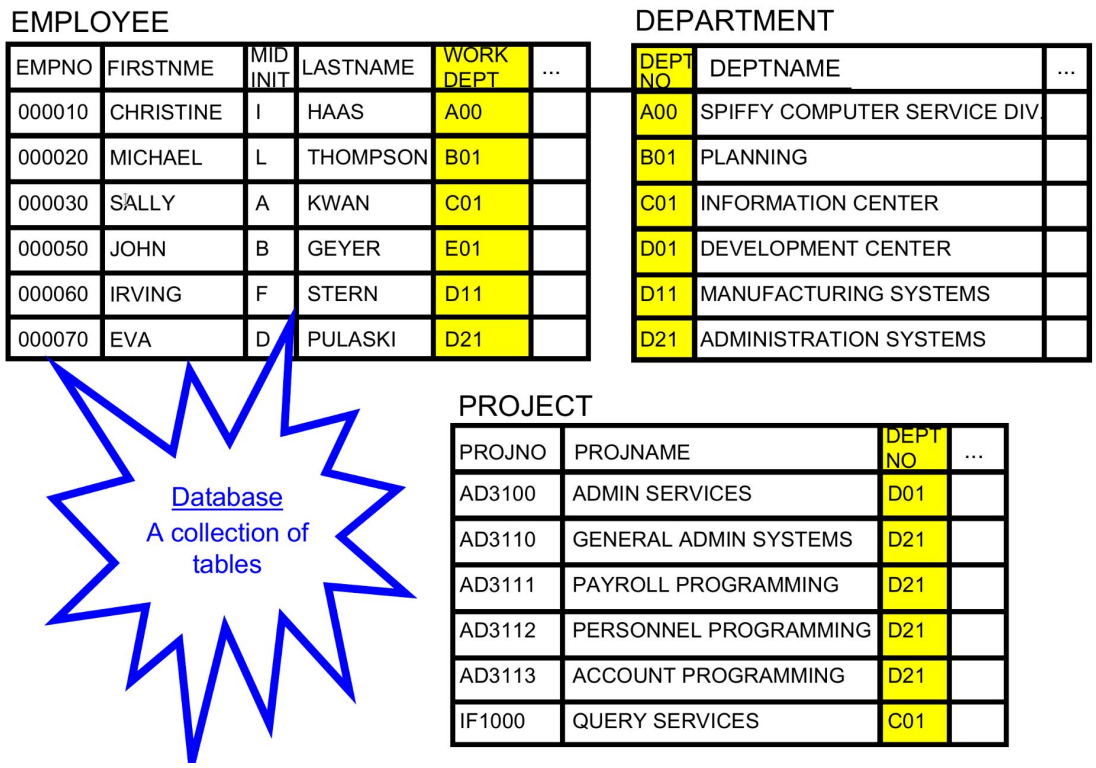
ORACLE SQL & SQL Server

FOUNDATION

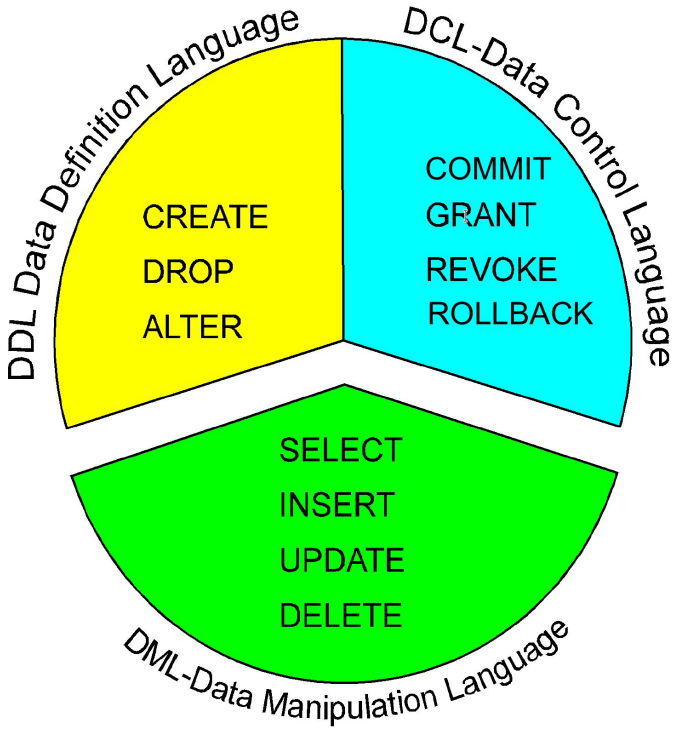
Objectives

- **SQL ANSI Standard**
- **Oracle Human Resources (HR) Sample Schema**
- **The SELECT Statement**
- **Joins**
- **Set Operations**
- **Pseudocolumns**

Relational Database Management System (RDBMS)



SQL - Structured Query Language



ANSI STANDARDS FOR SQL

ANSI Standards for SQL

Year	Standard Name (and Aliases)	Oracle Database
1986	SQL-86 / SQL-87	
1989	SQL-89 / FIPS 127-1	
1992	SQL-92 / SQL2 / FIPS 127-2	
1999	SQL:1999 / SQL3	
2003	SQL:2003	Oracle 10g Release 1 Oracle 10g Release 2 Oracle 11g Release 1
2006	SQL:2006	
2008	SQL:2008	Oracle 11g Release 2
2011	SQL:2011	

ANSI/ISO Standard Structure

Standard Part	Name	Content
ISO/IEC 9075-1:2011 Part 1	Framework (SQL/Framework)	Concepts
ISO/IEC 9075-2:2011 Part 2	Foundation (SQL/Foundation)	Language elements
ISO/IEC 9075-3:2008 Part 3	Call-Level Interface (SQL/CLI)	Interfacing components
ISO/IEC 9075-4:2011 Part 4	Persistent Stored Modules (SQL/PSM)	Procedural extensions
ISO/IEC 9075-9:2008 Part 9	Management of External Data (SQL/MED)	Foreign-data and Datalinks
ISO/IEC 9075-10:2008 Part 10	Object Language Bindings (SQL/OLB)	SQLJ
ISO/IEC 9075-11:2011 Part 11	Information and Definition Schemas (SQL/Schemata)	Self-describing objects
ISO/IEC 9075-13:2008 Part 13	SQL Routines and Types Using the Java Programming Language (SQL/JRT)	Using Java in the database
ISO/IEC 9075-14:2011 Part 14	XML-Related Specifications (SQL/XML)	Using XML

Core SQL Language Syntax and Semantic

ISO/IEC 9075-1:2008 Part 1: Framework (SQL/Framework)

Provides logical concepts.

ISO/IEC 9075-2:2008 Part 2: Foundation (SQL/Foundation) Contains the most central elements of the language and consists of both mandatory and optional features.

ISO/IEC 9075-11:2008 Part 11: Information and Definition Schemas (SQL/Schemata)

Defines the Information Schema and Definition Schema, providing a common set of tools to make SQL databases and objects self-describing.

Core SQL:2008

DATA TYPES

ANSI Data Types

SQL supports three sorts of data types:

1. **predefined data types**
2. **constructed types**
3. **user-defined types**

Oracle supports constructed (reference, rowtype, collection) and user-defined types. These constructions mostly used for PL/SQL programming.

There is no TIME equivalents in Oracle.
BOOLEAN is allowed in PL/SQL only.

ANSI Predefined data types:

1. CHARACTER
2. CHARACTER VARYING
3. CHARACTER LARGE OBJECT
4. BINARY
5. BINARY VARYING
6. BINARY LARGE OBJECT
7. NUMERIC
8. DECIMAL
9. SMALLINT
10. INTEGER
11. BIGINT
12. FLOAT
13. REAL
14. DOUBLE PRECISION
15. BOOLEAN
16. DATE
17. TIME
18. TIMESTAMP
19. INTERVAL

Oracle Data Types

Oracle data types:

1. **Oracle Built-in Types**
2. **ANSI, DB2, and SQL/DS Data Types**
3. **User-Defined Types**
4. **Oracle-Supplied Types**

Oracle built-in data types:

1. CHAR[(size[BYTE| CHAR])]
2. NCHAR[(size[BYTE| CHAR])]
3. VARCHAR2(size[BYTE| CHAR])
4. NVARCHAR2(size)

Character

5. NUMBER [(precision[, scale])]
6. FLOAT[(precision)]
7. BINARY_FLOAT
8. BINARY_DOUBLE

Numeric

9. DATE
10. TIMESTAMP[(fractional_seconds_precision)]
11. TIMESTAMP[(fractional_seconds)] WITH TIME ZONE
12. TIMESTAMP[(fractional_seconds)] WITH LOCAL TIME ZONE
13. INTERVAL YEAR[(year_precision)] TO MONTH
14. INTERVAL DAY[(day_precision)] TO SECOND[(fract_sec)]

Date / Timestamp / Interval

15. CLOB
16. NCLOB
17. BLOB
18. BFILE

LOB

19. LONG
20. RAW(size)
21. LONG RAW

Raw data

22. ROWID
23. UROWID[(size)]

Rowid

The LONG is legacy data type and provided for backward compatibility only. Only one LONG field can be in a table.

Internal Representation of ANSI Data Types in Oracle

```
CREATE TABLE test_ansi_data_types  (
  CHARACTER1 CHARACTER(10),
  CHAR1 CHAR(10),
  CHARACTER_VARYING
    CHARACTER VARYING(10),
  CHAR_VARYING CHAR VARYING(10),
  NATIONAL_CHARACTER
    NATIONAL CHARACTER(10),
  NATIONAL_CHAR NATIONAL CHAR(10),
  NCHAR1 NCHAR(10),
  NATIONAL_CHARACTER_VARYING
    NATIONAL CHARACTER VARYING(10),
  NATIONAL_CHAR_VARYING
    NATIONAL CHAR VARYING(10),
  NCHAR_VARYING NCHAR VARYING(10),
  NUMERIC1 NUMERIC(5,2),
  DECIMAL1 DECIMAL(5,2),
  INTEGER1 INTEGER,
  INT1 INT,
  SMALLINT1 SMALLINT,
  FLOAT1 FLOAT,
  DOUBLE_PRECISION DOUBLE PRECISION,
  REAL1 REAL
);
DESC test_ansi_data_types
DROP TABLE test_ansi_data_types;
```

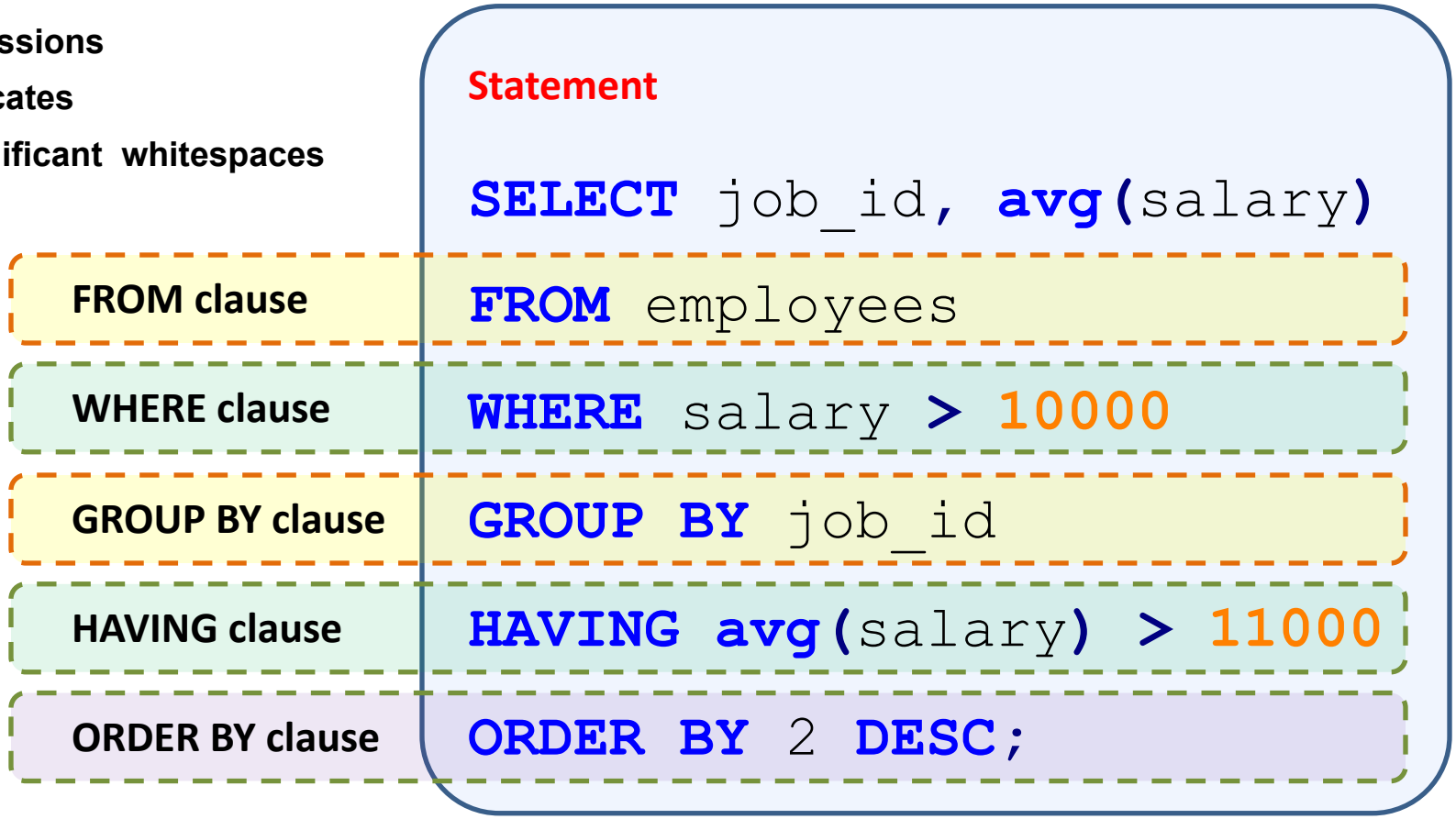
```
table TEST_ANSI_DATA_TYPES created.
DESC test_ansi_data_types
Name                          Null Type
----------------------------- ---- -------------
CHARACTER1                         CHAR(10)
CHAR1                              CHAR(10)
CHARACTER_VARYING                  VARCHAR2(10)
CHAR_VARYING                       VARCHAR2(10)
NATIONAL_CHARACTER                 NCHAR(10)
NATIONAL_CHAR                      NCHAR(10)
NCHAR1                             NCHAR(10)
NATIONAL_CHARACTER_VARYING         NVARCHAR2(10)
NATIONAL_CHAR_VARYING              NVARCHAR2(10)
NCHAR_VARYING                      NVARCHAR2(10)
NUMERIC1                           NUMBER(5,2)
DECIMAL1                           NUMBER(5,2)
INTEGER1                           NUMBER(38)
INT1                               NUMBER(38)
SMALLINT1                          NUMBER(38)
FLOAT1                             FLOAT(126)
DOUBLE_PRECISION                   FLOAT(126)
REAL1                              FLOAT(63)

table TEST_ANSI_DATA_TYPES dropped.
```

ANSY data types have been converted to Oracle native data types

THE SELECT STATEMENT

Basic Language Elements

- Statements
- Queries
- Clauses
- Expressions
- Predicates
- Insignificant whitespaces



Tables Aliases

- Table aliases is optional mechanism to make queries easier to read, understand and maintain.
- **Aliases should be meaningful!**
- Aliases can be used with asterisk, like SELECT emp.*
- Optional AS keyword between table name and its alias throws error in Oracle (non-standard behavior).

```
SELECT emp.job_id, avg(emp.salary)
FROM employees emp
WHERE emp.salary > 10000
GROUP BY emp.job_id
HAVING avg(emp.salary) > 11000
ORDER BY avg(emp.salary) DESC;
```

	JOB_ID	AVG(EMP.SALARY)
1	AD_PRES	24000
2	AD_VP	17000
3	MK_MAN	13000
4	SA_MAN	12200
5	AC_MGR	12000
6	FI_MGR	12000

Field Aliases

Naming Rules:

- Must not exceed 30 characters.
- First character must be a letter
- The rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).
- Identifier enclosed by double quotation marks (") can contain any combination of legal characters, including spaces but excluding quotation marks.
- Identifiers are not case sensitive except within double quotation marks.

```
SELECT
  emp.job_id AS "Group by job",
  avg(emp.salary) "Salary, AVG"
FROM employees "EMP"
WHERE "EMP".salary > 10000
GROUP BY emp.job_id
HAVING avg(emp.salary) > 11000
ORDER BY -"Salary, AVG";
```

	Group by job	Salary, AVG
1	AD_PRES	24000
2	AD_VP	17000
3	MK_MAN	13000
4	SA_MAN	12200
5	AC_MGR	12000
6	FI_MGR	12000

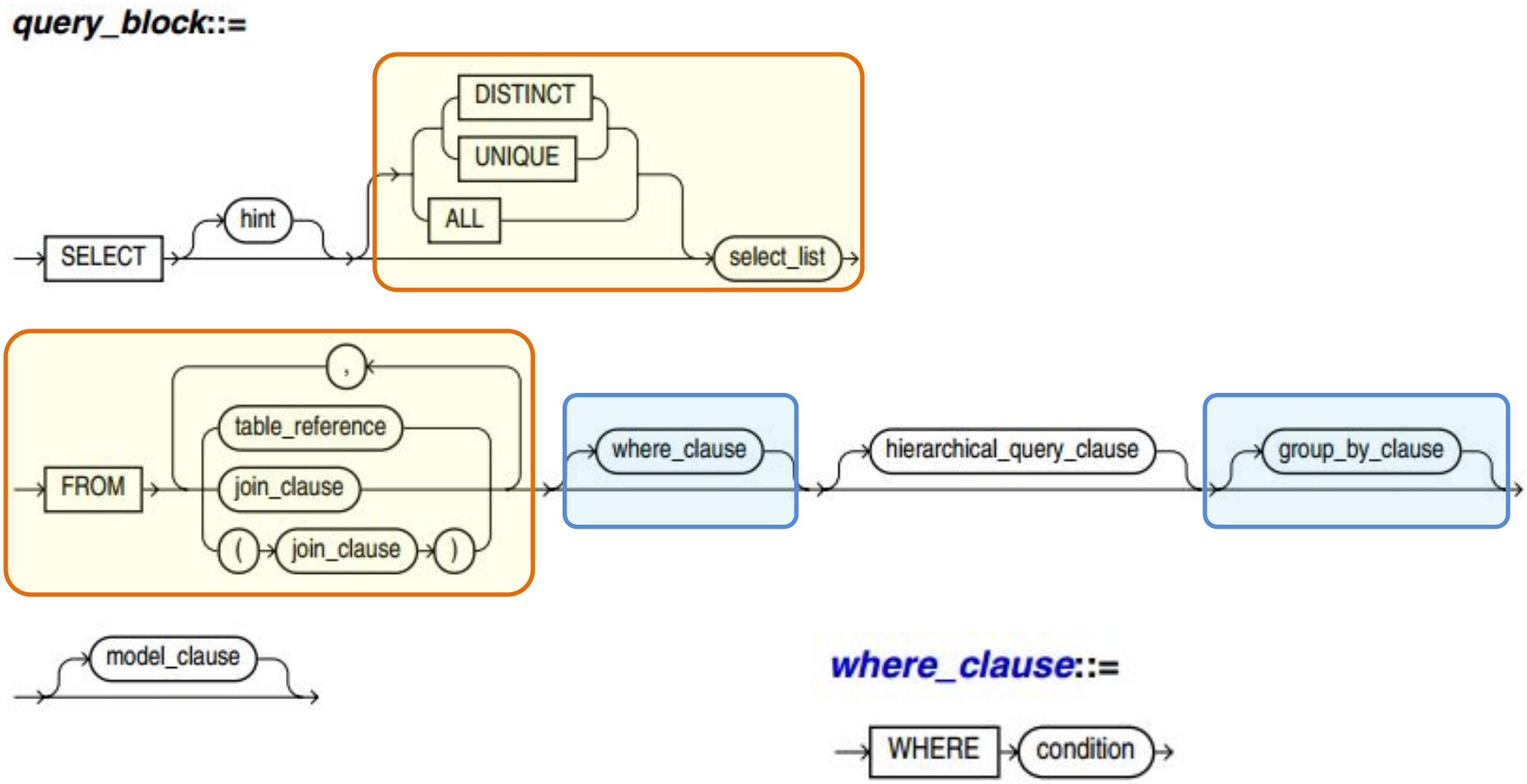
ORDER BY clause (NULLs Ordering)

- ASC | DESC
  Specify the ordering sequence. ASC is the default.
- NULLS FIRST | NULLS LAST
  Specify whether returned rows containing nulls should appear first or last in the ordering sequence.
- NULLS LAST is the default for ascending order, and NULLS FIRST is the default for descending order.

```
SELECT e.job_id AS "Group by job",
  avg(e.commission_pct) "Commission, AVG"
FROM employees e
WHERE "E".salary > 9000
GROUP BY e.job_id
--HAVING min(e.commission_pct) > 0
ORDER BY 2 DESC NULLS LAST;
```

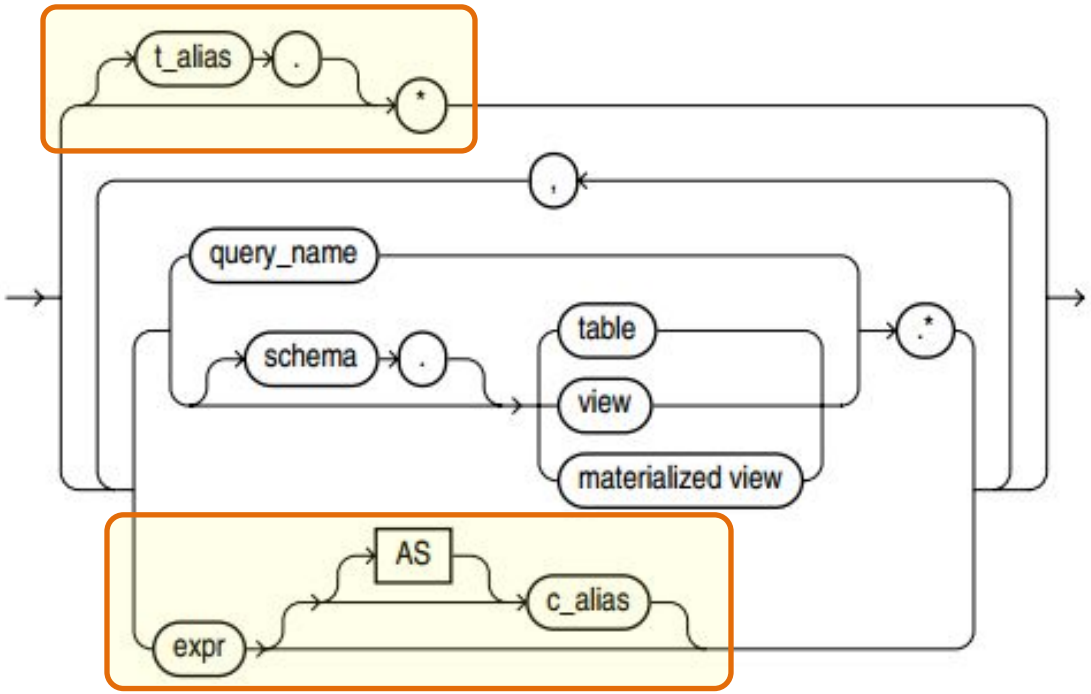
	Group by job	Commission, AVG
1	SA_MAN	0.3
2	SA_REP	0.26
3	PU_MAN	(null)
4	AD_VP	(null)
5	FI_MGR	(null)
6	MK_MAN	(null)
7	PR_REP	(null)
8	AD_PRES	(null)
9	AC_MGR	(null)

Oracle Query Block Structure and WHERE Clause



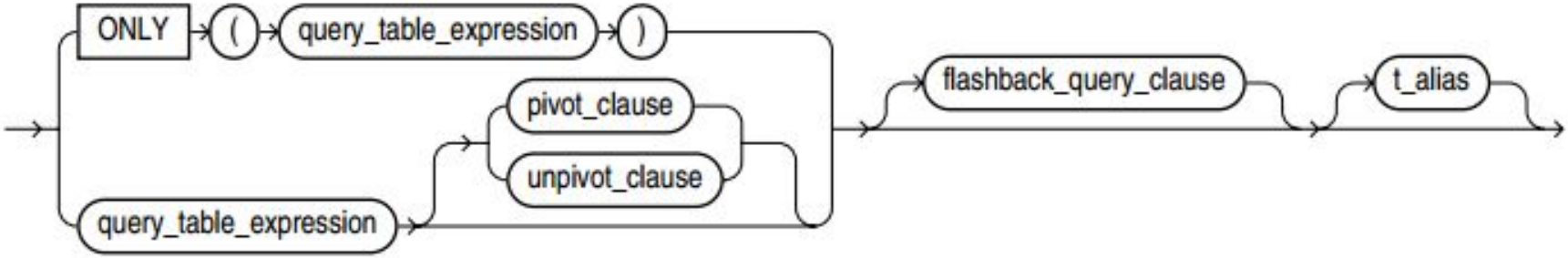
SELECT Columns List

***select_list*::=**

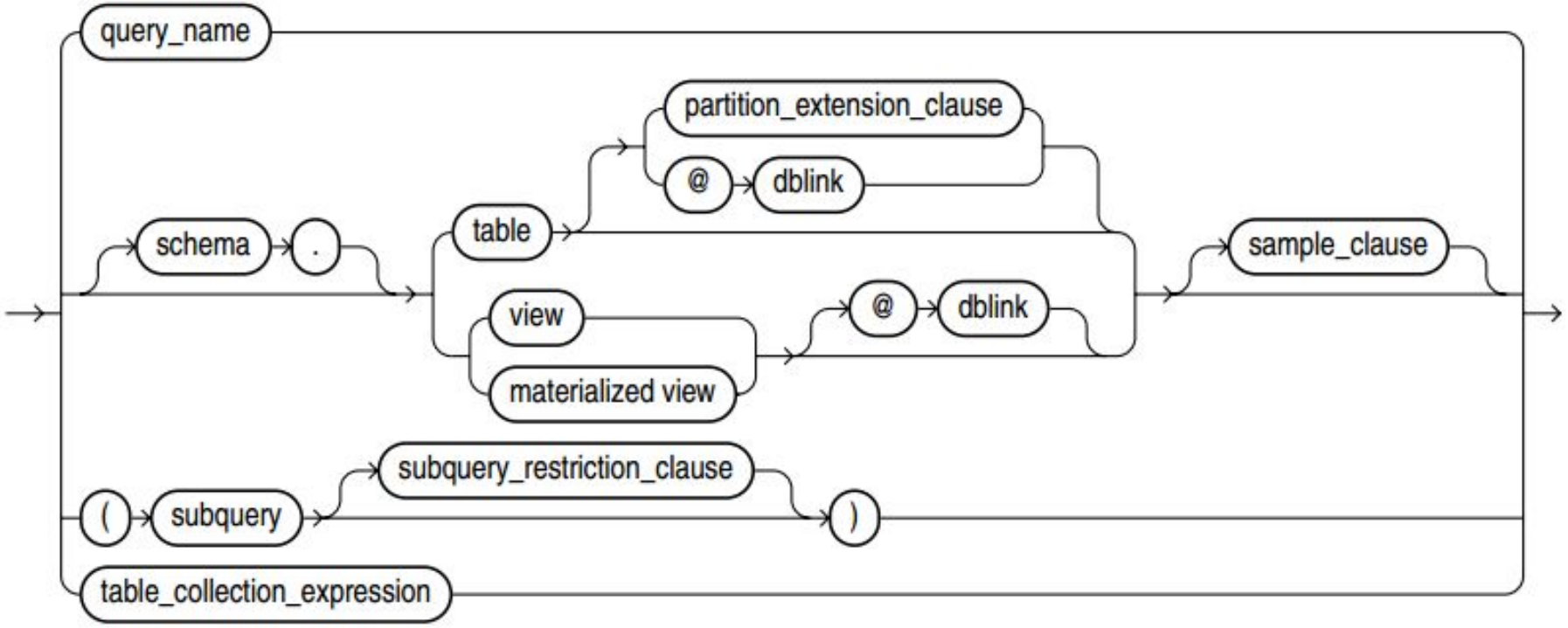


Tables References (simplified FROM clause)

***table_reference*::=**

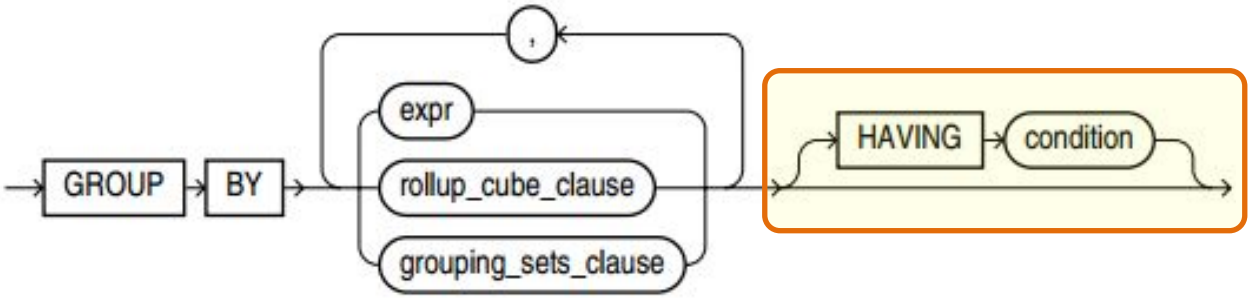


***query_table_expression*::=**

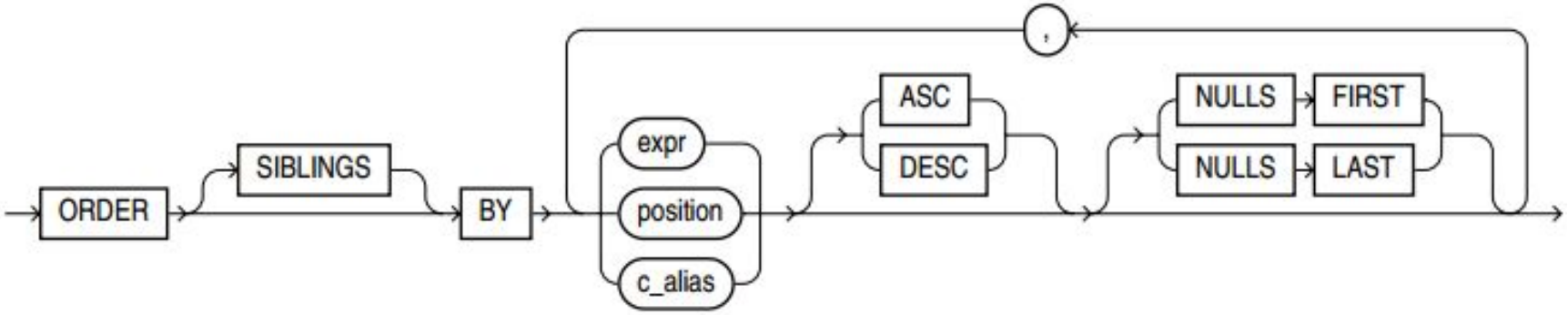


GROUP BY and HAVING clauses, ORDER BY clause

***group_by_clause*::=**



***order_by_clause*::=**



ORACLE FUNCTIONS

Oracle Functions

Function definition

- **A function is a subprogram that returns a single value (or "result") based of its arguments values.**
- **A function is subroutine used to encapsulate frequently performed logic. Any code that must perform the logic incorporated in a function can call the function rather than having to repeat all of the function logic.**

Function may operate on zero, one, two, or more arguments:

```
Function_name
Function_name(argument, argument, ...)
```

A function without any arguments is similar to a pseudocolumn.

However, a pseudocolumn typically returns a different value for each row in the result set, whereas a function without any arguments typically returns the same value for each row.

Functions Classification

1. **Built-in functions. Operate as defined in the Oracle SQL Language Reference and cannot be modified.**
2. **User-defined functions. Allow you to define your own logic, implemented as block of PL/SQL code.**

Oracle Functions

1. **Single-Row (Scalar) Functions**
   SQL scalar functions return a single value, based on the input value(s).
2. **Aggregate Functions**
   SQL aggregate functions return a single value, calculated from values in a column.
3. **Analytic Functions**
   Analytic functions compute an aggregate value based on a group of rows.
   They differ from aggregate functions in that they return multiple rows for each group.
4. **Object Reference Functions**
   Object reference functions manipulate REF values (references to objects).
5. **Model Functions**
   Facilitate SQL for Modeling operations.
6. **OLAP Functions**
   OLAP functions returns data from a dimensional object in two-dimension relational format.
7. **Data Cartridge Functions**
   Facilitate Data Cartridge development.

Single-Row Functions

1. Numeric Functions
2. Character Functions Returning Character Values
3. Character Functions Returning Number Values
4. NLS Character Functions
5. Datetime Functions
6. General Comparison Functions
7. Conversion Functions
8. NULL-Related Functions
9. Large Object Functions
10. Collection Functions
11. Hierarchical Functions
12. Data Mining Functions
13. XML Functions
14. Encoding and Decoding Functions
15. Environment and Identifier Functions

Subject of this module

Numeric Functions

1. **ABS**
2. **ACOS, ASIN**
3. **ATAN, ATAN2**
4. **BITAND**
5. **CEIL**
6. **COS, COSH**
7. **EXP**
8. **FLOOR**
9. **LN, LOG**
10. **MOD**
11. **NANVL**
12. **POWER**
13. **REMAINDER**
14. **ROUND (number), TRUNC (number)**
15. **SIGN**
16. **SIN, SINH**
17. **SQRT**
18. **TAN, TANH**
19. **WIDTH_BUCKET**

The most useful functions are marked blue

Character Functions

Character Functions Returning Character Values

1. CHR, NCHR
2. CONCAT
3. INITCAP, LOWER, UPPER
4. NLS_INITCAP, NLS_LOWER, NLS_UPPER
5. LPAD, RPAD
6. TRIM, LTRIM, RTRIM
7. NLSSORT
8. REGEXP_REPLACE, REGEXP_SUBSTR
9. REPLACE
10. SOUNDEX
11. SUBSTR
12. TRANSLATE

Character Functions Returning Number Values

1. ASCII
2. INSTR
3. LENGTH
4. REGEXP_COUNT, REGEXP_INSTR

Datetime Functions

1. ADD_MONTHS
2. CURRENT_DATE, CURRENT_TIMESTAMP
3. DBTIMEZONE
4. EXTRACT (datetime)
5. FROM_TZ
6. LAST_DAY, NEXT_DAY
7. LOCALTIMESTAMP
8. MONTHS_BETWEEN
9. NEW_TIME
10. NUMTODSINTERVAL, NUMTOYMINTERVAL
11. ORA_DST_AFFECTED, ORA_DST_CONVERT, ORA_DST_ERROR
12. ROUND (date), TRUNC (date)
13. SESSIONTIMEZONE
14. SYS_EXTRACT_UTC
15. SYSDATE, SYSTIMESTAMP
16. TO_CHAR (datetime)
17. TO_DSINTERVAL, TO_YMINTERVAL
18. TO_TIMESTAMP, TO_TIMESTAMP_TZ
19. TZ_OFFSET

General Comparison Functions

1. **GREATEST**
2. **LEAST**

Conversion Functions

1. ASCIISTR
2. BIN_TO_NUM
3. CAST
4. CHARTOROWID
5. COMPOSE
6. CONVERT
7. DECOMPOSE
8. HEXTORAW
9. NUMTODSINTERVAL
10. NUMTOYMINTERVAL
11. RAWTOHEX
12. RAWTONHEX
13. ROWIDTOCHAR
14. ROWIDTONCHAR
15. SCN_TO_TIMESTAMP
16. TIMESTAMP_TO_SCN
17. TO_BINARY_DOUBLE
18. TO_BINARY_FLOAT
19. TO_BLOB
20. TO_CHAR (character)
21. TO_CHAR (datetime)
22. TO_CHAR (number)
23. TO_CLOB
24. TO_DATE
25. TO_DSINTERVAL
26. TO_LOB
27. TO_MULTI_BYTE
28. TO_NCHAR (character)
29. TO_NCHAR (datetime)
30. TO_NCHAR (number)
31. TO_NCLOB
32. TO_NUMBER
33. TO_SINGLE_BYTE
34. TO_TIMESTAMP
35. TO_TIMESTAMP_TZ
36. TO_YMINTERVAL
37. TRANSLATE ... USING
38. UNISTR

Large Object Functions

1. **BFILENAME**
2. **EMPTY_BLOB**
3. **EMPTY_CLOB**

Hierarchical Functions

1. **SYS_CONNECT_BY_PATH**

Data Mining Functions

1. **CLUSTER_ID**
2. **CLUSTER_PROBABILITY**
3. **CLUSTER_SET**
4. **FEATURE_ID**
5. **FEATURE_SET**
6. **FEATURE_VALUE**
7. **PREDICTION**
8. **PREDICTION_BOUNDS**
9. **PREDICTION_COST**
10. **PREDICTION_DETAILS**
11. **PREDICTION_PROBABILITY**
12. **PREDICTION_SET**

XML Functions

1. APPENDCHILDXML
2. DELETEXML
3. DEPTH
4. EXISTSNODE
5. EXTRACT (XML)
6. EXTRACTVALUE
7. INSERTCHILDXML
8. INSERTCHILDXMLAFTER
9. INSERTCHILDXMLBEFORE
10. INSERTXMLAFTER
11. INSERTXMLBEFORE
12. PATH
13. SYS_DBURIGEN
14. SYS_XMLAGG
15. SYS_XMLGEN
16. UPDATEXML
17. XMLAGG
18. XMLCAST
19. XMLCDATA
20. XMLCOLATTVAL
21. XMLCOMMENT
22. XMLCONCAT
23. XMLDIFF
24. XMLELEMENT
25. XMLEXISTS
26. XMLFOREST
27. XMLISVALID
28. XMLPARSE
29. XMLPATCH
30. XMLPI
31. XMLQUERY
32. XMLROOT
33. XMLSEQUENCE
34. XMLSERIALIZE
35. XMLTABLE
36. XMLTRANSFORM

Encoding and Decoding Functions

1. **DECODE**
2. **DUMP**
3. **ORA_HASH**
4. **VSIZE**

NULL-Related Functions

1. **COALESCE**
2. **LNNVL**
3. **NANVL**
4. **NULLIF**
5. **NVL**
6. **NVL2**

Environment and Identifier Functions

1. **SYS_CONTEXT**
2. **SYS_GUID**
3. **SYS_TYPEID**
4. **UID**
5. **USER**
6. **USERENV**

Aggregate Functions

Aggregate functions return a single result row based on groups of rows, rather than on single rows.

Aggregate functions can appear in select lists, in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where Oracle Database divides the rows of a queried table or view into groups.

In a query containing a GROUP BY clause, the elements of the select list can be aggregate functions, GROUP BY expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.

If you omit the GROUP BY clause, then Oracle applies aggregate functions in the select list to all the rows in the queried table or view.

Use aggregate functions in the HAVING clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.

DISTINCT (UNIQUE) / ALL

Many (but not all) aggregate functions that take a single argument accept these clauses:

- **DISTINCT and UNIQUE, which are synonymous, cause an aggregate function to consider only distinct values of the argument expression.**
- **ALL causes an aggregate function to consider all values, including all duplicates.**

- **DISTINCT average of 1, 1, 1, and 3 is 2.**
- **The ALL average is 1.5.**
- **If you specify neither, then the default is ALL.**

All aggregate functions except COUNT(*), GROUPING, and GROUPING_ID ignore nulls. You can use the NVL function in the argument to an aggregate function to substitute a value for a null.

COUNT and REGR_COUNT never return null, but return either a number or zero.

For all the remaining aggregate functions, if the data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns null.

Nested Aggregates

You can nest aggregate functions. For example, the following statement calculates the average of the maximum salaries of all the departments in the sample schema HR:

```
SELECT AVG(MAX(salary))
FROM employees
GROUP BY department_id;
```

```
AVG(MAX(SALARY))
----------------
10926.3333
```

Aggregate Functions

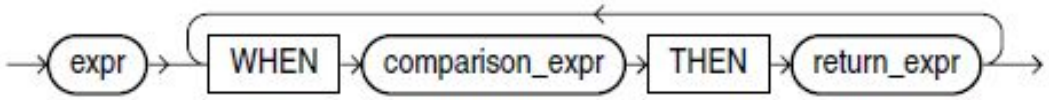
1. AVG
2. COLLECT
3. CORR
4. CORR_*
5. COUNT
6. COVAR_POP
7. COVAR_SAMP
8. CUME_DIST
9. DENSE_RANK
10. FIRST
11. GROUP_ID
12. GROUPING
13. GROUPING_ID
14. LAST
15. LISTAGG
16. MAX
17. MEDIAN
18. MIN
19. PERCENT_RANK
20. PERCENTILE_CONT
19. PERCENTILE_DISC
20. RANK
21. REGR_ (Linear Regression) Functions
22. STATS_BINOMIAL_TEST
23. STATS_CROSSTAB
24. STATS_F_TEST
25. STATS_KS_TEST
26. STATS_MODE
27. STATS_MW_TEST
28. STATS_ONE_WAY_ANOVA
29. STATS_T_TEST_*
30. STATS_WSR_TEST
31. STDDEV
32. STDDEV_POP
33. STDDEV_SAMP
34. SUM
35. SYS_XMLAGG
36. VAR_POP
37. VAR_SAMP
38. VARIANCE
39. XMLAGG

CASE EXPRESSION

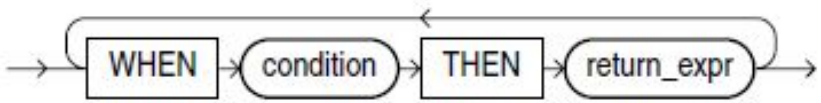
CASE Expressions

CASE expressions let you use IF ... THEN ... ELSE logic in SQL statements without having to invoke procedures.

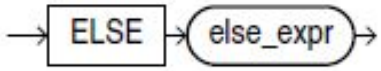
***simple_case_expression*::=**



***searched_case_expression*::=**



***else_clause*::=**



Simple Case (or Case with Selector) Example

```
SELECT first_name, last_name,
  CASE trunc(salary/5000)
    WHEN 0 THEN 'LOW-PAID'
    WHEN 1 THEN 'MID-PAID'
    WHEN 2 THEN 'WELL-PAID'
    ELSE 'EXCELLENT'
  END AS SALARY_CATEGORY
FROM employees;

SELECT first_name, last_name,
  DECODE(trunc(salary/5000),
    0, 'LOW-PAID',
    1, 'MID-PAID',
    2, 'WELL-PAID',
    'EXCELLENT') SALARY_CATEGORY
FROM employees;
```

	FIRST_NAME	LAST_NAME	SALARY_CATEGORY
1	Steven	King	EXCELLENT
2	Neena	Kochhar	EXCELLENT
3	Lex	De Haan	EXCELLENT
4	Alexander	Hunold	MID-PAID
5	Bruce	Ernst	MID-PAID
6	David	Austin	LOW-PAID
7	Valli	Pataballa	LOW-PAID
8	Diana	Lorentz	LOW-PAID
9	Nancy	Greenberg	WELL-PAID
10	Daniel	Faviet	MID-PAID
11	John	Chen	MID-PAID
12	Ismael	Sciarra	MID-PAID
13	Jose Manuel	Urman	MID-PAID
14	Luis	Popp	MID-PAID
15	Den	Raphaely	WELL-PAID
16	Alexander	Khoo	LOW-PAID

Searched Case Example

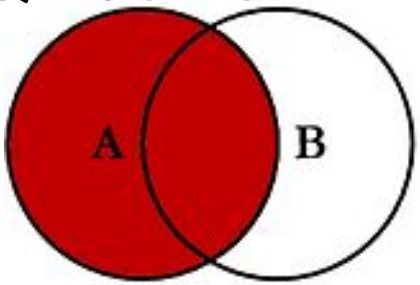
```
SELECT first_name, last_name,
  CASE
    WHEN salary < 5000 THEN 'LOW-PAID'
    WHEN salary >= 5000 AND salary < 10000 THEN 'MID-PAID'
    WHEN salary >= 10000 AND salary < 15000 THEN 'WELL-PAID'
    ELSE 'EXCELLENT'
  END AS SALARY_CATEGORY
FROM employees;
```

```
SELECT first_name, last_name,
  CASE
    WHEN salary < 5000
      THEN 'LOW-PAID'
    WHEN salary < 10000
      THEN 'MID-PAID'
    WHEN salary < 15000
      THEN 'WELL-PAID'
    ELSE 'EXCELLENT'
END AS SALARY_CATEGORY
FROM employees;
```

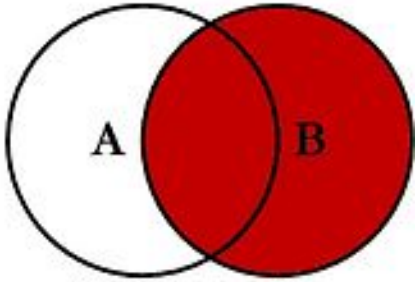
	FIRST_NAME	LAST_NAME	SALARY_CATEGORY
1	Steven	King	EXCELLENT
2	Neena	Kochhar	EXCELLENT
3	Lex	De Haan	EXCELLENT
4	Alexander	Hunold	MID-PAID
5	Bruce	Ernst	MID-PAID
6	David	Austin	LOW-PAID
7	Valli	Pataballa	LOW-PAID
8	Diana	Lorentz	LOW-PAID
9	Nancy	Greenberg	WELL-PAID
10	Daniel	Faviet	MID-PAID
11	John	Chen	MID-PAID
12	Ismael	Sciarra	MID-PAID
13	Jose Manuel	Urman	MID-PAID
14	Luis	Popp	MID-PAID
15	Den	Raphaely	WELL-PAID
16	Alexander	Khoo	LOW-PAID

JOIN TABLES

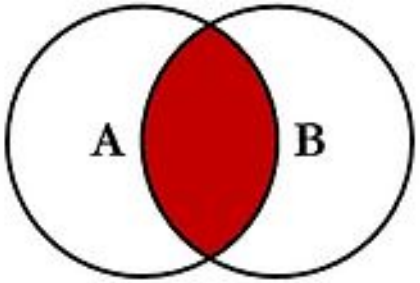
SQL Joins



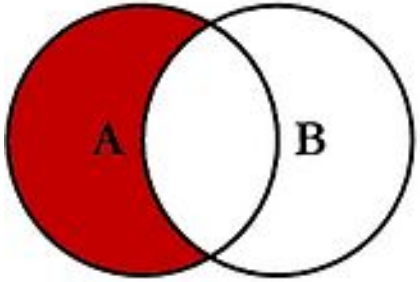

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



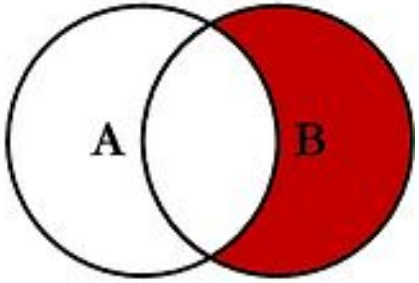

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



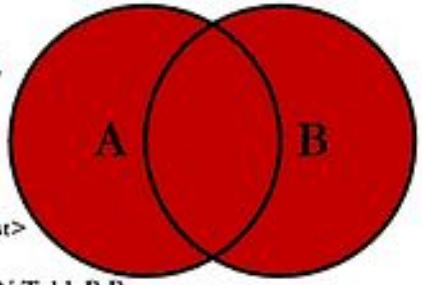

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



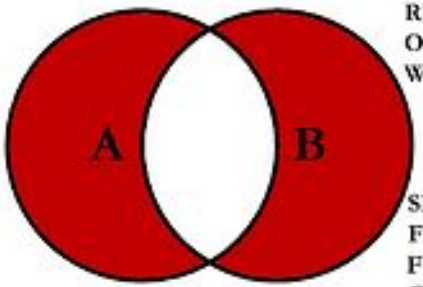

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

SQL Joins Classification

Qualified joins

- Inner join
  - Equi-join
    - Natural join
- Outer joins
  - Left outer join
  - Right outer join
  - Full outer join
- Cross join
- Self-join

Simple Join Example (cross join Employees and Jobs)

```
SELECT emp.first_name, emp.last_name,
  emp.job_id, emp.salary, jb.*
FROM employees emp, jobs jb;
```

```
SELECT emp.first_name, emp.last_name,
  emp.job_id, emp.salary, jb.*
FROM employees emp CROSS JOIN jobs jb;
```

	FIRST_NAME	LAST_NAME	JOB_ID	SALARY	JOB_ID_1	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	Steven	King	AD_PRES	24000	AD_PRES	President	20000	40000
2	Neena	Kochhar	AD_VP	17000	AD_PRES	President	20000	40000
3	Lex	De Haan	AD_VP	17000	AD_PRES	President	20000	40000
4	Alexander	Hunold	IT_PROG	9000	AD_PRES	President	20000	40000
5	Bruce	Ernst	IT_PROG	6000	AD_PRES	President	20000	40000
6	David	Austin	IT_PROG	4800	AD_PRES	President	20000	40000
7	Valli	Pataballa	IT_PROG	4800	AD_PRES	President	20000	40000
8	Diana	Lorentz	IT_PROG	4200	AD_PRES	President	20000	40000
9	Nancy	Greenberg	FI_MGR	12000	AD_PRES	President	20000	40000
10	Daniel	Faviet	FI_ACCOUNT	9000	AD_PRES	President	20000	40000
11	John	Chen	FI_ACCOUNT	8200	AD_PRES	President	20000	40000

Prove Cross Join

```
SELECT count(*) AS cnt
FROM employees emp, jobs jb;
```

```
CNT
---
2033
```

```
SELECT count(*) AS cnt
FROM employees emp CROSS JOIN jobs jb;
```

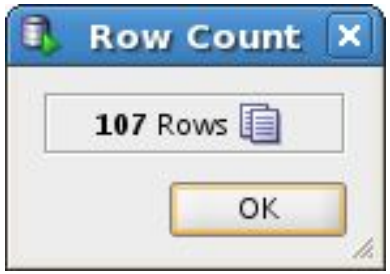
```
CNT
---
2033
```

```
SELECT
  (SELECT count(*) FROM employees emp)
  * (SELECT count(*) FROM jobs jb) cnt
FROM dual;
```

```
CNT
---
2033
```

Reducing Cartesian Product to get meaningful result

```
SELECT emp.first_name,  emp.last_name,
  emp.job_id, emp.salary, jb.*
FROM employees emp, jobs jb
WHERE emp.job_id = jb.job_id;
```

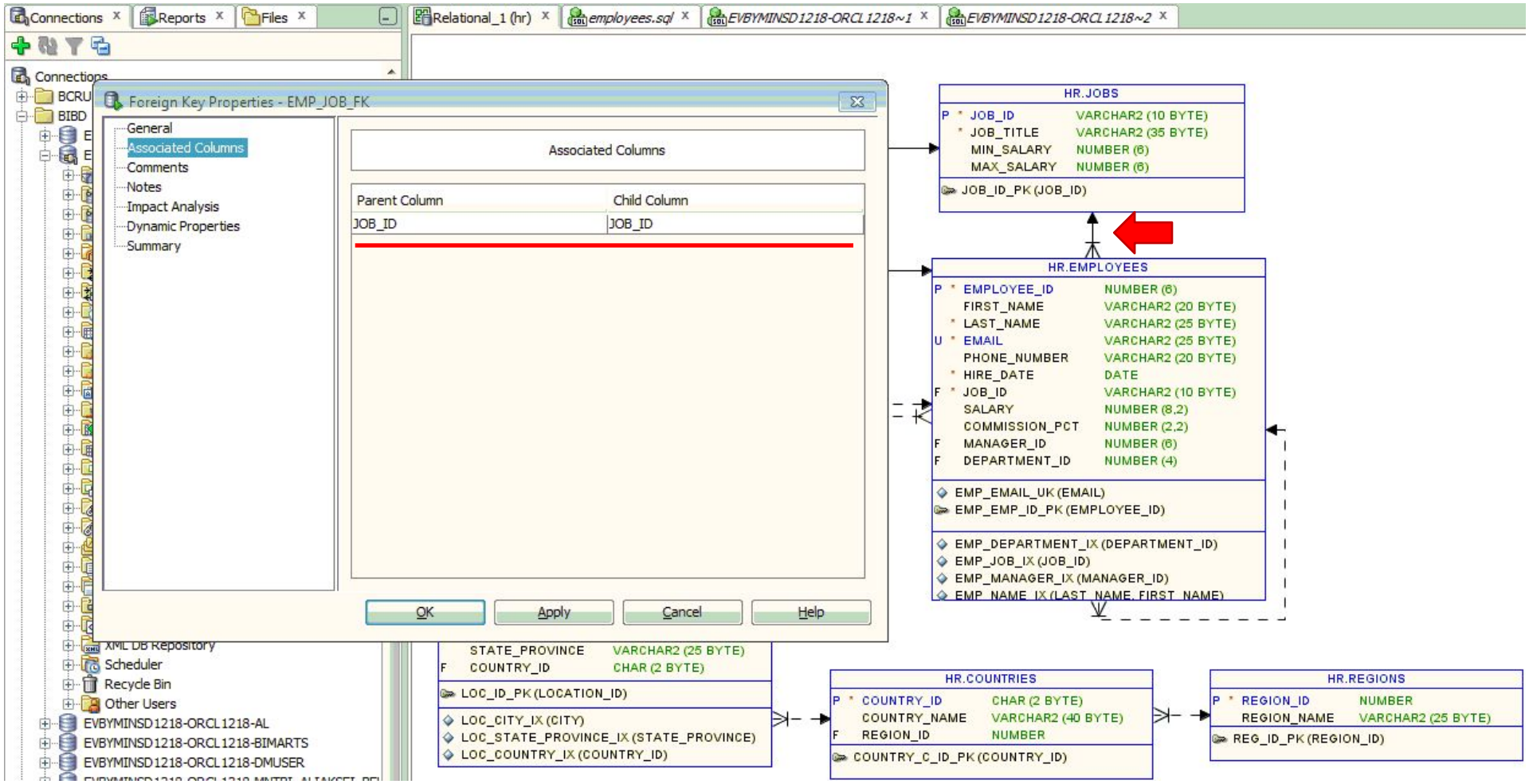



```
SELECT emp.first_name, emp.last_name,
  emp.job_id, emp.salary, jb.*
FROM employees emp CROSS JOIN jobs jb
WHERE emp.job_id = jb.job_id;
```

Senseless syntax

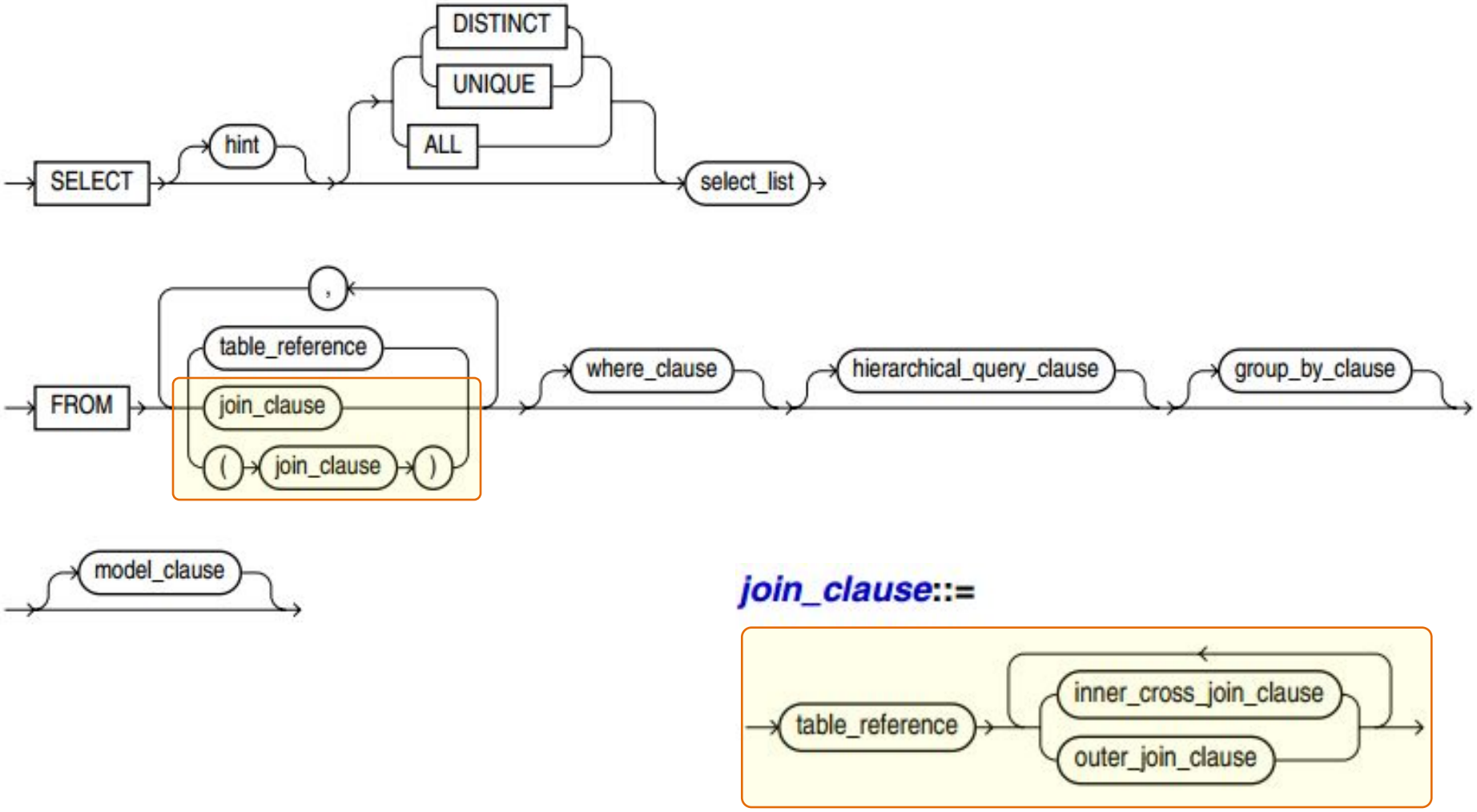
	FIRST_NAME	LAST_NAME	JOB_ID	SALARY	JOB_ID_1	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	Steven	King	AD_PRES	24000	AD_PRES	President	20000	40000
2	Neena	Kochhar	AD_VP	17000	AD_VP	Administration Vice...	15000	30000
3	Lex	De Haan	AD_VP	17000	AD_VP	Administration Vice...	15000	30000
4	Alexander	Hunold	IT_PROG	9000	IT_PROG	Programmer	4000	10000
5	Bruce	Ernst	IT_PROG	6000	IT_PROG	Programmer	4000	10000
6	David	Austin	IT_PROG	4800	IT_PROG	Programmer	4000	10000
7	Valli	Pataballa	IT_PROG	4800	IT_PROG	Programmer	4000	10000
8	Diana	Lorentz	IT_PROG	4200	IT_PROG	Programmer	4000	10000
9	Nancy	Greenberg	FI_MGR	12000	FI_MGR	Finance Manager	8200	16000
10	Daniel	Faviet	FI_ACCOUNT	9000	FI_ACCOUNT	Accountant	4200	9000

Check Your Join (Using foreign keys)



Join Syntax

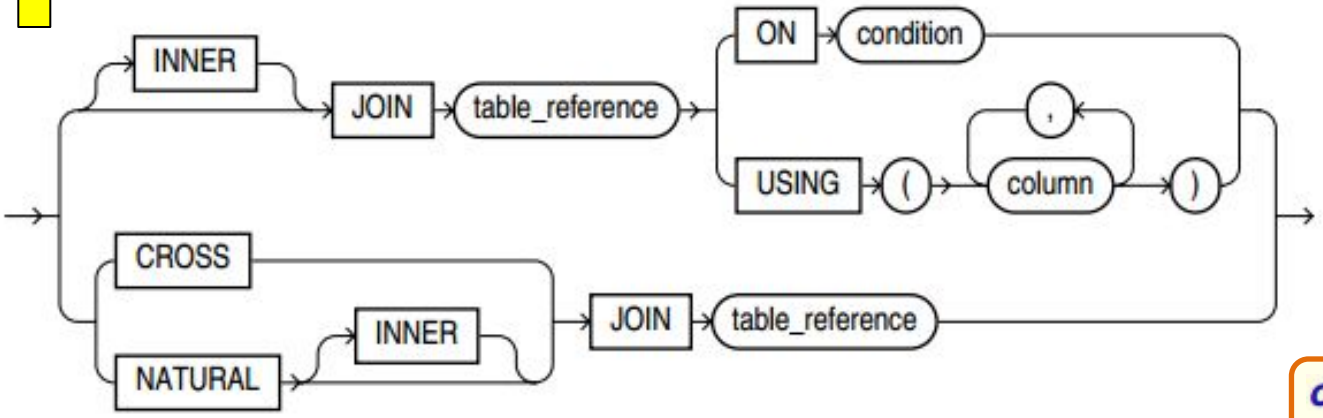
***query_block*::=**



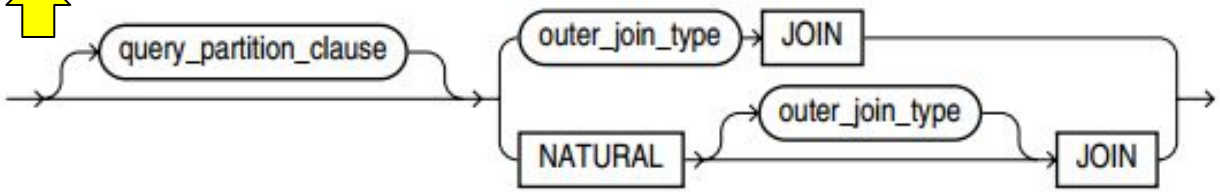
***join_clause*::=**

Inner / Outer / Cross Joins Syntax

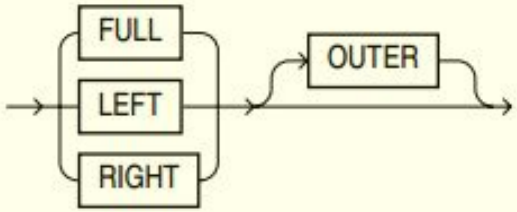
***inner_cross_join_clause*::=**

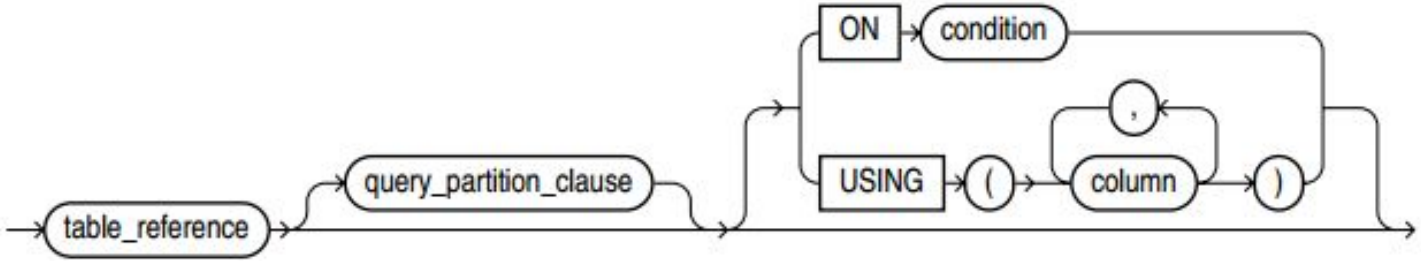


***outer_join_clause*::=**



***outer_join_type*::=**



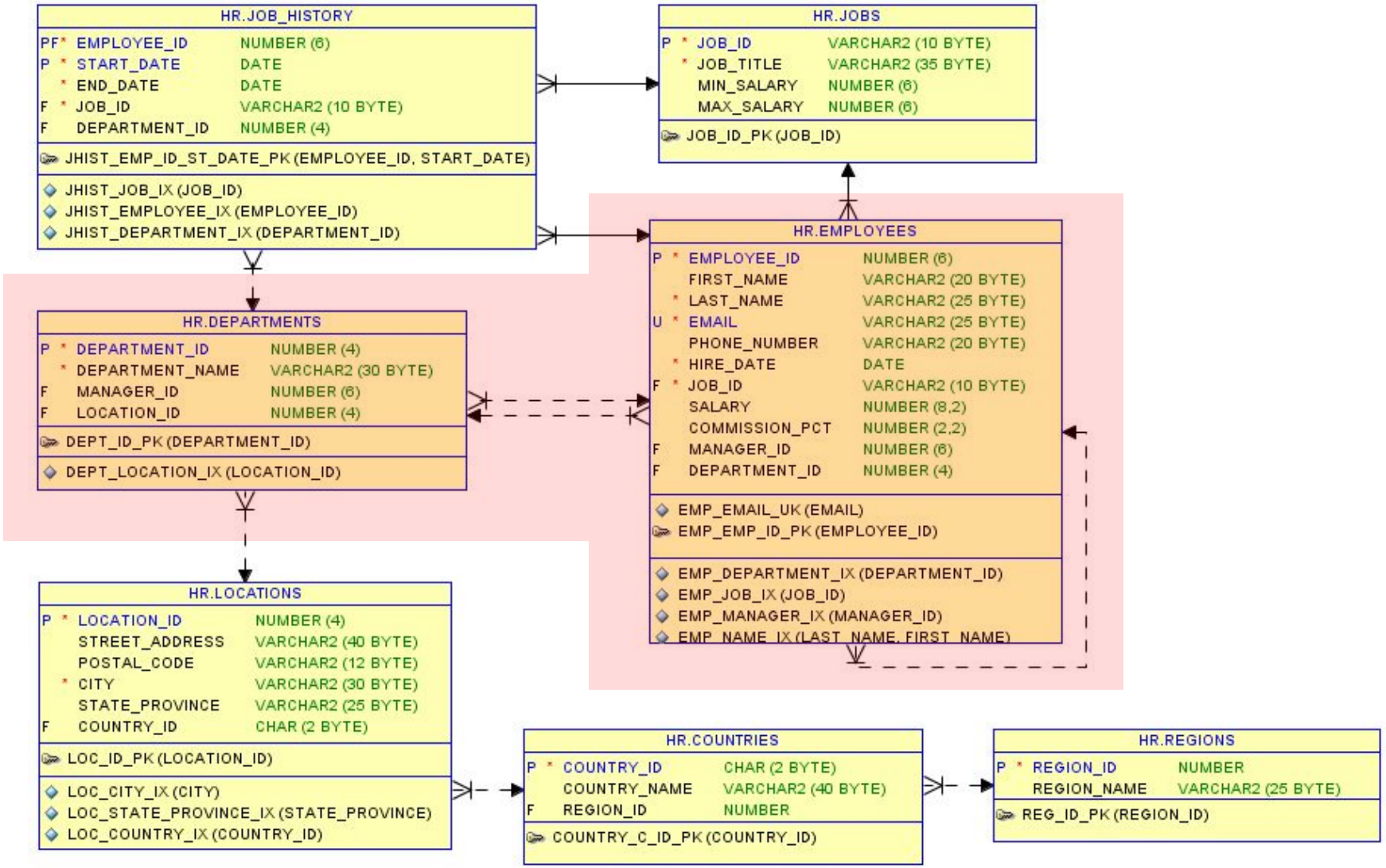


Inner Equi-joins

```
SELECT emp.first_name, emp.last_name, emp.salary, jb.*
FROM employees emp, jobs jb
WHERE emp.job_id = jb.job_id;

SELECT emp.first_name, emp.last_name, emp.salary,
  job_id, jb.job_title, jb.min_salary, jb.max_salary
FROM employees emp NATURAL JOIN jobs jb;

SELECT emp.first_name, emp.last_name, emp.salary,
  job_id, jb.job_title, jb.min_salary, jb.max_salary
FROM employees emp JOIN jobs jb USING(job_id);

SELECT emp.first_name, emp.last_name, emp.salary, jb.*
FROM employees emp JOIN jobs jb ON emp.job_id=jb.job_id;
```

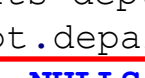
	FIRST_NAME	LAST_NAME	SALARY	JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	Steven	King	24000	AD_PRES	President	20000	40000
2	Neena	Kochhar	17000	AD_VP	Administration Vice President	15000	30000
3	Lex	De Haan	17000	AD_VP	Administration Vice President	15000	30000
4	Alexander	Hunold	9000	IT_PROG	Programmer	4000	10000
5	Bruce	Ernst	6000	IT_PROG	Programmer	4000	10000
6	David	Austin	4800	IT_PROG	Programmer	4000	10000
7	Valli	Pataballa	4800	IT_PROG	Programmer	4000	10000

Outer Equi-joins



Left Outer Equi-joins

```
SELECT emp.first_name, emp.last_name, emp.salary, dept.department_name
FROM employees emp, departments dept
WHERE emp.department_id = dept.department_id(+)
ORDER BY dept.department_name NULLS FIRST;
```



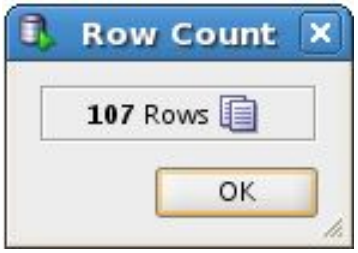
Old Oracle's syntax

```
SELECT emp.first_name, emp.last_name, emp.salary, dept.department_name
FROM employees emp NATURAL LEFT OUTER JOIN departments dept
ORDER BY dept.department_name NULLS FIRST;
```

```
SELECT emp.first_name, emp.last_name, emp.salary, dept.department_name
FROM employees emp LEFT OUTER JOIN departments dept USING (department_id)
ORDER BY dept.department_name NULLS FIRST;
```

```
SELECT emp.first_name, emp.last_name, emp.salary, dept.department_name
FROM employees emp LEFT OUTER JOIN departments dept
  ON (emp.department_id = dept.department_id)
ORDER BY dept.department_name NULLS FIRST;
```

	FIRST_NAME	LAST_NAME	SALARY	DEPARTMENT_NAME
1	Kimberely	Grant	7000	(null)
2	William	Gietz	8300	Accounting
3	Shelley	Higgins	12000	Accounting
4	Jennifer	Whalen	4400	Administration
5	Steven	King	24000	Executive
6	Neena	Kochhar	17000	Executive



Right Outer Equi-joins

```
SELECT dept.department_name, max(emp.salary)
FROM employees emp, departments dept
WHERE emp.department_id(+) = dept.department_id
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

Old Oracle's syntax

```
SELECT dept.department_name, max(emp.salary)
FROM employees emp NATURAL RIGHT JOIN departments dept
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

Do you really want this?

```
SELECT dept.department_name, max(emp.salary)
FROM employees emp RIGHT OUTER JOIN departments dept
  USING (department_id)
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

```
SELECT dept.department_name, max(emp.salary)
FROM employees emp RIGHT OUTER JOIN departments dept
  ON (emp.department_id = dept.department_id)
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

	DEPARTMENT_NAME	MAX(EMP.SALARY)
1	Accounting	12000
2	Administration	4400
3	Executive	24000
4	Finance	12000
5	Human Resources	6500
6	IT	9000
7	Marketing	13000
8	Public Relations	10000
9	Purchasing	11000
10	Sales	14000
11	Shipping	8200

Full Outer Equi-joins

```
SELECT dept.department_name , max(emp.salary)
FROM employees emp , departments dept
WHERE emp.department_id (+) = ept.department_id (+)
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

ORA-01468: a predicate may reference only one outer-joined table

```
SELECT dept.department_name , max(emp.salary)
FROM employees emp NATURAL FULL JOIN departments dept
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

```
SELECT dept.department_name , max(emp.salary)
FROM employees emp FULL OUTER JOIN departments dept
  USING (department_id)
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

```
SELECT dept.department_name , max(emp.salary)
FROM employees emp FULL OUTER JOIN departments dept
  ON (emp.department_id = dept.department_id)
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY dept.department_name NULLS FIRST;
```

	DEPARTMENT_NAME	MAX(EMP.SALARY)
1	(null)	7000
2	Accounting	12000
3	Administration	4400
4	Executive	24000
5	Finance	12000
6	Human Resources	6500
7	IT	9000
8	Marketing	13000
9	Public Relations	10000
10	Purchasing	11000
11	Sales	14000
12	Shipping	8200

Self-join

```
SELECT emp.first_name, emp.last_name, emp.salary,
  mng.first_name manager_first_name, mng.last_name manager_last_name
FROM employees emp LEFT JOIN employees mng
  ON emp.manager_id = mng.employee_id;

SELECT emp.first_name, emp.last_name, emp.salary,
  mng.first_name manager_first_name, mng.last_name manager_last_name
FROM employees emp, employees mng
WHERE emp.manager_id = mng.employee_id(+);
```

	FIRST_NAME	LAST_NAME	SALARY	MANAGER_FIRST_NAME	MANAGER_LAST_NAME
1	Steven	King	24000	(null)	(null)
2	Neena	Kochhar	17000	Steven	King
3	Lex	De Haan	17000	Steven	King
4	Alexander	Hunold	9000	Lex	De Haan
5	Bruce	Ernst	6000	Alexander	Hunold
6	David	Austin	4800	Alexander	Hunold
7	Valli	Pataballa	4800	Alexander	Hunold
8	Diana	Lorentz	4200	Alexander	Hunold
9	Nancy	Greenberg	12000	Neena	Kochhar
10	Daniel	Faviet	9000	Nancy	Greenberg

Complex Join Example

```
SELECT dept.department_name "Dept",
  dept_mng.first_name || ' ' || dept_mng.last_name "Dept Manager",
  emp.first_name || ' ' || emp.last_name "Employee",
  emp_mng.first_name || ' ' || emp_mng.last_name "Emp Manager"
FROM departments dept
  LEFT OUTER JOIN employees dept_mng
    ON (dept.manager_id = dept_mng.employee_id)
  FULL OUTER JOIN employees emp
    ON (emp.department_id = dept.department_id)
  LEFT OUTER JOIN employees emp_mng
    ON (emp.manager_id=emp_mng.employee_id)
ORDER BY 1 NULLS FIRST, 2, 3, 4;
```

Resulting dataset contains 123 rows:

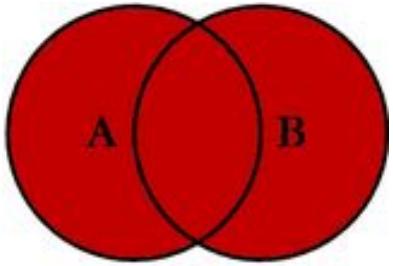
- 107 employees
- 16 empty departments

	Dept	Dept Manager	Employee	Emp Manager
1	(null)		Kimberely Grant	Eleni Zlotkey
2	Accounting	Shelley Higgins	Shelley Higgins	Neena Kochhar
3	Accounting	Shelley Higgins	William Gietz	Shelley Higgins
4	Administration	Jennifer Whalen	Jennifer Whalen	Neena Kochhar
5	Benefits			
6	Construction			
7	Contracting			
8	Control And...			
9	Corporate Tax			
10	Executive	Steven King	Lex De Haan	Steven King
11	Executive	Steven King	Neena Kochhar	Steven King
12	Executive	Steven King	Steven King	
13	Finance	Nancy Greenberg	Daniel Faviet	Nancy Greenberg

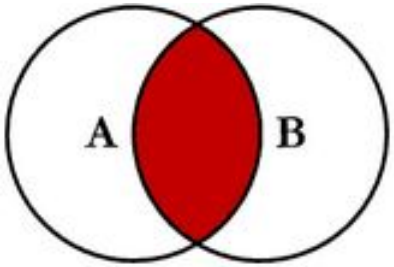
SET OPERATIONS

Set Operations

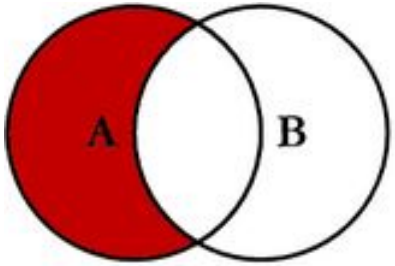
UNION



INTERSECT

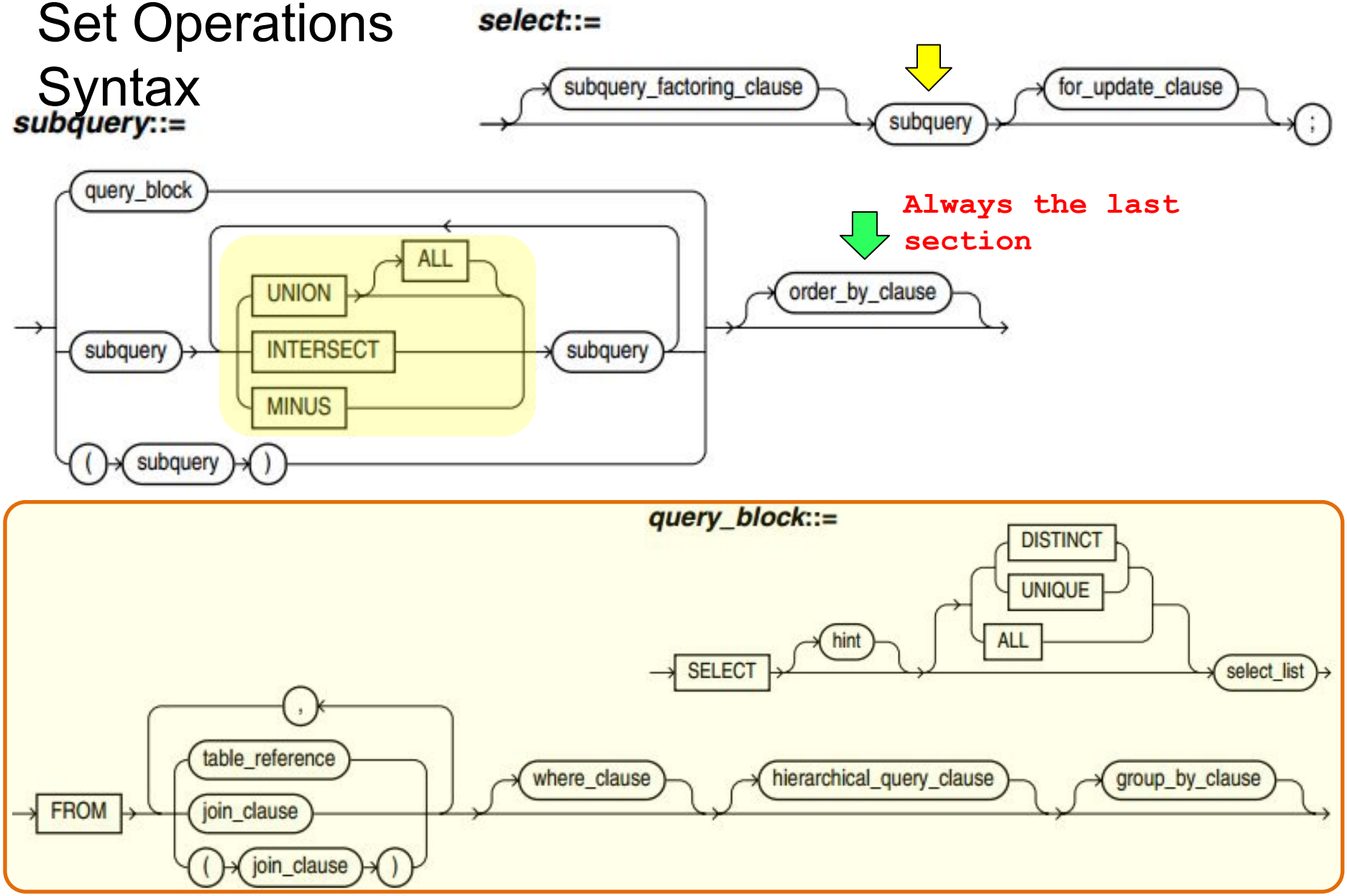


EXCEPT



Operation	ANSI Standard	Oracle
UNION	UNION ALL	UNION ALL
	UNION DISTINCT	UNION
INTERSECT	INTERSECT ALL	
	INTERSECT DISTINCT	INTERSECT
EXCEPT	EXCEPT ALL	
	EXCEPT DISTINCT	MINUS

Set Operations Syntax



Union Operation

```
SELECT dept.department_name,
  max(emp.salary)
FROM employees emp, departments dept
WHERE
  emp.department_id(+) = dept.department_id
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
UNION
SELECT dept.department_name, max(emp.salary)
FROM employees emp, departments dept
WHERE emp.department_id = dept.department_id(+)
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
ORDER BY 1 NULLS FIRST;

SELECT dept.department_name, max(emp.salary)
FROM employees emp, departments dept
WHERE emp.department_id(+) = dept.department_id
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 0
UNION
SELECT NULL, max(salary)
FROM employees emp
WHERE department_id IS NULL
ORDER BY 1 NULLS FIRST;
```

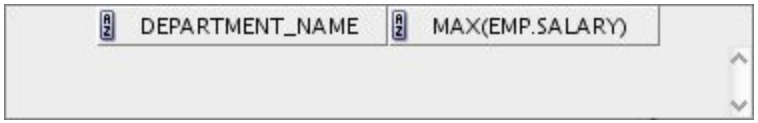
	DEPARTMENT_NAME	MAX(EMP.SALARY)
1	(null)	7000
2	Accounting	12000
3	Administration	4400
4	Executive	24000
5	Finance	12000
6	Human Resources	6500
7	IT	9000
8	Marketing	13000
9	Public Relations	10000
10	Purchasing	11000
11	Sales	14000
12	Shipping	8200

Minus Operation (Check datasets equivalence)

```
(
  SELECT dept.department_name, max(emp.salary)
  FROM employees emp FULL OUTER JOIN departments dept
    USING (department_id)
  GROUP BY dept.department_name
  HAVING count(emp.employee_id) > 0
)
MINUS
(
  SELECT dept.department_name, max(emp.salary)
  FROM employees emp, departments dept
  WHERE emp.department_id(+) = dept.department_id
  GROUP BY dept.department_name
  HAVING count(emp.employee_id) > 0
  UNION
  SELECT dept.department_name, max(emp.salary)
  FROM employees emp, departments dept
  WHERE emp.department_id = dept.department_id(+)
  GROUP BY dept.department_name
  HAVING count(emp.employee_id) > 0
);
```

Full Outer Join

Right Outer Join
Union
Left Outer join



Minus Operation (Check datasets equivalence)

```
(
  SELECT dept.department_name, max(emp.salary)
  FROM employees emp, departments dept
  WHERE emp.department_id(+) = dept.department_id
  GROUP BY dept.department_name
  HAVING count(emp.employee_id) > 0
  UNION
  SELECT dept.department_name, max(emp.salary)
  FROM employees emp, departments dept
  WHERE emp.department_id = dept.department_id(+)
  GROUP BY dept.department_name
  HAVING count(emp.employee_id) > 0
)
MINUS
(
  SELECT dept.department_name, max(emp.salary)
  FROM employees emp FULL OUTER JOIN departments dept
    USING (department_id)
  GROUP BY dept.department_name
  HAVING count(emp.employee_id) > 0
);
```

Right Outer Join
Union
Left Outer join

Full Outer Join

DEPARTMENT_NAME	MAX(EMP.SALARY)

Intersect Operation

```
SELECT dept.department_name
FROM employees emp, departments dept
WHERE emp.department_id(+) = dept.department_id
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 3
INTERSECT
SELECT dept.department_name
FROM employees emp, departments dept
WHERE emp.department_id(+) = dept.department_id
GROUP BY dept.department_name
HAVING MAX(emp.salary) > 9000;
```

	DEPARTMENT_NAME
1	Finance
2	Purchasing
3	Sales

```
SELECT dept.department_name
FROM employees emp, departments dept
WHERE emp.department_id(+) = dept.department_id
GROUP BY dept.department_name
HAVING count(emp.employee_id) > 3 and max(emp.salary) > 9000;
```

UNION ALL Operation

```
SELECT 'Dept' AS "Dept/Job",
  dept.department_name "Name",
  avg(emp.salary) "Avg Salary"
FROM employees emp
  JOIN departments dept
    USING (department_id)
GROUP BY department_id, dept.department_name
HAVING avg(emp.salary) > 9000
UNION ALL
SELECT 'Job',
  jb.job_title,
  avg(emp.salary)
FROM employees emp
  JOIN jobs jb
    USING (job_id)
GROUP BY job_id, jb.job_title
HAVING avg(emp.salary) > 9000
ORDER BY 1, 2, 3;
```

	Dept/Job	Name	Avg Salary
1	Dept	Accounting	10150
2	Dept	Executive	19333.33...
3	Dept	Marketing	9500
4	Dept	Public Relations	10000
5	Job	Accounting Manager	12000
6	Job	Administration Vice Pr...	17000
7	Job	Finance Manager	12000
8	Job	Marketing Manager	13000
9	Job	President	24000
10	Job	Public Relations Repre...	10000
11	Job	Purchasing Manager	11000
12	Job	Sales Manager	12200

PSEUDOCOLUMNS

Pseudocolumns

Oracle Pseudocolumns Overview

- Hierarchical Query Pseudocolumns
- Sequence Pseudocolumns
- Version Query Pseudocolumns
- COLUMN_VALUE Pseudocolumn
- OBJECT_ID Pseudocolumn
- OBJECT_VALUE Pseudocolumn
- ORA_ROWSCN Pseudocolumn
- **ROWID Pseudocolumn**
- **ROWNUM Pseudocolumn**
- XMLDATA Pseudocolumn

ROWNUM Pseudocolumn

```
SELECT ROWNUM, employee_id,
  first_name, last_name
FROM employees;
```

	ROWNUM	EMPLOYEE_ID	FIRST_NAME	LAST_NAME
1	1	174	Ellen	Abel
2	2	166	Sundar	Ande
3	3	130	Mozhe	Atkinson
4	4	105	David	Austin
5	5	204	Hermann	Baer
6	6	116	Shelli	Baida
7	7	167	Amit	Banda
8	8	172	Elizabeth	Bates
9	9	192	Sarah	Bell
10	10	151	David	Bernstein
11	11	129	Laura	Bissot
12	12	169	Harrison	Bloom

```
SELECT ROWNUM, employee_id,
  first_name, last_name
FROM employees
ORDER BY first_name, last_name;
```

	ROWNUM	EMPLOYEE_ID	FIRST_NAME	LAST_NAME
1	33	121	Adam	Fripp
2	104	196	Alana	Walsh
3	26	147	Alberto	Errazuriz
4	46	103	Alexander	Hunold
5	51	115	Alexander	Khoo
6	13	185	Alexis	Bull
7	68	158	Allan	McEwen
8	47	175	Alyssa	Hutton
9	7	167	Amit	Banda
10	14	187	Anthony	Cabrio
11	27	193	Britney	Everett
12	25	104	Bruce	Ernst

Isn't good idea if we need employee number into the list

ROWNUM Pseudocolumn

```
SELECT ROWNUM,
   first_name,
   last_name,
   salary
FROM employees
ORDER BY salary DESC;
```

	ROWNUM	FIRST_NAME	LAST_NAME	SALARY
1	1	Steven	King	24000
2	2	Neena	Kochhar	17000
3	3	Lex	De Haan	17000
4	46	John	Russell	14000
5	47	Karen	Partners	13500
6	102	Michael	Hartstein	13000
7	9	Nancy	Greenberg	12000
8	48	Alberto	Errazuriz	12000
9	106	Shelley	Higgins	12000
10	69	Lisa	Ozer	11500
11	75	Ellen	Abel	11000
12	49	Gerald	Cambrault	11000

```
SELECT ROWNUM, first_name,
   last_name,
   salary
FROM (
   SELECT first_name,
      last_name,
      salary
   FROM employees
   ORDER BY salary DESC
);
```

	ROWNUM	FIRST_NAME	LAST_NAME	SALARY
1	1	Steven	King	24000
2	2	Neena	Kochhar	17000
3	3	Lex	De Haan	17000
4	4	John	Russell	14000
5	5	Karen	Partners	13500
6	6	Michael	Hartstein	13000
7	7	Nancy	Greenberg	12000
8	8	Alberto	Errazuriz	12000
9	9	Shelley	Higgins	12000
10	10	Lisa	Ozer	11500
11	11	Ellen	Abel	11000
12	12	Gerald	Cambrault	11000

Limiting result set of SELECT query

```
SELECT ROWNUM, first_name, last_name, salary
FROM (
  SELECT first_name, last_name, salary
  FROM employees
  ORDER BY salary DESC
)
WHERE ROWNUM <= 5;
```

	ROWNUM	FIRST_NAME	LAST_NAME	SALARY
1	1	Steven	King	24000
2	2	Neena	Kochhar	17000
3	3	Lex	De Haan	17000
4	4	John	Russell	14000
5	5	Karen	Partners	13500

```
SELECT ROWNUM, first_name, last_name, salary
FROM (
  SELECT first_name, last_name, salary
  FROM employees
  ORDER BY salary DESC
)
WHERE ROWNUM BETWEEN 3 AND 5;
```

ROWNUM	FIRST_N...	LAST_N...	SALARY

ROWID Pseudocolumn

For each row in the database, the ROWID pseudocolumn returns the address of the row.

Oracle Database rowid values contain information necessary to locate a row:

- The data object number of the object
- The data block in the data file in which the row resides
- The position of the row in the data block (first row is 0)
- The data file in which the row resides (first file is 1). The file number is relative to the tablespace.

Rowid values have several important uses:

- *They are the fastest way to access a single row.*
- They can show you how the rows in a table are stored.
- They are unique identifiers for rows in a table.

ROWID Pseudocolumn

```
SELECT first_name,
  last_name,
  ROWID,
  DBMS_ROWID.ROWID_RELATIVE_FNO(ROWID) FILE_NO,
  DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) BLOCK_NO,
  DBMS_ROWID.ROWID_ROW_NUMBER(ROWID) ROW_NO
FROM employees
ORDER BY 4, 5, 6;
```

Data file Block Row

	FIRST_NAME	LAST_NAME	ROWID	FILE_NO	BLOCK_NO	ROW_NO
1	Steven	King	AAAXPXAAEAAAAD1AAA	4	245	0
2	Neena	Kochhar	AAAXPXAAEAAAAD1AAB	4	245	1
3	Lex	De Haan	AAAXPXAAEAAAAD1AAC	4	245	2
4	Alexander	Hunold	AAAXPXAAEAAAAD1AAD	4	245	3
5	Bruce	Ernst	AAAXPXAAEAAAAD1AAE	4	245	4
6	David	Austin	AAAXPXAAEAAAAD1AAF	4	245	5
7	Valli	Pataballa	AAAXPXAAEAAAAD1AAG	4	245	6
8	Diana	Lorentz	AAAXPXAAEAAAAD1AAH	4	245	7
9	Nancy	Greenberg	AAAXPXAAEAAAAD1AAI	4	245	8
10	Daniel	Faviet	AAAXPXAAEAAAAD1AAJ	4	245	9

How many blocks table actually occupies

```
SELECT
  COUNT(DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID)) BLOCKS_NUM
FROM employees;
```

	BLOCKS_NUM
1	2

SUBQUERIES

Define Subqueries

Subquery is a SELECT statement that is nested within another SQL statement.

SQL statements those accept subqueries:

- **DML: SELECT, INSERT, UPDATE, DELETE, MERGE**
- **DDL: CREATE TABLE and CREATE VIEW**

A SQL statement that includes a subquery as part of its code is considered the parent (or outer) to the subquery (or inner query).

A parent SQL statement may include one or more subqueries in its syntax.

Subqueries may have their own subqueries.

```
SELECT FIRST_NAME,
  LAST_NAME,
  SALARY
FROM EMPLOYEES
WHERE SALARY >= (SELECT MAX(SALARY)*0.7 FROM EMPLOYEES)
ORDER BY SALARY DESC;
```

FIRST_NAME	LAST_NAME	SALARY
Steven	King	24000
Lex	De Haan	17000
Neena	Kochhar	17000

Compare two queries

```
SELECT FIRST_NAME,
  LAST_NAME,
  SALARY
FROM EMPLOYEES
WHERE SALARY = (SELECT MAX(SALARY) FROM EMPLOYEES)
ORDER BY SALARY DESC;
```

```
SELECT *
FROM (
  SELECT FIRST_NAME,
    LAST_NAME,
    SALARY
  FROM EMPLOYEES
  ORDER BY SALARY DESC
)
WHERE ROWNUM=1;
```

- **Both return identical dataset**
- **The second causes a mistake when more than one person has the greatest salary**
- **Both use subqueries, but the nature of subqueries is different:**
  - In the first – **subquery acts like scalar** (just number – maximum salary value)
  - In the second – **subquery acts like dataset** (row source).

```
FIRST_NAME       LAST_NAME      SALARY
---------------- ----------- --------
Steven           King          24000
```

Subqueries Classification

- **Single-row subqueries**
  Return a single row in its result
- **Multiple-row subqueries**
  Return zero, one, or more rows
- **Multiple-column subqueries**
  Return more than one column in its result
- **Scalar subqueries**
  A single-row subquery consists of only one column
- **Correlated subqueries**
  Reference column(columns) from the parent query(queries)

```
SELECT FIRST_NAME,
  LAST_NAME,
  SALARY
FROM EMPLOYEES
WHERE SALARY = (
  SELECT MAX(SALARY)
  FROM EMPLOYEES
)
ORDER BY SALARY DESC;
```

```
SELECT *
FROM (
  SELECT FIRST_NAME,
    LAST_NAME,
    SALARY
  FROM EMPLOYEES
  ORDER BY SALARY DESC
)
WHERE ROWNUM=1;
```

Single-row subquery

```
SELECT EMP.FIRST_NAME,
  EMP.LAST_NAME,
  EMP.JOB_ID
FROM EMPLOYEES EMP
WHERE
  (SUBSTR(EMP.LAST_NAME,1,1), SUBSTR(EMP.LAST_NAME,2,1)) = (
    SELECT CHR(ROUND(AVG(ASCII(SUBSTR(LAST_NAME,1,1)))),
      CHR(MEDIAN(ASCII(SUBSTR(LAST_NAME,2,1))))
    FROM EMPLOYEES E
  );
```

FIRST_NAME	LAST_NAME	JOB_ID
Alexander	Khoo	PU_CLERK

```
SELECT EMP.FIRST_NAME,
  EMP.LAST_NAME,
  EMP.JOB_ID
FROM EMPLOYEES EMP
WHERE
  SUBSTR(EMP.LAST_NAME,1,1) || SUBSTR(EMP.LAST_NAME,2,1) = (
    SELECT CHR(ROUND(AVG(ASCII(SUBSTR(LAST_NAME,1,1)))) ||
      CHR(MEDIAN(ASCII(SUBSTR(LAST_NAME,2,1))))
    FROM EMPLOYEES E
  );
```

Multiple-row subqueries

```
SELECT MANAGERS.EMPLOYEE_ID, MANAGERS.FIRST_NAME,
  MANAGERS.LAST_NAME, MANAGERS.SALARY
FROM (
  SELECT E.EMPLOYEE_ID, E.FIRST_NAME, E.LAST_NAME, E.SALARY
  FROM DEPARTMENTS D
    JOIN EMPLOYEES E ON (D.MANAGER_ID = E.EMPLOYEE_ID)
) HEADS_OF_DEPTS JOIN (
  SELECT DISTINCT MGR.EMPLOYEE_ID, MGR.FIRST_NAME,
    MGR.LAST_NAME, MGR.SALARY
  FROM EMPLOYEES E
    JOIN EMPLOYEES MGR ON (E.MANAGER_ID = MGR.EMPLOYEE_ID)
) MANAGERS
ON (HEADS_OF_DEPTS.EMPLOYEE_ID = MANAGERS.EMPLOYEE_ID);

SELECT E.EMPLOYEE_ID, E.FIRST_NAME, E.LAST_NAME, E.SALARY
FROM DEPARTMENTS D
  JOIN EMPLOYEES E ON (D.MANAGER_ID = E.EMPLOYEE_ID)
INTERSECT
SELECT DISTINCT MGR.EMPLOYEE_ID, MGR.FIRST_NAME,
  MGR.LAST_NAME, MGR.SALARY
FROM EMPLOYEES E
  JOIN EMPLOYEES MGR ON (E.MANAGER_ID = MGR.EMPLOYEE_ID);
```

JOIN-based equivalent

```
SELECT DISTINCT
  MGR.EMPLOYEE_ID, MGR.FIRST_NAME, MGR.LAST_NAME, MGR.SALARY
FROM (
    DEPARTMENTS D
    JOIN EMPLOYEES DMGR ON (D.MANAGER_ID = DMGR.EMPLOYEE_ID))
  JOIN (
    EMPLOYEES E
    JOIN EMPLOYEES MGR ON (E.MANAGER_ID = MGR.EMPLOYEE_ID))
  ON DMGR.EMPLOYEE_ID = MGR.EMPLOYEE_ID
;
```

MPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
100	Steven	King	24000
145	John	Russell	14000
201	Michael	Hartstein	13000
108	Nancy	Greenberg	12008
205	Shelley	Higgins	12008
114	Den	Raphaely	11000
103	Alexander	Hunold	9000
121	Adam	Fripp	8200

Comparison conditions for multiple-row subqueries

1. **IN. Compares a subject value to a set of values. Returns TRUE if the subject value equals any of the values in the set. Returns FALSE if the subquery returns no rows.**
2. **NOT IN. NOT used with IN to reverse the result. Returns TRUE if the subquery returns no rows.**
3. **EXISTS. An EXISTS condition tests for existence of rows in a subquery. Returns TRUE if a subquery returns at least one row.**

Employees with maximum salaries by jobs (ANY, IN)

```
-- ORA-01427: single-row subquery returns more than one row
SELECT FIRST_NAME, LAST_NAME, SALARY
FROM EMPLOYEES
WHERE
  SALARY = (SELECT MAX(SALARY) FROM EMPLOYEES GROUP BY JOB_ID)
ORDER BY SALARY DESC;


-- Invalid logic
SELECT FIRST_NAME, LAST_NAME, SALARY
FROM EMPLOYEES
WHERE
  SALARY IN (SELECT MAX(SALARY) FROM EMPLOYEES GROUP BY JOB_ID)
ORDER BY SALARY DESC;
```

CRUD

TESTEMP table

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000010	HAAS	A00	1965-01-01	52750.00	1000.00
000020	THOMPSON	B01	1973-10-10	41250.00	800.00
000030	KWAN	C01	1975-04-04	38250.00	800.00
000050	GEYER	E01	1949-08-17	40175.00	800.00
000111	SMITH	C01	1998-06-25	25000.00	-

Create Table

```
CREATE TABLE TESTEMP (
    EMPNO CHAR(6) NOT NULL,
    LASTNAME VARCHAR(15) NOT NULL,
    WORKDEPT CHAR(3),
    HIREDATE DATE,
    SALARY DECIMAL(9,2),
    BONUS DECIMAL(9,2)
);
```

Inserting Rows to Table

```
INSERT INTO TESTEMP
VALUES ('000111', 'SMITH', 'C01', '1998-06-25', 25000, NULL);
```

OR

```
INSERT INTO TESTEMP
(EMPNO, LASTNAME, WORKDEPT, HIREDATE, SALARY)
VALUES ('000111', 'SMITH', 'C01', '1998-06-25', 25000);
```

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000111	SMITH	C01	1998-06-25	25000.00	-

UPDATE multiple rows

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000111	SMITH	C01	1998-06-25	25000.00	-
000010	HAAS	A00	1965-01-01	52750.00	1000.00
000020	THOMPSON	B01	1973-10-10	41250.00	800.00
000030	KWAN	C01	1975-04-05	38250.00	800.00
000050	GEYER	E01	1949-08-17	40175.00	800.00

```
UPDATE TESTEMP
SET BONUS = 500,
SALARY = 26000
WHERE EMPNO = '000111' ;
```

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000111	SMITH	C01	1998-06-25	26000.00	500.00
000010	HAAS	A00	1965-01-01	52750.00	1000.00
000020	THOMPSON	B01	1973-10-10	41250.00	800.00
000030	KWAN	C01	1975-04-05	38250.00	800.00
000050	GEYER	E01	1949-08-17	40175.00	800.00

UPDATE multiple rows

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000111	SMITH	C01	1998-06-25	26000.00	500.00
000010	HAAS	A00	1965-01-01	52750.00	1000.00
000020	THOMPSON	B01	1973-10-10	41250.00	800.00
000030	KWAN	C01	1975-04-05	38250.00	800.00
000050	GEYER	E01	1949-08-17	40175.00	800.00

```
UPDATE TESTEMP
SET SALARY = SALARY + 1000
WHERE WORKDEPT = 'C01' ;
```

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000111	SMITH	C01	1998-06-25	27000.00	500.00
000010	HAAS	A00	1965-01-01	52750.00	1000.00
000020	THOMPSON	B01	1973-10-10	41250.00	800.00
000030	KWAN	C01	1975-04-05	39250.00	800.00
000050	GEYER	E01	1949-08-17	40175.00	800.00

Delete Rows

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000111	SMITH	C01	1998-06-25	27000.00	500.00
000010	HAAS	A00	1965-01-01	52750.00	1000.00
000020	THOMPSON	B01	1973-10-10	41250.00	800.00
000030	KWAN	C01	1975-04-05	39250.00	800.00
000050	GEYER	E01	1949-08-17	40175.00	800.00

```
DELETE FROM TESTEMP
WHERE EMPNO = '000111' ;
```

EMPNO	LASTNAME	WORKDEPT	HIREDATE	SALARY	BONUS
000010	HAAS	A00	1965-01-01	52750.00	1000.00
000020	THOMPSON	B01	1973-10-10	41250.00	800.00
000030	KWAN	C01	1975-04-05	39250.00	800.00
000050	GEYER	E01	1949-08-17	40175.00	800.00

Drop table

DROP TABLE TESTEMP ;

Oracle Stored Procedure & Functions

What can you do with PL/SQL?

- Allows sophisticated data processing
- Build complex business logic in a modular fashion
- Use over and over
- Execute rapidly – little network traffic
  - Stored procedures
  - Functions
  - Triggers

Stored Procedures

- Defined set of actions written using PL/SQL
- When called, the procedure performs actions
- Can be called directly from other blocks
- Two parts
  - Procedure specification or header
  - Procedure body

PROCEDURES

- A procedure is a module performing one or more actions; it does not need to return any values.
- The syntax for creating a procedure is as follows:

```
CREATE OR REPLACE PROCEDURE name
  [(parameter[, parameter, ...])]
AS
  [local declarations]
BEGIN
  executable statements
[EXCEPTION
  exception handlers]
END [name];
```

PROCEDURES

- A procedure may have 0 to many parameters.
- Every procedure has two parts:
  1. The header portion, which comes before AS (sometimes you will see IS—they are interchangeable), keyword (this contains the procedure name and the parameter list),
  2. The body, which is everything after the IS keyword.
- The word REPLACE is optional.
- When the word REPLACE is not used in the header of the procedure, in order to change the code in the procedure, it must be dropped first and then re-created.

Example: Procedure

```
CREATE OR REPLACE PROCEDURE hello IS
   Greetings VARCHAR(20);
BEGIN
     Greetings:= 'Hello World';
     DBMS_OUTPUT.PUT_LINE(greetings);
END hello;
```

Example: Procedure

```
CREATE OR REPLACE PROCEDURE Discount
AS
   CURSOR c_group_discount
   IS
        SELECT distinct s.course_no, c.description
        FROM section s,enrollment e,course c
        WHERE s.section_id = e.section_id
             AND c.course_no = s.course_no
   GROUP BY s.course_no, c.description,
             e.section_id, s.section_id
   HAVING COUNT(*) >=8;
BEGIN
FOR r_group_discount IN c_group_discount
   LOOP
        UPDATE course
             SET cost = cost * .95
        WHERE course_no = r_group_discount.course_no;
        DBMS_OUTPUT.PUT_LINE
             ('A 5% discount has been given to'||
             r_group_discount.course_no||' '||
             r_group_discount.description
             );
   END LOOP;
END;
```

Calling a Procedure

Execute it is from another PL/SQL block:

```
BEGIN
      hello;
END;
```

Arguments

- A value can be passed to a procedure when it is called (input)
- Must specify datatype
- Example (not actually a procedure):

```
increase_salary_find_tax(
increase_percent   IN          NUMBER:=7,
sal                IN OUT      NUMBER,
tax                 OUT        NUMBER)
```

- IN means the procedure can read an incoming value from that parameter when the procedure is called
- OUT means the procedure can use that parameter to send a value back to what called it
- increase_percent has a default value of 7

Arguments

- Following is a procedure with arguments:

```
CREATE OR REPLACE PROCEDURE increase
 (oldprice NUMBER, percent NUMBER := 5,
 newprice OUT NUMBER)
IS
BEGIN

 newprice:=oldprice+oldprice*percent/100;
END increase;
```

Calling a Procedure with Arguments

```
DECLARE
    price_increase NUMBER(6,2) := 20;
    newp NUMBER(6,2) := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Current price: '|| price_increase);
    increase(oldprice=>price_increase,newprice=>newp);
    DBMS_OUTPUT.PUT_LINE
          ('Price after increase: '|| newp);
END;
```

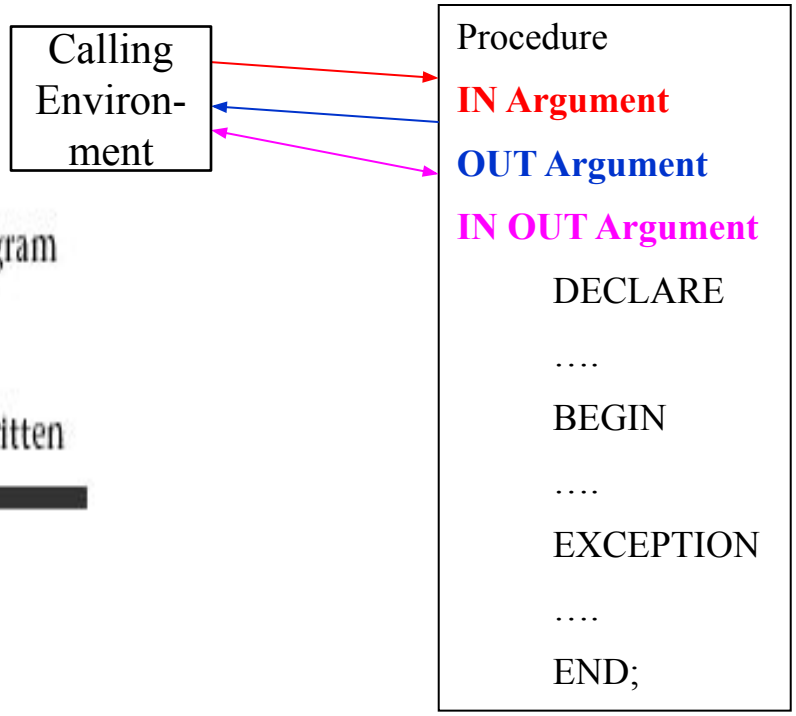
We should see a new price of 21

PARAMETERS

- Parameters are the means to pass values to and from the calling environment to the server.
- These are the values that will be processed or returned via the execution of the procedure.
- There are three types of parameters:
- IN, OUT, and IN OUT.
- Modes specify whether the parameter passed is read in or a receptacle for what comes out.

Types of Parameters

Mode	Description	Usage
IN	Passes a value into the program	Read only value Constants, literals, expressions Cannot be changed within program Default mode
OUT	Passes a value back from the program	Write only value Cannot assign default values Has to be a variable Value assigned only if the program is successful
IN OUT	Passes values in and also send values back	Has to be a variable Value will be read and then written



FORMAL AND ACTUAL PARAMETERS

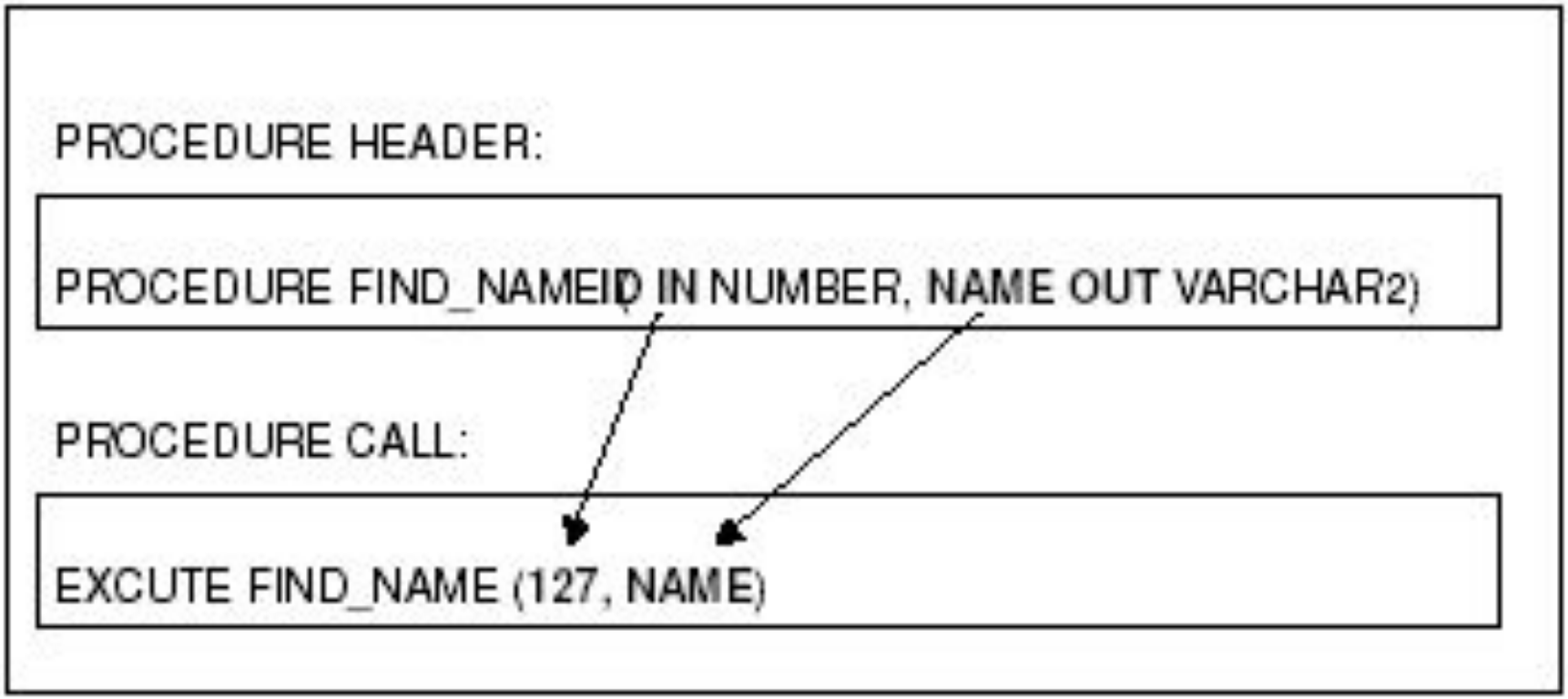
- *Formal parameters* are the names specified within parentheses as part of the header of a module.
- *Actual parameters* are the values—expressions specified within parentheses as a parameter list—when a call is made to the module.
- The formal parameter and the related actual parameter must be of the same or compatible data types.

MATCHING ACTUAL AND FORMAL PARAMETERS

- Two methods can be used to match actual and formal parameters: positional notation and named notation.
- *Positional notation* is simply association by position: The order of the parameters used when executing the procedure matches the order in the procedure's header exactly.
- *Named notation* is explicit association using the symbol =>
  - `Syntax: formal_parameter_name => argument_value`

- In named notation, the order does not matter.
- If you mix notation, list positional notation before named notation.

MATCHING ACTUAL AND FORMAL PARAMETERS



FUNCTIONS

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL block that *returns* a single value.
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.
- The datatype of the return value must be declared in the header of the function.
- A function is not a stand-alone executable in the way that a procedure is: It must be used in some context. You can think of it as a sentence fragment.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

FUNCTIONS

- The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
  (parameter list)
  RETURN datatype
IS
BEGIN
  <body>
  RETURN (return_value);
END;
```

FUNCTIONS

- The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.
- The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).
- A function may have IN, OUT, or IN OUT parameters. but you rarely see anything except IN parameters.

Example

```
CREATE OR REPLACE FUNCTION show_description
     (i_course_no number)
RETURN varchar2
AS
     v_description varchar2(50);
BEGIN
     SELECT description
            INTO v_description
            FROM course
            WHERE course_no = i_course_no;
     RETURN v_description;
EXCEPTION
     WHEN NO_DATA_FOUND
     THEN
            RETURN('The Course is not in the database');
     WHEN OTHERS
     THEN
            RETURN('Error in running show_description');
END;
```

Making Use Of Functions

- **In a anonymous block**

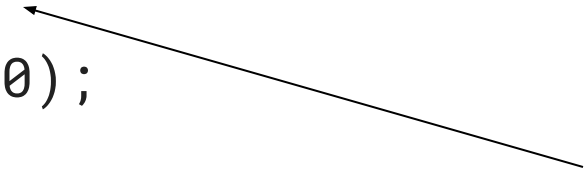
```
SET SERVEROUTPUT ON
DECLARE
  v_description VARCHAR2(50);
BEGIN
  v_description := show_description(&sv_cnumber);
  DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

- **In a SQL statement**

```
SELECT course_no, show_description(course_no)
  FROM course;
```

Example

```
CREATE OR REPLACE FUNCTION discount (amount
 NUMBER, percent NUMBER:=5)
RETURN NUMBER
IS
BEGIN
   IF (amount>=0) THEN
          return (amount*percent/100);
   ELSE
        return(0);
   END IF;
END discount;
```



The IF-THEN construct allows for error checking

Example : Calling the Function

```
DECLARE
       current_amt NUMBER:=100;
    incorrect_amt NUMBER:=-5;
BEGIN
   DBMS_OUTPUT.PUT_LINE(' Order and Discount');
    DBMS_OUTPUT.PUT_LINE(current_amt || '       '||
 discount(current_amt));
    DBMS_OUTPUT.PUT_LINE(incorrect_amt||'
 '||discount(incorrect_amt));
END;
```

Example

- Write a PL/SQL function that accepts price and onhand values, checks to be sure they are both greater than 0 and multiplies them together. If they are less than 0 return 0.

```
CREATE OR REPLACE FUNCTION total_amount (price NUMBER,
  onhand NUMBER)
RETURN NUMBER IS
BEGIN
    IF (price>0 AND onhand>0) THEN
           return (price*onhand);
    ELSE
         return(0);
    END IF;
END total_amount;
```

Ms SQL Server Stored Procedure & Functions

- In SQL Server, many administrative and informational activities can be performed by using system stored procedures.

- System stored procedures are prefixed by sp_, so it is not advisable to use sp_ for any of the stored procedures that we create, unless they form a part of our SQL Server installation.

- Stored procedures can be:
  - system / sp_ help …; sp_helptext …./
  - local
  - temporary
  - remote
  - extended

Stored procedures, user-defined functions, and prepared statements

- A stored procedure is a collection of SQL statements that can be called via a CALL statement.

- A user-defined function is also a collection of SQL statements, but it can be called and used like any Built-in function

- A prepared statement is a query that is stored on the server and that can be executed in the future

Stored procedures

- Stored procedures must be declared before they can be called.
- The declaration can include parameters.
- If parameters are changed inside the procedure, their modified values are accessible after the call.

Creating a simple stored procedure

To create a stored procedure to do this the code would look like this:

```
CREATE PROCEDURE getCountries
AS
select * from sample.dbo.countries
GO
```

To call the procedure to return the contents from the table specified, the code would be:

```
exec getCountries

--or just simply

getCountries
```

How to create a SQL Server stored procedure with parameters

The real power of stored procedures is the ability to pass parameters and have the stored procedure handle the differing requests that are made.

```
CREATE OR ALTER PROCEDURE getCountries2 @CountryId
char(2)
AS
begin
select *, @CountryId from sample.dbo.countries
where country_id = @CountryId
end
```

```
EXEC getCountries2 @CountryId = 'AR'
```

```
CREATE OR ALTER PROCEDURE getCountries3
@CountryName VARCHAR(40)
AS
begin
select * from sample.dbo.countries where
country_name like @CountryName+'%'
end
```

Default Parameter Values

- In most cases it is always a good practice to pass in all parameter values, but sometimes it is not possible. So in this example we use the NULL option to allow you to not pass in a parameter value.

- If we create and run this stored procedure as is it will not return any data, because it is looking for any City values that equal NULL.

```
CREATE OR ALTER PROCEDURE getCountries4 @CountryId
char(2) = NULL
AS
begin
select *, @CountryId from sample.dbo.countries
where country_id = @CountryId
end
```

We could change this procedure and use the ISNULL function to get around this.
So if a value is passed it will use the value to narrow the result set and if a value is not passed it will return all records.

```
CREATE OR ALTER PROCEDURE getCountries4 @CountryId
char(2) = NULL
AS
begin
select *, @CountryId from sample.dbo.countries where
country_id = ISNULL(@CountryId, country_id )
end
```

Multiple Parameters

- Setting up multiple parameters is very easy to do. You just need to list each parameter and the data type separated by a comma as shown below.

```
CREATE OR ALTER PROCEDURE getCountries5 @RegionId int = NULL,
@CountryId char(2) = NULL
AS
begin
select *, @CountryId, @RegionId
from sample.dbo.countries
where
    region_id = ISNULL(@RegionId, region_id)
    and country_id = ISNULL(@CountryId, country_id )
end
```

To execute this you could do any of the following:

```
EXEC getCountries5 @RegionId = 2
--or
EXEC getCountries5 @RegionId = 2, @CountryId = 'AR'
```

Returning stored procedure parameter values to a calling stored procedure

- **Overview**
  In a previous topic we discussed how to pass parameters into a stored procedure, but another option is to pass parameter values back out from a stored procedure.

  One option for this may be that you call another stored procedure that does not return any data, but returns parameter values to be used by the calling stored procedure.

- **Explanation**
  Setting up output paramters for a stored procedure is basically the same as setting up input parameters, the only difference is that you use the OUTPUT clause after the parameter name to specify that it should return a value.

  The output clause can be specified by either using the keyword "OUTPUT" or just "OUT".

Simple Output

```
CREATE PROCEDURE uspGetAddressCount @City nvarchar(30),
@AddressCount int OUTPUT
AS
SELECT @AddressCount = count(*)
FROM AdventureWorks.Person.Address
WHERE City = @City
```

To call this stored procedure we would execute it as follows. First we are going to declare a variable, execute the stored procedure and then select the returned valued.

```
DECLARE @CountriesCount int
EXEC getCountriesCount @CountriesCount = @CountriesCount
OUTPUT
SELECT @CountriesCount
```

Deleting a SQL Server stored procedure

- **Overview**
  In addition to creating stored procedures there is also the need to delete stored procedures. This topic shows you how you can delete stored procedures that are no longer needed.
- **Explanation**
  The syntax is very straightforward to drop a stored procedure, here are some examples.

Dropping Single Stored Procedure

To drop a single stored procedure you use the DROP PROCEDURE or DROP PROC command as follows.

```
DROP PROCEDURE  getCountriesCount
```