



SQL: Structure Query Language

D. J. Berndt
College of Business
University of South Florida
dberndt@usf.edu



SQL Landscape

SQL Category	Description
DML: Data Manipulation Language	<p>DML statements are used to retrieve, store and modify data. DML can be further sub-divided into read and write statements. In fact, the 80/20 rule is often a good guideline for the read/write ratio.</p> <p>DML-Read: SELECT (SELECT-FROM-WHERE)</p> <p>DML-Write: INSERT, UPDATE, DELETE, and TRUNCATE</p>
DDL: Data Definition Language	<p>DDL statements are used to create and modify the schema or structure of the database. Example statements: CREATE, ALTER, and DROP</p>
DCL: Data Control Language	<p>DCL statements are used to manage database access rights through roles and permissions. Example statements: GRANT and REVOKE</p>
TCL: Transaction Control Language	<p>TCL statements are used to manage database transactions. Example statements: COMMIT, ROLLBACK, and SAVEPOINT</p>



SQL Clauses

Oracle SQL Developer : ReIMDB on CDB1~1

Start Page | ReIMDB on CDB1 | ReIMDB on CDB1~1

Worksheet | Query Builder

```
SELECT
  genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY genre
HAVING COUNT(*) > 5
ORDER BY COUNT(*) DESC;
```

Query Result

All Rows Fetched: 10 in 0.032 seconds

GENRE	FILM_COUNT
1 Drama	27
2 Crime	12
3 Thriller	12
4 Adventure	10
5 Action	9
6 War	7
7 Comedy	7
8 Mystery	6

Log

ReIMDB on CDB1 | Line 1 Column 7 | Insert | Modified | Uni

-- SQL DML-Read Clauses

SELECT

-- one or more attributes (columns)

FROM

-- one or more relations (tables)

WHERE

-- each tuple (row) meets the specified conditions

GROUP BY

-- one or more attributes

HAVING

-- each grouping meets the specified conditions

ORDER BY

-- one or more attributes



Aggregate Functions

There are five basic aggregate functions in SQL:

1. COUNT(x)
2. SUM(x)
3. AVG(x)
4. MIN(x)
5. MAX(x)

```
SELECT
    AVG(runtime) AS avg_runtime
FROM
    movies
WHERE
    imdb_rank <= 100;
```

This query returns a single record with only the average runtime. Note: NULL values do not participate in aggregate functions (or other arithmetic calculations).

We have already used a couple of these aggregate functions.



GROUP BY

- Interesting reports can often be formed by dynamically grouping the result set. Any attributes not in the GROUP BY must be used as a parameter in an aggregate function.

```
SELECT
  genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY genre;
```

Oracle SQL Developer : ReIMDB on CDB1

Query Builder

```
SELECT
  genre,
  COUNT(*) AS film_count
FROM
  movies
  INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY
  genre;
```

Query Result

All Rows Fetched: 26 in 0.072 seconds

GENRE	FILM_COUNT
1 Sci-Fi	6
2 Mystery	6
3 Comedy	7
4 Sci-fi	2
5 Animation	2
6 Family	2
7 Biography	2
8 Action	1
9 Thriller	12
10 Film-Noir	3

Log

Table RELMDB.GENRES@ReIMDB on CDB1 | Line 1 Column 7 | Insert | Modified | Editing



Detour: Data Quality

- The repeated and inconsistent values for genres can lead to data quality issues.
- This is a big reason for normalization!
- We can clean up the results with some string processing functions.

LOWER(s) – Lower case of s
UPPER(s) – Upper case of s
LTRIM(s,t) – Left Trim t from s
RTRIM(s,t) – Right trim t from s

Oracle SQL Developer : RelMDB on CDB1

Start Page | RelMDB on CDB1 | GENRES

Worksheet | Query Builder

```
SELECT
genre,
COUNT(*) AS film_count
FROM
movies
INNER JOIN genres
ON movies.film_id = genres.film_id
GROUP BY
genre;
```

Query Result

All Rows Fetched: 26 in 0.072 seconds

GENRE	COUNT
1 Sci-Fi	6
2 Mystery	6
3 Comedy	7
4 Sci-fi	2
5 Animation	2
6 Family	2
7 Biography	2
8 Action	1
9 Thriller	12
10 Film-Noir	3

Log

Table RELMDB.GENRES@RelMDB on CDB1 | Line 1 Column 7 | Insert | Modified | Editing



Detour: Data Quality

- The results look much better after trimming and standardizing the case.
- A normalized GENRES table makes good sense.

```
SELECT
  LOWER(LTRIM(RTRIM(genre))) AS genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY LOWER(LTRIM(RTRIM(genre)))
ORDER BY COUNT(*) DESC;
```

Oracle SQL Developer : ReIMDB on CDB1

Worksheet Query Builder

```
SELECT
  LOWER(LTRIM(RTRIM(genre))) AS genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY LOWER(LTRIM(RTRIM(genre)))
ORDER BY COUNT(*) DESC;
```

Query Result

All Rows Fetched: 17 in 0.333 seconds

GENRE	FILM_COUNT
1 drama	28
2 crime	13
3 thriller	12
4 action	11
5 adventure	11
6 sci-fi	9
7 comedy	7
8 war	7
9 romance	6
10 mystery	6

Log

Line 1 Column 7 | Insert | Modified | Editing



GROUP BY for Histograms

- Histograms and other data aggregations can be created using a GROUP BY.

```
SELECT
  ROUND(runtime, -1) AS approx_runtime,
  COUNT(*) AS film_count
FROM
  movies
WHERE
  runtime IS NOT NULL
GROUP BY
  ROUND(runtime, -1)
ORDER BY
  ROUND(runtime, -1) DESC;
```

The screenshot shows the Oracle SQL Developer interface with the query executed. The 'Query Result' tab is active, displaying a table with 11 rows. The columns are 'APPROX_RUNTIME' and 'FILM_COUNT'. The data is sorted in descending order of approximate runtime.

	APPROX_RUNTIME	FILM_COUNT
1	200	3
2	180	4
3	170	3
4	160	1
5	150	8
6	140	7
7	130	5
8	120	8



HAVING

- Conditions can be applied to the groups using the HAVING clause.
- Akin to the WHERE clause.

```
SELECT
  genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY genre
HAVING COUNT(*) > 5
ORDER BY COUNT(*) DESC;
```

Oracle SQL Developer : RelMDB on CDB1~1

Query Builder

```
SELECT
  genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY genre
HAVING COUNT(*) > 5
ORDER BY COUNT(*) DESC;
```

Query Result

All Rows Fetched: 10 in 0.032 seconds

GENRE	FILM_COUNT
1 Drama	27
2 Crime	12
3 Thriller	12
4 Adventure	10
5 Action	9
6 War	7
7 Comedy	7
8 Mystery	6

Log

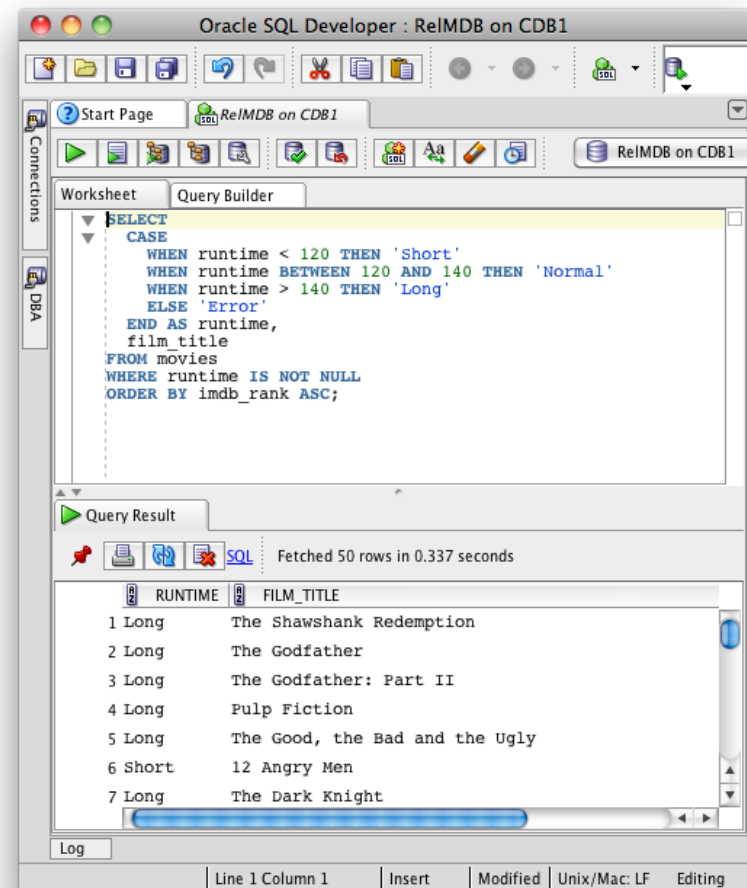
RelMDB on CDB1 | Line 1 Column 7 | Insert | Modified | Unix/Mac: LF | Editing



CASE Statement

- The CASE statement provides a succinct method of including if-then-else logic in a query.

```
CASE [expression]
  WHEN condition_1 THEN result_1
  WHEN condition_2 THEN result_2
  ...
  WHEN condition_n THEN result_n
  ELSE result
END
```





CASE Statement

Oracle SQL Developer : ReIMDB on CDB1

Start Page ReIMDB on CDB1

Worksheet Query Builder

```
SELECT
CASE
  WHEN runtime < 120 THEN 'Short'
  WHEN runtime BETWEEN 120 AND 140 THEN 'Normal'
  WHEN runtime > 140 THEN 'Long'
  ELSE 'Error'
END AS runtime,
film_title
FROM movies
WHERE runtime IS NOT NULL
ORDER BY imdb_rank ASC;
```

Query Result

Fetches 50 rows in 0.337 seconds

	RUNTIME	FILM_TITLE
1	Long	The Shawshank Redemption
2	Long	The Godfather
3	Long	The Godfather: Part II
4	Long	Pulp Fiction
5	Long	The Good, the Bad and the Ugly
6	Short	12 Angry Men
7	Long	The Dark Knight

Log

Line 1 Column 1 | Insert | Modified | Unix/Mac: LF | Editing

```
SELECT
CASE
  WHEN runtime < 120 THEN 'Short'
  WHEN runtime BETWEEN 120 AND 140 THEN 'Normal'
  WHEN runtime > 140 THEN 'Long'
  ELSE 'Error'
END AS runtime,
film_title
FROM movies
WHERE runtime IS NOT NULL
ORDER BY imdb_rank ASC;
```



Views

- Database views are “virtual relations” defined by stored queries.
- While this virtual relation “contains” the rows specified by any query conditions, nothing is pre-computed and materialized, the results are computed at query time.
- Views are can be used anywhere a normal relation is used.
- Views are useful for delivering only the columns and rows necessary to particular users for convenience and security reasons.

```
CREATE VIEW <virtual relation name> AS  
SELECT ...
```



Views

Create a view of only the “thriller” films with only a few columns.

The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL code:

```
CREATE VIEW thrillers AS
SELECT
  movies.film_id,
  movies.film_year,
  genres.genre
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
WHERE genres.genre LIKE 'Thriller'
ORDER BY film_id;

SELECT * FROM thrillers;
```

The 'Query Result' tab shows the results of the query, displaying 12 rows of data. The columns are FILM_ID, FILM_YEAR, and GENRE.

FILM_ID	FILM_YEAR	GENRE
1	4	1994 Thriller
2	14	2010 Thriller
3	15	1990 Thriller
4	31	1994 Thriller
5	34	1998 Thriller
6	36	1991 Thriller
7	51	1958 Thriller

```
CREATE VIEW thrillers AS
SELECT
  movies.film_id,
  movies.film_year,
  genres.genre
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
WHERE genres.genre LIKE 'Thriller'
ORDER BY film_id;
```



Materialized Views

- Some database systems (such as the Oracle DBMS) support **materialized views**.
- Essentially, a view can be defined as before, but the result is pre-computed and stored like a physical relation.
- Queries against a materialized view are much faster since the underlying query has already been executed.
- However, this performance comes at a cost since there must now be some type of “view maintenance” that keeps the underlying relations synchronized with the view. That is, when the underlying relations change, the materialized view must be updated. The “tightness” of this coupling must be chosen by the database designer!

INSERT, UPDATE, and DELETE



- The basic syntax is shown below.

-- Insert statement

```
INSERT INTO table [(col1, col2, ...)]  
VALUES (value1, value2, ...);
```

-- Update statement

```
UPDATE table  
SET column = 'value'  
[WHERE condition];
```

-- Delete statement

```
DELETE FROM table  
[WHERE condition];
```

INSERT INTO casts

```
(film_id, cast_member, cast_role)  
VALUES (117, 'Tim Allen', 'Buzz');
```

UPDATE casts

```
SET cast_role = 'Buzz Lightyear'  
WHERE film_id = 117  
AND cast_member = 'Tim Allen';
```

DELETE FROM casts

```
WHERE film_id = 117  
AND cast_member = 'Tim Allen';
```

UPDATE and DELETE assume a cast member only plays one role.
How can we modify the statements if this is not true?



INSERT INTO

The screenshot shows the Oracle SQL Developer interface with the title 'Oracle SQL Developer : MED00 on CDB1'. The 'Query Builder' tab is active, displaying the following SQL script:

```
TRUNCATE TABLE movies;  
  
INSERT INTO movies  
SELECT * from relmdb.movies;  
  
SELECT * FROM movies;|
```

Below the script, the 'Query Result' tab shows the output of the query. It indicates 'Fetched 100 rows in 0.215 seconds'. The results are displayed in a table with three columns: IMDB_RANK, IMDB_RATING, and FILM_TITLE.

IMDB_RANK	IMDB_RATING	FILM_TITLE
1	176	8.1 Stand by Me
2	177	8.1 Donnie Darko
3	178	8.1 Groundhog Day
4	179	8.1 Twelve Monkeys
5	180	8.1 Dog Day Afternoon
6	181	8.1 Black Swan
7	182	8.1 Amores Perros

The status bar at the bottom indicates 'Table MED00.MOVIES@MED00 on CDB1 | Line 6 Column 22 | Insert | Modified | Unix/Mac: LF | Editing'.

INSERT INTO can take the results of a query and easily add it to an existing table. Usually the columns are explicitly listed!

INSERT INTO movies
SELECT * from relmdb.movies;



Transaction Control Language (TCL)

- A transaction is the fundamental unit of work in a database system consisting of one or more SQL statements (more on this later).
- After modifying data using INSERT, UPDATE, or DELETE as part of a transaction, the final disposition of work can be specified via a basic TCL statement.
 - ❑ COMMIT – Instructs the database to complete and log the work for persistent storage.
 - ❑ ROLLBACK – Instructs the database to “undo” any previously uncommitted changes.
- Client side tools usually have a default behavior with regard to COMMIT or ROLLBACK upon exit.