SQL:
Structure Query Language

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

USF UNIVERSITY OF SOUTH FLORIDA

Why SQL?

- SQL is a "high-level" language.
  - SQL is a **declarative language** (expressing "what to do"), rather than a **procedural language** (expressing "how to do it"), typically a much simpler task.
  - SQL avoids the implementation details faced in procedural languages like C/C++ and Java.
  - The goal is to make database users more productive!

- A database management systems (DBMS) must transform a high-level query into an executable task.
  - The **query optimizer** considers alternative **execution plans**, picking the most efficient plan for a query.

SELECT-FROM-WHERE Queries

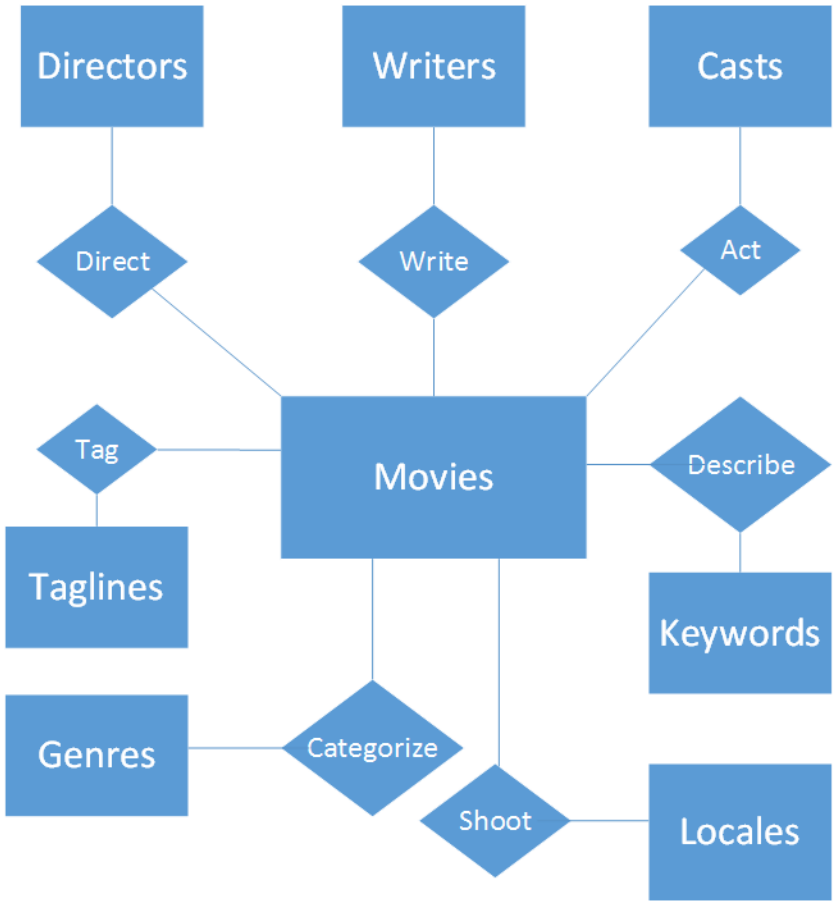
- The most basic SQL query uses the SELECT, FROM, and WHERE clauses.

```
SELECT
        one or more attributes (columns)
FROM
        one or more relations (tables)
WHERE
        each tuple (row) meets the specified conditions
```

```
SELECT film_title
FROM movies
WHERE imdb_rank <= 10;
```

An Example Database: Movies

- The relational movie database (RelMDB) was built to serve as an example for query writing and database design.
- The data was sourced from the Internet movie database (IMDb).
- The core entity set is obviously MOVIES, with associated DIRECTORS, WRITERS, and CASTS .



Simple (Single Table) Queries

- The simplest query (a database "hello world"), just lists all data on movies.

```
SELECT * FROM movies;
```

The "*" means show all the attributes (columns).

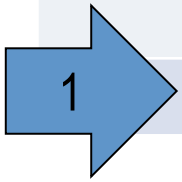
- Adding a WHERE clause limits the result set by forcing tuples (rows) to meet the specified condition.

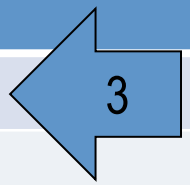
```
SELECT
        film_title
FROM
        movies
WHERE
        imdb_rank = 1;
```

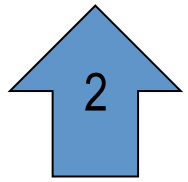
So the highest ranked film (at least for now) is *The Shawshank Redemption* based on the IMDb Top 250 list (www.imdb.com/chart/top).

Operational Semantics

Film ID	Film Title	IMDb Rating	IMDb Rank
1	The Shawshank Redemption	9.2	1
2	The Godfather	9.2	2
3	The Godfather: Part II	9	3
4	Pulp Fiction	8.9	4
5	The Good, the Bad and the Ugly	8.9	5

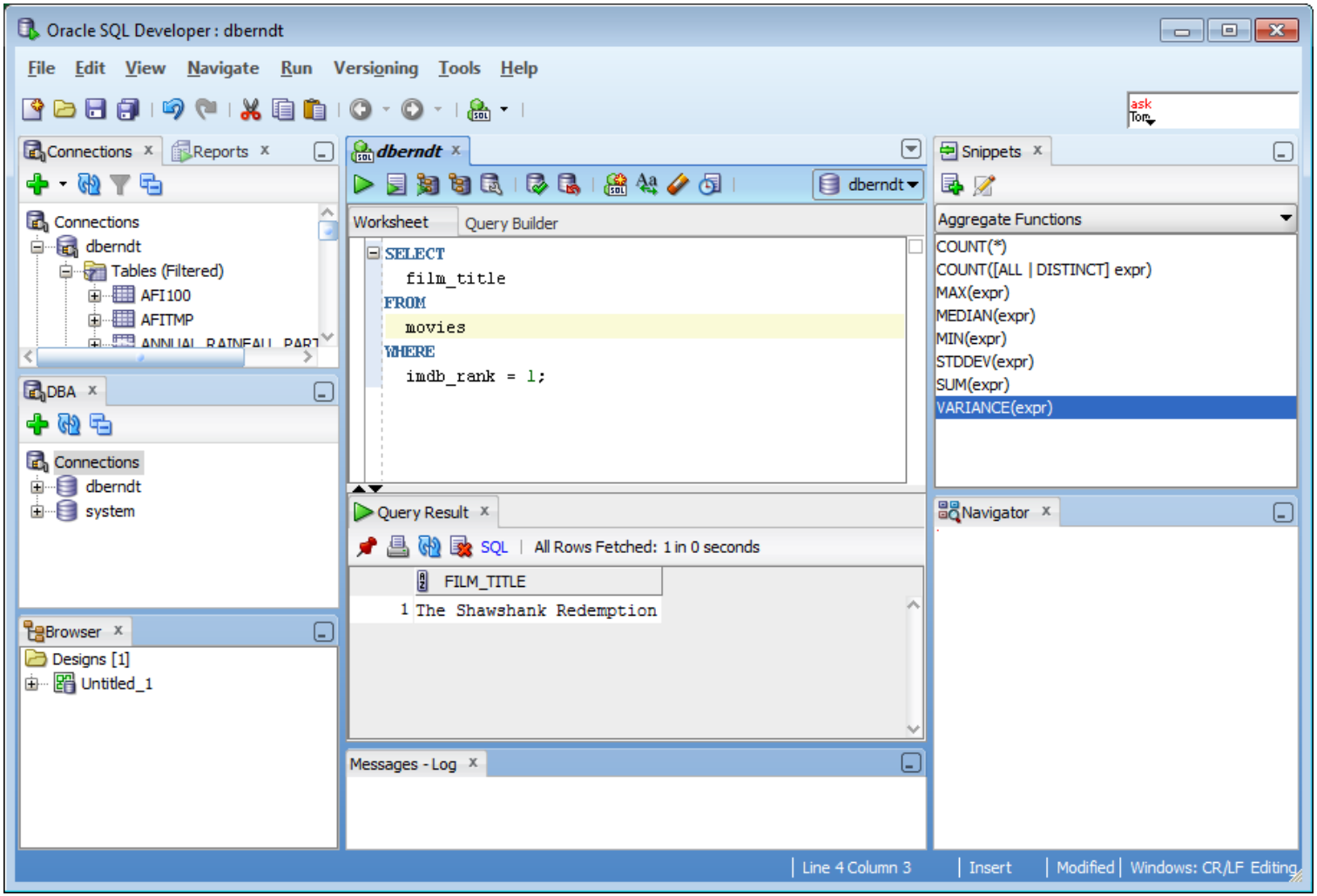






1. Begin with the relation or table in the FROM clause.
   - A "tuple variable" visits each tuple or row in the relation.
2. Apply the restrictions specified in the WHERE clause.
   - Check if the "current tuple" meets the conditions.
3. Project the columns specified in the SELECT clause.
   - If conditions are met, compute any expressions and add the selected attributes to the answer set.

Simple (Single Table) Queries



Simple (Single Table) Queries

- Rename attributes with AS <name> to use an alias.
- Arbitrary expressions can also be used in the SELECT clause.

```
SELECT
        film_title AS "Title",
        (imdb_rank + afi_rank) AS "Rank"
FROM
        movies
WHERE
        (imdb_rank IS NOT NULL) AND
        (afi_rank IS NOT NULL);
```

The expression for "rank" adds two attributes, but NULL values could play an unexpected role.

So, restrict NULL values or better yet use the ISNULL() function.

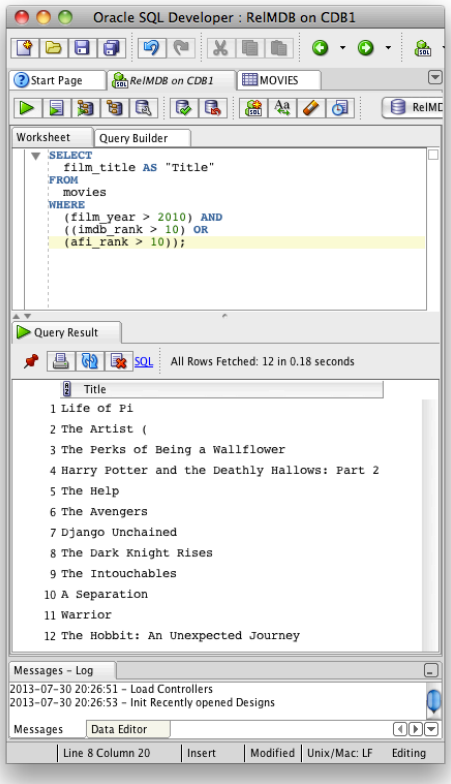
NULL Values & Three-Valued Logic

- Attributes can have NULL values by default.
  - Can be prevented by using a NOT NULL constraint.
- NULLs can have different interpretations, such as a missing value or an attribute that is not applicable to an entity.
- SQL uses three-valued logic: TRUE, FALSE, and UNKNOWN.
- Comparing a NULL value with any other value (even another NULL value) evaluates to UNKNOWN!
  - This can lead to counter intuitive results.
  - Use the ISNULL() function to provide alternative values.
- A tuple is added to an answer set only if the WHERE clause evaluates to TRUE (not FALSE or UNKNOWN).

Simple (Single Table) Queries

- Boolean operators: AND, OR, and NOT.
- Comparison operators: =, >, >=, <, <=, and <>.
- Other boolean-valued operators (e.g. IN, NOT IN, EXISTS, NOT EXISTS, …).

```
SELECT
        film_title AS "Title",
        imdb_rank AS "IMDb Rank",
        afi_rank AS "AFI Rank"
FROM
        movies
WHERE
        (film_year > 2010) AND
        ((imdb_rank > 10) OR (afi_rank > 10));
```



SELECT-FROM-WHERE Queries Revisited

- This simple query maps to the relational algebra operators.

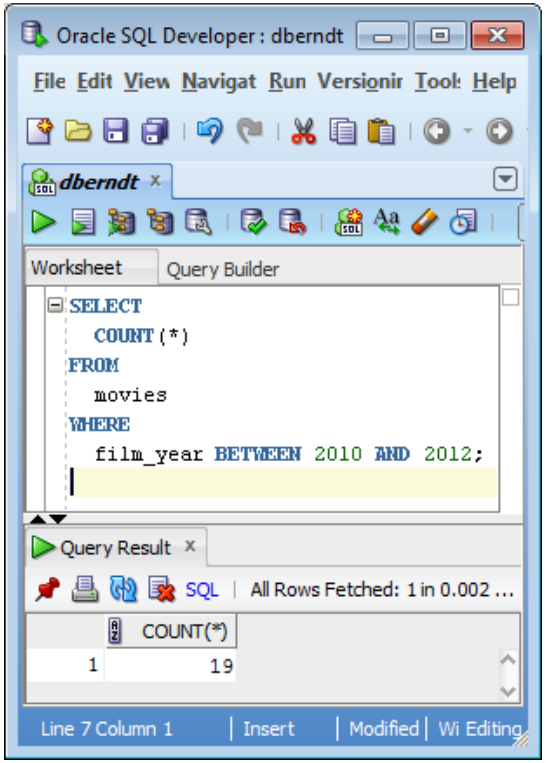
```
SELECT
        one or more attributes (columns)
FROM
        one or more relations (tables)
WHERE
        each tuple (row) meets the specified conditions
```

Simple Queries: BETWEEN

- The relational algebra selection operator, plus much more can be implemented in the WHERE clause. Here BETWEEN is used to look at recent films (and then COUNT them).

```
SELECT
        film_title
FROM
        movies
WHERE
        film_year BETWEEN 2010 AND 2012;
```

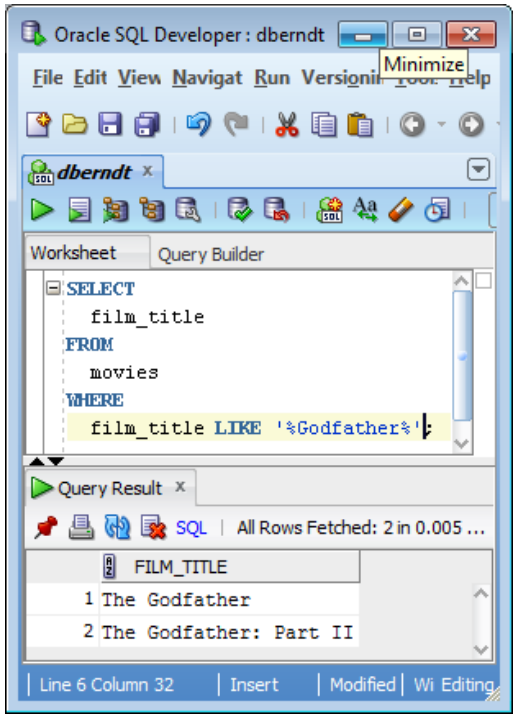
```
-- Count the recent movies.
SELECT COUNT(*) FROM movies …;
```



Simple Queries: LIKE / NOT LIKE

- A common task is to compare strings in the WHERE clause with LIKE providing a **pattern matching** capability. Note: comparing strings with "greater than," "less than," or other comparison operators is based on lexicographic order.

```
SELECT
        film_title
FROM
        movies
WHERE
        film_title LIKE '%Godfather%';
```

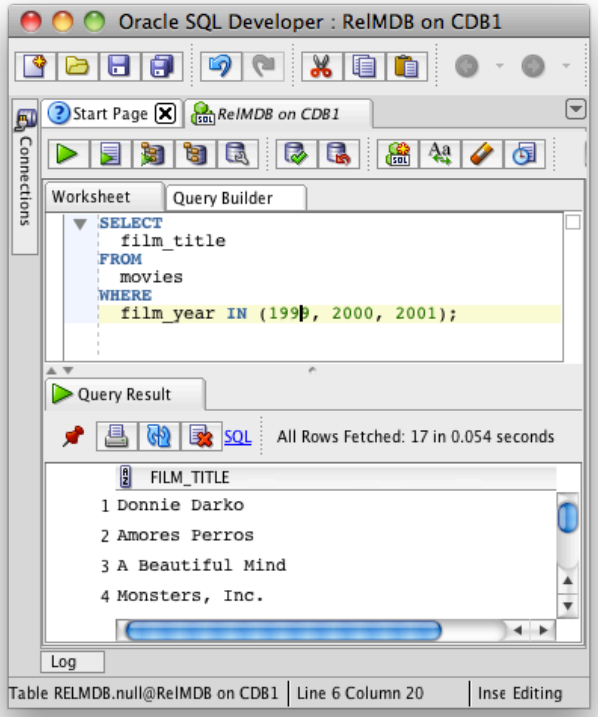


The **pattern** is expressed as a quoted string, with the special wildcard symbols "%" matching zero or more characters and "_" matching any single character.

Simple Queries: IN / NOT IN

- IN / NOT IN allow tests based on set membership.
  - <attribute> IN (<relation>)
- Most useful when the relation is computed in a subquery.
  - Here a set of years is statically defined.

```
SELECT
        film_title
FROM
        movies
WHERE
        film_year IN (1999, 2000, 2001);
```

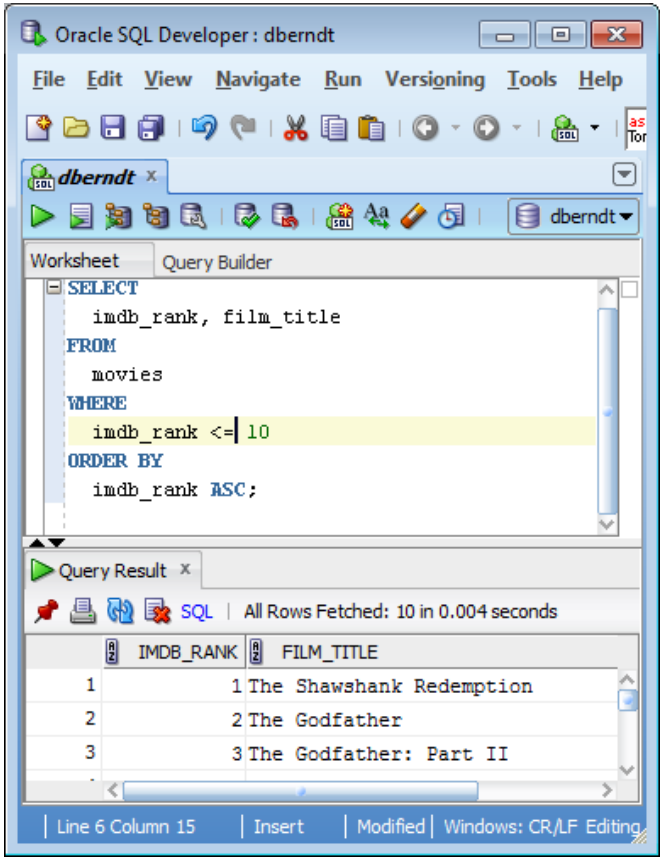


Simple Queries: ORDER BY

- Sorting the result set is accomplished using the ORDER BY clause, with the order specified by as ASCending (the default) or DESCending.

```
SELECT
        imdb_rank, film_title
FROM
        movies
WHERE
        imdb_rank < 10
ORDER BY
        imdb_rank ASC;
```

List the top 10 films in ascending order.

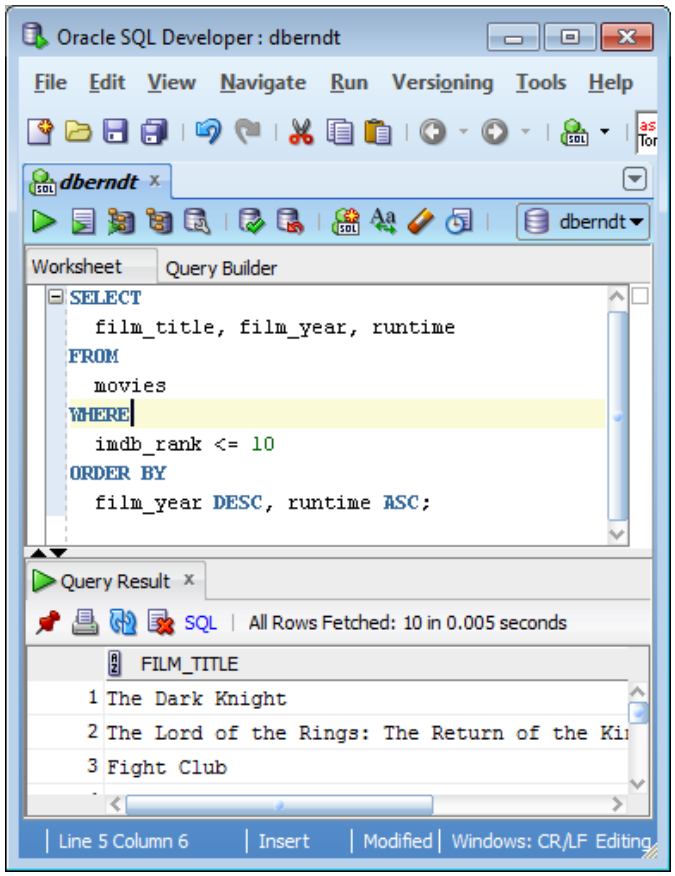


Simple Queries: ORDER BY

- Sorting the result set is accomplished using the ORDER BY clause, with the order specified by as ASCending (the default) or DESCending.

```
SELECT
        film_title, film_year, runtime
FROM
        movies
WHERE
        imdb_rank <= 10
ORDER BY
        film_year DESC, runtime ASC;
```

List the top 10 films by year in descending order and runtime in ascending order.



Multi-Relation Queries

- A relation can be joined to itself by using table aliases to explicitly refer to each copy.

```
SELECT
        m1.film_title,
        m2.film_title
FROM
        movies m1,
        movies m2
WHERE
        m1.director_id = m2.director_id AND
        m1.film_title < m2.film_title
```

Scenario: Imagine that you are implementing a movie rating app, which asks fans to rate pairs of movies by a director. The query should produce all pair-wise combinations, so each vote requires a simple choice.

Subqueries (Scalar)

- A **subquery** evaluates to a relation, extending the WHERE clause from conditions using constants to computations.

```
SELECT
        imdb_rank,
        film_title
FROM
        movies
WHERE
        runtime >
                (SELECT AVG(runtime)
                FROM movies)
ORDER BY
        imdb_rank ASC;
```

Find the films (and IMDb rank) that run longer than the average runtime.

The subquery computes the average runtime for use in the outer query.

Subqueries: IN / NOT IN

- Since a **subquery** evaluates to a relation, IN or NOT IN fits naturally in the WHERE clause.
- See also: EXISTS / NOT EXISTS, ANY (SOME), and ALL.

```
SELECT
        film_title,
        film_year
FROM
        movies
WHERE
        film_id IN
                (SELECT film_id
                FROM genres
                WHERE genre IN ('Crime', 'Drama'));
```

Scenario: Find the films that are in crime or drama genres.

Subqueries: ANY or ALL

- <a> >= ALL(<subquery>) – TRUE if and only if there is no result that exceeds <a>.
- <a> = ANY(<subquery> - TRUE if and only if there <a> matches at least one result.

```
SELECT
        film_title,
        budget
FROM
        movies
WHERE
        budget >= ALL(
                SELECT budget
                FROM movies
                WHERE budget IS NOT NULL);
```

Scenario: Find the film with the largest budget.

Subqueries (Correlated)

- A **correlated subquery** evaluates to a relation, but it is re-computed many times based on a tuple variable from outside the subquery.

```
SELECT
  imdb_rank,
  film_title
FROM movies
WHERE 'Harrison Ford' IN
  (SELECT cast_member
  FROM casts
  WHERE film_id = movies.film_id)
ORDER BY imdb_rank;
```

Find the films (and IMDb rank) that include “Harrison Ford” in the cast.

The subquery checks for the specified cast member amongst the entire cast. The subquery is evaluated again for each row (a film) from the main query.

Bag versus Set Semantics

- SQL treats relations as **bags** (not sets), except in a few contexts.
- A bag or multiset is a generalization of a set in which members may appear multiple times (as duplicates).
- Duplicates can be explicitly eliminated (using DISTINCT).
- UNION, INTERSECT, and EXCEPT use set semantics.
  - These operations combine (or pull apart) two relations using set-theoretic operations.

<r> UNION <s>
<r> INTERSECT <s>
<r> EXCEPT <s>

Eliminating Duplicates: DISTINCT

- SQL operations typically use relations that are bags, rather than sets (except for the UNION, INTERSECTION, and DIFFERENCE operators).

```
SELECT film_year
FROM movies
ORDER BY film_year DESC
```

```
SELECT DISTINCT film_year
FROM movies
ORDER BY film_year DESC
```

The first query returns a relation as a **bag**, with an ORDER BY to highlight the duplicates. The DISTINCT keyword modifies the SELECT to remove duplicates and return a relation as a **set**.

Eliminating Duplicates: DISTINCT

- Eliminating duplicates is expensive!
- One of the best methods is to sort the relation (a bag), removing adjacent duplicates to form a result set.
- While there are efficient sorting algorithms, relations are often very large and costly to sort. Use DISTINCT only when necessary.
- UNION, INTERSECT, and EXCEPT also remove duplicates and can be expensive. The keyword ALL modifies these operators for bag semantics, preserving duplicates and reducing costs.

```
<r> UNION ALL <s>
<r> INTERSECT ALL <s>
<r> EXCEPT ALL <s>
```

Set Operations: INTERSECTION

- UNION – Combines two sets (based on common columns).
- INTERSECTION – Finds the overlap between two sets.
- EXCEPT – Set difference.

```
(SELECT film_title
FROM likes, movies
WHERE
        likes.film_id = movies.film_id AND
        fan_id = 22)
INTERSECT
(SELECT film_title
FROM showtimes, movies
WHERE
        showtimes.film_id = movies.film_id;
```

Scenario: Assume we have extended our design to include both "fans that like movies" and "theaters that show movies." We can use INTERSECT to find movies that a specific fan likes that are also playing right now (given a showtimes).

Multi-Relation Queries

- Most queries require a combination of data from multiple relations (tables).
- Essentially, the data that was carefully separated in the database design to control redundancy must be re-integrated to answer interesting queries.
- Multiple relations can be combined or "joined" together in the FROM clause.
- Individual attributes can be referenced by specifying both the relation and attribute names to clarify any ambiguities.

```
SELECT <relation1>.<attribute>, <relation2>.<attribute>
FROM <relation1>, <relation2>
```

Multi-Relation Queries

```
SELECT film_title, tagline
FROM movies, taglines;
```

- A **Cartesian product** is formed by crossing members of each set with one another. So, two sets of size *m* and *n*, yield a product set of size *m* times *n* (all ordered pairs).
- The CROSS JOIN is an explicit relational operator that computes this product set (more on JOINS later). Try it.

```
SELECT COUNT(*)
FROM movies CROSS JOIN taglines;
```

Multi-Relation Queries

- For example, a **Cartesian square** can be formed from two sets (the row and column headers) with the cells containing the ordered pairs.
- For two relations, it means pairing every row in one relation with every row in the other!

Cartesian Square	a	b	c
1	a1	b1	c1
2	a2	b2	c2
3	a3	b3	c3

Multi-Relation Queries

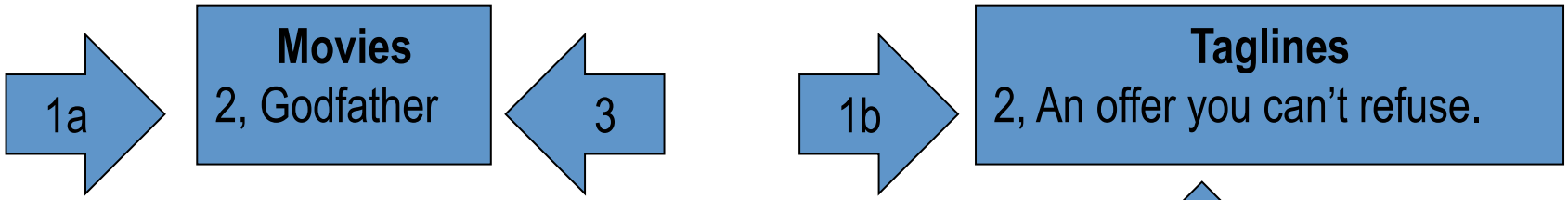
- Whenever a pair of tuples from the combined relations meet the WHERE clause conditions, the film title and tagline are added to the result.

```
SELECT
        film_title,
        tagline
FROM
        movies,
        taglines
WHERE
        film_title LIKE '%Godfather%' AND
        movies.film_id = taglines.film_id
```

The relation names fully specify the ambiguous attributes. Of course, relation names can be used even when there is no ambiguity.

Operational Semantics

1. Begin with the cross product of the two (or more) relations in the FROM clause. Logical semantics, but not an efficient JOIN.
   - "Tuple variables" visit each tuple or row in both relations.
2. Apply the restrictions specified in the WHERE clause.
   - Check if the "current tuples" meet the conditions.
3. Project the columns specified in the SELECT clause.

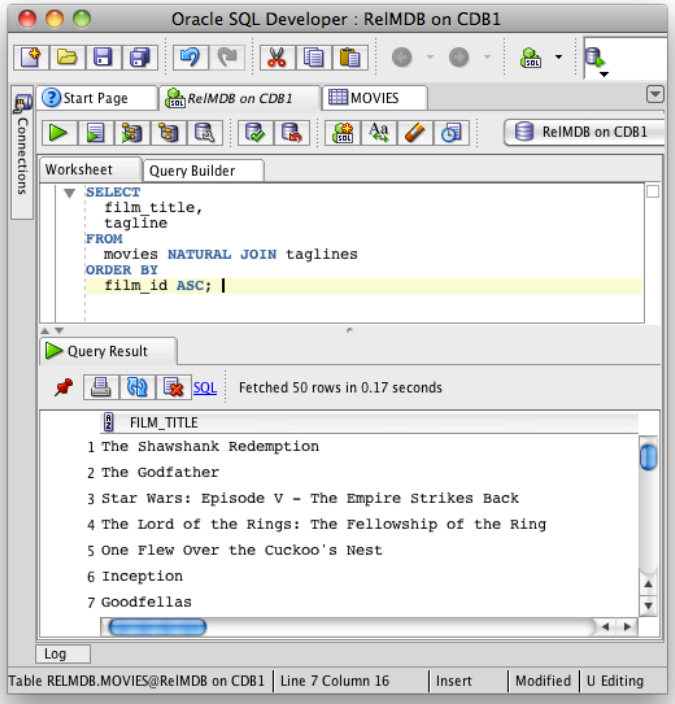



```
SELECT film_title
FROM movies, taglines
WHERE  moves.film_id = taglines.film_id AND
       tagline LIKE 'An offer you can't refuse.';
```

JOIN Queries: NATURAL JOIN

- This is the basic join that uses naturally occurring common attributes shared by each of the relations.
  - Here FILM_ID is a key shared attribute.
- Not used too often since the attributes are left unspecified.

```
SELECT
        film_title,
        tagline
FROM
        movies NATURAL JOIN taglines
ORDER BY
        film_id ASC;
```



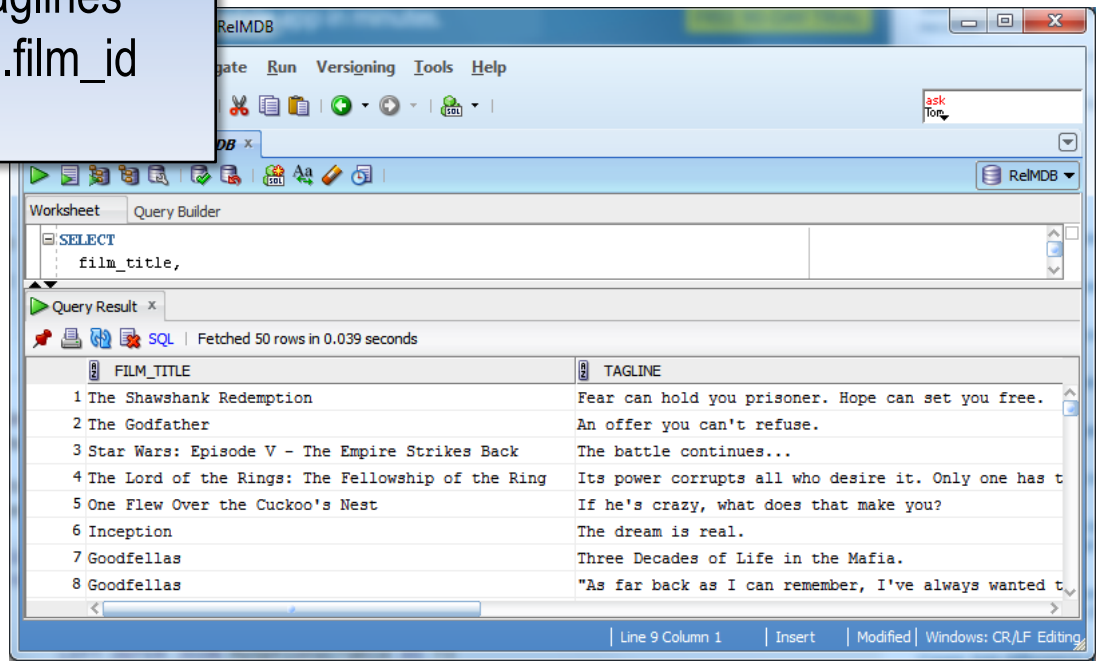
JOIN Queries: INNER JOIN

- The most common join operation is an INNER JOIN (or just JOIN) that combines the tuples from each relation, whenever the specified ON condition is met.

```
SELECT film_title, tagline
FROM movies INNER JOIN taglines
  ON movies.film_id = taglines.film_id
ORDER BY imdb_rank ASC;
```

[INNER] JOIN

INNER is optional, but more fully documents the type of join.



JOIN Queries: Implicit or Explicit?

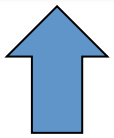
- SQL includes both implicit syntax and (as of SQL2) explicit syntax INNER JOIN operations. Which do you prefer?

Implicit Syntax

```
SELECT
  film_title,
  tagline
FROM
  movies,
  taglines
WHERE
  film_title LIKE '%Godfather%' AND
  movies.film_id = taglines.film_id
```

Explicit Syntax

```
SELECT
  film_title,
  tagline
FROM
  movies,
  INNER JOIN taglines
    ON  movies.film_id = taglines.film_id
WHERE
  film_title LIKE '%Godfather%'
```



The explicit notation does document the SQL code somewhat better, differentiating the JOIN types.

JOIN Queries: INNER JOINs

- Most interesting queries involve joining multiple relations, often mixing different types of joins.

```
SELECT
  film_title,
  tagline
FROM
  movies
  INNER JOIN taglines
    ON movies.film_id = taglines.film_id
  INNER JOIN genres
    ON movies.film_id = genres.film_id
WHERE genre LIKE 'Crime'
ORDER BY imdb_rank ASC;
```

Join the **movies** to their **taglines** and **genres**, keeping only the "crime" films (ordering by imdb_rank).

JOIN Queries: OUTER JOIN

- The OUTER JOIN includes all the tuples from either relation (LEFT or RIGHT), as well the tuples from the other relation whenever they exist.

```
SELECT
  film_title,
  tagline
FROM
  movies
  LEFT OUTER JOIN taglines
    ON movies.film_id = taglines.film_id
ORDER BY imdb_rank ASC;
```

List the **movies** and their **taglines**, whenever a tagline exists (ordering by imdb_rank).

LEFT [OUTER] JOIN

OUTER is optional, but more fully documents the type of join.

JOIN Queries: OUTER JOIN

- The FULL OUTER JOIN includes all the tuples from both relations, irrespective of whether a corresponding tuple exists in the paired relation.

```
SELECT
  film_title,
  keyword
FROM
  movies
  FULL OUTER JOIN keywords
    ON movies.film_id = keywords.film_id
ORDER BY imdb_rank ASC;
```

List the **movies** and their **keywords**, whenever either exists (ordering by imdb_rank).

FULL [OUTER] JOIN

OUTER is optional, but more fully documents the type of join.