Intersection

```
MovieStar(name,address,gender,birthDate)
MovieExec(name,address,cert#,netWorth)
```

Query: Names and addresses of all female movie stars who are also movie executives with netWorth over 10.000.000

First, find female movie stars

```
SELECT name, address
FROM MovieStar
WHERE gender = 'F';
```

Next, find movie executives with netWorth over 10.000.000

```
SELECT name, address
FROM MovieExec
WHERE netWorth > 10000000;
```

Then take the intersection (note that the schemas are the same)

```
(SELECT name, address
FROM MovieStar
WHERE gender = 'F')
    INTERSECT
(SELECT name, address
FROM MovieExec
WHERE netWorth > 10000000);
```

Difference: Movie Stars who are not Movie Executives

```
(SELECT name, address FROM MovieStar)
  EXCEPT
(SELECT name, address FROM MovieExec);
```

What is wrong with this?

```
(SELECT * FROM MovieStar)
  EXCEPT
(SELECT * FROM MovieExec);
```

Union

```
Movies(title,year,length,genre,studioName,producerC#)
StarsIn(movieTitle, movieYear, starName)
```

We might have movies in StarsIn that are missing from Movies and vice versa. Find all movies in the database

```
(SELECT  title, year  FROM Movies)
  UNION
(SELECT movieTitle, movieYear FROM  StarsIn);
```

Doesn't work, as the schemas are different. Need to rename:

```
(SELECT  title, year  FROM Movies)
  UNION
(SELECT movieTitle AS title, movieYear AS year
                                     FROM StarsIn);
```

Note on duplication

- SQL uses bag semantics
- Duplicates are always preserved (remember Disney example?)
- Exception: `INTERSECT`, `UNION` and `EXCEPT` use set semantics as default
- If you want to preserve duplicates, you must use `INTERSECT ALL`, `UNION ALL` and `EXCEPT ALL` instead

Eliminating duplication

Keyword `DISTINCT`

```
Movies(title,year,length,genre,studioName,producerC#)
```

```
SELECT studioName
FROM Movies;
```

Gives us a list of Studios with many repetitions

```
SELECT DISTINCT studioName
FROM Movies;
```

Gives us each Studio only once

Aggregation

The basic concepts are similar to those that we saw for the operator γ

Keyword is `GROUP BY`

```
Movies(title,year,length,genre,studioName,producerC#)
```

Find, for each studio, the sum of the lengths of the movies produced by this Studio

```
SELECT studioName, SUM(length) AS totalLength
FROM Movies
GROUP BY studioName;
```

After using a `GROUP BY` clause, *only* the attributes in this clause can be used for the selection.

Each StudioName is then associated with a set of all titles, a set of all years, etc, and these can only be accessed using aggregation

Aggregates are `AVG`, `SUM`, `COUNT`, `MAX`, `MIN`

The following queries are legal (though don't make much sense. . .).

```
SELECT studioName, SUM(length) AS totallength,
     MAX(title) AS maxtitle,
     AVG(year) AS avgyear, SUM(producerC#) AS sumprod
FROM Movies
GROUP BY studioName;
```

```
SELECT studioName, title, SUM(length) AS totallength,
     AVG(year) AS avgyear
FROM Movies
GROUP BY studioName, title;
```

GROUP BY attributes *may* be used in the SELECT-clause, but don't have to be

On the other hand, this is not legal:

```
SELECT studioName, title, SUM(length) AS totallength,
     AVG(year) AS avgyear
FROM Movies
GROUP BY studioName;
```

After using `GROUP BY`, `title` can only be used with an aggregate

We can use `DISTINCT` with aggregation.

Suppose that the movies by Disney have lengths 100, 100, and 120.

Then this query will contain a tuple `('Disney',320)`

```
SELECT studioName, SUM(length) AS totalLength
FROM Movies
GROUP BY studioName;
```

while this will contain a tuple `('Disney',220)`

```
SELECT studioName, SUM(DISTINCT length) AS totalLength
FROM Movies
GROUP BY studioName;
```

Another example

Find the total length of film for each producer (by name)

We need to use two relations, one for the film information, and one to find the producer's name

After taking the join, we can proceed with the aggregation on that single (new) relation

```
Movies(title,year,length,genre,studioName,producerC#)
MovieExec(name,address,cert#,netWorth)
```

```
SELECT name, SUM(length) AS totalLength
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name;
```

Note use of the `WHERE`-clause. This is applied first to create the join, then the `GROUP BY` is applied

More on selection

Suppose we want to evaluate the previous query only for producers with netWorth more than 10.000.000

```
SELECT name, SUM(length) AS totalLength
FROM MovieExec, Movies
WHERE producerC# = cert# AND netWorth > 10000000
GROUP BY name;
```

What is we want to evaluate the query only for producers who made at least one movie before 1930?

```
SELECT name, SUM(length) AS totalLength
FROM MovieExec, Movies
WHERE producerC# = cert# AND year < 1930
GROUP BY name;
```

Why doesn't this work?

`HAVING` clause

This computes the total length of movies made before 1930, while we want to include later movies in the sum

Our condition is actually a condition on each group. We want the group of years corresponding to a producer to contain at least one date before 1930

We could do this with two queries. First, define `R` as

```
SELECT name, SUM(length) AS totalLength, MIN(year) AS minYear
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name;
```

And then select those tuples from `R` with `minYear` less than 1930

- Requires two queries (we'll see later how to do this)
- Rather complicated.

SQL provides the `HAVING` clause so that we can do this in one query

The `HAVING` clause is a condition on aggregates

```
SELECT name, SUM(length) AS totalLength
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

Subqueries

- An SQL query can use subqueries
- Can be used to make the structure of a complex query clearer

Subqueries can produce

- Single values, to be used in `WHERE` clauses
- Relations, that can be used in `WHERE`-clauses and `FROM`-clauses (with a tuple variable)

Subqueries with a single value

Single value: called a *scalar*

User must be sure that such a subquery will only produce a single value

Recall:

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND year=1977
                 AND producerC# = cert#;
```

If we knew `producerC#` (producer code of *Star Wars*) was 1100, we could write

```
SELECT name
FROM  MovieExec
WHERE cert# = 1100;
```

We can find producerC# with the query

```
SELECT producerC#
FROM Movies
WHERE title = 'Star Wars' and year = 1977;
```

Furthermore, since (title,year) is a key for Movies, we know that this query produces a single value, so we can write

```
SELECT name
FROM  MovieExec
WHERE cert# = (
   SELECT producerC#
   FROM Movies
   WHERE title = 'Star Wars' and year = 1977);
```

Note that the schema of the subquery doesn't have to match

What happens if we omit year=1977?

Conditions on relations

What if the subquery does not producer a scalar?

In this example, it will be an error.

Instead of =, we can use other conditions

For now, assume that `R` is a unary relation

- `EXISTS R`: There exists a tuple in R, i.e., R is not empty
- `s IN R`: Holds if s is one of the values in R
- `s NOT IN R`: Holds if s is not equal to any of the values in R
- `s > ALL R`: Holds if s is greater than all the values in R
- `s > ANY R`: Holds if s is greater than some value in R

We can also use >, <=, >=, =, <>.

What do `s=ALL R`, `s=ANY R`, `s<>ALL R`, `s<>=ANY R` mean?

Another example

Find the producers of Harrison Ford's movies

Step 1: Find the title and year of these movies

```
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = 'Harrison Ford';
```

Results include $(Star\ Wars, 1977)$ and $(Raiders\ of\ the\ Lost\ Ark, 1981)$

Step 2: Find the producer codes for each of these.

```
SELECT producerC#
FROM Movies
WHERE title = 'Star Wars' and year = 1977;
```

and

```
SELECT producerC#
FROM Movies
WHERE title = 'Raiders of the lost ark' and year = 1981;
```

We get two codes, say 100 and 111.

We could combine these two queries

```
SELECT producerC#
FROM Movies
WHERE (title,year) IN
   (SELECT movieTitle, movieYear
    FROM StarsIn
    WHERE starName = 'Harrison Ford');
```

Note use of tuple notation for the `IN` clause when the relation is binary

Note also that the two schemas don't have to match

Step 3: Find the names of the producers.

```
SELECT name
FROM MovieExec
WHERE cert# = 100;
```

Yields the name “Gary Kurtz”

```
SELECT name
FROM MovieExec
WHERE cert# = 111;
```

Yields the name “Howard Kazanjian”

We can combine all three parts into

```
SELECT name
FROM MovieExec
WHERE cert# IN
  (SELECT producerC#
   FROM Movies
   WHERE (title,year) IN
      (SELECT movieTitle, movieYear
       FROM StarsIn
       WHERE starName = 'Harrison Ford');
```

Since the first subquery produces a single value, tuple notation is not required

Without subqueries

```
SELECT name
FROM MovieExec, Movies, StarsIn
WHERE cert# = producerC#
  AND title =  movieTitle
  AND movieYear = year
  AND starName = 'Harrison Ford';
```

Alternative

```
WITH R AS
 SELECT producerC#
 FROM Movies
 WHERE (title,year) IN
    (SELECT movieTitle, movieYear
     FROM StarsIn
     WHERE starName = 'Harrison Ford')
SELECT name
FROM MovieExec
WHERE cert# IN R;
```

`WITH` defines R to be a temporary relation, to be used only in this query

Correlated subqueries

Much more complicated use of subqueries

`Movies(title,year,length,genre,studioName,producerC#)`

Find all the titles that have been used more than once

Our running example: “King Kong”, with years 1933, 1976, and 2005

If we already knew this title, we could find the years with

```
SELECT year
FROM Movies
WHERE title = 'King Kong';
```

Title has been reused if we get more than one value, here $\{1933, 1976, 2005\}$

Alternatively, it has been reused, if we can find some year for this movie (1933 or 1976) which is smaller than some value in this set

In other words, if our relation only contained the *King Kong*'s

```
SELECT title
FROM Movies
WHERE year < ANY
    (SELECT year
     FROM Movies
     WHERE title = 'King Kong');
```

In general, introduce a tuple variable `Old` to range over `Movies` and use `Old.title` in the subquery

```
SELECT title
FROM Movies Old
WHERE year < ANY
    (SELECT year
     FROM Movies
     WHERE title = Old.title);
```

Check this gives the same result for *King Kong*. Then check what it does for other movies

Subqueries in FROM clause

Harrison Ford query again. First create a relation with the producer codes

```
SELECT producerC#
FROM Movies, StarsIn
WHERE title =  movieTitle AND
      movieYear = year AND
      starName = 'Harrison Ford';
```

Then use this in the FROM clause

```
SELECT name
FROM MovieExec, (SELECT producerC#
                 FROM Movies, StarsIn
                 WHERE title =  movieTitle AND
                       movieYear = year AND
                       starName = 'Harrison Ford'
                ) Prod
WHERE cert# = Prod.producerC#;
```

Other options

The following keywords can be used as abbreiviations for more complex queries, or simply for readability. They correspond to the algebraic operations that we have seen earlier

```
Movies CROSS JOIN StarsIn
```

Creates $\mathrm{Movies} \times \mathrm{StarsIn}$

Can be used in SQL `FROM`-clause (with tuple variable, if needed)

Theta-Join

```
Movies JOIN StarsIn ON
   title = movieTitle AND year = movieYear;
```

For example,

```
SELECT title, year, length, starName
FROM Movies JOIN StarsIn ON
   title = movieTitle AND year = movieYear;
```

Advantage: Separate join conditions from other conditions for readability

```
SELECT title, length, starName
FROM Movies JOIN StarsIn ON
   title = movieTitle AND year = movieYear;
WHERE year = 1977
```

Natural join

When the attributes for the join are the same

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth
```

```
MovieStar NATURAL JOIN MovieExec
```

creates a relation with attributes

```
name, address, gender, birthdate, cert#, netWorth
```

Outerjoins

Other commands, with the same semantics are the corresponding algebraic operations

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;
MovieStar NATURAL LEFT OUTER JOIN MovieExec;
MovieStar NATURAL RIGHT OUTER JOIN MovieExec;
```

If the attribute names differ, we can use

```
MovieStar FULL OUTER JOIN MovieExec ON
    movieTitle = title AND movieYear = year;
MovieStar LEFT OUTER JOIN MovieExec ON
    movieTitle = title AND movieYear = year;
MovieStar RIGHT OUTER JOIN MovieExec ON
    movieTitle = title AND movieYear = year;
```

Database modification

Database modifications

- Insert: insert new tuples
- Delete: delete duples from the database
- Modify: modify existing tuples

Why modify? We could always insert and delete tuples.

Some of the reasons include

- Efficiency
- Semantics (example later, when we discuss foreign keys)
- Security: A user may be allowed to modify specific fields ony

Insertions

```
INSERT INTO R(A1,...,An) VALUES (v1,...,vn);
```

First part lists the names of the columns in the order that corresponds to the values (remember that the order of the columns has no meaning)

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
   VALUES ('The Maltese Falcon', 1942,
                       'Sydney Greenstreet');
```

But: If we use the order of attributes from the `CREATE TABLE` command, we can omit the attributes here

```
INSERT INTO StarsIn VALUES
   ('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

What if `name` is a key for a relation and we try to insert a tuple with a value that is already in the relation?

The system should give an error message and refuse to update the relation

```
Studio(name,address, presC#)
```

```
INSERT INTO Studio(name) VALUES ('MGM');
```

What happens to the other columns? They become `NULL` (if allowed, otherwise update disallowed)

We get the tuple `('MGM',NULL,NULL)`

```
Movies(title,year,length,genre,studioName,producerC#)
```

```
Studio(name,address, presC#)
```

Suppose we want to insert in `Studio` those studio names that are missing from `Movies` (and only these studios). Other attribute values will have to be `NULL`

```
INSERT INTO Studio(name)
  SELECT DISTINCT studioName
  FROM Movies
  WHERE studioName NOT IN
     (SELECT name
      FROM Studio);
```