Part 1
    • Will be out soon!
    • Individual project
    • More like an assignment
    • Writing a few `real’ SQL queries.
• Finish SQL (Chapter 5, 6, and 7)

• We will start Chapter 3 in next lecture
What did we learn?

• Queries on single tables:
  – SELECT ... FROM ... WHERE ...
  – DISTINCT
  – ORDER BY
  – AND, OR, NOT

• What Else:
  – NULL Values
  – DEFAULT Values
  – PRIMARY KEYs
  – UNIQUE
  – LIKE
  – LIMIT
What did we learn?

• Queries on multiple tables:
  – Set operations (UNION, INTERSECT, EXCEPT)
  – Cross Product and JOIN

• What Else:
  – Aliases (AS)
  – IN
  – ANY, ALL
  – EXISTS
  – Foreign Key
Finally

• Aggregation:
  – MIN, MAX, COUNT, AVG, SUM
  – GROUP BY
  – HAVING
What Will We Learn Today?

• Different types of JOIN
  – We only covered (INNER JOIN)

• Constraints

• Other SQL statements related to Data Manipulation
  – INSERT INTO <TB>
  – DELETE FROM <TB>
  – UPDATE <TB> SET

• Other SQL statement related to Data Definition
  – CREATE TABLE
  – DELETE TABLE
  – ALTER TABLE
RECAP: Inner Joins

By default, joins in SQL are “inner joins”:

```
Product(name, category)
Purchase(prodName, store)

SELECT Product.name, Purchase.store
FROM Product
JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```

Both equivalent: Both INNER JOINS!
By default, joins in SQL are “inner joins”:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM Product
JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```

However: Products that never sold (with no Purchase tuple) will be lost!
Outer Joins

• An **outer join** returns tuples from the joined relations that don’t have a corresponding tuple in the other relations
  – I.e. If we join relations A and B on \( a.X = b.X \), and
  – there is an entry in A with \( X=5 \), but none in B with \( X=5 \)...
    • A LEFT OUTER JOIN will return a tuple \((a, \text{NULL})\)!

• Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM Product
    LEFT OUTER JOIN Purchase
        ON Product.name = Purchase.prodName
```

Now we’ll get products even if they didn’t sell
INNER JOIN:

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

```
SELECT Product.name, Purchase.store
FROM Product
INNER JOIN Purchase
ON Product.name = Purchase.prodName
```

Note: another equivalent way to write an INNER JOIN!
**LEFT OUTER JOIN:**

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>prodName</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Camera</td>
</tr>
</tbody>
</table>

**SQL Query:**

```sql
SELECT Product.name, Purchase.store
FROM Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
```
Other Outer Joins

• Left outer join:
  – Include the left tuple even if there’s no match

• Right outer join:
  – Include the right tuple even if there’s no match

• Full outer join:
  – Include the both left and right tuples even if there’s no match
TWO OTHER SQL CONSTRUCTS: WITH & CASE
Use of WITH

• The WITH clause allows a user to define a table that will only be used in a particular query (not available in all SQL implementations)

• Used for convenience to create a temporary “View” and use that immediately in a query

• Allows a more straightforward way of looking a step-by-step query

Latest MySQL (MySQL 8.0 supports WITH)
Example of WITH

• See an alternate approach to doing Q28:

• Q28':

```
WITH BIGDEPTS (Dno) AS
    (SELECT Dno FROM EMPLOYEE
     GROUP BY Dno
     HAVING COUNT(*) > 5)
SELECT Dno, COUNT(*)
FROM EMPLOYEE
WHERE Salary>40000 AND Dno IN BIGDEPTS
GROUP BY Dno;
```

Retrieve the department number having more than 5 employees and the number of its employees who are making more than $40,000)
Use of CASE

• SQL also has a **CASE** construct

• Used when a value can be different based on certain conditions.

• Can be used in any part of an SQL query where a value is expected

• Applicable when querying, inserting or updating tuples
• The following example shows that employees are receiving different raises in different departments

```sql
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
END);
```
Another Example of CASE

SELECT OrderID, Quantity,
    CASE
        WHEN Quantity > 30 THEN "The quantity is greater than 30"
        WHEN Quantity = 30 THEN "The quantity is 30"
        ELSE "The quantity is something else"
    END
AS NOTE
FROM OrderDetails;

<table>
<thead>
<tr>
<th>OrderID</th>
<th>Quantity</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10255</td>
<td>20</td>
<td>The quantity is something else</td>
</tr>
<tr>
<td>10255</td>
<td>35</td>
<td>The quantity is greater than 30</td>
</tr>
<tr>
<td>10255</td>
<td>25</td>
<td>The quantity is something else</td>
</tr>
<tr>
<td>10255</td>
<td>30</td>
<td>The quantity is 30</td>
</tr>
<tr>
<td>10255</td>
<td>15</td>
<td>The quantity is something else</td>
</tr>
</tbody>
</table>
CONSTRAINTS
Types of Constraints

- **Implicit Constraints:**
  - Inherent model-based constraints
  - Ex: Relations cannot contain duplicates

- **Explicit Constraints:**
  - Schema based constraints
  - We will mostly focus on these constraints.
  - Ex: Domain, Primary Key, Foreign Key, Not NULL,

- **Semantic Integrity constraints:**
  - Usually enforced within the application program.
  - SQL uses Triggers and Assertions as a way to handle this.
Domain Constraints

- States, within each tuple, each attribute A must be an atomic value from the domain of A.
  - Ex: You can not pass a string as EmployeeID
Entity Integrity and Referential Integrity

- **Entity integrity constraint** states that no primary key value can be NULL.

- **Referential integrity constraint** states that every value of a foreign key must match a value of an existing primary key.

**Self study**: Key, superkey, candidate key and primary key
Assertions and Triggers

• Specifying
  – Constraints as Assertions
  – Actions as Triggers

• **CREATE ASSERTION**
  – Specify additional types of constraints outside scope of built-in relational model constraints

• **CREATE TRIGGER**
  – Specify automatic actions that database system will perform when certain events and conditions occur
• **CREATE ASSERTION**

  – Specify a query that selects any tuples that violate the desired condition
  – Use only in cases where it goes beyond a simple `CHECK` which applies to individual attributes and domains

```sql
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT * FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D WHERE E.Salary>M.Salary AND E.Dno=D.Dnumber AND D.Mgr_ssn=M.Ssn ) );
```
**Introduction to Triggers in SQL**

- **CREATE TRIGGER** statement
  - Used to monitor the database
- Typical trigger has **three** components which make it a rule for an “active database”:
  - Event(s)
  - Condition
  - Action
CREATE TRIGGER upd_check AFTER INSERT ON account
  FOR EACH ROW
  BEGIN
    IF NEW.amount < 0 THEN
      SET NEW.amount = 0;
    ELSEIF NEW.amount > 100 THEN
      SET NEW.amount = 100;
    END IF;
  END;
We are going backwards!

GOING BEYOND SELECT
Recall: CRUD

CRUD refers to all of the major functions that are implemented by each DBMS. Each letter corresponds to a standard SQL statement.

<table>
<thead>
<tr>
<th>Operation</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>INSERT</td>
</tr>
<tr>
<td>Update (Modify)</td>
<td>UPDATE</td>
</tr>
<tr>
<td>Delete (Destroy)</td>
<td>DELETE</td>
</tr>
</tbody>
</table>
INSERT INTO

Syntax

```
INSERT INTO table_name (column1, column2, column3, ...) 
VALUES (value1, value2, value3, ...);
```

Only if adding values for all columns

```
INSERT INTO table_name 
VALUES (value1, value2, value3, ...);
```

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country) 
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');
```
**UPDATE <TB> SET**

**Syntax**

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

**Example**

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City = 'Frankfurt'
WHERE CustomerID = 1;
```

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

https://www.w3schools.com/sql/sql_update.asp
DELETE FROM

Syntax

```
DELETE FROM table_name
WHERE condition;
```

Example

```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste';
```
Recall: CRUD

- CRUD refers to all of the major functions that are implemented by each DBMS. Each letter corresponds to a standard SQL statement.

<table>
<thead>
<tr>
<th>Operation</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>INSERT</td>
</tr>
<tr>
<td>Read (Retrieve)</td>
<td>SELECT</td>
</tr>
<tr>
<td>Update (Modify)</td>
<td>UPDATE</td>
</tr>
<tr>
<td>Delete (Destroy)</td>
<td>DELETE</td>
</tr>
</tbody>
</table>

CRUD refers to all of the major functions that are implemented by each DBMS. Each letter corresponds to a standard SQL statement.
Are we done?

• No.

• Can we apply CRUD to the database itself?
  —Yes!
## CRUD on Databases

<table>
<thead>
<tr>
<th>Operation</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>CREATE TABLE</td>
</tr>
<tr>
<td>Read (Retrieve)</td>
<td>SHOW TABLES</td>
</tr>
<tr>
<td>Update (Modify)</td>
<td>ALTER TABLE</td>
</tr>
<tr>
<td>Delete (Destroy)</td>
<td>DROP TABLE</td>
</tr>
</tbody>
</table>
CREATE TABLE

ten_name (  
column1 datatype,  
column2 datatype,  
column3 datatype,  
....  
);

Example

CREATE TABLE Orders (  
OrderID int NOT NULL,  
OrderNumber int NOT NULL,  
PersonID int,  
PRIMARY KEY (OrderID),  
CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)  
REFERENCES Persons (PersonID)  
);
SHOW TABLES

Syntax

SHOW TABLES
FROM db_name
WHERE expr

Real Syntax

SHOW [FULL] TABLES
[FROM | IN] db_name
[LIKE 'pattern' | WHERE expr]

Adding FULL shows VIEWs too. Coming up soon!

ALTER TABLE
ADD column_name datatype;

ALTER TABLE table_name
DROP COLUMN column_name;

ALTER TABLE table_name
MODIFY COLUMN column_name datatype;

Note: This conversion may result in alteration of data.

DROP TABLE

**Syntax**

```sql
DROP TABLE table_name;
```

**Example**

```sql
DROP TABLE Shippers;
```
VIEWS
Views (Virtual Tables) in SQL

• Concept of a view in SQL
  – Single table derived from other tables called the defining tables
  – Considered to be a virtual table that is not necessarily populated
CREATE VIEW syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
CREATE VIEW [Current Product List] AS
SELECT ProductID, ProductName
FROM Products
WHERE Discontinued = No;
```

```
SELECT * FROM [Current Product List];
```
View Implementation

• Complex problem of efficiently implementing a view for querying

• View is always up-to-date
  – Responsibility of the DBMS and not the user
View Materialization

• **Strategy 1: Query modification** approach
  – Compute the view as and when needed. Do not store permanently
  – Modify view query into a query on underlying base tables
  – Disadvantage:
    • inefficient for views defined via complex queries that are time-consuming to execute

• **Strategy 2: View materialization**
  – Physically create a temporary view table when the view is first queried
  – Keep that table on the assumption that other queries on the view will follow
  – Requires **efficient strategy** for automatically updating the view table when the base tables are updated
Multiple ways to handle materialization:

- **Immediate update** strategy updates a view as soon as the base tables are changed.

- **Lazy update** strategy updates the view when needed by a view query.

- **Periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).
  - This is commonly used in Banks, Retail store operations, etc.
FOREIGN KEY CONSTRAINT
Note the `CONSTRAINT` keyword. You can name any constraint.
Creating a Foreign Key after Table Creation

ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
Using FOREIGN KEY Constraints

```sql
[CONSTRAINT [symbol]] FOREIGN KEY
  [index_name] (index_col_name, ...)
  REFERENCES tbl_name (index_col_name,...)
  [ON DELETE reference_option]
  [ON UPDATE reference_option]

reference_option:
  RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```
• **CASCADE:**
  – Delete or update the row from the parent table, and automatically delete or update the matching rows in the child table.
  – Both ON DELETE CASCADE and ON UPDATE CASCADE are supported

• **SET NULL:**
  – Delete or update the row from the parent table, and set the foreign key column or columns in the child table to NULL.
  – Both ON DELETE SET NULL and ON UPDATE SET NULL clauses are supported.

• **RESTRICT / NO ACTION:**
  – Rejects the delete or update operation for the parent table.
  – Specifying RESTRICT (or NO ACTION) is the same as omitting the ON DELETE or ON UPDATE clause.

• **SET DEFAULT:**
  – This action is recognized by the MySQL parser but no action is taken.
## CRUD: Differences between Database and Tables

<table>
<thead>
<tr>
<th>CRUD</th>
<th>Table</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>INSERT INTO &lt;TB&gt;</td>
<td>CREATE TABLE</td>
</tr>
<tr>
<td>Read</td>
<td>SELECT</td>
<td>SHOW TABLES</td>
</tr>
<tr>
<td>Update</td>
<td>UPDATE &lt;TB&gt; SET</td>
<td>ALTER TABLE</td>
</tr>
<tr>
<td>Delete</td>
<td>DELECT FROM &lt;TB&gt;</td>
<td>DROP TABLE</td>
</tr>
</tbody>
</table>
Some of the slides in this presentation are taken from the slides provided by the authors.

Many of these slides are taken from cs145 course offered by Stanford University.

https://www.w3schools.com/

MySQL 5.5 Reference Manual