**SQL Notes**

* *Describes the table by listing column names and metadata (data types)*

**DESCRIBE** TABLE;

* *Select all data elements from a table*

**SELECT** \*   
**FROM** TABLE;

* *Selecting one column*

**SELECT** COLUMN1  
**FROM** TABLE;

* *Selecting more than one column*

**SELECT** COLUMN1, COLUMN2  
**FROM** TABLE;

* *Setting an alias for columns*

**SELECT** COLUMN1 AS “New Alias”  
**FROM** TABLE;

**SELECT** COLUMN1 “New Alias”  
**FROM** TABLE;

**SELECT** COLUMN1 NEW\_ALIAS  
**FROM** TABLE;

* *Selecting distinct rows (this query will get rows distinct to COLUMN1, COLUMN2 and COLUMN3)*

**SELECT** **DISTINCT**(COLUMN1, COLUMN2, COLUMN3)  
**FROM** TABLE;

* *Concatenating columns (i.e. adding columns together)*

**SELECT** COLUMN1 || ‘ ‘ || COLUMN2 || ‘ ‘ || COLUMN3 AS “Composite Attribute”  
**FROM** TABLE;

* *Note that you can’t have single quotations within single quotations, need to add a delimeter q’<>’ so that SQL knows that the ‘ ‘ within the single quotations are part of the string*

**SELECT** COLUMN1 || q’<’s name is >’ || COLUMN2 AS “Name”  
**FROM** TABLE;

* *Arithmetic operations within SELECT*

**SELECT** COLUMN1 \* COLUMN2 AS “Col1 times Col2”  
**FROM** TABLE;

* *Selecting records with a corresponding column value using WHERE*

**SELECT** COLUMN1   
**FROM** TABLE  
**WHERE** COLUMN1 =/>=/<=/<> 1001;

Note that the WHERE clause can check for conditions on any other column other than the one SELECTed

* *Using WHERE for strings*

**SELECT** COLUMN1   
**FROM** TABLE  
**WHERE** COLUMN1 = ‘King’;

Use single quotation when working when comparing strings. Also, strings are case-sensitive.

* *Using AND/OR in WHERE clause*

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** COLUMN1 = 1001 AND/OR COLUMN2 = ‘King’;

* *Using BETWEEN in WHERE clause*

**SELECT** COLUMN1   
**FROM** TABLE  
**WHERE** COLUMN1 **BETWEEN** 1001 **AND** 1005;  
 **EQUIVALENT TO:** **WHERE** COLUMN1 >= 1001 **AND** COLUMN1 <= 1005

* *Using IN*

Returns rows where COLUMN1 has a value that is IN the given set of possible values

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** COLUMN1 **IN** (Value1, Value2, Value3);

Returns rows where COLUMN1 doesn’t have a value that is IN the given set of possible values

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** COLUMN1 **NOT IN** (Value1, Value2, Value3);

* *Using NULL*

Note that you can’t compare NULL using logical operators like = or <>. You can only use IS or IS NOT.

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** COLUMN1 **IS NULL**;

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** COLUMN1 **IS** **NOT NULL**;

* *Wildcards (%, \_ on strings or even on dates)*

%: preceding, in between or trailing slots

%bon: babon, sabon, taaaabon, daaaaabon, bon [any string that ends with ‘bon’]  
b%on: blon, balloon, boon [any string that starts with ‘b’ and ends with ‘on’ and has any substring within those start and end strings]  
bon%: bonita, bong, bonk, bon [any string that ends starts with bon]  
b%o%: balloon, baobao, blaona [any string that starts with ‘b’ and has any substring followed by an ‘o’ following by another substring]  
%-09 / %/09: any date in the year 2009

\_: single character slots

\_bon: abon, ebon, ubon, ibon  
b\_on: blon, baon, bron  
bo\_n: born, boln  
bon\_: bong, bonk, bons

For wildcards you don’t use logical operators, instead you use LIKE:

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** COLUMN1 **LIKE** ‘**%**bon**%**’ **AND** COLUMN2 **LIKE** ‘**\_**bon’ **AND** COLUMN3 **NOT** **LIKE** ‘b**\_**on’;

* Using ORDER BY [DESC]

By default if [] is not included, records are ordered in ascending order

**SELECT** \*  
**FROM** TABLE  
**WHERE** COLUMN1 > 100  
**ORDER** **BY** COLUMN1;

Can also order by more than one column

**SELECT** \*  
**FROM** TABLE  
**WHERE** COLUMN1 > 100  
**ORDER** **BY** COLUMN1, COLUMN2 **DESC**;

Note that the column you want to ORDER BY need not be in SELECT

Also, by default, NULL values are placed at the bottom of the list, or on top (if DESC).  
We can retain ascending order and add NULLS FIRST to keep NULL at the top.  
Or we can retain descending order and add NULLS LAST to keep NULL at the bottom

* *Dummy Table (DUAL)*

**SELECT** ANYTHING\_YOU\_WANT  
**FROM** **DUAL**;

* *Case Conversion Functions*

**SELECT** **UPPER**(COLUMN1), **LOWER**(COLUMN2), **INITCAP**(**LOWER**(COLUMN2)), **INITCAP**(**UPPER**(COLUMN1))  
**FROM** TABLE  
**WHERE** **UPPER**(COLUMN1) = ‘KING’;

Converts COLUMN1 to uppercase, COLUMN2 to lowercase, LOWER(COLUMN2) to an initial upper character and lowercase for trailing characters, UPPER(COLUMN1) to an initial upper character and lowercase for trailing characters  
  
e.g. INITCAP(UPPER(‘hello’)) = ‘Hello’

* *Character Manipulation Functions*

**SELECT SUBSTR**(COLUMN1, 1, 3), **LENGTH**(COLUMN1), **LPAD**(COLUMN1, 10, ‘\*’), **RPAD**(COLUMN1, 10, ‘\*’)  
**FROM** TABLE;

Returns the substring from index 1 to index 3 of COLUMN1 values (note that strings index from 1).  
Returns the length of the string in COLUMN.  
Pads to string to desired length (second argument).  
Third argument is optional and if left empty, it is replaced as ‘ ‘ but in the query above it is padded as ‘\*’.

* *Number Functions* (ROUND, TRUNC, MOD, POWER, ABS)

**SELECT** **ROUND**(X, Y)  
**FROM** **DUAL**;

**SELECT TRUNC**(X, Y)  
**FROM** **DUAL**;

Instead of rounding, TRUNC trims off decimal Y digits

X = the integer or float  
Y = the number of decimal digits you want, can be negative which will take the closest whole 10|Y|ths. By default, it is Y = 0.

2 5 . 0 0

(-2) (-1) (0) (1) (2)

* *Replace NULLs (NVL and NVL2)*

**SELECT** **NVL**(X, Y)   
**FROM** TABLE;

If X is NULL, replace it with Y (useful because multiplications with NULL returns NULL)

**SELECT** **NVL2**(X, Y, Z)  
**FROM** TABLE;

If X record is NULL, replace it with Z else replace it with Y

* *Using NULLIF*

**SELECT** COLUMN1, **NULLIF**(COLUMN2, 100)  
**FROM** TABLE;

COLUMN2 will return NULL if it COLUMN2 = 100 OR COLUMN2 = NULL

* *Changing format of data values (TO\_CHAR or TO\_NUMBER)*

**SELECT** **TO\_CHAR**(30, ‘$99’)  
**FROM** **DUAL**;   
You get $30

**TO\_CHAR**(30, ‘$99.99’)  
You get $30.00

**TO\_CHAR**(0.1, ‘0.99’)  
You get 0.10

**TO\_CHAR**(’01-MAR-09’, ‘MM/YY’)  
You get 03/09

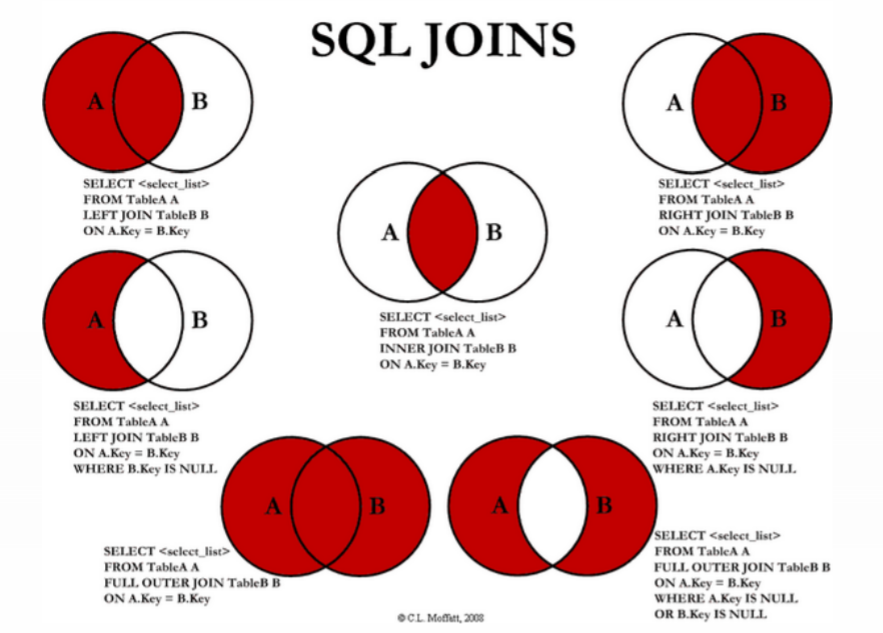
* *SOUNDEX*

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** **SOUNDEX**(COLUMN1) = **SOUNDEX**(‘Alan’);

* *Using CASE*

**SELECT** COLUMN1,  
**CASE**  
 **WHEN** (COLUMN1 >= 10) **THEN** RESULT  
 **WHEN** (COLUMN1 >= 100) **THEN** RESULT2  
 **WHEN** (COLUMN1 >= 1000) **THEN** RESULT3  
 **ELSE** RESULT4  
**END** **AS** “CASE”  
**FROM** TABLE;

Note that RESULT, RESULT2, RESULT3 and RESULT4 can be any data type and is what will be returned as column values for each record.

* *CROSS JOIN*

**SELECT** \*   
**FROM** R, S;

**SELECT** \*   
**FROM** R  
**CROSS** **JOIN** S;

* *Traditional Join using CROSS JOIN*

**SELECT** \*  
**FROM** R, S  
**WHERE** R.COLUMN1 = S.COLUMN2;

Note that COLUMN1 and COLUMN2 must be matching attributes, also you need to identify which attribute comes from which relation using aliases

**SELECT** \*  
**FROM** R r  
**CROSS** **JOIN** S s  
**WHERE** r.COLUMN1 = s.COLUMN2;

* *INNER JOIN*

**SELECT** \*  
**FROM** R r  
**JOIN** S s  
**ON** r.COLUMN1 = s.COLUMN2;

**SELECT** \*  
**FROM** R r  
**INNER** **JOIN** S s  
**USING**(COLUMN1);

Where COLUMN1 exists in R and S.

**SELECT** \*  
**FROM** R r  
**JOIN** S s  
**ON** r.COLUMN1 = s.COLUMN2 // **USING**(COLUMN1)  
**JOIN** T t  
**ON** t.COLUMN1 = s.COLUMN2 // **USING**(COLUMN2)  
**WHERE** t.COLUMN1 = 90;

Can JOIN multiple tables. Can also have WHERE clause.

Note that when using USING, if you want to SELECT the column, you should specify the column name without its associated relation.  
E.g. instead of SELECT r.COLUMN1 you do SELECT COLUMN1 when you do USING(COLUMN1)

* *Natural Join*

**SELECT** \*  
**FROM** R  
**NATURAL** **JOIN** S;

Joins table R with S with the first matching column in both relations (not very useful)

* *Self Join*

**SELECT** \*  
**FROM** R r  
**JOIN** R rcopy  
**ON** r.COLUMN1 = rcopy.COLUMN2;

Can also perform self JOINs.

* *Outer Join*

**SELECT** \*  
**FROM** R r  
**LEFT** **OUTER** **JOIN** S s  
**ON** r.COLUMN1 = s.COLUMN2;

Matches tuples from R with S on COLUMN1 (COLUMN2) where for each non-matching tuple in S, it will be returned as NULL (Think of it as joining S to the left which is R)

**SELECT** \*  
**FROM** R r  
**RIGHT** **OUTER** **JOIN** S s  
**USING** (COLUMN1);

The converse.

* *Aggregate Functions*

**SELECT** **MIN**(COLUMN1), **MAX**(COLUMN1), **SUM**(COLUMN1), **AVG**(COLUMN1), **COUNT**(\*)  
**FROM** TABLE  
**WHERE** COLUMN1 > 100;

These aggregate the column to compute the min, max, sum and average. They all yield a single value result.

* *Count Distinct*

**SELECT COUNT**(**DISTINCT**(COLUMN1 **||** COLUMN2 **||** COLUMN3))  
**FROM** TABLE;

Can select more than one distinct column using ‘||’.

* *Group By*

**SELECT** COLUMN2, COLUMN3, **ANY\_AGGREGATEFUNCTION**(COLUMN1)  
**FROM** TABLE  
**GROUP** **BY** COLUMN2, COLUMN3;

Separates the aggregate function and groups it to a column;  
e.g. COUNT(DISTINCT EMPLOYEES) & Group By DepartmentID gives records of unique DepartmentID and the number of Employees in each department.

* *Having*

**SELECT** COLUMN1  
**FROM** TABLE  
**WHERE** COLUMN1 < 100000  
**GROUP** **BY** COLUMN1  
**HAVING** **AVG**(COLUMN2) > 10000  
**ORDER** **BY** COLUMN1;

In order for you to use HAVING you need to use GROUP BY because HAVING is used for conditions based on grouped aggregates. So for the above, it will return records from COLUMN1 that have an AVG(COLUMN2) > 10000   
(i.e. SUM(COLUMN2) / COUNT(COLUMN2))

Note also the ordering of the clauses.

1. SELECT
2. FROM
3. JOIN/ON/USING
4. WHERE
5. GROUP BY
6. HAVING
7. ORDER BY