# KE UNIT 3 DATA WAREHOUSING FOR BUSINESS ANALYTICS

## DAY 2

Dr TIAN Jing

tianjing@nus.edu.sg

## Data management

Database ≠ Database Management System

# Data management

- How can we **collect and store** large amounts of data?
  - Build tools and data structures to efficiently index and serve data
- How can we **efficiently query** data?
  - Compile high-level declarative queries into efficient low-level plans
- How can we **safely update** data?
  - Manage concurrent access to state as it is read and written

# Module objective and outline

- Objective
  - Discuss fundamentals of data management, how to use SQL to query databases for data analytics
  - Not study how to be a database administrator; and how to optimize data query commands
- Outline
  - Relational model
  - SQL query: Data definition language and data manipulation language

# Module references

- CS145 Introduction to Databases, Stanford University, http://web.stanford.edu/class/cs145/
- Michael Cafarella, Introduction to SQL and the Relational Model, Data Boot Camp, 2014, http://ibug-um.github.io/2014-summer-camp/
- Online SQL hands-on exercise, http://sqlfiddle.com/

# Relational database management systems (RDBMS)

- A schema / database is a collection of logical structures of data or objects
- Some types of objects:
  - Tables
    - stores data / records
  - Indexes
- Mapping from ERD
  - Each Entity -> one relational table
  - Each Attributes-> one column
  - Identifier (key)-> primary key

# Relational database table

- All data is stored in tables
  - Organized into rows and columns.
  - Example: employee table.

| Emp_No | Emp_Name | IC_No | Dept_No |
|---|---|---|---|
| 179 | Chang | P28493 | 7 |
| 857 | Robinson | S95843 | 4 |
| 342 | Bill | T04842 | 7 |

Rows

Columns

- All relationship information is presented as data values in tables
  - No ordering

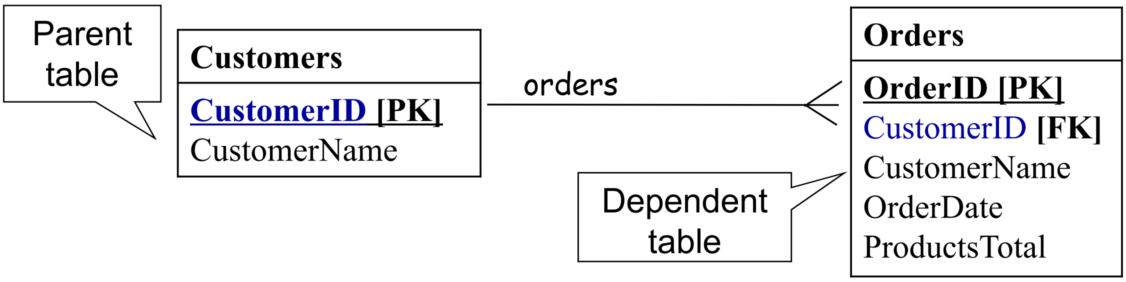
# Data type

- Each column in a relational database has a datatype
- Common datatypes:

| Data Type | |
|---|---|
| **CHAR(size)** | Fixed character data |
| **VARCHAR(size)** | Variable length character |
| **INT** | Whole numbers |
| **DECIMAL(p,s)** | Number with precision and scale<br>Precision: number of digits in a number.<br>Scale: number of digits to the right of the decimal point. For example, the number 123.45 has a precision of 5 and a scale of 2. |
| **DATE** | Date field |

# Primary key (PK) and foreign key (FK)

- A primary key is unique identifiers of the table. At most 1 primary key can be defined on a table.
- A foreign key enforces referential integrity. Foreign key attribute must correspond to an existing primary key value in the parent table (unless the foreign key value is null)



Parent table

**Customers**

**CustomerID [PK]**

CustomerName

orders

**Orders**

**OrderID [PK]**

CustomerID **[FK]**

CustomerName

OrderDate

ProductsTotal

Dependent table

| CustomerID [PK] | Customer Name |
|---|---|
| S009 | Lynn Wang |
| S010 | Suzan Tan |

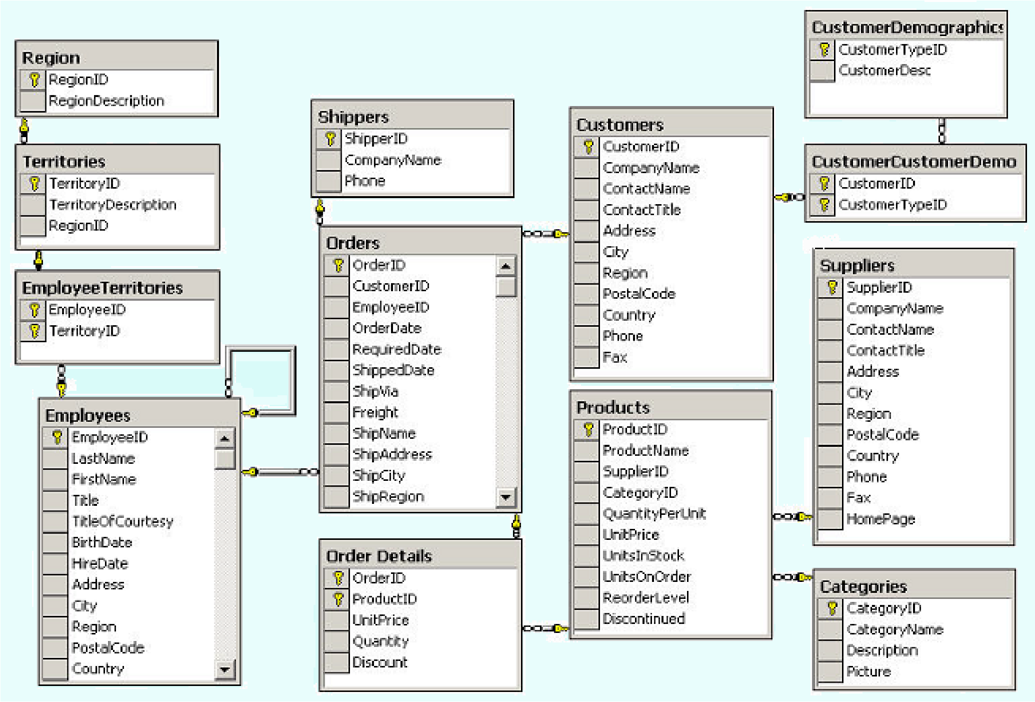
| Order ID [PK] | CustomerID [FK] | Order Date | Products Total |
|---|---|---|---|
| A1091 | S009 | 21/7/2011 | 3 |
| A1092 | S010 | 12/1/2011 | 2 |

# Database ERD

- In RDBMS, databases are illustrated using an ERD (entity relationship diagram)
  - Example of a E-R Model is shown in the next slide.
  - This diagram illustrates Northwind Database provided along with MS SQL Server.
  - This system typically depicts a trading system consisting Sales and Purchases.
  - Each entity is represent by a table : eg. Customers table, Employees table, etc. Each row in the table is called the entity data.
  - The lines joining the tables are the relationships.
  - There are a number of relationships between entities in an ERD.
    - 1 to 1 relationship
    - 1 to many relationship
    - Also entity related to itself ( Employees)

# Example: Northwind ERD

## Northwind Entity Relationship Diagram



**Region**
- RegionID
- RegionDescription

**Territories**
- TerritoryID
- TerritoryDescription
- RegionID

**EmployeeTerritories**
- EmployeeID
- TerritoryID

**Employees**
- EmployeeID
- LastName
- FirstName
- Title
- TitleOfCourtesy
- BirthDate
- HireDate
- Address
- City
- Region
- PostalCode
- Country

**Shippers**
- ShipperID
- CompanyName
- Phone

**Orders**
- OrderID
- CustomerID
- EmployeeID
- OrderDate
- RequiredDate
- ShippedDate
- ShipVia
- Freight
- ShipName
- ShipAddress
- ShipCity
- ShipRegion

**Order Details**
- OrderID
- ProductID
- UnitPrice
- Quantity
- Discount

**Customers**
- CustomerID
- CompanyName
- ContactName
- ContactTitle
- Address
- City
- Region
- PostalCode
- Country
- Phone
- Fax

**Products**
- ProductID
- ProductName
- SupplierID
- CategoryID
- QuantityPerUnit
- UnitPrice
- UnitsInStock
- UnitsOnOrder
- ReorderLevel
- Discontinued

**CustomerDemographics**
- CustomerTypeID
- CustomerDesc

**CustomerCustomerDemo**
- CustomerID
- CustomerTypeID

**Suppliers**
- SupplierID
- CompanyName
- ContactName
- ContactTitle
- Address
- City
- Region
- PostalCode
- Country
- Phone
- Fax
- HomePage

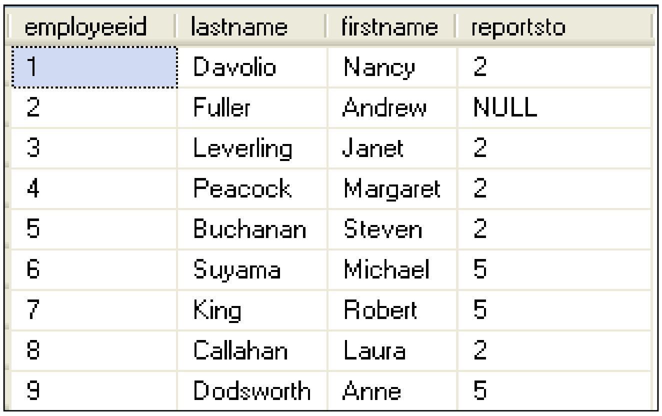
**Categories**
- CategoryID
- CategoryName
- Description
- Picture

# Example: Northwind ERD

- Interpretation of the Northwind ERD is as follows :
  - Notice that there is an employee id field under the Orders and Employees table.
  - The employee id in  Employees table is the primary key while the same id in the Orders table is the foreign key. In other words, for every row in the orders table, that row's employee id must be found in the Employee table.
  - The relationship between the Employee and Orders table is a 1 to many relationship. This means that for each employee id in the Employee table, there could be repetition of the same id in the Orders table.
  - Note that the (One to Many) relationship is determined by the a row of the Employee table with respect to many rows of the Orders table having the same EmployeeID

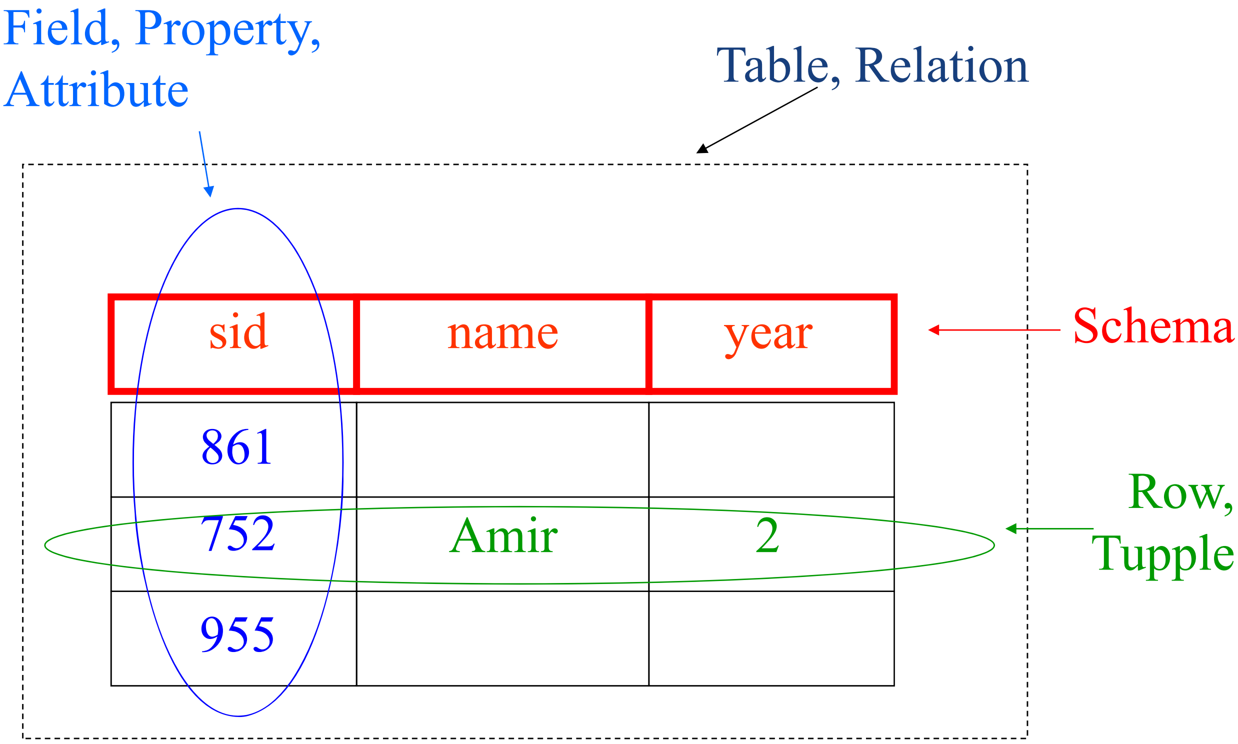
# Example: Northwind ERD

- Interpretation of the Northwind ERD (cont'd):
  - Employees Table has a self referencing relationship. The table has a "report to" column (apart from the employeeID) which requires the employee number to be insert
  - Each employee reports to another employee (I.e., the Boss). Hence each row in the employee table (ie each employee) has a column that requires another EmployeeID for "reporting to" relationship to be established.



| employeeid | lastname | firstname | reportsto |
|---|---|---|---|
| 1 | Davolio | Nancy | 2 |
| 2 | Fuller | Andrew | NULL |
| 3 | Leverling | Janet | 2 |
| 4 | Peacock | Margaret | 2 |
| 5 | Buchanan | Steven | 2 |
| 6 | Suyama | Michael | 5 |
| 7 | King | Robert | 5 |
| 8 | Callahan | Laura | 2 |
| 9 | Dodsworth | Anne | 5 |

# Summary: Relational model



Field, Property, Attribute

Table, Relation

Schema

Row, Tupple

| sid | name | year |
|---|---|---|
| 861 | | |
| 752 | Amir | 2 |
| 955 | | |

# History of SQL

- Stands for *Structured Query Language*
- Developed at IBM by Donald D. Chamberlin and Raymond F. Boyce
  - [Chamberlin, Boyce: SEQUEL: A Structured English Query Language. SIGMOD Workshop, Vol. 1 1974: 249-264]
- Originally called SEQUEL
  - Now written SQL but still pronounced “SEQUEL”
- Standardized as ANSI (1986), ISO (1987)

# CRUD: Create

- Create the Athlete relation
  - Type constraint enforced when tuples added or modified

```
CREATE TABLE Athlete
(aid INTEGER,
 name CHAR(30),
 country CHAR(20),
 sport CHAR(20));
```

- Create the Olympics relation

```
CREATE TABLE Olympics
(oid INTEGER,
 year INTEGER,
 city CHAR(20));
```

- Create the Compete relation

```
CREATE TABLE Compete
(aid INTEGER,
 oid INTEGER);
```

# CRUD: Read

- Find all athletes from USA:

```
SELECT *
FROM Athlete;
```

| AID | Name | Country | Sport |
|---|---|---|---|
| 1 | Mary Lou Retton | USA | Gymnastics |
| 2 | Jackie Joyner-Kersee | USA | Track |
| 3 | Michael Phelps | USA | Swimming |

# CRUD: Update

- Can insert a single tuple using

```
INSERT INTO  Athlete (aid, name, country, sport)
VALUES  (4, ‘Johann Koss’ , ‘Norway’ , ‘Speedskating’ );
```

## CRUD: Delete

- Can delete all tuples satisfying some condition (e.g., name = Smith)

```
DELETE
FROM Athlete A
WHERE A.name = ‘Smith’;
```

- Destroys the relation Olympics.

```
DROP TABLE Olympics;
```

## Table used in example

**Students**

| **name** | **gpa** | **age** | **dept** | **gender** |
|---|---|---|---|---|
| Sergey Brin | 4 | 40 | CS | M |
| Danah Boyd | 4 | 35 | CS | F |
| Bill Gates | 1 | 60 | CS | M |
| Hillary Mason | 4 | 35 | DATASCI | F |
| Mike Olson | 4 | 50 | CS | M |
| Mark Zuckerberg | 4 | 30 | CS | M |
| Cheryl Sandberg | 4 | 47 | BUSINESS | F |
| Susan Wojcicki | 4 | 46 | BUSINESS | F |
| Marissa Meyer | 4 | 45 | BUSINESS | F |

DEMO: http://sqlfiddle.com/#!18/b2a3b/1

# Basic SELECT (1)

-
  ```
  SELECT S.name, S.gpa
    FROM students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

- Produce all tuples in the table that satisfy the predicate, output the expressions in the SELECT list
- Expression can be a column reference, or an arithmetic expression over column references.
- The WHERE clause allows to build arbitrary propositional logic over built-in predicates over attributes
  - Logical operators: AND, OR, NOT
  - Comparisons on numbers/strings (lexicographic): =, !=, >, <, >=, <=,
  - Membership in lists: IN, NOT IN

# Basic SELECT (2)

-
  ```
  SELECT S.name, S.gpa
    FROM students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

| name | gpa |
|---|---|
| Sergey Brin | 4 |
| Danah Boyd | 4 |
| Bill Gates | 1 |
| Mike Olson | 4 |
| Mark Zuckerberg | 4 |

- *Try changing the WHERE clause,*
  *e.g. 'WHERE S.gpa < 4.0'*
- *Try changing the SELECT clause,*
  *e.g. 'SELECT S.sname, S.gender' (error!)*

DEMO: http://sqlfiddle.com/#!18/b2a3b/1

# SELECT DISTINCT

-
  ```
  SELECT DISTINCT S.name, S.gpa
    FROM students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

- DISTINCT specifies removal of duplicate rows before output

# Renaming

-
  ```
  SELECT S.name, S.gpa, S.age*2 AS a2
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

  - Attributes can be *renamed*
  - Attributes can be invented as *functions of other attributes*

| name | gpa | a2 |
|---|---|---|
| Sergey Brin | 4 | 80 |
| Danah Boyd | 4 | 70 |
| Bill Gates | 1 | 120 |
| Mike Olson | 4 | 100 |
| Mark Zuckerberg | 4 | 60 |

DEMO: http://sqlfiddle.com/#!18/b2a3b/19

# ORDER BY

```
• SELECT S.name, S.gpa, S.age*2 AS a2
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
   ORDER BY S.gpa DESC, S.name ASC, a2
```

Ascending order by default, but can be overriden

- DESC flag for descending, ASC for ascending
- Can mix and match, lexicographically

| name | gpa | a2 |
|---|---|---|
| Danah Boyd | 4 | 70 |
| Mark Zuckerberg | 4 | 60 |
| Mike Olson | 4 | 100 |
| Sergey Brin | 4 | 80 |
| Bill Gates | 1 | 120 |

DEMO: http://sqlfiddle.com/#!18/b2a3b/7

# Aggregates

```
• SELECT [DISTINCT] AVG(S.gpa)
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>] ;
```

Result is 3.4

Common aggregate functions
SUM(C) – sum over all numbers in C
COUNT(C) - number of rows in C
AVG(C) – SUM(C)/COUNT(C)
MAX(C) – largest value
MIN(C) – smallest value

DEMO: http://sqlfiddle.com/#!18/b2a3b/14

# GROUP BY

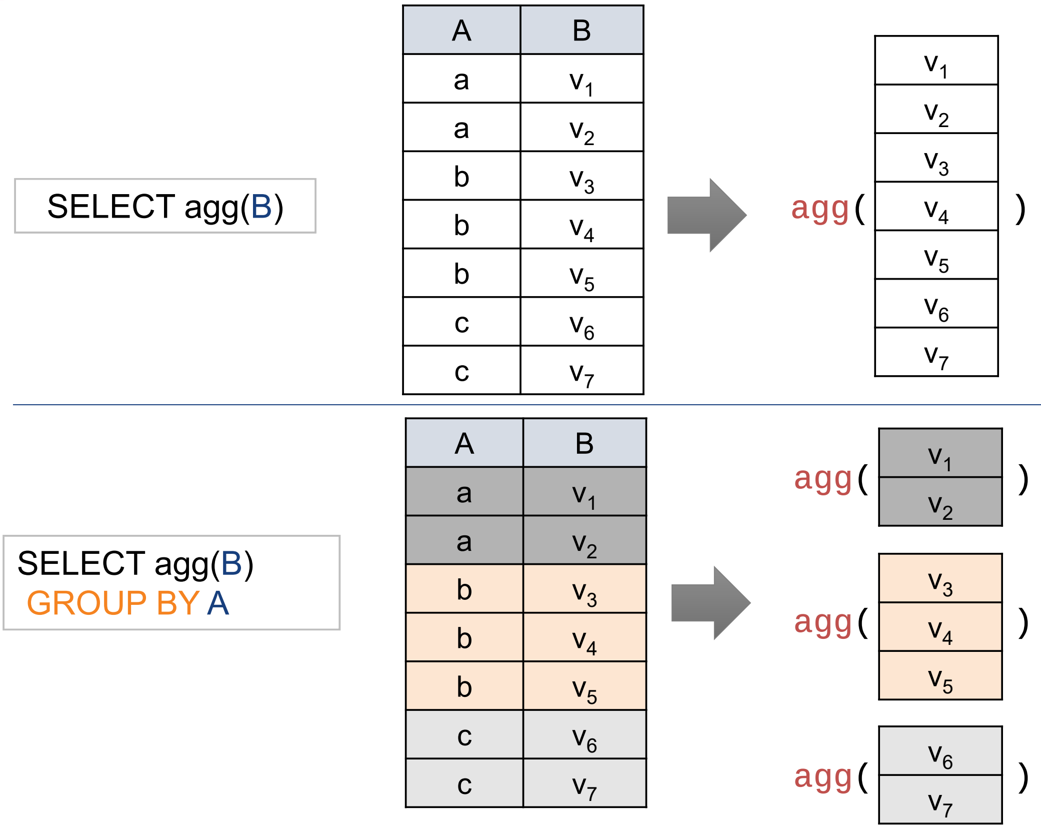
```
SELECT [DISTINCT] AVG(S.gpa), S.dept
  FROM Students S
[WHERE <predicate>]
 GROUP BY S.dept
 [HAVING <predicate>]
[ORDER BY <column list>] ;
```

- Partition table into groups with same GROUP BY column values
- Produce an aggregate result per group
- Note: can put grouping columns in SELECT list

| | dept |
|---|---|
| 4 | BUSINESS |
| 3.4 | CS |
| 4 | DATASCI |

DEMO: http://sqlfiddle.com/#!18/b2a3b/16

# Grouping idea



SELECT agg(B)

| A | B |
|---|---|
| a | $v_1$ |
| a | $v_2$ |
| b | $v_3$ |
| b | $v_4$ |
| b | $v_5$ |
| c | $v_6$ |
| c | $v_7$ |

agg( $v_1$, $v_2$, $v_3$, $v_4$, $v_5$, $v_6$, $v_7$ )

SELECT agg(B)
GROUP BY A

| A | B |
|---|---|
| a | $v_1$ |
| a | $v_2$ |
| b | $v_3$ |
| b | $v_4$ |
| b | $v_5$ |
| c | $v_6$ |
| c | $v_7$ |

agg( $v_1$, $v_2$ )

agg( $v_3$, $v_4$, $v_5$ )

agg( $v_6$, $v_7$ )

# HAVING

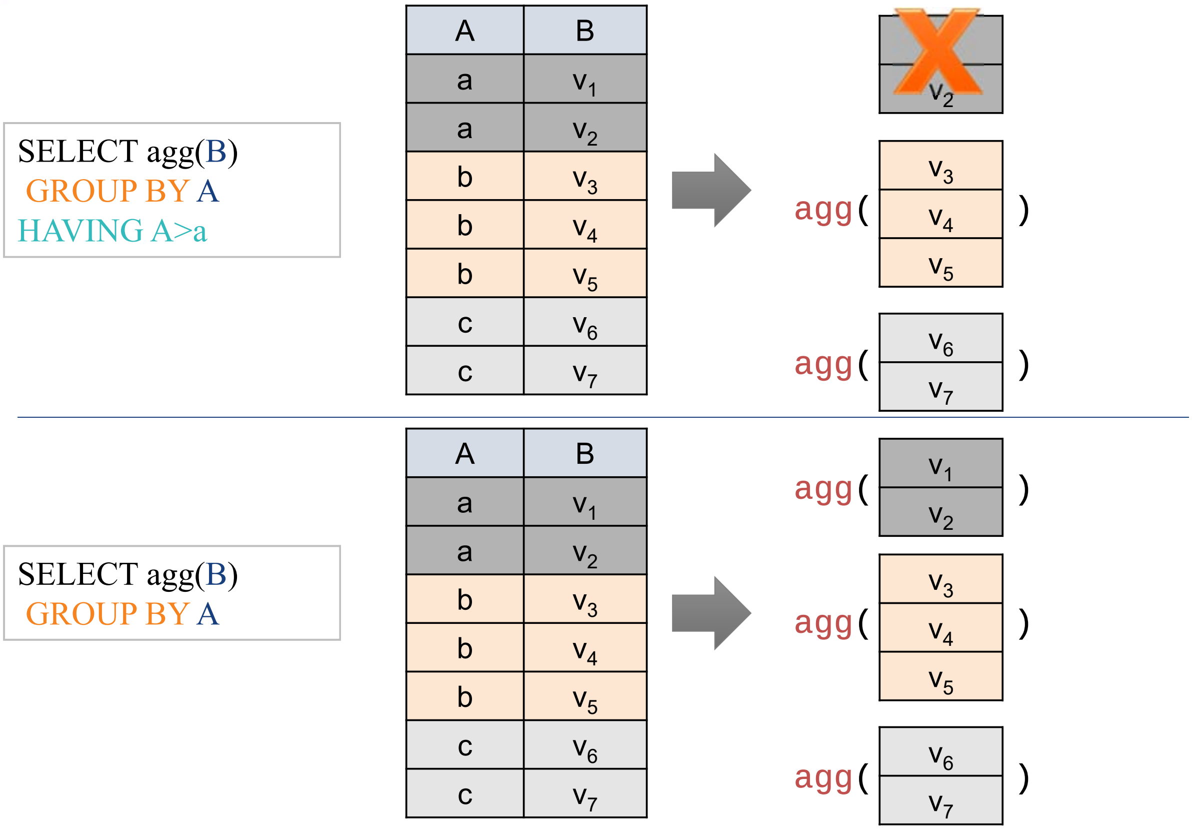
```
SELECT [DISTINCT] AVG(S.gpa), S.dept
  FROM Students S
[WHERE <predicate>]
 GROUP BY S.dept
  HAVING COUNT(*) > 2
[ORDER BY <column list>] ;
```

- The HAVING predicate filters groups
- HAVING is applied *after* grouping and aggregation
  - Hence can contain anything that could go in the SELECT list, i.e. aggs or GROUP BY columns
- HAVING can only be used in aggregate queries
- It's an optional clause

| | dept |
|---|---|
| 4 | BUSINESS |
| 3.4 | CS |

DEMO: http://sqlfiddle.com/#!18/b2a3b/17

# Group selection



```
SELECT agg(B)
 GROUP BY A
HAVING A>a
```

| A | B |
|---|---|
| a | $v_1$ |
| a | $v_2$ |
| b | $v_3$ |
| b | $v_4$ |
| b | $v_5$ |
| c | $v_6$ |
| c | $v_7$ |

$v_2$

agg( $v_3$ $v_4$ $v_5$ )

agg( $v_6$ $v_7$ )

```
SELECT agg(B)
 GROUP BY A
```

| A | B |
|---|---|
| a | $v_1$ |
| a | $v_2$ |
| b | $v_3$ |
| b | $v_4$ |
| b | $v_5$ |
| c | $v_6$ |
| c | $v_7$ |

agg( $v_1$ $v_2$ )

agg( $v_3$ $v_4$ $v_5$ )

agg( $v_6$ $v_7$ )

# Putting it all together

```
• SELECT S.dept, AVG(S.gpa), COUNT(*)
   FROM Students S
  WHERE S.gender = 'F'
  GROUP BY S.dept
 HAVING COUNT(*) >= 2
  ORDER BY S.dept;
```

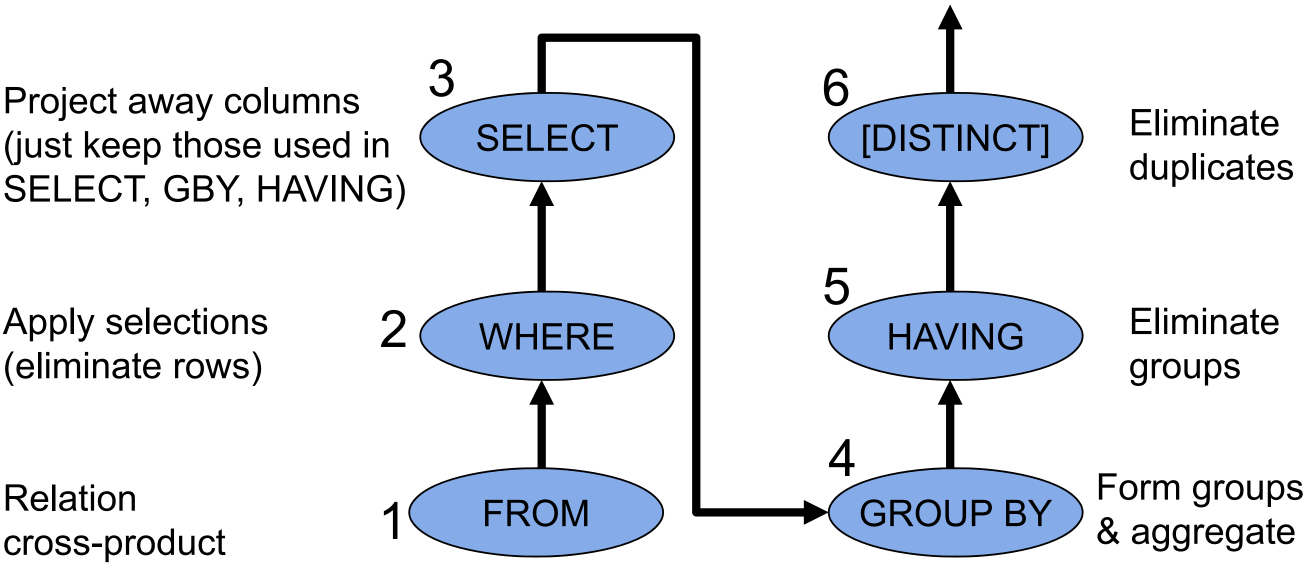
| dept | | |
|---|---|---|
| BUSINESS | 4 | 3 |

| name | gpa | age | dept | gender |
|---|---|---|---|---|
| Sergey Brin | 4 | 40 | CS | M |
| Danah Boyd | 4 | 35 | CS | F |
| Bill Gates | 1 | 60 | CS | M |
| Hillary Mason | 4 | 35 | DATASCI | F |
| Mike Olson | 4 | 50 | CS | M |
| Mark Zuckerberg | 4 | 30 | CS | M |
| Cheryl Sandberg | 4 | 47 | BUSINESS | F |
| Susan Wojcicki | 4 | 46 | BUSINESS | F |
| Marissa Meyer | 4 | 45 | BUSINESS | F |

DEMO: http://sqlfiddle.com/#!18/b2a3b/18

# Brief summary

SELECT [DISTINCT] *target-list*
FROM *relation-list*
WHERE *qualification*
GROUP BY *grouping-list*
HAVING *group-qualification*



Project away columns (just keep those used in SELECT, GBY, HAVING)

Apply selections (eliminate rows)

Relation cross-product

Eliminate duplicates

Eliminate groups

Form groups & aggregate

# Query from multiple tables

- SELECT [DISTINCT] <column expression list>
  FROM <table1 [AS t1], ... , tableN [AS tn]>
  [WHERE <predicate>]
  [GROUP BY <column list>
  [HAVING <predicate>] ]
  [ORDER BY <column list>];

1. FROM : compute *cross product* of tables.
2. WHERE : Check conditions, discard tuples that fail.
3. SELECT : Specify desired fields in output.
4. DISTINCT (optional) : eliminate duplicate rows.

# Cross product of two tables

All pairs of tuples, concatenated

**U**

| uid | uname | followers | age |
|---|---|---|---|
| 1 | Anurag | 2224 | 22 |
| 2 | Ryan | 12 | 39 |
| 3 | Rolando | 2 | 27 |
| 4 | Valerie | 5034 | 19 |

**P**

| uid | cid | time |
|---|---|---|
| 1 | 102 | 2018-01-16 09:00 |
| 2 | 102 | 2018-01-17 09:02 |
| 1 | 101 | 2018-01-22 23:00 |

| U.uid | U.uname | U.followers | U.age | P.uid | P.cid | P.time |
|---|---|---|---|---|---|---|
| 1 | Anurag | 2224 | 22 | 1 | 102 | 2018-01-16 09:00 |
| 1 | Anurag | 2224 | 22 | 2 | 102 | 2018-01-17 09:02 |
| 1 | Anurag | 2224 | 22 | 1 | 101 | 2018-01-22 23:00 |
| 2 | Ryan | 12 | 39 | 1 | 102 | 2018-01-16 09:00 |
| … | … | … | … | … | … | ... |

# Tables used in example

Users

| uid | uname | followers | age |
|---|---|---|---|
| 1 | Anurag | 2224 | 22 |
| 2 | Ryan | 12 | 39 |
| 3 | Rolando | 2 | 27 |
| 4 | Valerie | 5034 | 19 |

Channels

| cid | cname | bkcolor |
|---|---|---|
| 101 | homeworks | red |
| 102 | midterms | magenta |
| 103 | lectures | red |

Posts

| uid | cid | time |
|---|---|---|
| 1 | 101 | 2018-01-22 23:00:00 |
| 1 | 102 | 2018-01-16 09:00:00 |
| 2 | 102 | 2018-01-17 09:02:00 |

# Find users who have posted

```
SELECT Users.uid, uname
  FROM Users, Posts
 WHERE Users.uid = Posts.uid
```

| uid | uname |
|---|---|
| 1 | Anurag |
| 1 | Anurag |
| 2 | Ryan |

```
SELECT U.uid, uname
  FROM Users AS U, Posts AS P
 WHERE U.uid=P.uid
```

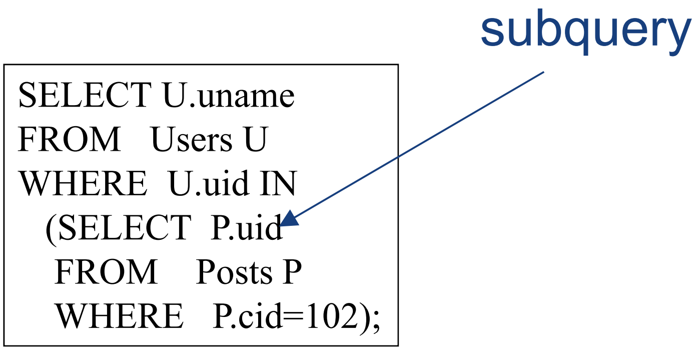
DEMO: http://sqlfiddle.com/#!18/ed664/1

# SQL query: Nesting

- Nesting: one query is nested in another query as a relation/value component
- The nested query is called a subquery
- Where are we nesting?
  - SELECT
    - Select a value from a subquery
  - FROM
    - Use a subquery instead of an existing relation
  - WHERE
    - Conditions phrased via subqueries

# Nested queries: IN

*Names of users who’ve posted on channel #102:*



subquery

```
SELECT U.uname
FROM   Users U
WHERE  U.uid IN
  (SELECT  P.uid
   FROM    Posts P
   WHERE   P.cid=102);
```

```
SELECT  P.uid
   FROM    Posts P
   WHERE   P.cid=102;

SELECT U.uname
FROM   Users U
WHERE  U.uid IN
   ('1', '2');
```

| uname |
|---|
| Anurag |
| Ryan |

| uid |
|---|
| 1 |
| 2 |

| uname |
|---|
| Anurag |
| Ryan |

DEMO: http://sqlfiddle.com/#!18/a7ca4/7

# Nested queries: NOT IN

*Names of users who've **not** posted on channel #103:*

```
SELECT  U.uname
FROM    Users U
WHERE   U.uid NOT IN
   (SELECT  P.uid
    FROM    Posts P
    WHERE   P.cid=103)
```

| uname |
|---|
| Anurag |
| Ryan |
| Rolando |
| Valerie |

DEMO: http://sqlfiddle.com/#!18/a7ca4/9

# Join tables

- INNER is default
- Inner join is akin to what we've learned so far, just with different syntax.

```
SELECT (column_list)
FROM  table_name
  [INNER | {LEFT |RIGHT | FULL }  {OUTER}] JOIN table_name
    ON qualification_list
WHERE …
```

# Inner Joins

```
 SELECT u.uid, u.uname, p.uid
 FROM Users u, Posts p
WHERE u.uid = p.uid
  AND u.age > 20;
SELECT u.uid, u.uname, p.uid
 FROM Users u INNER JOIN Posts p
   ON u.uid = p.uid
  AND u.age > 20;
```

**Both are equivalent!**

| uid | uname | uid |
|---|---|---|
| 1 | Anurag | 1 |
| 1 | Anurag | 1 |
| 2 | Ryan | 2 |

DEMO: http://sqlfiddle.com/#!18/a7ca4/11

# Tables used in example

## Sailors

| sid | sname | rating | age |
|---|---|---|---|
| 22 | Dustin | 7 | 45 |
| 31 | Lubber | 8 | 55.5 |
| 95 | Bob | 3 | 63.5 |

## Boats

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | green |
| 103 | Santa Maria | blue |

## Reserves

| sid | bid | day |
|---|---|---|
| 22 | 101 | 10/10/96 |
| 95 | 103 | 11/12/96 |

# Left Outer Join

Returns all matched rows, and *preserves* all unmatched rows from the table on the left of the join clause
(use nulls in fields of non-matching tuples)

| sid | sname | bid |
|---|---|---|
| 22 | Dustin | 101 |
| 95 | Bob | 103 |
| 31 | Lubber | (null) |

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid;
```

DEMO: http://sqlfiddle.com/#!17/78155/2

# Right Outer Join

Returns all matched rows, and *preserves* all unmatched rows from the table on the right of the join clause
(use nulls in fields of non-matching tuples)

| sid | sname | bid |
|---|---|---|
| 22 | Dustin | 101 |
| 95 | Bob | 103 |
| 31 | Lubber | (null) |

```
SELECT s.sid, s.sname, r.bid
FROM Reserves r RIGHT OUTER JOIN Sailors s
ON r.sid = s.sid;
```

DEMO: http://sqlfiddle.com/#!17/78155/4

# Full Outer Join

Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

| sid | bid | bname |
|---|---|---|
| 22 | 101 | Nina |
| 95 | 103 | Santa Maria |
| (null) | 102 | Pinta |

```
SELECT r.sid, b.bid, b.bname
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid
```

DEMO: http://sqlfiddle.com/#!17/78155/5

# Null values

- Field values are sometimes unknown
  - SQL provides a special value NULL for such situations.
  - Every data type can be NULL
- The presence of null complicates many issues. E.g.:
  - Selection predicates (WHERE)
  - Aggregation
- NULLs also come from various joins

# Tables used in example

**Sailors**

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Popeye | 10 | 22 |
| 2 | OliveOyl | 11 | 39 |
| 3 | Garfield | 1 | 27 |
| 4 | Bob | 5 | 19 |
| 5 | SpongeBob | 11 | 2 |
| 11 | Jack Sparrow | (null) | 35 |

**Boats**

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | green |
| 103 | Santa Maria | blue |

**Reserves**

| sid | bid | day |
|---|---|---|
| 1 | 101 | 10/01/2017 |
| 1 | 102 | 9/12/2017 |
| 2 | 102 | 9/13/2017 |

# NULL in WHERE clause

- WHERE clause eliminates rows that don't evaluate to true

```
SELECT * FROM sailors WHERE rating > 8;
SELECT * FROM sailors WHERE rating > 8 AND 'True'='True';
SELECT * FROM sailors WHERE rating > 8 OR 'True'='True';
SELECT * FROM sailors WHERE NOT (rating > 8);
```

Three-valued logic:

NOT(A)

| A | ¬A |
|---|---|
| F | T |
| U | U |
| T | F |

AND(A, B)

| A ∧ B | | B | | |
|---|---|---|---|---|
| | | F | U | T |
| A | F | F | F | F |
| | U | F | U | U |
| | T | F | U | T |

OR(A, B)

| A ∨ B | | B | | |
|---|---|---|---|---|
| | | F | U | T |
| A | F | F | U | T |
| | U | U | U | T |
| | T | T | T | T |

DEMO: http://sqlfiddle.com/#!18/0f6bd/21

# NULL in WHERE clause

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Popeye | 10 | 22 |
| 2 | OliveOyl | 11 | 39 |
| 5 | SpongeBob | 11 | 2 |

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Popeye | 10 | 22 |
| 2 | OliveOyl | 11 | 39 |
| 5 | SpongeBob | 11 | 2 |

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Popeye | 10 | 22 |
| 2 | OliveOyl | 11 | 39 |
| 3 | Garfield | 1 | 27 |
| 4 | Bob | 5 | 19 |
| 5 | SpongeBob | 11 | 2 |
| 11 | Jack Sparrow | (null) | 35 |

| sid | sname | rating | age |
|---|---|---|---|
| 3 | Garfield | 1 | 27 |
| 4 | Bob | 5 | 19 |

# NULL and aggregation

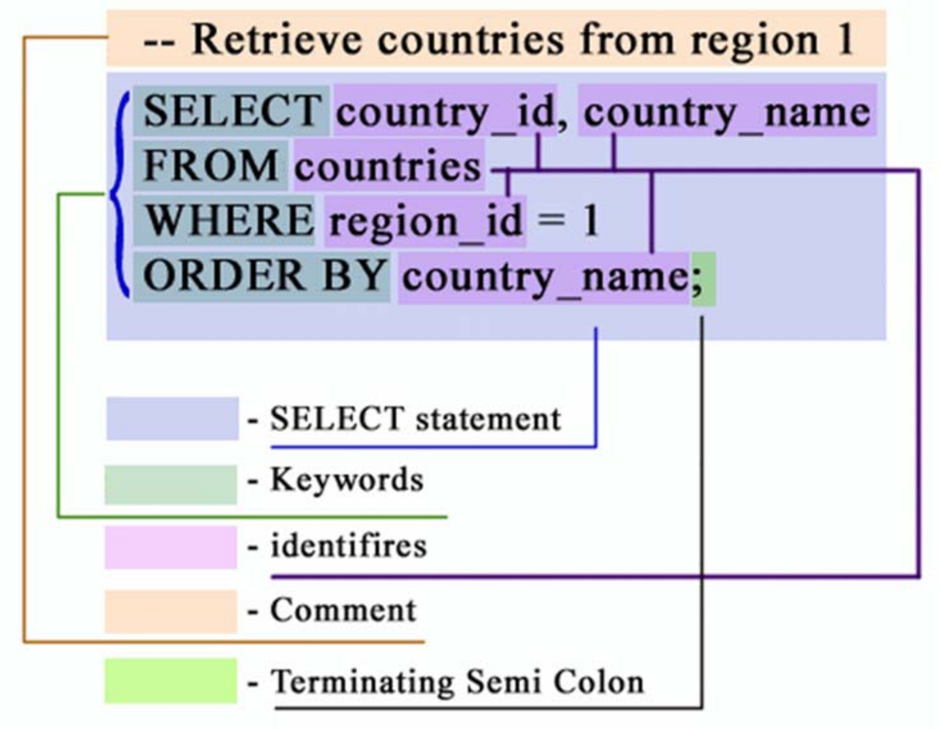
NULL values generally ignored when computing aggregates

```
SELECT count(*) FROM sailors;
SELECT count(rating) FROM sailors;
SELECT sum(rating) FROM sailors;
SELECT avg(rating) FROM sailors;
```

| |
|---|
| **6** |
| 5 |
| 38 |
| 7 |

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Popeye | 10 | 22 |
| 2 | OliveOyl | 11 | 39 |
| 3 | Garfield | 1 | 27 |
| 4 | Bob | 5 | 19 |
| 5 | SpongeBob | 11 | 2 |
| 11 | Jack Sparrow | (null) | 35 |

# Summary: SQL syntax

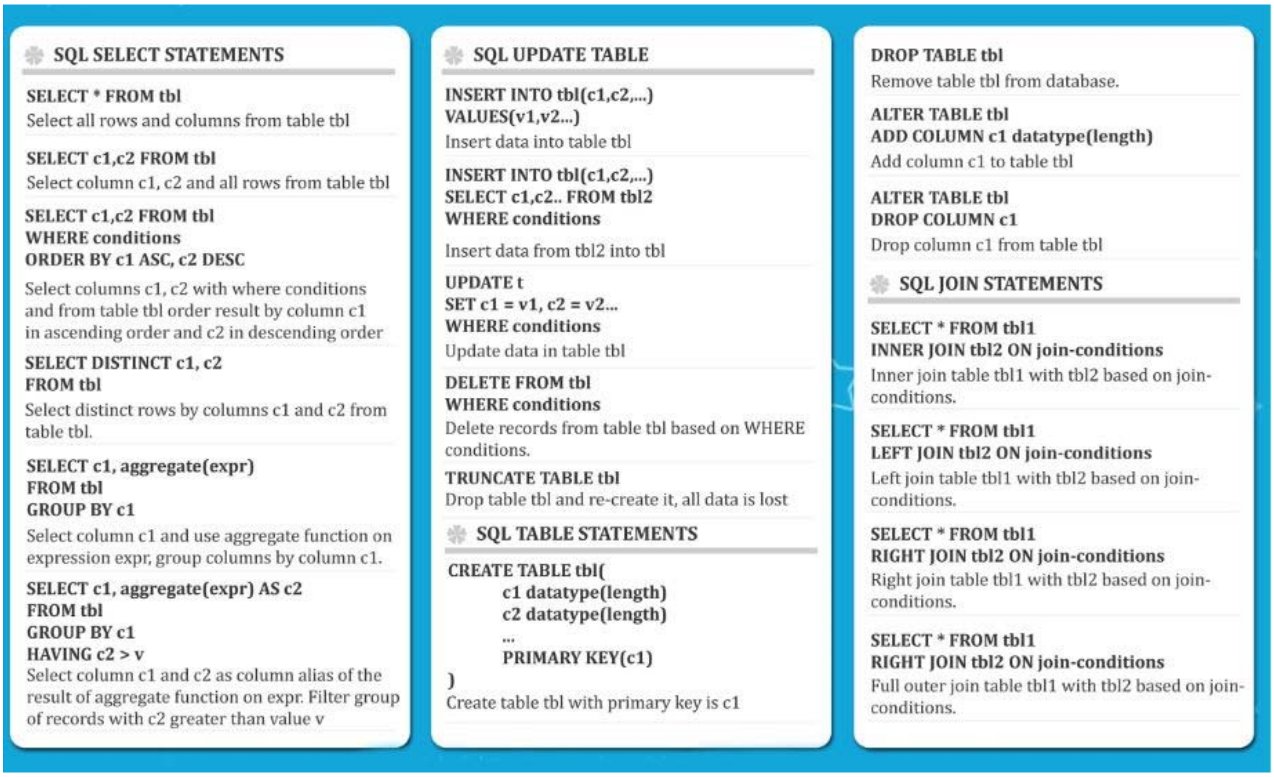


```
-- Retrieve countries from region 1
SELECT country_id, country_name
FROM countries
WHERE region_id = 1
ORDER BY country_name;
```

- SELECT statement
- Keywords
- identifires
- Comment
- Terminating Semi Colon

Source: https://www.w3resource.com/sql/sql-syntax.php

# Summary: SQL command



**SQL SELECT STATEMENTS**

```
SELECT * FROM tbl
```
Select all rows and columns from table tbl

```
SELECT c1,c2 FROM tbl
```
Select column c1, c2 and all rows from table tbl

```
SELECT c1,c2 FROM tbl
WHERE conditions
ORDER BY c1 ASC, c2 DESC
```
Select columns c1, c2 with where conditions and from table tbl order result by column c1 in ascending order and c2 in descending order

```
SELECT DISTINCT c1, c2
FROM tbl
```
Select distinct rows by columns c1 and c2 from table tbl.

```
SELECT c1, aggregate(expr)
FROM tbl
GROUP BY c1
```
Select column c1 and use aggregate function on expression expr, group columns by column c1.

```
SELECT c1, aggregate(expr) AS c2
FROM tbl
GROUP BY c1
HAVING c2 > v
```
Select column c1 and c2 as column alias of the result of aggregate function on expr. Filter group of records with c2 greater than value v

**SQL UPDATE TABLE**

```
INSERT INTO tbl(c1,c2,...)
VALUES(v1,v2...)
```
Insert data into table tbl

```
INSERT INTO tbl(c1,c2,...)
SELECT c1,c2.. FROM tbl2
WHERE conditions
```
Insert data from tbl2 into tbl

```
UPDATE t
SET c1 = v1, c2 = v2...
WHERE conditions
```
Update data in table tbl

```
DELETE FROM tbl
WHERE conditions
```
Delete records from table tbl based on WHERE conditions.

```
TRUNCATE TABLE tbl
```
Drop table tbl and re-create it, all data is lost

**SQL TABLE STATEMENTS**

```
CREATE TABLE tbl(
        c1 datatype(length)
        c2 datatype(length)
        ...
        PRIMARY KEY(c1)
)
```
Create table tbl with primary key is c1

```
DROP TABLE tbl
```
Remove table tbl from database.

```
ALTER TABLE tbl
ADD COLUMN c1 datatype(length)
```
Add column c1 to table tbl

```
ALTER TABLE tbl
DROP COLUMN c1
```
Drop column c1 from table tbl

**SQL JOIN STATEMENTS**

```
SELECT * FROM tbl1
INNER JOIN tbl2 ON join-conditions
```
Inner join table tbl1 with tbl2 based on join-conditions.

```
SELECT * FROM tbl1
LEFT JOIN tbl2 ON join-conditions
```
Left join table tbl1 with tbl2 based on join-conditions.

```
SELECT * FROM tbl1
RIGHT JOIN tbl2 ON join-conditions
```
Right join table tbl1 with tbl2 based on join-conditions.

```
SELECT * FROM tbl1
RIGHT JOIN tbl2 ON join-conditions
```
Full outer join table tbl1 with tbl2 based on join-conditions.

# Hands-On #1

- Write SQL queries to find
  1. Team names for all teams with attendance more than 2,000,000
  2. Player ID and home stadium for all Allstars
  3. TeamID, attendance values for teams that had an all-star player ORDERED BY ATTENDANCE

```
SELECT name FROM Teams WHERE attendance > 2000000;

SELECT playerID, park FROM Allstars, Teams WHERE Allstars.teamID =
Teams.teamID;

SELECT DISTINCT Allstars.teamID, attendance FROM Teams, Allstars
WHERE Teams.teamID = Allstars.teamID ORDER BY attendance DESC;
```

DEMO: http://sqlfiddle.com/#!18/66363/111

# Hands-On #2

- Write SQL queries to find
  1. Average attendance for all teams
  2. Average attendance among teams that had an all-star player

```
SELECT AVG(attendance) FROM Teams;

SELECT AVG(attendance) FROM Teams, Allstars WHERE Teams.teamID
= Allstars.teamID;
```

DEMO: http://sqlfiddle.com/#!18/66363/114

# Hands-On #3

1. Show all teamIds that had an all-star, along with number of all-star players;
2. Show all team names that had an all-star, along with number of all-star players;
3. Show all team names that had an all-star, along with number of all-star players, SORTED IN DESCENDING ORDER OF NUM ALLSTARS AND: only show teams with at least 2 players

```
SELECT teamID, COUNT(*) FROM Allstars GROUP BY teamID;

SELECT name, COUNT(Allstars.playerID) FROM Allstars, Teams WHERE
Allstars.teamID = Teams.teamID GROUP BY name;

SELECT name, COUNT(Allstars.playerID) AS playerCount FROM Allstars,
Teams WHERE Allstars.teamID = Teams.teamID GROUP BY name HAVING
COUNT(Allstars.playerID) >= 2 ORDER BY playerCount DESC;
```

DEMO: http://sqlfiddle.com/#!18/66363/113

# Thank you!

Dr TIAN Jing
Email: tianjing@nus.edu.sg