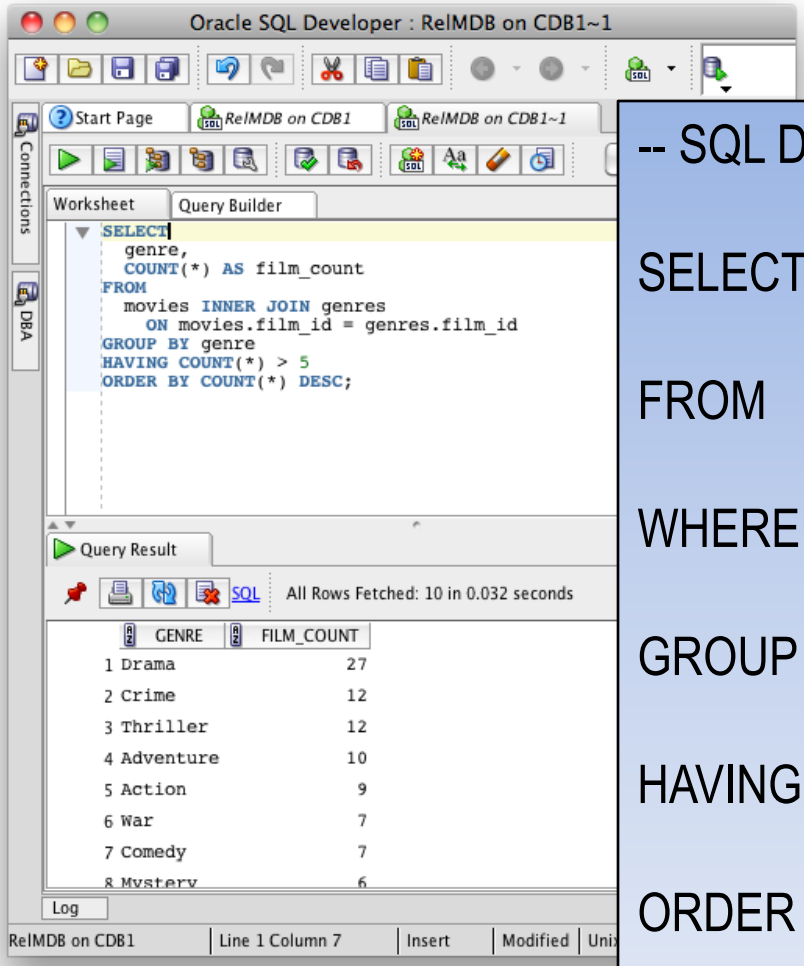
SQL:
Structure Query Language

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

SQL Landscape

SQL Category	Description
DML: Data Manipulation Language	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. DML-Read: SELECT (SELECT-FROM-WHERE) DML-Write: INSERT, UPDATE, DELETE, and TRUNCATE
DDL: Data Definition Language	DDL statements are used to create and modify the schema or structure of the database. Example statements: CREATE, ALTER, and DROP
DCL: Data Control Language	DCL statements are used to manage database access rights through roles and permissions. Example statements: GRANT and REVOKE
TCL: Transaction Control Language	TCL statements are used to manage database transactions. Example statements: COMMIT, ROLLBACK, and SAVEPOINT

SQL Clauses



-- 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;
```

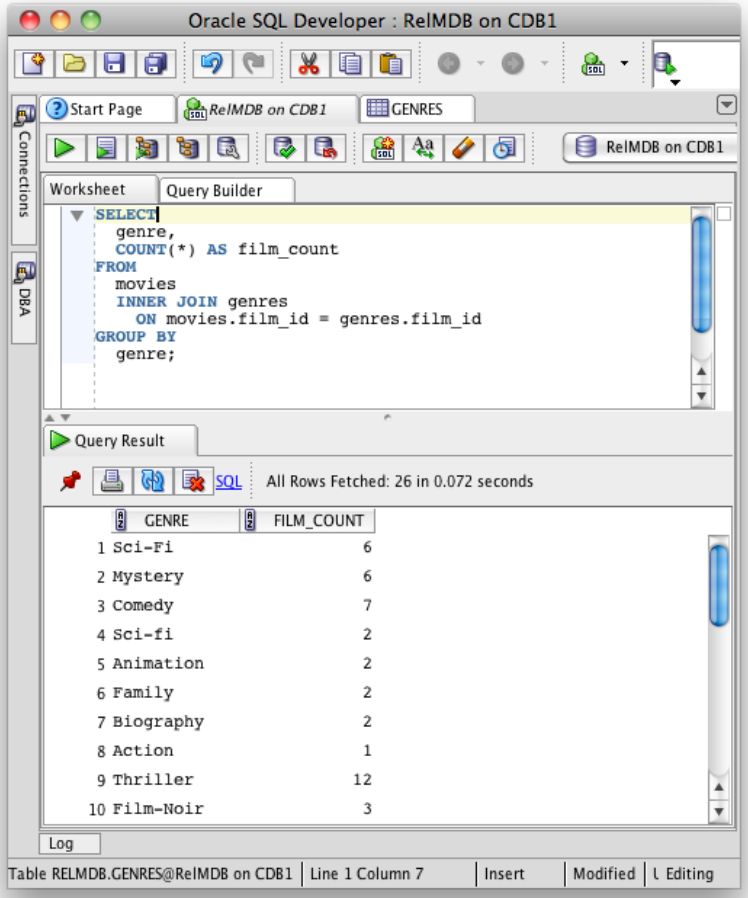
This query returns a single record with only the average runtime. Note: NULL values do not participate in aggregate functions (or other arithmetic calculations).

We have already used a couple of these aggregate functions.

GROUP BY

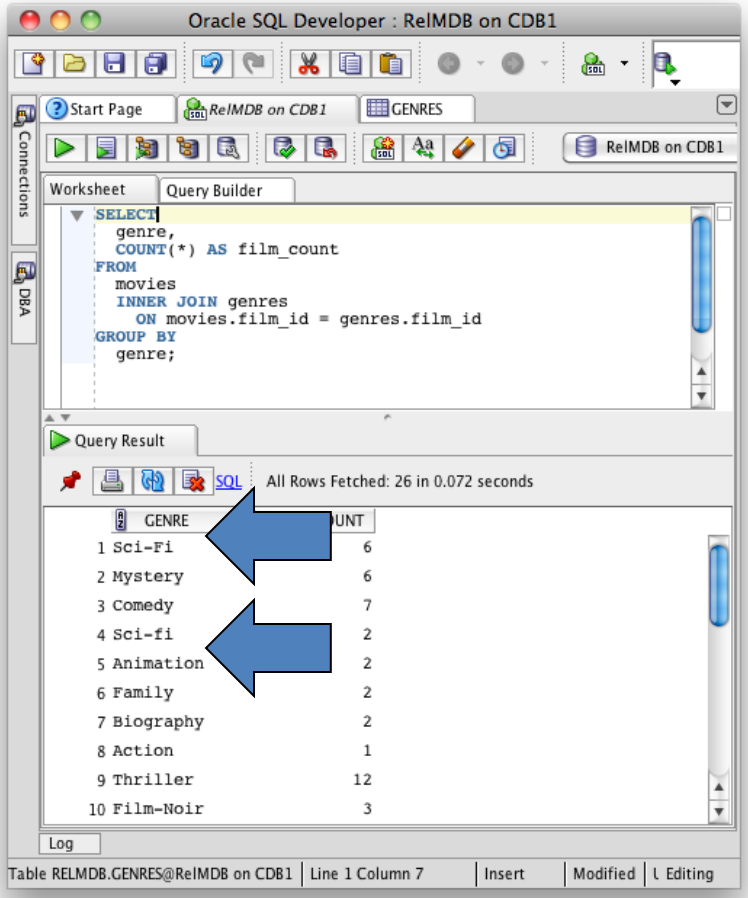
- Interesting reports can often be formed by dynamically grouping the result set. Any attributes not in the GROUP BY must be used as a parameter in an aggregate function.

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



Detour: Data Quality

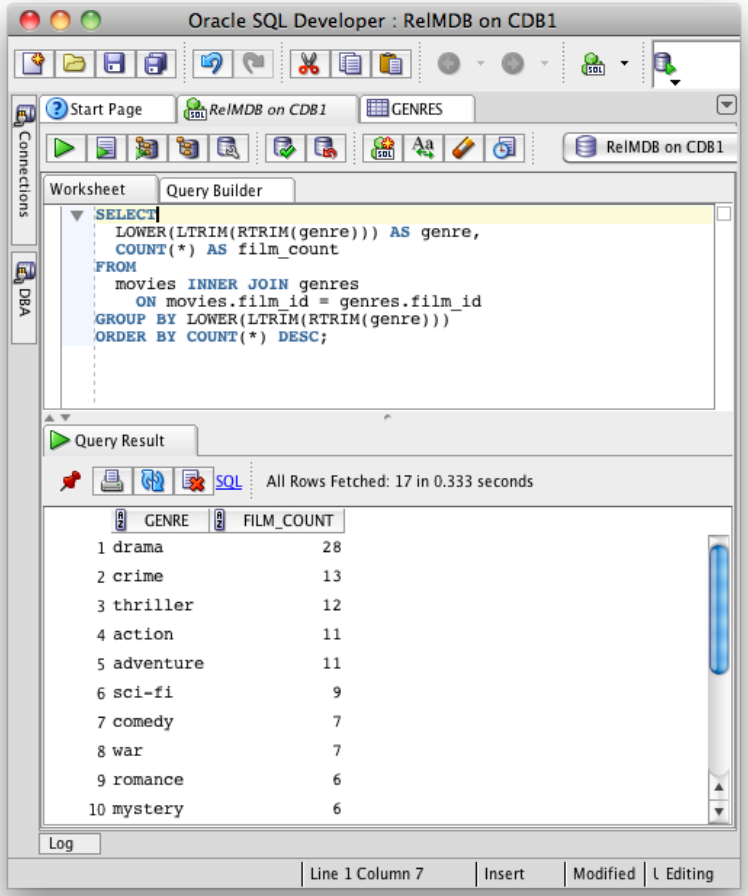
- The repeated and inconsistent values for genres can lead to data quality issues.
- This is a big reason for normalization!
- We can clean up the results with some string processing functions.



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

Detour: Data Quality

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

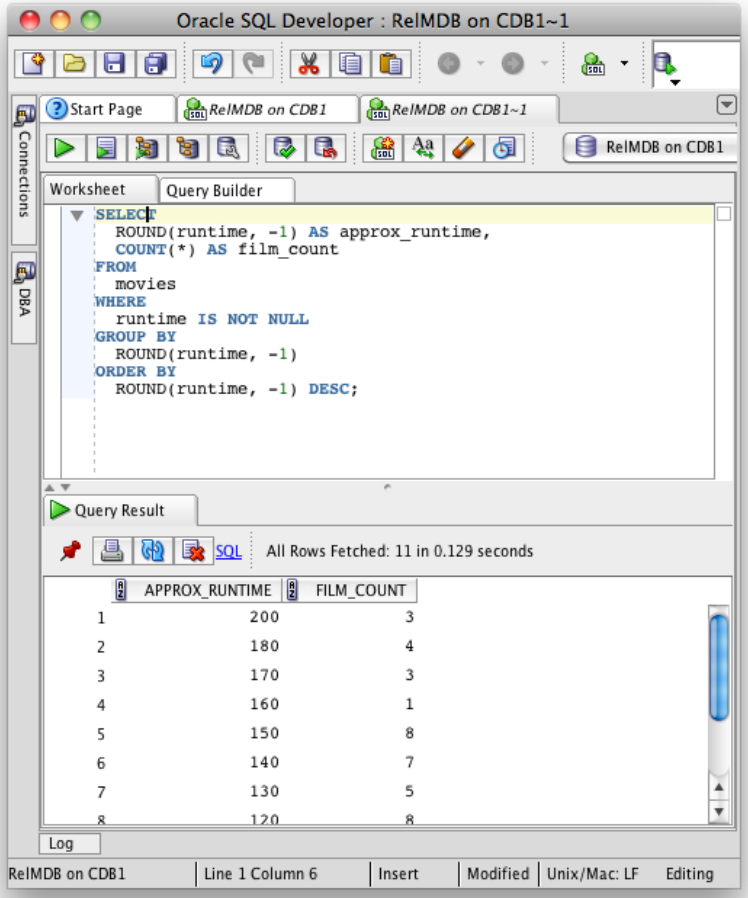



```
SELECT
  LOWER(LTRIM(RTRIM(genre))) AS genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY LOWER(LTRIM(RTRIM(genre)))
ORDER BY COUNT(*) DESC;
```

GROUP BY for Histograms

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

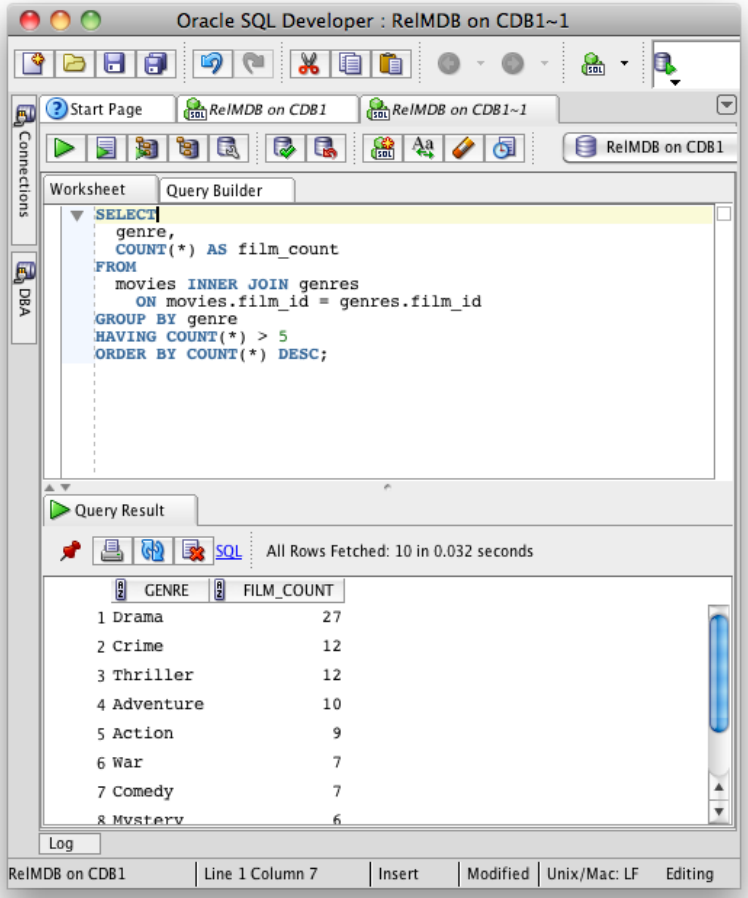
```
SELECT
  ROUND(runtime, -1) AS approx_runtime,
  COUNT(*) AS film_count
FROM
  movies
WHERE
  runtime IS NOT NULL
GROUP BY
  ROUND(runtime, -1)
ORDER BY
  ROUND(runtime, -1) DESC;
```



HAVING

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

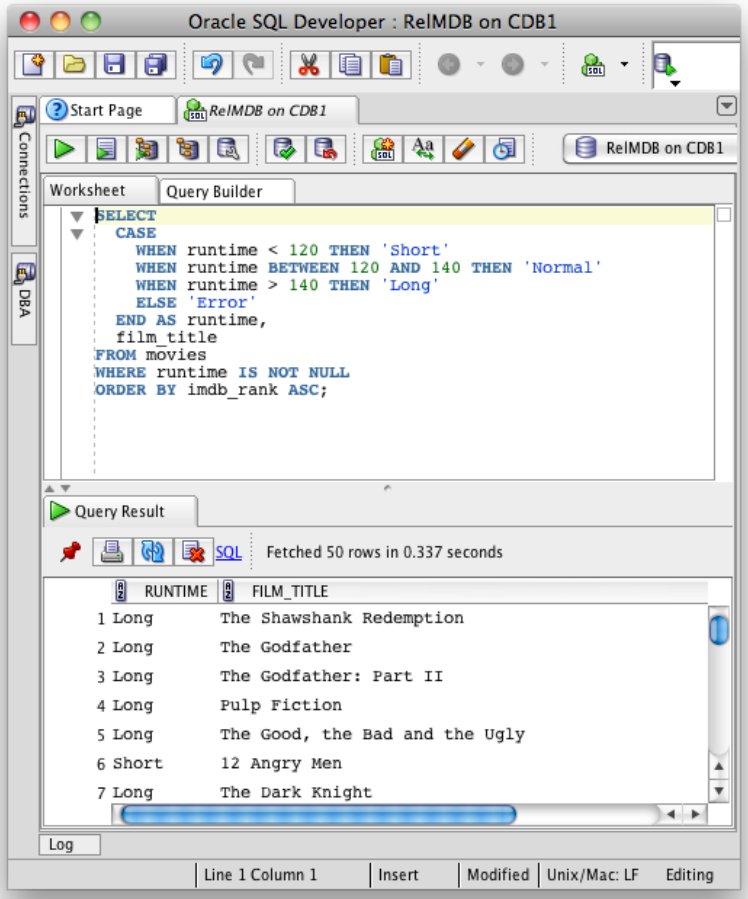
```
SELECT
  genre,
  COUNT(*) AS film_count
FROM
  movies INNER JOIN genres
    ON movies.film_id = genres.film_id
GROUP BY genre
HAVING COUNT(*) > 5
ORDER BY COUNT(*) DESC;
```



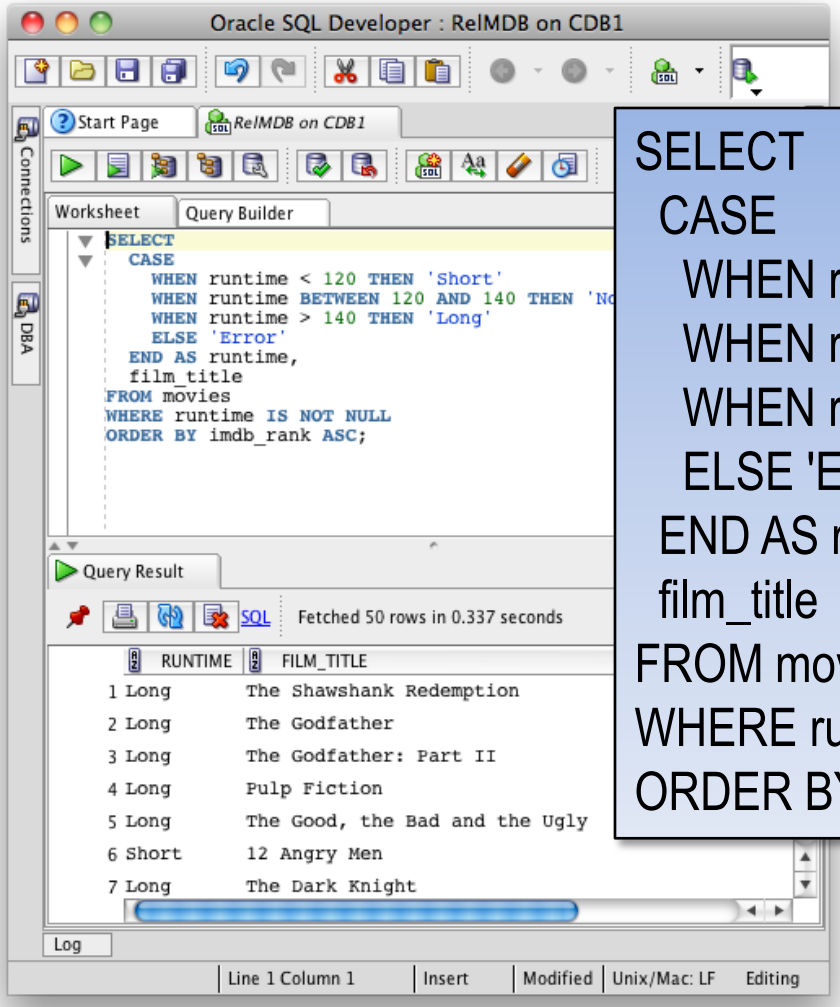
CASE Statement

- 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



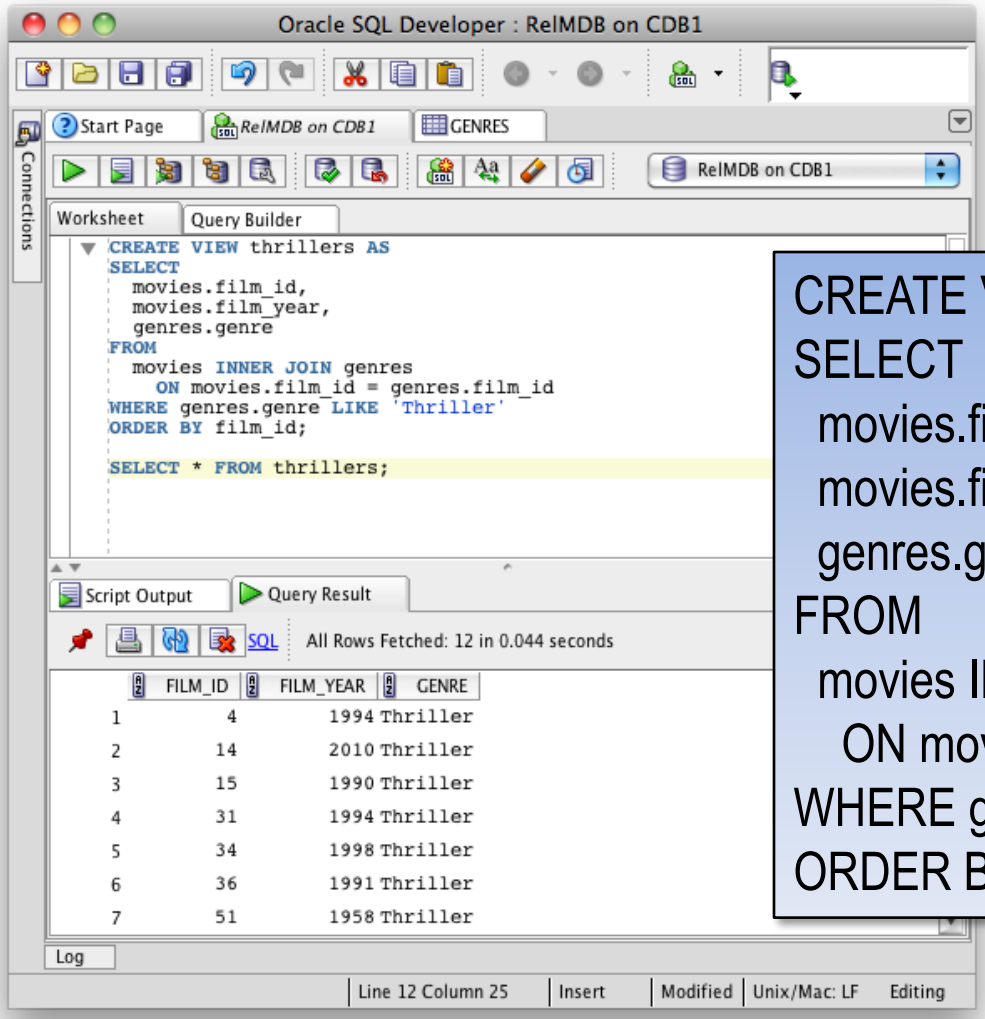

```
SELECT
  CASE
    WHEN runtime < 120 THEN 'Short'
    WHEN runtime BETWEEN 120 AND 140 THEN 'Normal'
    WHEN runtime > 140 THEN 'Long'
    ELSE 'Error'
  END AS runtime,
  film_title
FROM movies
WHERE runtime IS NOT NULL
ORDER BY imdb_rank ASC;
```

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.

```
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];
```

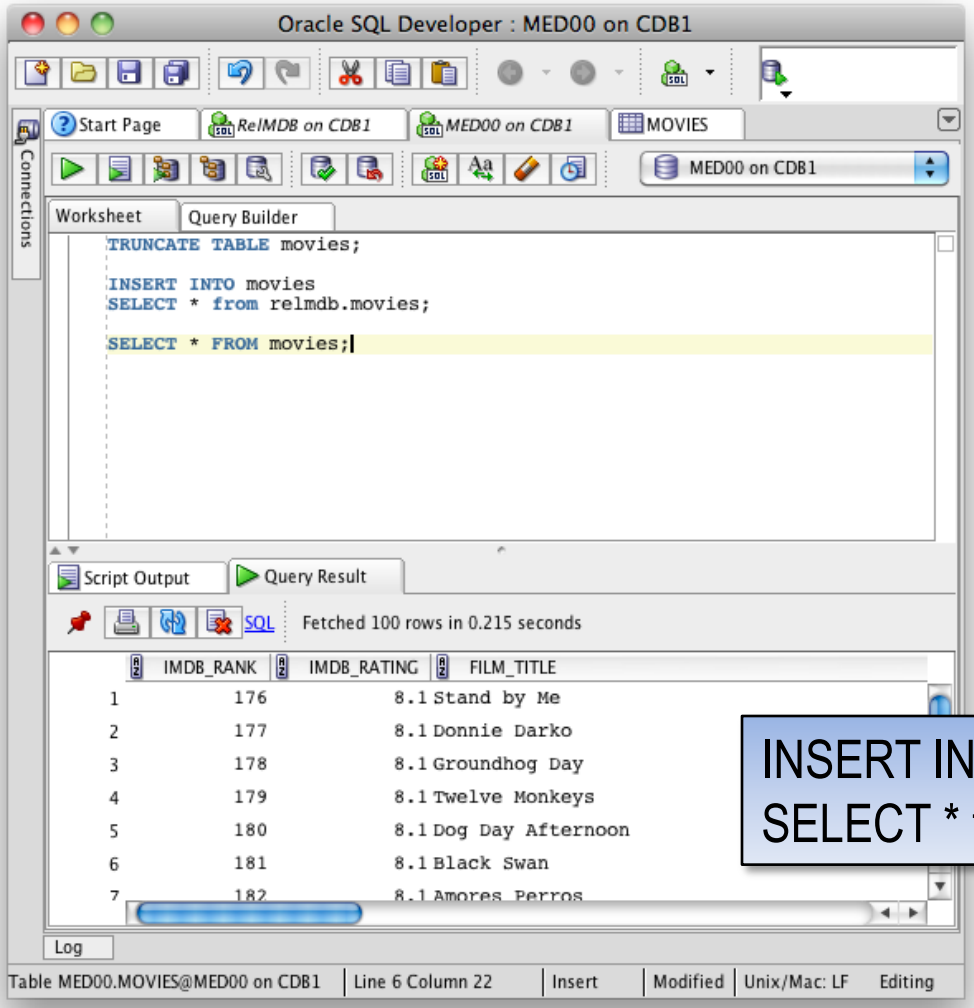
```
INSERT INTO casts
  (film_id, cast_member, cast_role)
VALUES (117, 'Tim Allen', 'Buzz');

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



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.