Chapter 3: Introduction to SQL

Database System Concepts, 7th Ed.

Outline

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

Division

- Notation: r / s or $r \div s$
- Useful for expressing queries that include a "for all" or "for every" phrase
- Let r and s be relations on schemas R and S respectively where
  - $R = (A_1, \dots, A_m, B_1, \dots, B_n)$
  - $S = (B_1, \dots, B_n)$

  Then r/s is a relation on schema

  $R - S = (A_1, \dots, A_m)$

  defined as

$$r / s = \{ t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s \, (tu \in r) \}$$

- Informally, r / s contains the (parts of) tuples of r that are associated with every tuple in s.

Examples of Division A/B

R

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

/

S

pno
p1
p2
p4

=

R / S

sno
s1

$$r / s = \{ t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s \, (tu \in r) \}$$

More on Division

cust (cid, cname, rating, salary)
ord (iid, cid, day, qty)

Query: Find items (iid) that are ordered by **every** customer.

Division (contd.)

cust (cid, cname, rating, salary)
ord (iid, cid, day, qty)

Query: Find items (iid) that are ordered by **every** customer.

$$\pi_{iid,cid}(ord) \div \pi_{cid}(cust)$$

- In RA, using only basic ops:

$$\pi_{iid}(ord) - \pi_{iid}((\pi_{cid}(cust) \times \pi_{iid}(ord)) - \pi_{cid,iid}(ord))$$

Expressing r÷s Using Basic *Operators*

- Generalizing from previous example …
- To express r÷s think as
- Idea:
  - let X = R-S (X is the set of attributes of R that are not in S)
  - (1) compute the X-projection of r
  - (2) compute all X-projection values of r that are `disqualified' by some value in s.
    - value *x* is *disqualified* if by attaching *y* value from *s*, we obtain an *xy* tuple that is not in *r*.
  - result is (1)-(2)
- So,
  - Disqualified x values: $\pi_X((\pi_X(r) \times s) - r)$
  - r ÷ s is $\pi_X(r) - \pi_X((\pi_X(r) \times s) - r)$

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

SQL Parts

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control –includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table *r*
  (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *r* is the name of the relation
  - each A_i is an attribute name in the schema of relation *r*
  - D_i is the data type of values in the domain of attribute A_i

- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20),
  dept_name **varchar**(20),
  salary **numeric**(8,2))

Integrity Constraints in Create Table

- Types of integrity constraints
  - **primary key** (A_1, ..., A_n)
  - **foreign key** (A_m, ..., A_n) **references** *r*
  - **not null**
- SQL prevents any update to the database that violates an integrity constraint.
- Example:

```
create table instructor (
        ID            char(5),
        name          varchar(20) not null,
        dept_name     varchar(20),
        salary        numeric(8,2),
        primary key (ID),
        foreign key (dept_name) references department);
```

And a Few More Relation Definitions

- **create table** *student* (
      ID   **varchar**(5),
      name   **varchar**(20) not null,
      dept_name   **varchar**(20),
      tot_cred   **numeric**(3,0),
      primary key *(ID),*
      foreign key (*dept_name*) **references** *department*);

- **create table** *takes* (
      ID   **varchar**(5),
      course_id   **varchar**(8),
      sec_id   **varchar**(8),
      semester   **varchar**(6),
      year   **numeric**(4,0),
      grade   **varchar**(2),
      primary key *(ID, course_id, sec_id, semester, year)* ,
      foreign key (*ID*) **references** *student,*
      foreign key (*course_id, sec_id, semester, year*) **references** *section*);

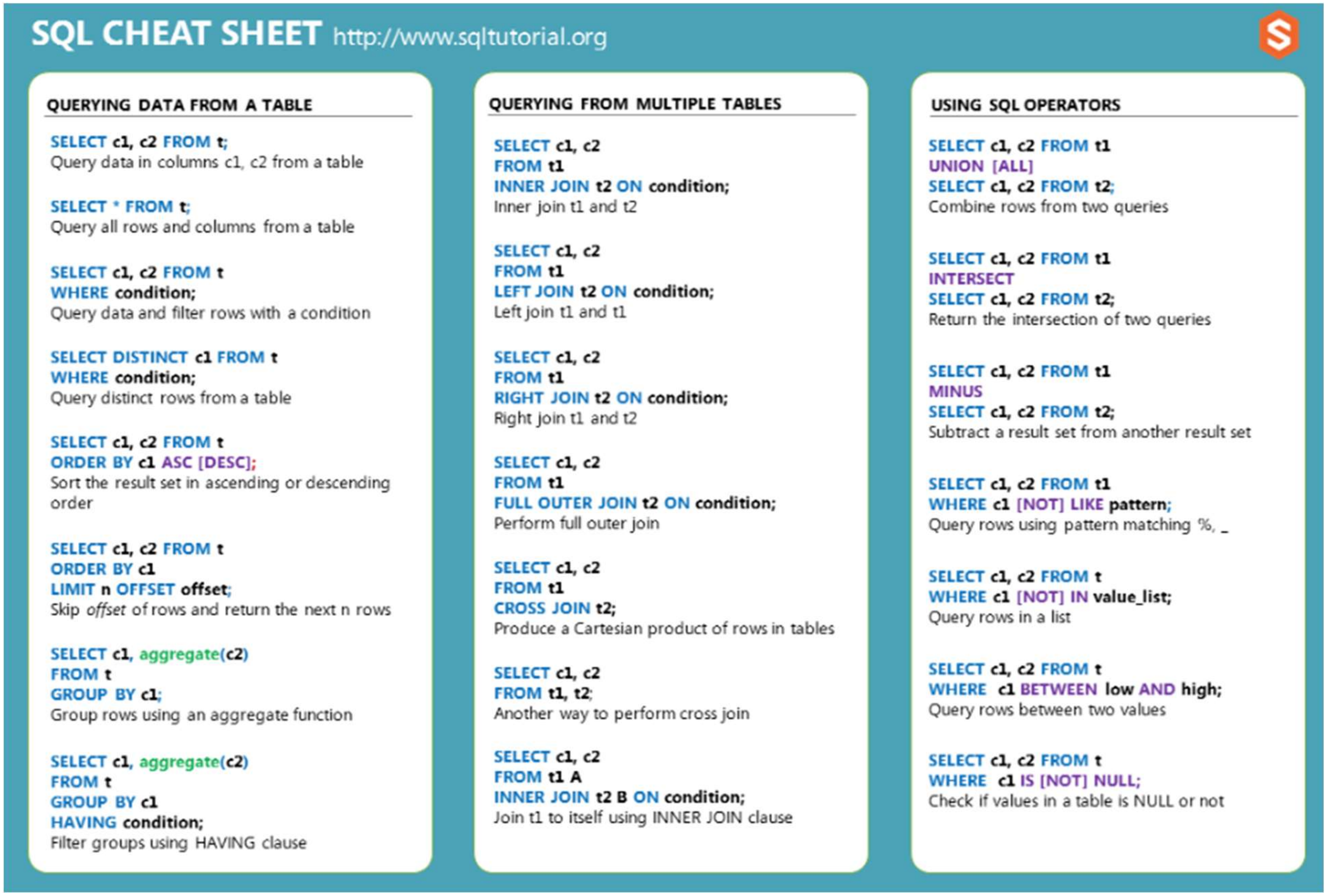
And more still

- **create table** *course* (
  course_id **varchar**(8),
  title **varchar(**50),
  dept_name **varchar**(20),
  credits **numeric**(2,0),
  primary key *(course_id),*
  foreign key (*dept_name*) **references** *department*);

Updates to tables

- **Insert**
  - **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
    - All exiting tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

SQL



SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

sqltutorial.org/sql-cheat-sheet

Basic Query Structure

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- Call this a **SFW** query.
- The result of an SQL query is a relation.

The select Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  select *name*
  from *instructor*

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.
- Values are **not:**

  Different: 'Seattle', 'seattle'
- Use single quotes for constants:

  'abc' - yes
  "abc" - no

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the department names of all instructors, and remove duplicates

  select distinct *dept_name*
  from *instructor*

- The keyword **all** specifies that duplicates should not be removed.

  select all *dept_name*
  from *instructor*

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

  select *
  from *instructor*

- An attribute can be a literal with **from** clause

  select ‘A’
  from *instructor*

  - Result is a table with one column and *N* rows (number of tuples in the *instructors* table), each row with value “A”

The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.
  - The query:

    select *ID, name, salary/12*
    from *instructor*

    would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.
  - Can rename "*salary/12"* using the **as** clause:

    select *ID, name, salary/12* **as** *monthly_salary*

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.'

- SQL allows the use of the logical connectives **and, or,** and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 80000

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  select ∗
  from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

Examples

- Find the names of all instructors who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID*

- Find the names of all instructors in the Art department who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID* ***and*** *instructor. dept_name* = 'Art'

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.
  - **select distinct** *T.name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary* > *S.salary* **and** *S.dept_name* = *‘Comp. Sci.’*

- Keyword **as** is optional and may be omitted
  instructor **as** *T* ≡ *instructor T*

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any single character.
- Find the names of all instructors whose name includes the substring "dar".

  select *name*
  from *instructor*
  where *name* **like** '%dar%'

- Match the string "100%"

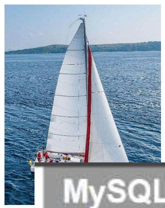
  like '100 \%' **escape** '\'

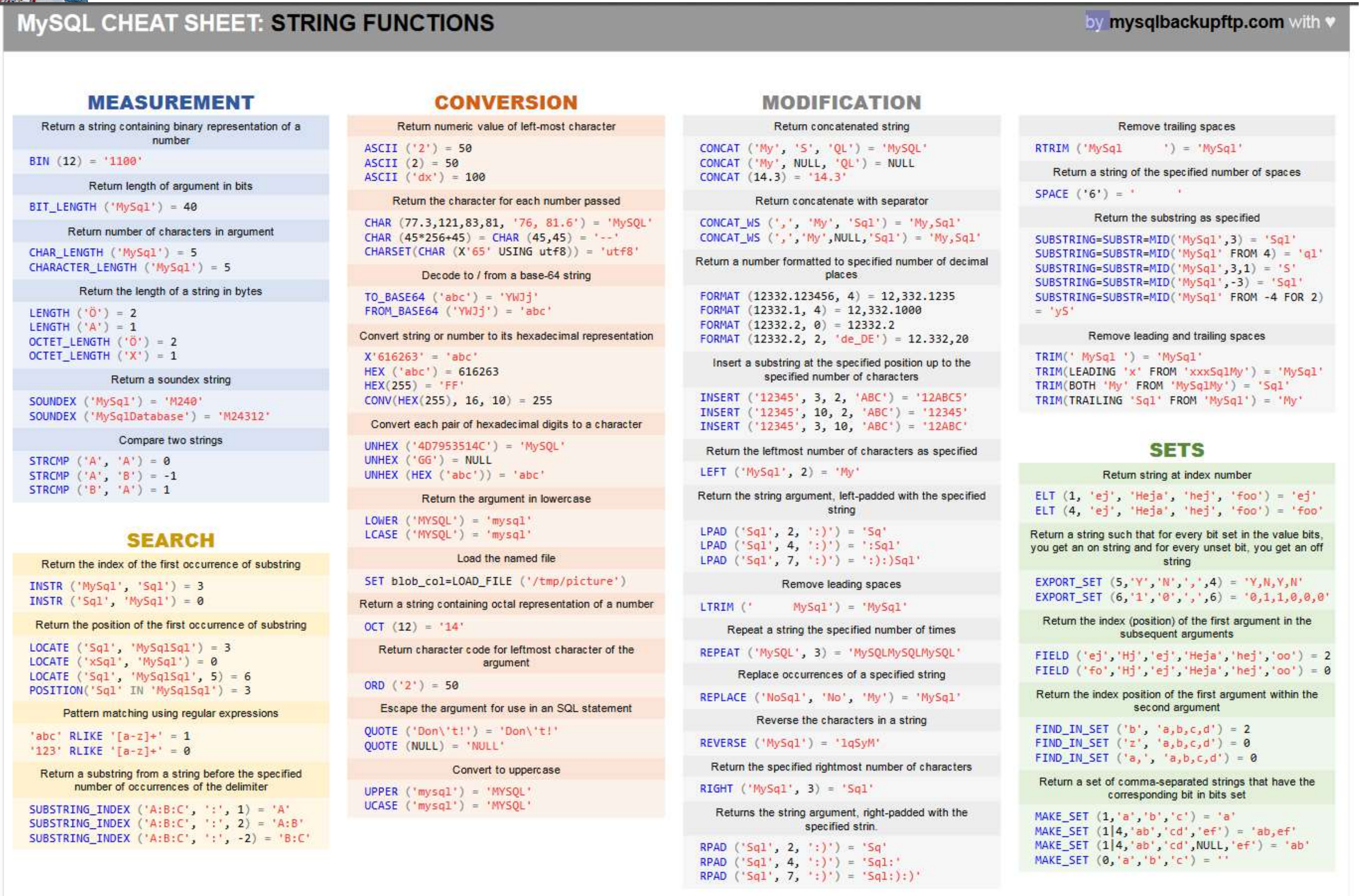
  in that above we use backslash (\) as the escape character.

String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
  - 'Intro%' matches any string beginning with "Intro".
  - '%Comp%' matches any string containing "Comp" as a substring.
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of at least three characters.

- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.





MySQL CHEAT SHEET: STRING FUNCTIONS

by mysqlbackupftp.com with ♥

MEASUREMENT

Return a string containing binary representation of a number

```
BIN (12) = '1100'
```

Return length of argument in bits

```
BIT_LENGTH ('MySql') = 40
```

Return number of characters in argument

```
CHAR_LENGTH ('MySql') = 5
CHARACTER_LENGTH ('MySql') = 5
```

Return the length of a string in bytes

```
LENGTH ('Ö') = 2
LENGTH ('A') = 1
OCTET_LENGTH ('Ö') = 2
OCTET_LENGTH ('X') = 1
```

Return a soundex string

```
SOUNDEX ('MySql') = 'M240'
SOUNDEX ('MySqlDatabase') = 'M24312'
```

Compare two strings

```
STRCMP ('A', 'A') = 0
STRCMP ('A', 'B') = -1
STRCMP ('B', 'A') = 1
```

SEARCH

Return the index of the first occurrence of substring

```
INSTR ('MySql', 'Sql') = 3
INSTR ('Sql', 'MySql') = 0
```

Return the position of the first occurrence of substring

```
LOCATE ('Sql', 'MySqlSql') = 3
LOCATE ('xSql', 'MySql') = 0
LOCATE ('Sql', 'MySqlSql', 5) = 6
POSITION('Sql' IN 'MySqlSql') = 3
```

Pattern matching using regular expressions

```
'abc' RLIKE '[a-z]+' = 1
'123' RLIKE '[a-z]+' = 0
```

Return a substring from a string before the specified number of occurrences of the delimiter

```
SUBSTRING_INDEX ('A:B:C', ':', 1) = 'A'
SUBSTRING_INDEX ('A:B:C', ':', 2) = 'A:B'
SUBSTRING_INDEX ('A:B:C', ':', -2) = 'B:C'
```

CONVERSION

Return numeric value of left-most character

```
ASCII ('2') = 50
ASCII (2) = 50
ASCII ('dx') = 100
```

Return the character for each number passed

```
CHAR (77.3,121,83,81, '76, 81.6') = 'MySQL'
CHAR (45*256+45) = CHAR (45,45) = '--'
CHARSET(CHAR (X'65' USING utf8)) = 'utf8'
```

Decode to / from a base-64 string

```
TO_BASE64 ('abc') = 'YWJj'
FROM_BASE64 ('YWJj') = 'abc'
```

Convert string or number to its hexadecimal representation

```
X'616263' = 'abc'
HEX ('abc') = 616263
HEX(255) = 'FF'
CONV(HEX(255), 16, 10) = 255
```

Convert each pair of hexadecimal digits to a character

```
UNHEX ('4D7953514C') = 'MySQL'
UNHEX ('GG') = NULL
UNHEX (HEX ('abc')) = 'abc'
```

Return the argument in lowercase

```
LOWER ('MYSQL') = 'mysql'
LCASE ('MYSQL') = 'mysql'
```

Load the named file

```
SET blob_col=LOAD_FILE ('/tmp/picture')
```

Return a string containing octal representation of a number

```
OCT (12) = '14'
```

Return character code for leftmost character of the argument

```
ORD ('2') = 50
```

Escape the argument for use in an SQL statement

```
QUOTE ('Don\'t!') = 'Don\'t!'
QUOTE (NULL) = 'NULL'
```

Convert to uppercase

```
UPPER ('mysql') = 'MYSQL'
UCASE ('mysql') = 'MYSQL'
```

MODIFICATION

Return concatenated string

```
CONCAT ('My', 'S', 'QL') = 'MySQL'
CONCAT ('My', NULL, 'QL') = NULL
CONCAT (14.3) = '14.3'
```

Return concatenate with separator

```
CONCAT_WS (',', 'My', 'Sql') = 'My,Sql'
CONCAT_WS (',','My',NULL,'Sql') = 'My,Sql'
```

Return a number formatted to specified number of decimal places

```
FORMAT (12332.123456, 4) = 12,332.1235
FORMAT (12332.1, 4) = 12,332.1000
FORMAT (12332.2, 0) = 12332.2
FORMAT (12332.2, 2, 'de_DE') = 12.332,20
```

Insert a substring at the specified position up to the specified number of characters

```
INSERT ('12345', 3, 2, 'ABC') = '12ABC5'
INSERT ('12345', 10, 2, 'ABC') = '12345'
INSERT ('12345', 3, 10, 'ABC') = '12ABC'
```

Return the leftmost number of characters as specified

```
LEFT ('MySql', 2) = 'My'
```

Return the string argument, left-padded with the specified string

```
LPAD ('Sql', 2, ':)') = 'Sq'
LPAD ('Sql', 4, ':)') = ':Sql'
LPAD ('Sql', 7, ':)') = ':):)Sql'
```

Remove leading spaces

```
LTRIM ('      MySql') = 'MySql'
```

Repeat a string the specified number of times

```
REPEAT ('MySQL', 3) = 'MySQLMySQLMySQL'
```

Replace occurrences of a specified string

```
REPLACE ('NoSql', 'No', 'My') = 'MySql'
```

Reverse the characters in a string

```
REVERSE ('MySql') = 'lqSyM'
```

Return the specified rightmost number of characters

```
RIGHT ('MySql', 3) = 'Sql'
```

Returns the string argument, right-padded with the specified strin.

```
RPAD ('Sql', 2, ':)') = 'Sq'
RPAD ('Sql', 4, ':)') = 'Sql:'
RPAD ('Sql', 7, ':)') = 'Sql:):)'
```

Remove trailing spaces

```
RTRIM ('MySql      ') = 'MySql'
```

Return a string of the specified number of spaces

```
SPACE ('6') = '      '
```

Return the substring as specified

```
SUBSTRING=SUBSTR=MID('MySql',3) = 'Sql'
SUBSTRING=SUBSTR=MID('MySql' FROM 4) = 'ql'
SUBSTRING=SUBSTR=MID('MySql',3,1) = 'S'
SUBSTRING=SUBSTR=MID('MySql',-3) = 'Sql'
SUBSTRING=SUBSTR=MID('MySql' FROM -4 FOR 2)
= 'yS'
```

Remove leading and trailing spaces

```
TRIM(' MySql ') = 'MySql'
TRIM(LEADING 'x' FROM 'xxxSqlMy') = 'MySql'
TRIM(BOTH 'My' FROM 'MySqlMy') = 'Sql'
TRIM(TRAILING 'Sql' FROM 'MySql') = 'My'
```

SETS

Return string at index number

```
ELT (1, 'ej', 'Heja', 'hej', 'foo') = 'ej'
ELT (4, 'ej', 'Heja', 'hej', 'foo') = 'foo'
```

Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string

```
EXPORT_SET (5,'Y','N',',',4) = 'Y,N,Y,N'
EXPORT_SET (6,'1','0',',',6) = '0,1,1,0,0,0'
```

Return the index (position) of the first argument in the subsequent arguments

```
FIELD ('ej','Hj','ej','Heja','hej','oo') = 2
FIELD ('fo','Hj','ej','Heja','hej','oo') = 0
```

Return the index position of the first argument within the second argument

```
FIND_IN_SET ('b', 'a,b,c,d') = 2
FIND_IN_SET ('z', 'a,b,c,d') = 0
FIND_IN_SET ('a,', 'a,b,c,d') = 0
```

Return a set of comma-separated strings that have the corresponding bit in bits set

```
MAKE_SET (1,'a','b','c') = 'a'
MAKE_SET (1|4,'ab','cd','ef') = 'ab,ef'
MAKE_SET (1|4,'ab','cd',NULL,'ef') = 'ab'
MAKE_SET (0,'a','b','c') = ''
```

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

  select distinct *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018

  (**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2017)
  union
  (**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2018)

- Find courses that ran in Fall 2017 and in Spring 2018

  (**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2017)
  intersect
  (**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2018)

- Find courses that ran in Fall 2017 but not in Spring 2018

  (**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2017)
  except
  (**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2018)

Set Operations (Cont.)

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the
  - **union all,**
  - **intersect all**
  - **except all.**

Null Values

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving **null** is **null**
  - Example: 5 + **null** returns **null**
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

    select *name*
    from *instructor*
    where *salary* **is null**
- The predicate **is not null** succeeds if the value on which it is applied is not null.

Null Values (Cont.)

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
  - Example*: 5* < **null** or **null** <> **null** or **null** = **null**
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
  - **and** : *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - **or:** (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown) = unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg:	average value
min:	minimum value
max:	maximum value
sum:	sum of values
count:	number of values

Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count** (**distinct** *ID*)
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2018;
- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

Aggregation

Product(PName, Price, Category, Year, Maker)

```
select AVG(price)
from   Product
where  maker = “Toyota”
```

```
select COUNT(*)
from   Product
where  year > 1995
```

Except COUNT, all aggregations apply to a single attribute

Question: count(*) vs. count(price)?

Aggregation: count

```
Purchase(product, date, price, quantity)
```

count applies to duplicates, unless otherwise stated

```
select COUNT(category)
from   Product
where  year > 1995
```

We probably want:

```
select COUNT(distinct category)
from   Product
where  year > 1995
```

More Examples

```
Purchase(product, date, price, quantity)
```

```
SELECT SUM(price * quantity)
FROM   Purchase
```

What do these mean?

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = ‘bagel’
```

$\Rightarrow$ 50 (= 1*20 + 1.50*20)

Grouping and Aggregation

What GROUPings are possible?

- Type, Size, Color
- Number of holes
- Combination?

What GROUPings are possible?

Purchase

Product	**Date**	**Price**	**Quantity**
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

Possible Groups

- Product? (e.g. SUM(quantity) by product) # product units sold
- Date? (e.g., SUM(price*quantity) by date) # daily sales
- Price?
- Product, Date?
- <various column combinations>

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    from *instructor*
    group by *dept_name*;

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Grouping and Aggregation

Purchase (product, date, price, quantity)

Query: Find total sales after 10/1/2005 per product.

```
SELECT   product, SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > ‘10/1/2005’
GROUP BY product
```

Let’s see what this means…

Grouping and Aggregation

```
SELECT   product, SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Semantics of the query:

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM-WHERE

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

```
SELECT   product, SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

$\xrightarrow{\text{SELECT}}$

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

Purchase (product, date, price, quantity)

```
SELECT     product, SUM(price*quantity)
FROM       Purchase
WHERE      date > '10/1/2005'
GROUP BY   product
HAVING     SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples…***

General form of Grouping and Aggregation

```
SELECT      S
FROM        R1,…,Rn
WHERE       C1
GROUP BY a1,…,ak
HAVING      C2
```

- S: Can **ONLY** contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes
- C_1: is any condition on the attributes in $R_1,\ldots,R_n$
- C_2: is any condition on the aggregate expressions

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - /* erroneous query */
    select *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;
  - Error, Why?

General form of Grouping and Aggregation

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Evaluation steps:

1. Evaluate **FROM-WHERE**: apply condition C_1 on the attributes in $R_1,…,R_n$
2. **GROUP BY** the attributes $a_1,…,a_k$
3. **HAVING:** Apply condition C_2 to each group (may need to compute aggregates)
4. **SELECT:** Compute aggregates in S and return the result

Example

- Find the names and average salaries of all departments whose average salary is greater than 42000

select *dept_name*, **avg** (*salary*) **as** *avg_salary*
from *instructor*
group by *dept_name*
having avg (*salary*) > 42000;

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  as follows:
  - **From clause:** r_i can be replaced by any valid subquery
  - **Where clause:** P can be replaced with an expression of the form:

    B <operation> (subquery)

    Where *B* is an attribute and <operation> to be defined later.
  - **Select clause:**

    A_i can be replaced be a subquery that generates a single value.

Set Membership

Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = ’Fall’ and year= 2017 and
        course_id in (select course_id
                      from section
                      where semester = ’Spring’ and year= 2018);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = ’Fall’ and year= 2017 and
        course_id  not in (select course_id
                           from section
                           where semester = ’Spring’ and year= 2018);
```

Set Membership (Cont.)

- Name all instructors whose name is neither “Mozart” nor Einstein”

```
select distinct name
from instructor
where  name not in (‘Mozart’, ‘Einstein’)
```

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
                             (select course_id, sec_id, semester, year
                              from teaches
                              where teaches.ID= 10101);
```

- Note: Above query can be written in a much simpler manner.
  The formulation above is simply to illustrate SQL features

Subqueries in the From Clause

Subqueries in the Form Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
 select dept_name, avg_salary
from ( select dept_name, avg (salary) as avg_salary
          from instructor
          group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
 select dept_name, avg_salary
from ( select dept_name, avg (salary)
          from instructor
          group by dept_name)
           as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

  delete from *instructor*
  where *salary* < (**select avg** (*salary*)
  from *instructor*);

  - Problem: as we delete tuples from deposit, the average salary changes
  - Solution used in SQL:
    1. First, compute **avg** (salary) and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to *course*

  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values ('3003', 'Green', 'Finance', *null*);

Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

  insert into *instructor*
      select *ID, name, dept_name, 18000*
      from *student*
      where *dept_name* = 'Music' **and** *total_cred* > 144;

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

  Otherwise queries like

      insert into *table*1 **select** * **from** *table*1

  would cause problem

Updates

- Give a 5% salary raise to all instructors

  update *instructor*
  set *salary* = *salary* * 1.05

- Give a 5% salary raise to those instructors who Eran less than 70000

  update *instructor*
  set *salary* = *salary* * 1.05
  where *salary* < 70000;

- Give a 5% salary raise to instructors whose salary is less than average

  update *instructor*
  set *salary* = *salary* * 1.05
  where *salary* < (**select avg** (salary)
  from *instructor*);

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

    update *instructor*
        set *salary* = *salary* * 1.03
        where *salary* > 100000;
    update *instructor*
        set *salary* = *salary* * 1.05
        where *salary* <= 100000;
  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
    set salary = case
                    when salary <= 100000 then salary * 1.05
                    else salary * 1.03
                   end
```

Updates with Scalar Subqueries

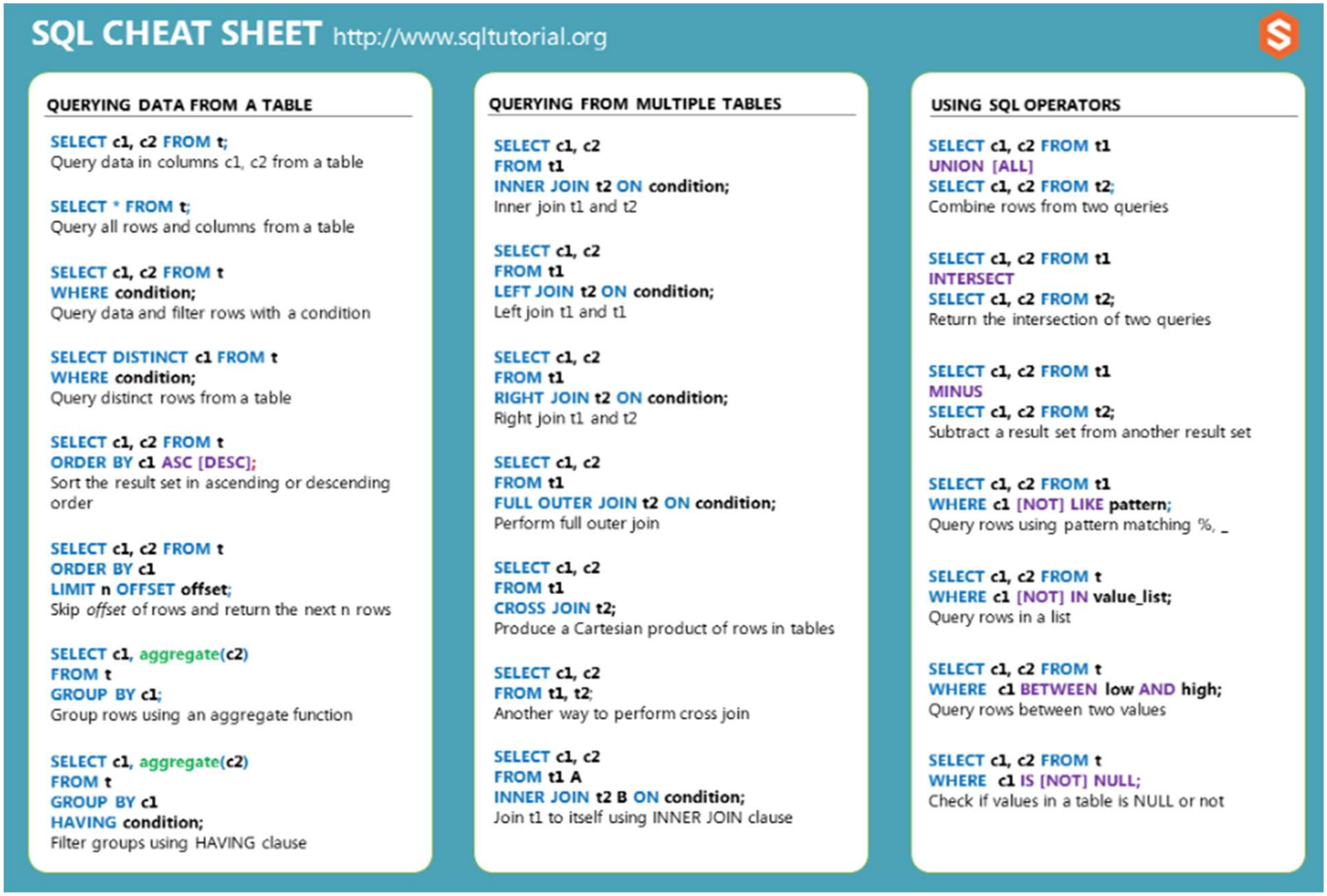
- Recompute and update tot_creds value for all students

```
 update student S
set tot_cred = (select sum(credits)
                     from takes, course
                     where takes.course_id = course.course_id and
                          S.ID= takes.ID.and
                         takes.grade <> 'F' and
                          takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
    when sum(credits) is not null then sum(credits)
    else 0
end
```

SQL



SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

sqltutorial.org/sql-cheat-sheet

End of Chapter 3