SQL Programming (3)

Readings:

-”The Database language SQL” chapter of the textbook
- SQLite tutorial https://www.sqlitetutorial.net/

What we discussed in SQL1 and SQL2

- The SELECT-FROM-WHERE structure
- Single relation queries
  - The Where condition to extract rows from tables
  - Simple aggregation on whole tables
- Multi-relation queries
  - Join
  - Sub-queries

What we discussed in SQL1 and SQL2: Exercise

Explain in English what the following SQL queries are doing.

```
select rating, length
from movie
where strftime(’%Y’, rel_date)= ‘2009’;
```

```
select avg(length)
from movie;
```

Note: The ”strftime()’ function in SQLite is equivalent to the ”to_char()” function in SQL*Plus. It can extract specific parts of a date value:

https://www.sqlitetutorial.net/sqlite-date-functions/sqlite-strftime-function/

In this lecture

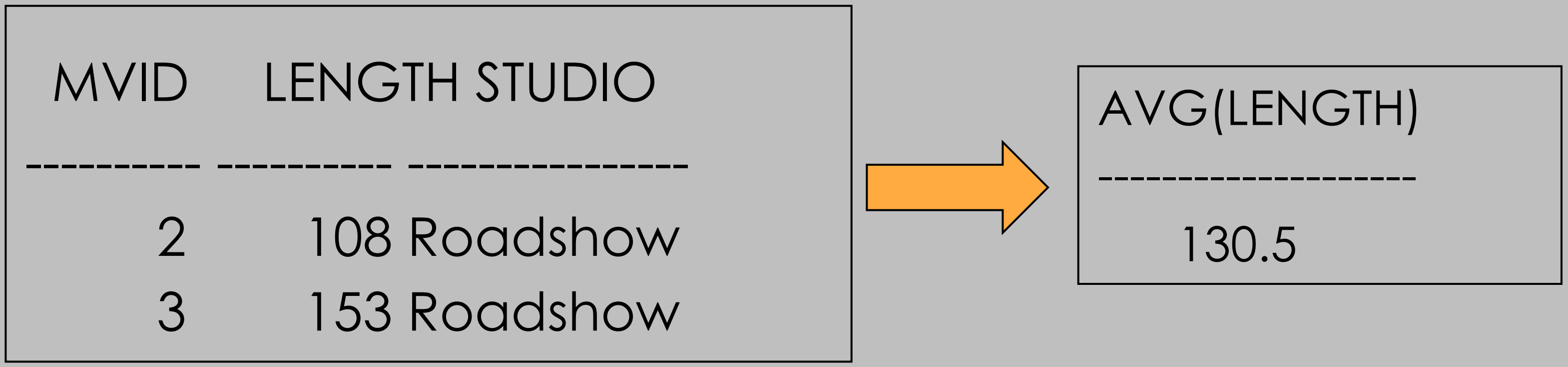
- Complex Aggregations
  - selective aggregation,
  - GROUP BY and HAVING
- SET Operators
  - UNION, INTERSECT, EXCEPT
- View
- Index

Note: EXCEPT in SQLite is equivalent to MINUS in SQL*Plus.

Selective Aggregation

- Aggregate the tuples selected by the WHERE clause.
- Example: What is the average length of movies produced by Roadshow?

```
SELECT AVG(length)
FROM Movie
WHERE studio='Roadshow';
```



```
MVID     LENGTH STUDIO
---------- ---------- ----------------
       2      108 Roadshow
       3      153 Roadshow
```

```
AVG(LENGTH)
-------------------
     130.5
```

GROUP BY: Grouping Tuples

- The GROUP BY operator groups tuples -- possibly selected by a WHERE clause – into groups for aggregation.
    - Each group of tuples have unique values for the group-by attribute list.
- Example: How many movies are there for each genre?

```
 MVID GENRE
---------- ----------
         1 Drama
         2 Drama
         3 Action
         3 Adventure
         3 Drama
         4 Comedy
         5 Animated
         5 Comedy
```

```
select genre, count(*)
from classification
group by genre;

GENRE       COUNT(*)
---------- ------------------
Adventure          1
Animated           1
Action             1
Comedy             2
Drama              3
```

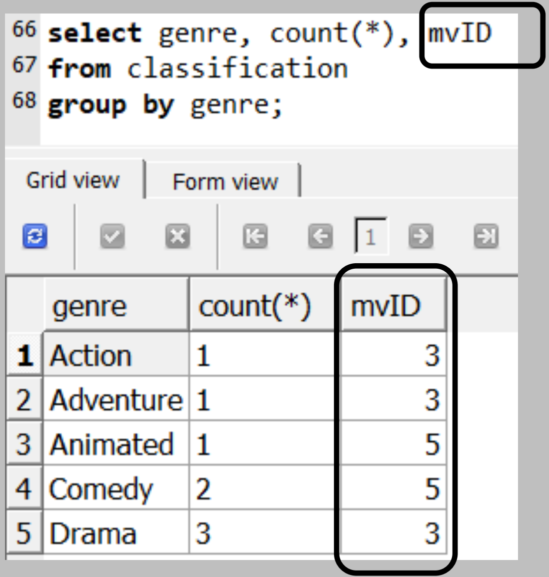
GROUP BY …

- With tuples grouped into groups, groups are represented by the group-by attributes, and tuples become unrecognisable.
- With GROUP BY, only group-by attributes and aggregates for groups should be output. But SQLite does not report errors when you output non-GROUP BY attributes (although SQL*Plus reports errors).

In SQL*Plus:

```
 1  select genre, mvID
  2  from classification
  3  group by genre
SQL> /
select genre, mvID
          *
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

In SQLite:



```
select genre, count(*), mvID
from classification
group by genre;
```

	genre	count(*)	mvID
1	Action	1	3
2	Adventure	1	3
3	Animated	1	5
4	Comedy	2	5
5	Drama	3	3

Non-GROUP BY attribute.

HAVING: select groups and aggregates

- The HAVING operator specifies conditions for choosing groups and aggregates to output.
- Example: What are the genres that have at least two movies? Output these genres and their total number of movies.

```
 MVID GENRE
---------- ----------
         1 Drama
         2 Drama
         3 Action
         3 Adventure
         3 Drama
         4 Comedy
         5 Animated
         5 Comedy
```

```
select genre, count(*)
from classification
group by genre
having count(*) >=2;

GENRE       COUNT(*)
---------- --------------------
Comedy            2
Drama             3
```

WHERE vs. HAVING

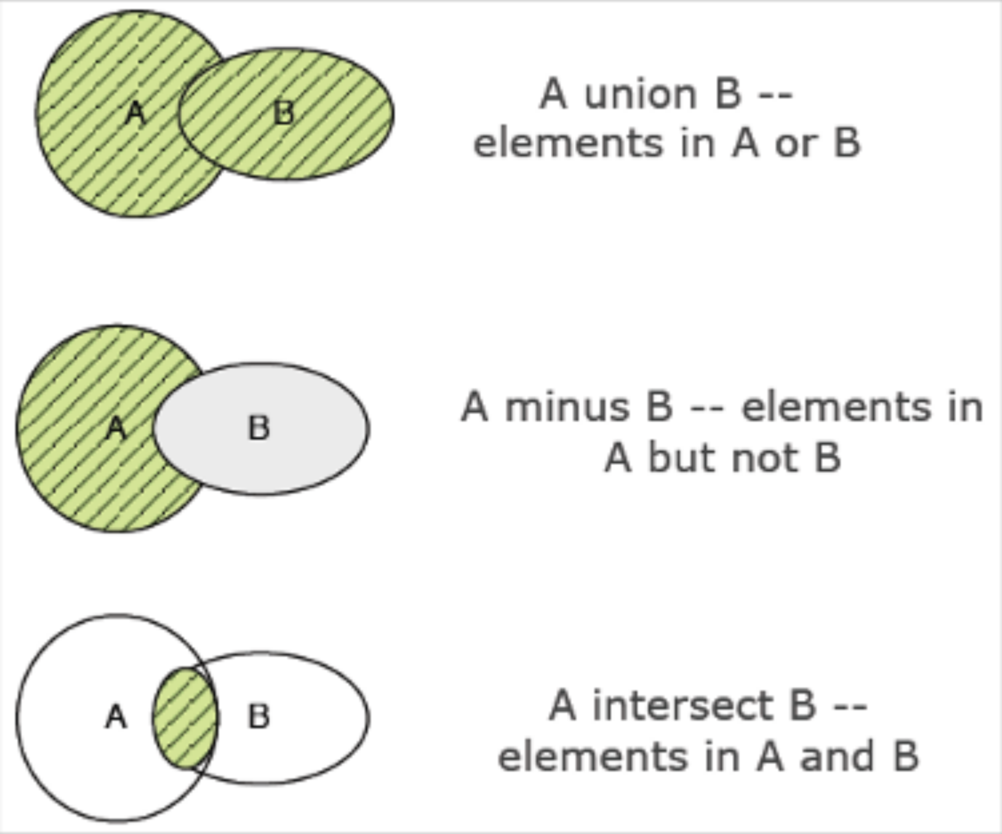
- A WHERE clause chooses tuples for output or further aggregation. A HAVING clause chooses groups and aggregates for output.
- A WHERE clause must appear before the GROUP BY and HAVING clauses.
- Example: For genres starting with C or D, output those that have at least 3 movies.

```
 MVID GENRE
---------- ----------
         1 Drama
         2 Drama
         3 Action
         3 Adventure
         3 Drama
         4 Comedy
         5 Animated
         5 Comedy
```

```
select genre, count(*)
from classification
where genre like 'C%' or genre like 'D%'
group by genre
having count(*) >=3

GENRE      COUNT(*)
---------- --------------------
Drama               3
```

Set operations



A union B -- elements in A or B

A minus B -- elements in A but not B

A intersect B -- elements in A and B

A and B are sets of elements of the same structure/domain.

- If A contains integers B must also contains integers.
- If A is a set of tuples of 3 components, then so is B.

Set operators: Union, Intersect and Minus

- The Union, Intersect and Except operators applied to relations are expressed by the following expressions possibly involving subqueries:
  - <subquery> UNION <subquery>
  - <subquery> INTERSECT <subquery>
  - <subquery> EXCEPT <subquery>

Our previous failed query

Find the movies (mvID) that have both “Marie Gillain” and “Audrey Tautou”.

```
select mvID
from Cast
where actor='Marie Gillain'
        and actor='Audrey Tautou';
```

INTERSECT

- Solution:
  1. Find the movies that have “Marie Gillain”.
  2. Find the movies that have “Audrey Tautou”.
  3. Find the intersection of results from 1 and 2, which is the solution.

Solution

Movies that have "Marie Gillain".

```
Select mvID
From Cast
Where actor=‘Marie Gillain’
```

INTERSECT

```
Select mvID
From Cast
Where actor=‘Audrey Tautou’;
```

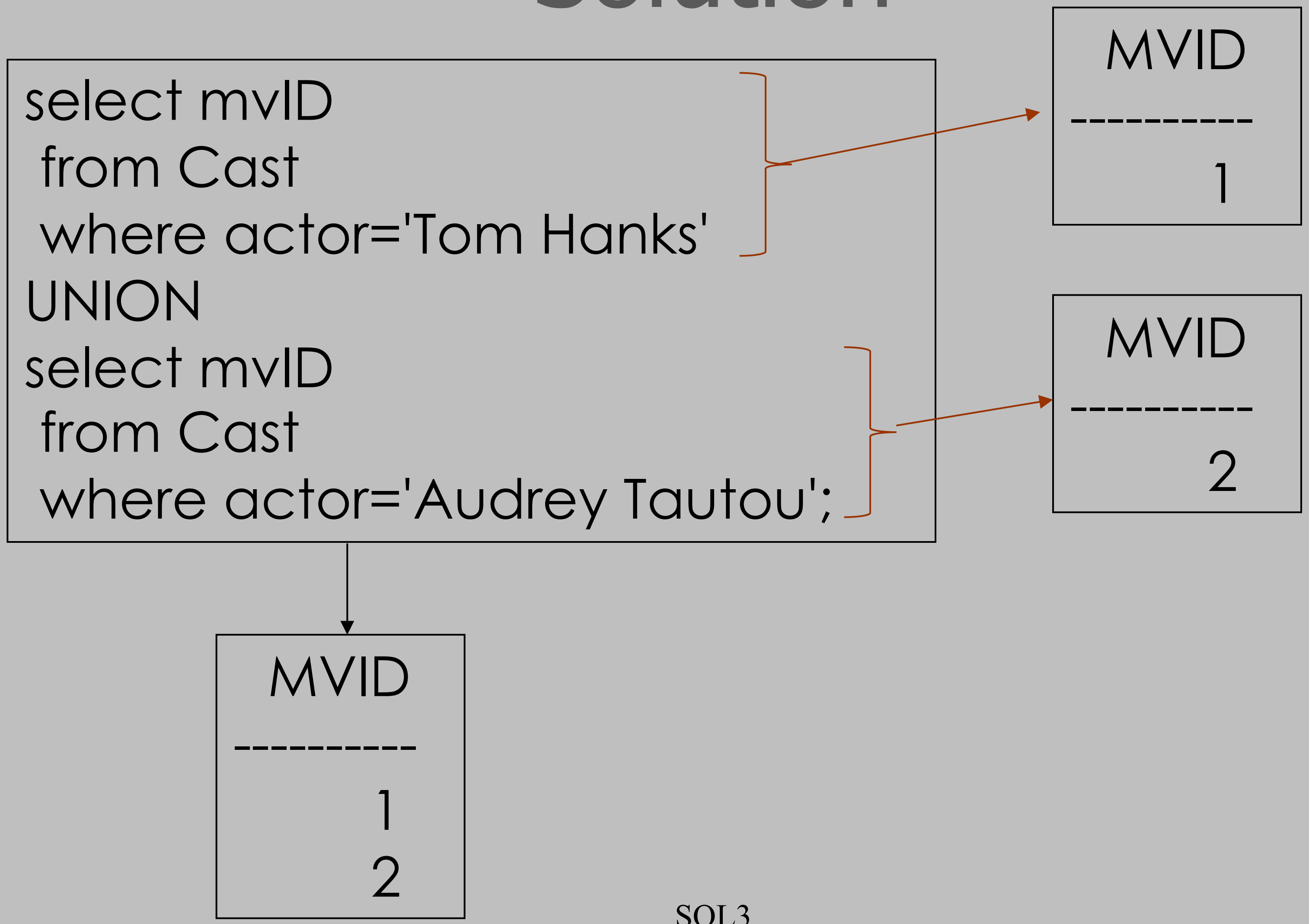
Movies that have "Audrey Tautou".

UNION

- Find the movies (mvID) that have either “Tom Hanks” or “Audrey Tautou”.
- Solution:

1. Find the movies by “Tom Hanks”.
2. Find the movies by “Audrey Tautou”.
3. Union the results from 1 and 2, and form the solution.

Solution



```
select mvID
 from Cast
 where actor='Tom Hanks'
UNION
select mvID
 from Cast
 where actor='Audrey Tautou';
```

```
MVID
----------
         1
```

```
MVID
----------
         2
```

```
MVID
----------
         1
         2
```

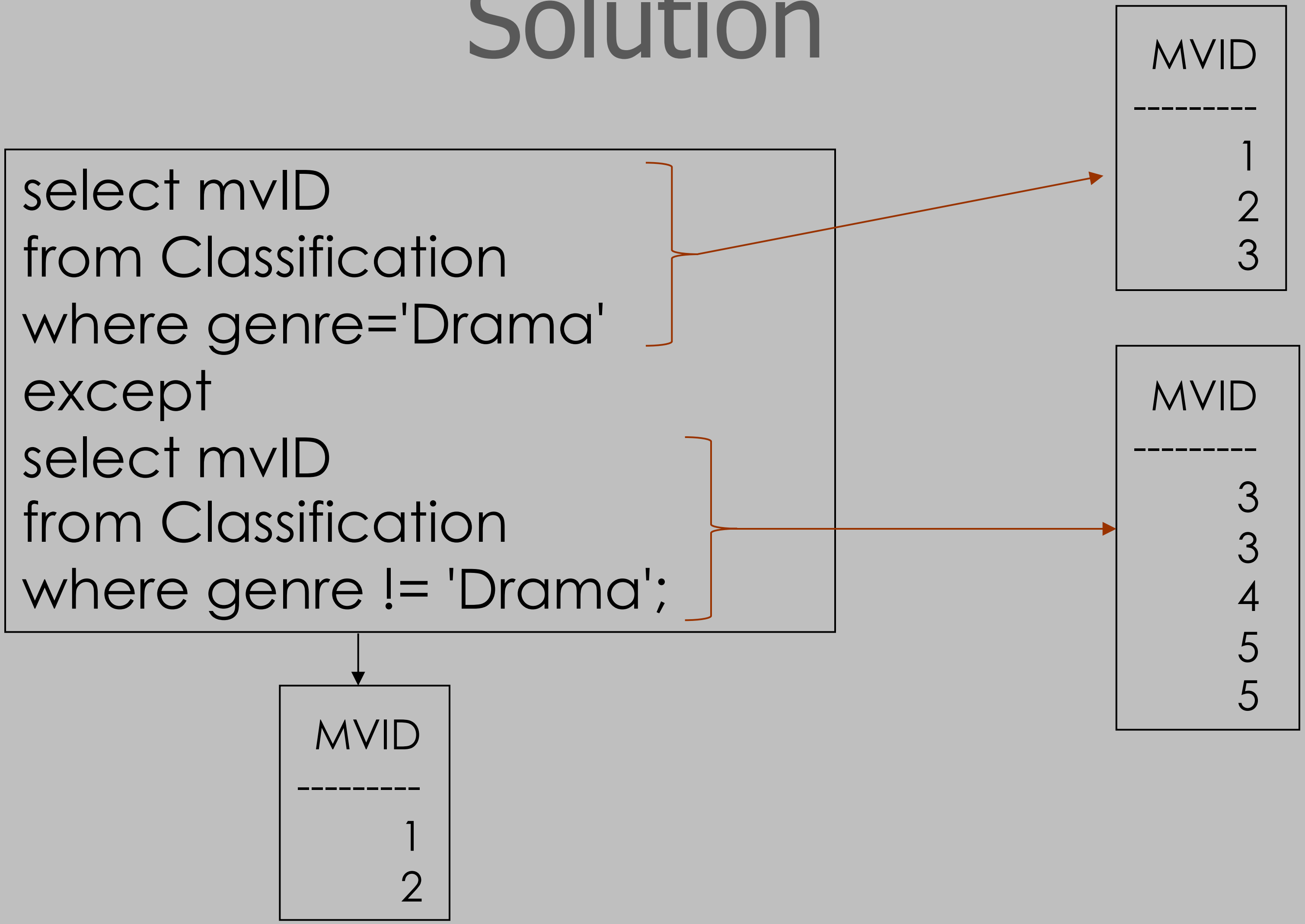
EXCEPT

Find the movies that are *only* in the genre “Drama”.

1. Find the movies in the genre “Drama”.
2. Find the movies in genres other than “Drama”.
3. Take the difference of 1 and 2, and form the solution.

Note that EXCEPT in SQLite is equivalent to MINUS in SQL*PLUS.

Solution



```
select mvID
from Classification
where genre='Drama'
except
select mvID
from Classification
where genre != 'Drama';
```

```
MVID
---------
        1
        2
        3
```

```
MVID
---------
        3
        3
        4
        5
        5
```

```
MVID
---------
        1
        2
```

Set operators Remove duplicates!

- The default SELECT-FROM-WHERE statement keeps duplicates (the bag semantics).
- The default union, intersection, and except expressions remove duplicates (set semantics).

Set operators remove duplicates ...

- Find the movies (mvID) that are not in the genre “Drama”

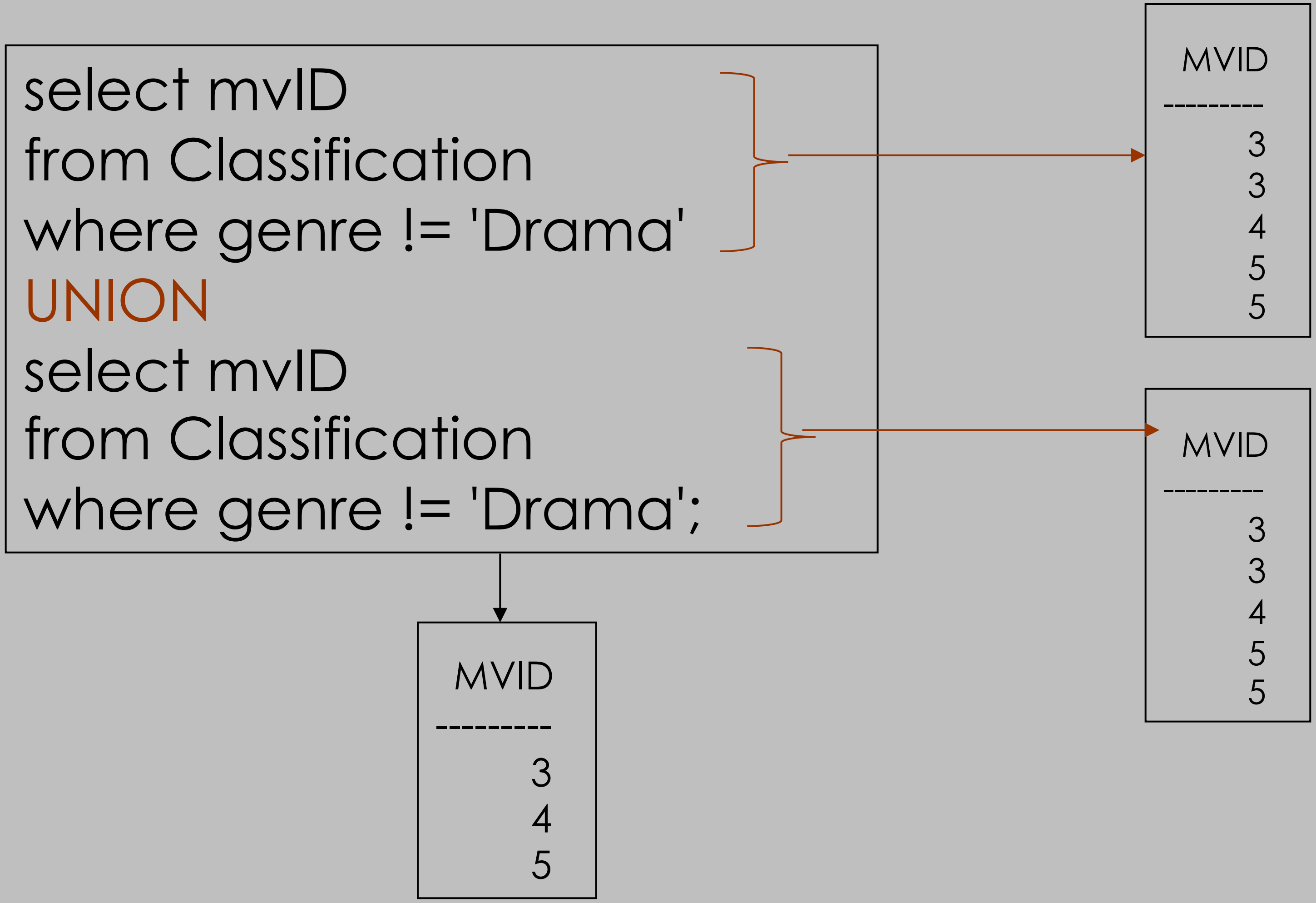
```
select mvID
from Classification
where genre != 'Drama’;
```

```
MVID
---------
       3
       3
       4
       5
       5
```

```
select DISTINCT mvID
from Classification
where genre != 'Drama’;
```

```
MVID
---------
       3
       4
       5
```

Set operators remove duplicates ...



```
select mvID
from Classification
where genre != 'Drama'
UNION
select mvID
from Classification
where genre != 'Drama';
```

```
MVID
---------
       3
       3
       4
       5
       5
```

```
MVID
---------
       3
       3
       4
       5
       5
```

```
MVID
---------
       3
       4
       5
```

Views

- A view is a virtual table defined by an SQL query.
- Views do not keep data, but can be queried. When a view is queried, it is replaced by its definition to execute the query.
- Views generally are for not for update.
- View definition:

  CREATE VIEW <name>AS <view-definition>

Views …

- Define a view for the total number of actors for each movie – attributes can be renamed.
- The view can be queried and queries often become simpler.

```
CREATE VIEW genreCount (mvid,numGenre) AS
SELECT mvid, count(genre)
FROM Classification
group by mvid;
```

Querying the view GenreCount(mvid, numGenre) :

```
select mvid
from GenreCount
where numGenre >1;
```

```
select avg(numGenre)
from GenreCount;
```

Indexes

- CS convention: indexes not indices.
- An Index is a data structure used to speed up access to tuples of a relation.
  - A DBMS uses an index on a table to search for a row rather than scanning the whole table. This greatly reduces search time and disk input/output.
- An index for a table is always a balanced search tree with giant nodes (a full disk page) called a B-tree.
  - This topic will be discussed in more details in the course *Database Systems (COSC2406/2407).*

Creating Indexes

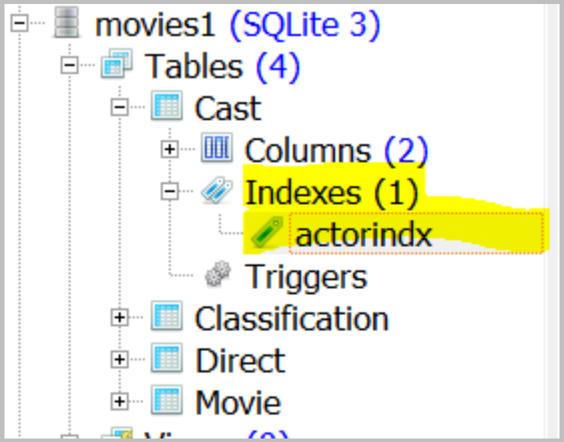
- In most if not all DBMSs, implicit indexes are created automatically when the PRIMARY KEY constraint is defined.
- An index can also be created in SQLite using a CREATE INDEX statement.

```
CREATE INDEX indexname
        ON tablename (col1, col2, …);
```

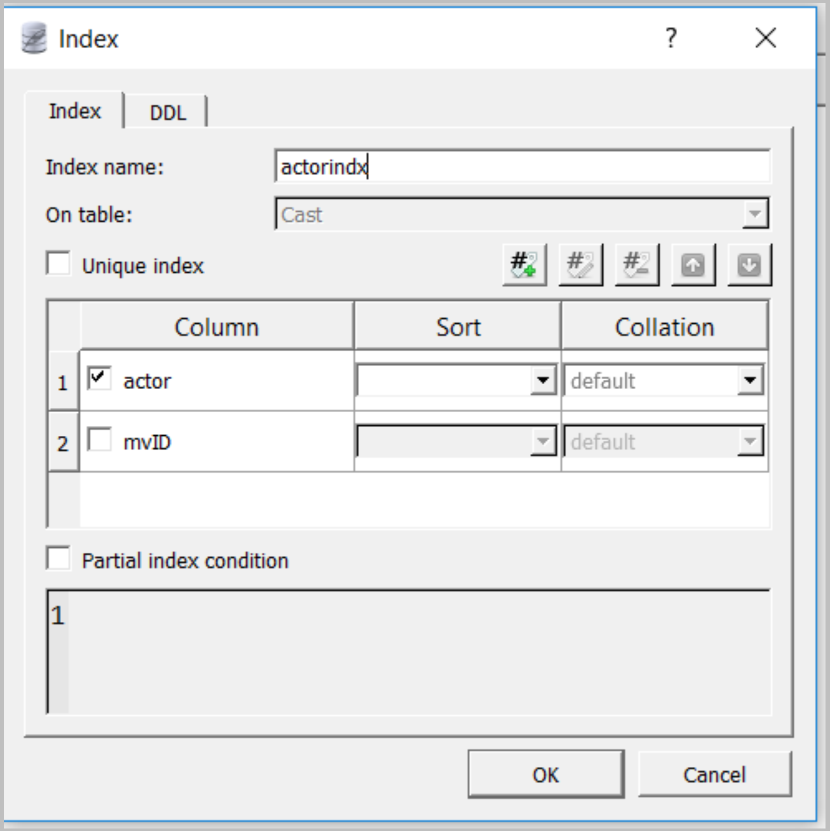
Creating Indexes ...

- An index named actorindx is created.

```
CREATE INDEX actorindx
ON Cast(actor);
```



movies1 (SQLite 3)
- Tables (4)
  - Cast
    - Columns (2)
    - Indexes (1)
      - actorindx
    - Triggers
  - Classification
  - Direct
  - Movie



Index name: actorindx

On table: Cast

□ Unique index

	Column	Sort	Collation
1	☑ actor		default
2	□ mvID		default

□ Partial index condition

1

Using Indexes

- When an index is created, a user does not need to open or use the index with a command. Rather an index is used by the DBMS for processing queries.
- Indexes are kept separately from their base tables.
- Every insertion or deletion in a table updates the index, which is added overhead on the system.

Using Index …

```
CREATE INDEX actorindx
ON Cast(actor);
```

The DBMS SQL engine will automatically use the “actorindx” index when processing queries involving the column “actor”. Queries run more efficiently.

```
select *
from Cast
where actor > 'A%';
```

```
select C1.mvid, C2.mvid, C1.actor
from Cast C1, Cast C2
where C1.actor=C2.actor
   and C1.mvid < C2.mvid;
```

Database Tuning

- A main task in making a database run fast is deciding which indexes to create.
  - An index speeds up queries that can use it.
  - An index slows down all modifications on its relation because the index must be modified too.

Database Tuning …

- Generally an index is created on a column if the column
  - is used very often in querying and joining,
  - has a big domain of values, or
  - contains many Null values.
    - Null values are removed in indexes.
- An index should not be created for
  - a very small table,
  - a column not used often in queries, or
  - a table that often gets updated.