Database1

Kandahar University
Computer Science Faculty

Sayed Ahmad Sahim

DBMS 5_SQL

The Lecture

1. Schema definitions and constraints in SQL
2. Data Definition and Data Types
3. Basic Retrieval Queries in SQL

What is SQL?

- Considered one of the major reasons for the commercial success of relational databases
- a language for querying and manipulating data
- developed in the 1970s as the first commercial language for Codd's relational model.
- Core specification
  - many dialects of SQL depending on the server vendor
    - most implement a superset of standarts
    - we cover the basic SQL syntax

SQL

- Statements for data definitions, queries, and updates
- Data Definition Language (DDL)
  - Define relational schemata
  - CREATE/ALTER/DELETE tables and their attributes
  - CONSTRAINTS
  - VIEWS
- Data Manipulation Language (DML)
  - SELECT - Query one or more tables
  - INSERT/DELETE/UPDATE tuples in tables
- Other topics
  - Transaction controls
  - Indexing
  - Security specification
  - Procedural extensions
  - ...

Terminology

- A **relation** or **table** is a
  - **multiset of tuples** (duplicate instances allowed).
- A **tuple** or **row** is a single entry in the table
  - The number of tuples is the **cardinality** of the relation.
- An **attribute** (or **column**) is a **typed data entry** present in each tuple in the relation.
  - The number of attributes is the **arity** of the relation.

SQL conventions

- SQL **commands** are **case insensitive**
  - *SELECT* $\equiv$ *Select* $\equiv$ *select*
- Identifiers
  - depend on DBMS, config
- **Values** are **case sensitive**
  - '*German*' $\neq$ '*german*'
- Each statement in SQL ends with a Semicolon

Database Schema & State

A relational **database schema** is a

- **set of schemas** for its **tables** together with a
- **set of integrity constraints**
- views, domains, and other constructs

- CREATE SCHEMA ||*DATABASEstatement*

A relational **database state** is a

- **set of states** of its tables
- such that **no integrity constraint is violated**.

Schema and Constraints are how databases understand the semantics (meaning) of data.

CREATE TABLE Command in SQL

- Specifying a new relation
  - Provide name of table
  - Specify attributes, their types and initial constraints
- Can optionally specify schema:
  - **CREATE TABLE COMPANY.EMPLOYEE** ...
    - or
  - **CREATE TABLE EMPLOYEE** ...

CREATE / ALTER TABLE

```
CREATE TABLE City (
    ID primary key,
    Name varchar(10) not null,
    CountryCode char(3),
    District varchar(10),
    Population float
);

ALTER TABLE City RENAME COLUMN Name Cityname;
ALTER TABLE City ADD COLUMN Mayor varchar(10);
ALTER TABLE City DROP COLUMN Mayor;
```

City

ID	Name	CountryCode	District	Population
1	Kabul	AFG	Kabol	1780000
1523	Wien	AUT	Wien	8091800
3425	Kampala	UGA	Central	890800
...	...	...	...	...

CREATE TABLE Command in SQL

- **Base tables (base relations)**
  - Relation and its tuples are actually created and stored as a file by the DBMS
- **Virtual relations (views)**
  - Created through the CREATE VIEW statement. Do not correspond to any physical file.

Attributes in SQL

An attribute (or column) is a typed data entry present in each tuple in a relation.

```
mysql> mysql> explain city;
+-------------+----------+------+-----+---------+----------------+
| Field       | Type     | Null | Key | Default | Extra          |
+-------------+----------+------+-----+---------+----------------+
| ID          | int(11)  | NO   | PRI | NULL    | auto_increment |
| Name        | char(35) | NO   |     |         |                |
| CountryCode | char(3)  | NO   | MUL |         |                |
| District    | char(20) | NO   |     |         |                |
| Population  | int(11)  | NO   |     | 0       |                |
+-------------+----------+------+-----+---------+----------------+
5 rows in set (0,01 sec)
```

Attribute types

Attribute types

- Characters: CHAR(20), VARCHAR(50)
- Numbers: INT, BIGINT, SMALLINT, FLOAT
- Boolean: TRUE, FALSE, NULL
- Binary object: BLOB
- Time: DATE, TIME, TIMESTAMP ...
- Others:
  - ARRAY, MULTISET, JSON, XML

.. many more, depending on the type of SQL-Server.

Attributes have an atomic type in standard SQL. However, sets and arrays are allowed in some SQL versions.

Boolean

- TRUE / FALSE values
- 3rd value: UNKNOWN
  - represented as NULL

NULL

NULL can mean many things:

- Value does not exists
- Value exists but is unknown
- Value not applicable

The schema specifies for each attribute if can be null (nullable attribute) or not.

- **NOT NULL** constraint

Constraints in SQL

- **Constraints** are rules enforced on **data columns**.
- They limit the type of data that can go into a table.
  - to supportdata consistency and integrity.

Specifying Constraints in SQL

- **required data**: Whether a data field must contain a value
- **domain constraints**: A set of legal values for a field
- **entity integrity**: Each tuple contains a unique, non-null key value
- **referential integrity**: valid references to other relations
- **general constraints**: General, organization specific constraints
  - such as requiring that no staff member may handle > 100 properties

Domain Constraints

Attribute Domain:

- each attribute value must be either **NULL**
- or drawn from the **domain** of that attribute
- **NOT NULL** constraint upon an attribute
  - permits the value NULL

Default value

- **DEFAULT** $<$ *value* $>$
  - Provides a default value for a column when none is specified.
  - default: NULL

CHECK clause

- **CHECK (**$<$ *booleanexpression* $>$**)**
- not in MySql!

```
Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber <
    21);
```

Key Constraints

PRIMARY

- Uniquely identified each rows/records in a database table.

UNIQUE

- Ensures that all values in a column are different.

FOREIGN

- a field that matches the primary key column of another table

Specifying Key Constraints

- PRIMARY KEY clause
  - Specifies one or more attributes that make up the primary key of a relation
  - Dnumber INT PRIMARY KEY;
- UNIQUE clause
  - Specifies alternate (secondary) keys (called CANDIDATE keys in the relational model).
  - Dname VARCHAR(15) UNIQUE;

```
CREATE TABLE Student(
   sid CHAR(20) PRIMARY KEY,
   cid     CHAR(20)
)
CREATE TABLE Enrolled(
   student_id CHAR(20), cid     CHAR(20),
   PRIMARY KEY (student_id, cid),
)
```

Referential Integrity Constraints - Foreign Keys

A **foreign Key** of relation R is a set of attributes whose values match a **primary key** in a relation S.
It's used by tuples in R for identifying/referring to a tuple in S

- R is called the referencing relation
- S the referenced relation
- **FOREIGN KEY** clause

Foreign Key Constraints

```
Students(sid: string, name: string, gpa: float)
Enrolled( student_id: string ,cid: string, grade: string)

CREATE TABLE Enrolled(
   student_id CHAR(20),
   cid      CHAR(20),
   grade    CHAR(10),
   PRIMARY KEY (student_id, cid),
   FOREIGN KEY (student_id) REFERENCES Students(sid)
)
```

Giving Names to Constraints

- Using the Keyword CONSTRAINT c_name
  - Useful for later altering

Declaring Foreign Keys

- Some foreign keys may cause errors
  - Specified either via:
    - Circular references
    - Or because they refer to a table that has not yet been created
- DBA's have ways to stop referential integrity enforcement to get around this problem.

MySql Ex: stop referential integrity enforcement

```
SET FOREIGN_KEY_CHECKS=0;
```

Foreign Keys and Update Operations

```
Students(sid: string, name: string, gpa: float)
Enrolled( student_id: string ,cid: string, grade: string)
```

- What if we insert a tuple into *Enrolled* but no corresponding student?
  - INSERT is rejected (foreign keys are constraints)!

Options for what to do when a delete (ON DELETE) or update (ON UPDATE) operation would violate a constraint:

- RESTRICT: don't carry out the operation
- CASCADE
- SET NULL
- SET DEFAULT

Foreign Keys CASCADE - MySql Example

```
CREATE TABLE parent (
    id INT NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (id)
);

CREATE TABLE child (
    id INT NOT NULL AUTO_INCREMENT, parent_id INT,
    INDEX par_ind (parent_id),
    CONSTRAINT child_par
    FOREIGN KEY (parent_id)
        REFERENCES parent(id)
        ON DELETE CASCADE
        ON UPDATE ...
);
```

Summary DDL

- **Schema** and **Constraints** are how databases understand the semantics (meaning) of data
- SQL supports general constraints:
  - **Keys** and **foreign** keys are most important

The Lecture

1. The SFW query
2. Other useful operators: LIKE, DISTINCT, ORDER BY

The Basic SELECT FROM WHERE Query

- The SELECT statement is used to select data from a database.
- It's filtering **tables rows** on some **condition**.
- The result is stored in a result table, called the **result-set**.

```
SELECT column_name(s)
FROM table_name(s)
WHERE condition(s);
```

- WHERE condition(s)
  - Boolean condition that must be true for any retrieved tuple
  - may include join conditions
- Logical comparison operators
  - $=, <, <=, >, >=,$ *and* $<>$

Conditions and BOOLEAN operators

Logic	SQL
$\vee$	**or**
$\wedge$	**and**
$\neg$	**not**
$(a \geq x) \wedge (a \leq y)$	**between** x **and** y

```
SELECT <attr_list>
FROM <table_list>
[WHERE <condition> AND <condition> OR <condition >];
```

Aliases or tuple variables

- Table aliases: Declare alternative relation names E and S to refer to the EMPLOYEE relation twice in a query:

```
SELECT  E.Fname, E.Lname, S.Fname, S.Lname
FROM  EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

example: For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

- Attribute aliases: attribute names can also be aliased

```
SELECT column_name AS alias_name
FROM table_name;
```

Unspecified WHERE Clause

```
SELECT name, language from country, countrylanguage;
```

Missing WHERE clause

- Indicates no condition on tuple selection
  - could result in CROSS PRODUCT

Use of the Asterisk

Specify an asterisk (*)

- Retrieve **all the attribute values** of the selected tuples

```
SELECT * FROM table_name;
```

Specify *column_name(s)*

- Retrieve specified **attribute values** of the selected tuples

```
SELECT column_name(s) FROM table_name;
```

Substring Pattern Matching

- comparison operator **LIKE**
  - Used for string pattern matching
  - **%** matches an arbitrary number of zero or more characters
  - underscore _ matches a single character

```
SELECT * FROM CountryLanguage
WHERE CountryCode
LIKE 'AF%';
```

```
mysql> SELECT * FROM countrylanguage WHERE countrycode  LIKE "AF%";
+-------------+------------+------------+------------+
| CountryCode | Language   | IsOfficial | Percentage |
+-------------+------------+------------+------------+
| AFG         | Balochi    | F          |        0.9 |
| AFG         | Dari       | T          |       32.1 |
| AFG         | Pashto     | T          |       52.4 |
| AFG         | Turkmenian | F          |        1.9 |
| AFG         | Uzbek      | F          |        8.8 |
+-------------+------------+------------+------------+
5 rows in set (0,00 sec)

mysql>
```

String pattern Matching Examples

```
SELECT * FROM CountryLanguage WHERE CountryCode
LIKE 'A__';
```

```
SELECT * FROM CountryLanguage WHERE CountryCode='%F%';
```

```
SELECT * FROM CountryLanguage WHERE CountryCode >= 'AFG'
    ;
```

BETWEEN comparison operator

```
SELECT * FROM countrylanguage
WHERE percentage BETWEEN 5 AND 10;
```

same result as:

```
SELECT * FROM countrylanguage
WHERE percentage > 5 AND percentage < 10;
```

Arithmetic Operators

- Standard **arithmetic operators**:
  - Addition **(+)**
  - subtraction **(–)**
  - multiplication **(*)**
  - division **(/)**

```
SELECT   E.Fname, E.Lname,
1.1 * E.Salary AS Increased_sal
FROM   EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE   E.Ssn=W.Essn AND W.Pno=P.Pnumber AND P.Pname=
          ProductX      ;
```

Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

DISTINCT: Eliminating Duplicates

- SQL does not automatically eliminate duplicate tuples in query results
- Use the keyword **DISTINCT** in the SELECT clause
  - Only distinct tuples remain in the result

Ex: duplicates in result

```
SELECT Language FROM CountryLanguage;
```

[..., Dari,..., German,..., German, ..., Pashto, Pashto, ..]

Ex: **DISTINCT** keyword eliminates duplicates

```
SELECT DISTINCT Language FROM CountryLanguage;
```

[..., Dari,..., German, ..., Pashto, ..]

ORDER BY: Sorting the Results

- Keyword **DESC** to see result in a descending order of values
- Keyword **ASC** to specify ascending order explicitly
- **ASC** per default

```
SELECT Language FROM CountryLanguage ORDER BY Language;
```

```
mysql> select language from countrylanguage order by language;
+--------------------------+
| language                 |
+--------------------------+
| Abhyasi                  |
| Acholi                   |
| Adja                     |
| Afar                     |
| Afar                     |
| Afrikaans                |
| Afrikaans                |
| Aimará                   |
```

ORDER BY, LIMIT

```
SELECT    language
FROM      countrylanguage
ORDER BY  language DESC
LIMIT  1
```

```
mysql> select language from countrylanguage order by language DESC LIMIT 1;
+--------------+
| language     |
+--------------+
| [South]Mande |
+--------------+
1 row in set (0,00 sec)
```

- Postgres, MySQL uses LIMIT k
- SQL Server uses TOP k

INSERT, DELETE, and UPDATE Statements in SQL

Three commands used to modify the database state:

- **INSERT** inserts a tuple (row) in a relation (table)
- **UPDATE** update a number of tuples on a condition
- **DELETE** delete a number of tuples on a condition

INSERT

- **add** one or more **tuples** to a relation
- Constraints on data types are observed automatically
- Any integrity constraints as a part of the DDL specification are enforced
- Specify the **relation name**, a **list of attributes** and a **list of values** for the tuple.
  - provide NULL if necessary

INSERT

Specify the **relation name**, a **list of attributes** and a **list of values** for the tuple.

```
INSERT INTO TABLE_NAME
[ (col1, col2, col3,...colN)]
VALUES (value1, value2, value3,...valueN);
```

You may omit the attribute list if all values are provided.

- attribute values should be listed in the same order as they were
  - specified in the CREATE TABLE command

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

Insert Data

```
CREATE TABLE Product (
    pname varchar(10) primary key,
    price float,
    category char(20),
    manufacturer text
);

INSERT INTO Product VALUES ('Gizmo',19.99,'Gadgets','
    GizmoWorks');
INSERT INTO Product VALUES ('Powergizmo',29.99,'Gadgets'
    ,'GizmoWorks');
INSERT INTO Product VALUES ('SingleTouch',149.99,'
    Photography','Canon');
INSERT INTO Product VALUES ('MultiTouch',203.99,'
    Household','Hitachi');
```

Bulk Insert from Data file (MySql)

```
CREATE TABLE discounts (
    id INT NOT NULL,
    title VARCHAR(255) NOT NULL, expired_date DATE NOT
        NULL,
    amount DECIMAL(10, 2) NULL, PRIMARY KEY (id)
);
```

CSV data file that matches the number of columns and the data type of each column:

discounts.csv

```
1  id,title,expired date,amount
2  1,"Spring Break 2014",20140401,20
3  2,"Back to School 2014",20140901,25
4  3,"Summer 2014",20140825,10
```

MySql data import:

```
LOAD DATA INFILE 'c:/tmp/discounts.csv'
INTO TABLE discounts
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Other Ways to Bulk Insert

```
CREATE TABLE Product (
    pname varchar(10) primary key,
    price float,
    category char(20),
    manufacturer text
);

INSERT into Product (column_name(s))
       SELECT    ...
       FROM    ...
       WHERE    ...   ;
```

Quick method: create AND insert

```
CREATE TABLE Product AS
       SELECT    ...
       FROM    ...
       WHERE    ...   ;
```

UPDATE

- **modify attribute values** of one or more selected tuples
- A **WHERE**-clause selects the tuples to be modified
- **SET**-clause specifies the attributes to be modified and their new values
  - An assignment is of the form: **attribute=value**.
- Referential integrity specified as part of DDL specification is enforced

```
UPDATE table_name
SET column_name1 = value1, column_name2 = value2, ...
[WHERE condition]
```

```
UPDATE Products set category='gadets'
where pname='gizmo'
```

Deleting Tuples conditionally

To delete tuples from a table we use:

- **delete from** table **where** condition
  - The **where** clause works like in a **select** statement.

```
DELETE from Products where pname='gizmo';
```

Deleting a table

Attention:

Delete table and data:

```
DROP table Products
```

Truncate data only:

```
TRUNCATE TABLE table_name;
```

Views

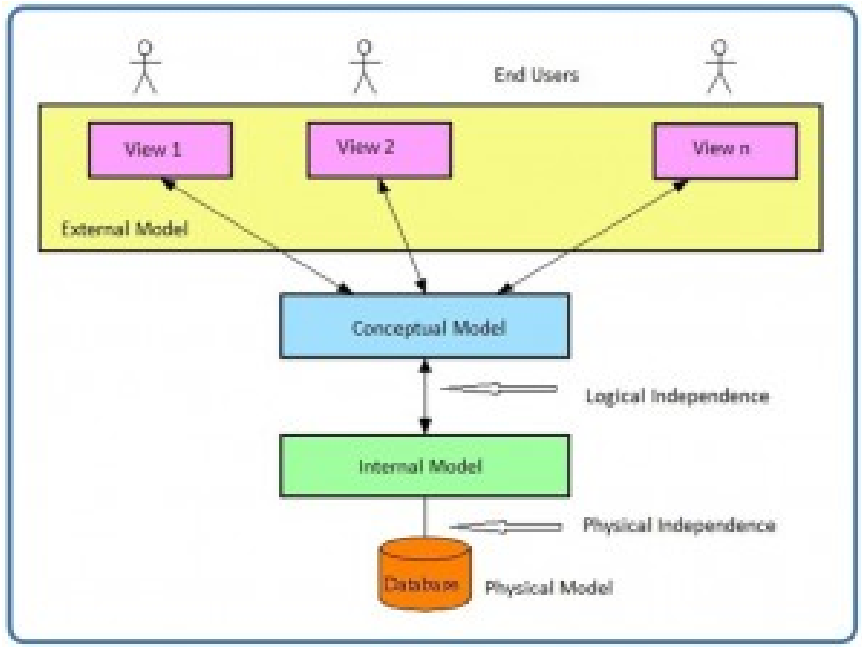
A view is a single table, that is

- **derived from other tables** and considered a
  - **virtual table**.
  - in contrast to **base tables**
- **logical view** on our data
  - may consist of virtual or base tables

Why Views?

A **View** is a logical view on the data, used to:

- hide complexity
- hide db structure,
- restrict access
- implement external model



Views

- Created through the **CREATE VIEW** statement.
- View always up-to-date
  - Responsibility of the DBMS and not the user
    - may applied dynamically (execute query at runtime)
    - may made persistent (store the results as a 'materialized view')
- **DROP VIEW** command
  - Dispose of a view

```
CREATE VIEW view_name AS
  SELECT column_name(s)
    FROM table_name
      WHERE condition
```

Create view statement

Create View Example

Example: Create a view that shows only asian countries

```
create view AsianCountries as
    select * from Country
      where Continent='Asia';
```

Operations Involving NULL

- **NULL** means **UNKNOWN**
  - Each individual NULL value considered to be different from every other NULL value.

Numerical operations:
If $x = \mathit{NULL}$ then $4 * (3 - x)/7$ is still **NULL**

Comparsions:
If $x = y = \mathit{NULL}$ then
$x ==' \mathit{Joe}'$ is **UNKNOWN**
$x == y$ is **UNKNOWN**

Comparsions involving NULL and three-valued logic

Table 5.1 Logical Connectives in Three-Valued Logic

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

NULL values

- Each individual NULL value considered to be different from every other NULL value!
- **Don't** test for equality e.g. **WHERE x = null**

Test for NULL explicitly:

- **x IS NULL**
- **x IS NOT NULL**

```
SELECT *
FROM Product
  WHERE price < 25 OR price >= 25
    OR price IS NULL
```

Aggregate Functions in SQL

Used to summarize information from

- **multiple tuples** into a
- **single-tuple summary**
- Built-in aggregate functions
  - **COUNT, SUM, MAX, MIN**, and **AVG**
    - attribute NULL values discarded

```
SELECT count(*) FROM countrylanguage
WHERE language="German";
```

```
mysql> select count(*) from countrylanguage where language="German";
+----------+
| count(*) |
+----------+
|       19 |
+----------+
1 row in set (0,01 sec)

mysql>
```

Grouping: The GROUP BY Clause

- **Partition** relation into **subsets** of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently
- **GROUP BY** clause
  - Specifies grouping attributes
- If **NULLs** exist in grouping attribute
  - Separate group for tuples with a NULL value

Grouping and Aggregate Functions Example

- Create subgroups of tuples before summarizing

```
SELECT countrycode, count(language)
FROM countrylanguage
GROUP BY countrycode;
```

```
mysql> SELECT countrycode, count(language) FROM countrylanguage GROUP BY
countrycode;
+-------------+-----------------+
| countrycode | count(language) |
+-------------+-----------------+
| ABW         |               4 |
| AFG         |               5 |
| AGO         |               9 |
| AIA         |               1 |
| ALB         |               3 |
| AND         |               4 |
| ANT         |               3 |
| ARE         |               2 |
| ARG         |               3 |
| ARM         |               2 |
```

GROUP BY and HAVING Clause Example

- **HAVING** clause
  - Provides a condition on the summary information
    - (as a WHERE clause doesn't work with aggregates)

```
SELECT countrycode, count(language) FROM countrylanguage
GROUP BY countrycode HAVING count(language) > 10;
```

```
mysql> SELECT countrycode, count(language) FROM countrylanguage
GROUP BY countrycode HAVING count(language) > 10;
+-------------+-----------------+
| countrycode | count(language) |
+-------------+-----------------+
| CAN         |              12 |
| CHN         |              12 |
| IND         |              12 |
| RUS         |              12 |
| TZA         |              11 |
| USA         |              12 |
| ZAF         |              11 |
+-------------+-----------------+
7 rows in set (0,00 sec)
```

Nested Queries

SQL is compositional. This is extremely powerful!

- Everything (**inputs / outputs**) is represented as **multisets**
  - the output of one query can be used as the input to another
    - (nesting)!
- **Nested queries**
  - **Outer query** and **nested subqueries**

Subqueries in the FROM clause

- We can use Subqueries in the **FROM** clause
  - to generate **derived tables**
  - we are **executing the outer query on the derived table**
    - derived tables need to have an alias

```
SELECT ... FROM (subquery) [AS] name ...
```

Subqueries in the FROM clause Example

```
select avg(population) from (
 (select population from country where continent='Asia')
  as AsianCountries
);
```

What's the average size of asian countries?

```
mysql> select name, population from country where continent='Asia';
+----------------------+------------+
| name                 | population |
+----------------------+------------+
| Afghanistan          |   22720000 |
| United Arab Emirates |    2441000 |
| Armenia              |    3520000 |
| Azerbaijan           |    7734000 |
| Bangladesh           |  129155000 |
| Bahrain              |     617000 |
| Brunei               |     328000 |
| Bhutan               |    2124000 |
| China                | 1277558000 |
| Cyprus               |     754700 |
| Georgia              |    4968000 |
| Hong Kong            |    6782000 |
| Indonesia            |  212107000 |
```

```
mysql> select avg(population) from (
    ->   (select population from country where continent='Asia')
    ->    as AsianCountries
    -> );
+-----------------+
| avg(population) |
+-----------------+
|   72647562.7451 |
+-----------------+
1 row in set (0,00 sec)

mysql>
```

Set membership comparsions with ANY, IN, or SOME

Compare value v with a set (or multiset) of values V

```
v IN (V)
v comparison_operator ANY (V)
v comparison_operator ALL (V)
```

Operators:

```
= > < >= <= <> !=
```

Set Membership IN

- Comparison operator **IN**
  - Compares value v with a set (or multiset) of values V
  - Evaluates to *TRUE* if v is one of the elements in V

```
select countrycode from countrylanguage
where language IN ("Pashto","German");
```

Set Membership IN

- Comparison operator **IN**
  - Compares value v with a set (or multiset) of values V
  - Evaluates to *TRUE* if v is one of the elements in V

Using the result set of a subquery:

```
select name from city where countrycode IN (
  select code from country
  where continent = "Asia"
);
```

Cities in Asia.

Set Membership ANY

- evaluate to true if the result of an inner query contains **at least one row**
  - Suppose using greater than ($>$) with SOME means greater **than at least one value**.

```
select name, population from city
where population > ANY (
  select population from country
  where continent = "Asia"
);
```

Which cities are bigger than **at least one** Asian country?

Set Membership ALL

- return **TRUE** if the comparison is **TRUE for ALL** of the values in the result set.

```
SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);
```

Example: Which countries are bigger than all european countries?

```
select name from country
where population > all(
  select population from country
    where continent="Europe"
);
```

Test for Empty Sets - (NOT) EXISTS

- **EXISTS** or **NOT EXISTS** can test if a subquery has a result

```
select name from country
where exists (
  select * from City where
    City.CountryCode = Country.Code
    and
    City.Population > 5000000
)
```

Ex: Which countries have cities bigger than 5000000?

Tables as Sets in SQL

- Set operations in SQL work exactly like in relational algebra
  - $\cup$ is **UNION**
  - $\cap$ is **INTERSECT**
  - $-$ is **EXCEPT**
- Type compatibility is needed for these operations to be valid

```
(SELECT name FROM country)
  UNION
(SELECT name FROM city)
```

MySQL does not support **intersect**, **minus** and **except**

Corresponding multiset operations

- Set operations remove duplicate rows from result set
- Corresponding multiset operations
  - UNION ALL, EXCEPT ALL, INTERSECT ALL
- **ALL** indicates Multiset operations

Example

- The **UNION** operator removes duplicate rows from the final result set.
- The **UNION ALL** operator does not remove duplicate rows from the final result set.

JOIN

SQL joins are used to combine rows from two or more tables.

- A join between tables returns all
  - **unique combinations** of their tuples
    - which meet some specified **join condition**.
- in RA: $\mathcal{R}_1 \bowtie_{A\theta B} \mathcal{R}_2$

```
SELECT (attr_list)
  FROM (table_list)
    WHERE (join_condition)
      AND (more_conditions);
```

JOIN Example

- **JOIN**, also called **INNER JOIN**
- the **joined table** is specified in the **FROM** clause
- Default type of join in a joined table

```
Product(PName, Price, Category, Manufacturer)
Company(CName, StockPrice, Country)

SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
   AND Country="Japan"
   AND Price <= 200;
```

Ex: Find all products under $200 manufactured in Japan, return their names and prices.

INNER JOIN

Several equivalent ways to write ..

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
AND Country="Japan" AND Price <= 200
```

```
SELECT PName, Price FROM
Product INNER JOIN Company ON Manufacturer = Cname
WHERE  Price <= 200
AND Country ="Japan";
```

```
SELECT PName, Price
FROM Product JOIN Company ON Manufacturer = Cname
WHERE  Price <= 200
AND Country ="Japan";
```

NATURAL JOIN

- **NATURAL JOIN** on two relations R and S
- No join condition specified but
- **implicit** EQUIJOIN **condition**
  - for each pair of attributes
    - with **same name** from R and S

NATURAL JOIN:

```
Product(product_id, name, price)
Purchase(purchase_id, customer_id, product_id)

SELECT count(name) from Purchase NATURAL JOIN Product;
```

expressed as INNER JOIN:

```
SELECT count(name) from Purchase, Product
WHERE Puchchase.product_id = Product.product_id ;
```

Outer Joins

- **left [outer] join**
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table
- **right [outer] join**
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of left table
- **full [outer] join**
  - returns rows when there is a match in one of the tables
    - *not supported in MySQL*

Outer joins introduce **NULL** values in our result relation

Outer Joins and null

- One of the main uses of outer joins is to lookup NULL values in the joined table
- the **is null** condition checks for the null value.

For which countries are languages unknown?

```
SELECT name FROM country
LEFT JOIN countrylanguage ON code = countrycode
WHERE language IS NULL;
```

```
mysql> SELECT name FROM country  LEFT JOIN countrylanguage ON code = countrycode
 WHERE  language IS NULL;
+-----------------------------------------+
| name                                    |
+-----------------------------------------+
| Antarctica                              |
| French Southern territories             |
| Bouvet Island                           |
| Heard Island and McDonald Islands       |
| British Indian Ocean Territory          |
| South Georgia and the South Sandwich Islands |
+-----------------------------------------+
6 rows in set (0,00 sec)
```

Outer Join Alternative Example

For which countries are languages unknown?

```
SELECT name FROM country
LEFT JOIN countrylanguage ON code = countrycode
WHERE language IS NULL;
```

Alternatively we can use **exists** and a **subquery**.

```
SELECT name FROM country WHERE NOT EXISTS
(
  SELECT * FROM countrylanguage
  WHERE country.code = countrylanguage.countrycode
);
```

```
+-----------------------------------------+
| name                                    |
+-----------------------------------------+
| Antarctica                              |
| French Southern territories             |
| Bouvet Island                           |
| Heard Island and McDonald Islands       |
| British Indian Ocean Territory          |
| South Georgia and the South Sandwich Islands |
+-----------------------------------------+
6 rows in set (0,01 sec)

mysql> select name from country where NOT EXISTS (select * from countrylanguage
where country.code = countrylanguage.countrycode);
```

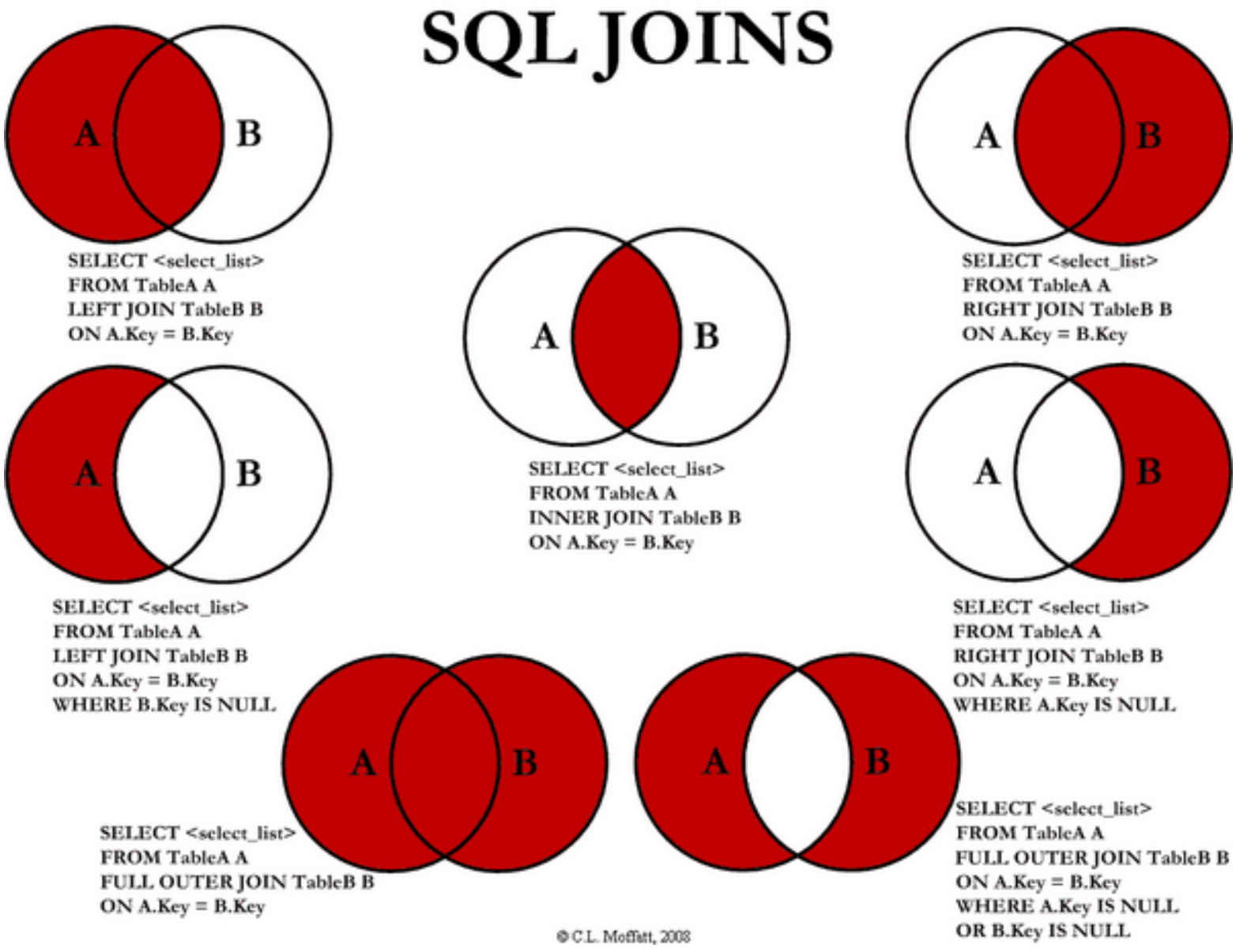
Resolve Variable Ambiguity in Multi-Table

equivalent ways to resolve variable ambiguity:

```
SELECT DISTINCT Person.name, Person.address
FROM            Person, Company
WHERE           Person.worksfor = Company.name
```

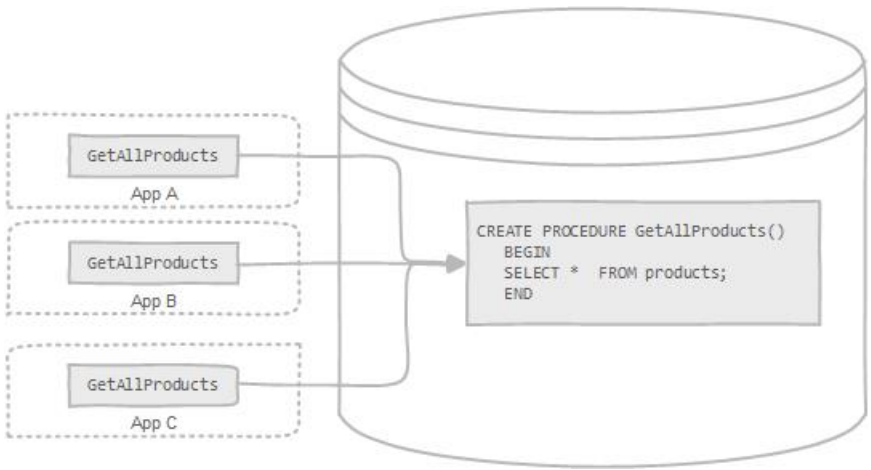
```
SELECT DISTINCT p.name, p.address
FROM            Person p, Company c
WHERE           p.worksfor = c.name
```

JOINS Overview



Stored Procedure

- a **segment of declarative SQL** statements
- **stored inside the database**, that can be
  - **invoked by triggers**, other
  - **stored procedures**, and
  - **external applications** such as Java, Python, PHP, etc.



CREATE PROCEDURE

```
DELIMITER //
CREATE PROCEDURE
   GetCountriesByContinent(IN continentName VARCHAR(255))
      BEGIN
         SELECT name, capital, population
         FROM country
         WHERE continent = continentName;
      END //
DELIMITER ;
```

more commands:

```
SHOW PROCEDURE STATUS;
SHOW PROCEDURE CODE proc_name;
DROP PROCEDURE proc_name;
```

CALL PROCEDURE

```
CALL GetCountriesByContinent("Europe");
```

```
mysql> CALL GetCountriesByContinent("Europe");
+------------------------------+---------+------------+
| name                         | capital | population |
+------------------------------+---------+------------+
| Albania                      |      34 |    3401200 |
| Andorra                      |      55 |      78000 |
| Austria                      |    1523 |    8091800 |
| Belgium                      |     179 |   10239000 |
| Bulgaria                     |     539 |    8190900 |
| Bosnia and Herzegovina       |     201 |    3972000 |
| Belarus                      |    3520 |   10236000 |
| Switzerland                  |    3248 |    7160400 |
```

Stored Procedures - IN,OUT, or INOUT Parameters

IN

- default mode
- caller has to pass an argument to the stored procedure
- the value of an IN parameter may NOT be changed

OUT

- initial value NOT available to SP
- the value of an OUT parameter can be changed
- new value is passed back to the caller

INOUT

- combination of IN and OUT parameters. It means that the calling program may pass the argument, and the
- SP can modify the INOUT parameter and pass the new value back to the caller

Stored Procedures Pro & Con

Advantages

- Share logic with other applications. Stored procedures encapsulate functionality
  - cohaerent, reusable and transparent to any applications
- To enhance modeling power provided by views
  - Isolate users from data tables.
- To reduce data transfer and communication
  - multiple calls can be melded into one.

Disadvantages

- may increase memory usage
  - more work on the server, less on the client
- may not be suited for complicated business logic
- rather difficult to debug
- additional layer of development and maintenance
  - may repeating application logic

Active Database Concepts

- **active database**
  - implements an **event-driven architecture**
    - often in the form of **Event-condition-action rules (ECA)**
- **ECA** rules
  - may respond to conditions both inside and outside the database.
  - Possible uses
    - security monitoring
    - alerting
    - statistics gathering
    - data integrity

Triggers

Trigger

- **stored procedure** that
- **automatically** executes when
- a specified **condition** occurs.

Event-Condition-Action (ECA) Model

Triggers follow an Event-condition-action (ECA) model

- **Event**:
  - Database modification
    - E.g., insert, delete, update
- **Condition**:
  - Any true/false expression
    - Optional: If no condition is specified then condition is always true
- **Action**:
  - Sequence of SQL statements that will be automatically executed

Trigger Example

Use case:
When a new employees is added to a department,
modify the *Total_sal* of the Department to include the new employees salary.

```
CREATE TRIGGER Total_sal1
  AFTER INSERT ON Employee
  FOR EACH ROW
  WHEN (NEW.Dno is NOT NULL)
    UPDATE DEPARTMENT
    SET Total_sal = Total_sal + NEW.Salary
    WHERE Dno = NEW.Dno;
```

CREATE or ALTER TRIGGER

- CREATE TRIGGER < *name* >
  - Creates a trigger
- ALTER TRIGGER < *name* >
  - Alters a trigger *assumingoneexists*
- CREATE OR ALTER TRIGGER < *name* >
  - Creates a trigger if one does not exist
  - Alters a trigger if one does exist
    - Works in both cases, whether a trigger exists or not

Conditions

- AFTER
  - Executes after the event
- BEFORE
  - Executes before the event
- INSTEAD OF
  - Executes **instead of** the event
    - Event does not execute in this case!
    - E.g., used for modifying views

```
...
  AFTER INSERT ON Employee
  ...
```

Row-Level versus Statement-level

- Triggers can be
  - **Row-level**
  - Executed separately for each affected row
    - FOR EACH ROW specifies a row-level trigger
- **Statement-level**
  - Default
  - Execute once for the SQL statement
    - (when FOR EACH ROW is not specified)

```
..
  FOR EACH ROW
   ...
```

Condition

- Any **true/false** condition to control whether a trigger is activated on not
  - Absence of condition means that the trigger will always execute for the even
    - Otherwise, condition is evaluated
- before the event for BEFORE trigger
- after the event for AFTER trigger

```
...
  WHEN (NEW.Dno is NOT NULL)
   ...
```

Action

- Generalized Model (cont.)
- Action can be

- One SQL statement
- A sequence of SQL statements enclosed between a BEGIN and an END

- Action specifies the relevant modifications

```
...
    UPDATE DEPARTMENT
    SET  Total_sal = Total_sal + NEW.Salary
    WHERE Dno = NEW.Dno;
    ...
```

Active Database Concepts and Triggers

Potential Applications for Active Databases

- Notification
  - Automatic notification when certain condition occurs
- Enforcing integrity constraints
  - Triggers are smarter and more powerful than constraints
- Maintenance of derived data
  - Automatically update derived data and avoid anomalies due to redundancy