Intro to Relational Database Design

This PDF is completed and recording is watched

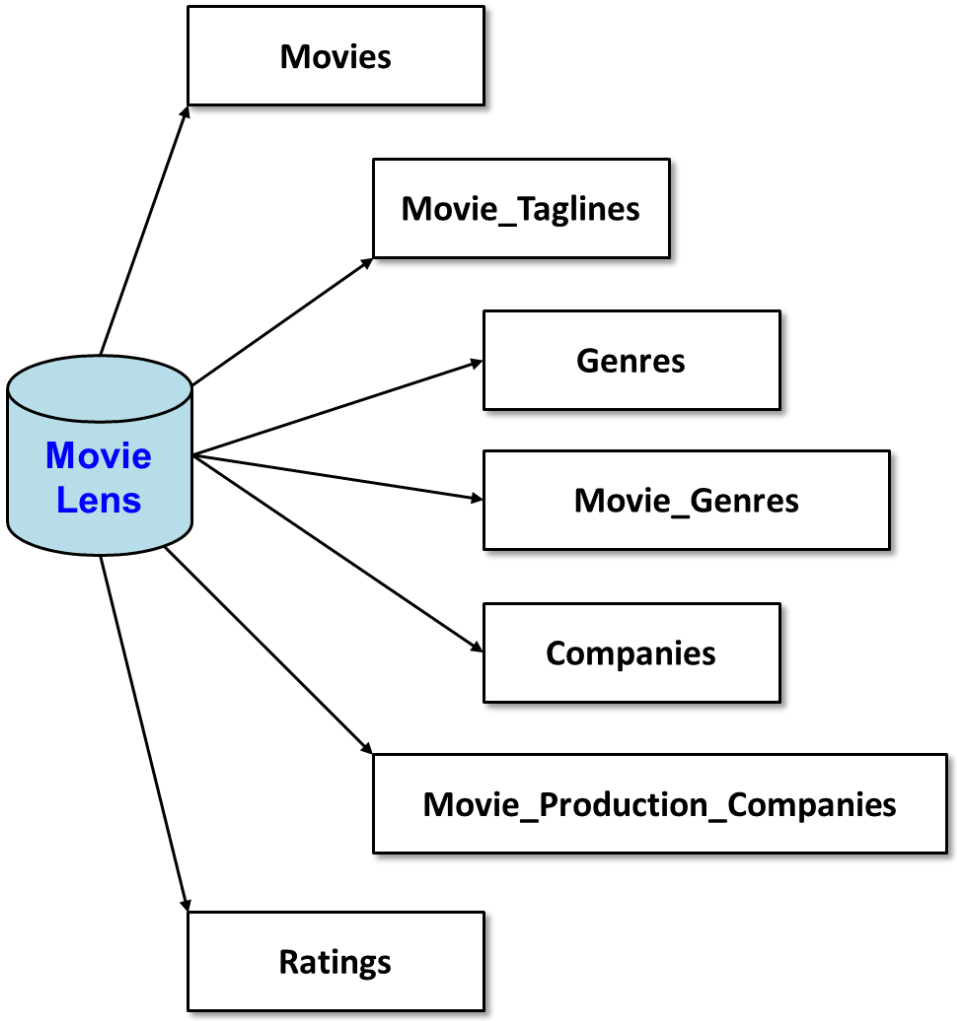
- The 3 rules of database design
- Example: MovieLens
- Indexing to improve performance
- Stored procedures
- Transactions

Database design

- **Database design is an art…**
- **But there are 3 fundamental rules to always consider**

MovieLens database

- **MovieLens**
  - *https://movielens.org/*
  - *45K movies*
  - *26M reviews*



Why 7 tables?

Three fundamental rules of DB design

Here, the table breaks rule 1, as it is storing genres, movie multiple times. We need to store data once. Therefore, break down the table into multiple table like movies, genres which will have them once.

1. **Store data exactly once**

Assign a unique identifier called primary ID to each row in a table

2. **Every row has a unique key to identify that row**

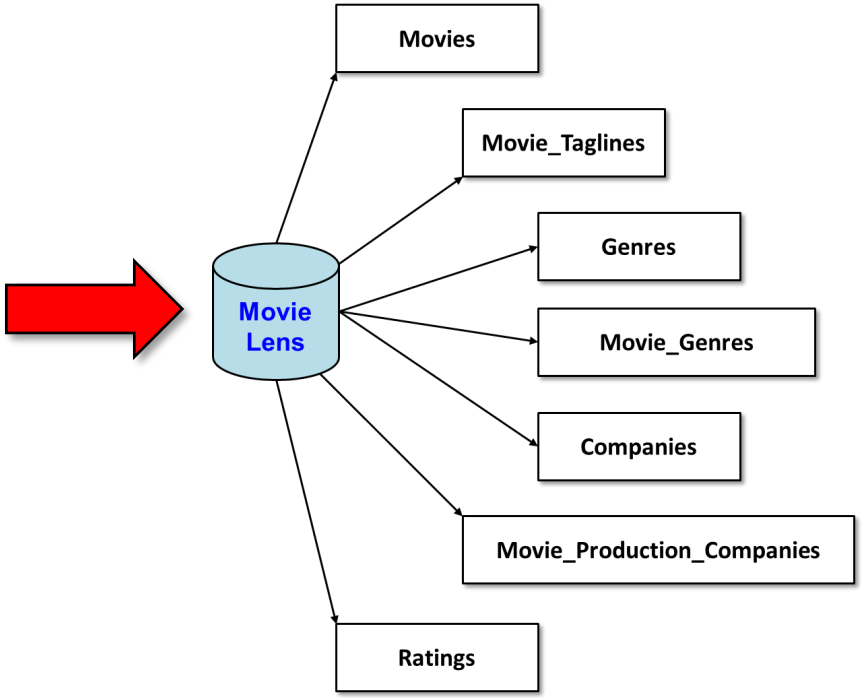
3. **Store one value per cell**

In Genres column, more than 1 value is stored. Its better to have 1 in each row. Eg Movie Genres -> This Table holds Genres for each movie.

Id	Movie	Genre
1	Matrix	Action
2.	Matrix	Science
3.	Matrix	Comedy

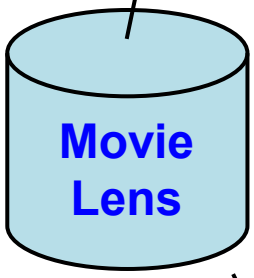
Original data (csv file)

Movie	Genres	Year	Revenue	Rating
The Matrix	Action; Science Fiction	1999	463517383	9
Toy Story	Animation; Comedy	1995	373554033	8
The Matrix	Action; Science Fiction	1999	463517383	10
The Matrix	Action; Science Fiction	1999	463517383	6
...	...	...	...	...



Movie_ID	Title	Release_Date	Runtime	Original_Language	Budget	Revenue
603	The Matrix	1999-03-30 00:00:00.000	136	en	63000000	463517383
862	Toy Story	1995-10-30 00:00:00.000	81	en	30000000	373554033
…	…	…	…	…	…	…

Movies



Ratings

Movie_ID	Rating
605	8
603	6
605	10
605	6
…	…

Genres

Genre_ID	Genre_Name
28	Action
878	Science Fiction
16	Animation
35	Comedy
…	…

Movie_Genres

Linking Table

Movie_ID	Genre_ID
862	16
862	35
603	28
603	878
…	…

Primary Key

Primary key *--- unique row identifier.*
Every table should have a PK.

Movie_ID	Title	Release_Date	Runtime	Original_Language	Budget	Revenue
603	The Matrix	1999-03-30 00:00:00.000	136	en	63000000	463517383
862	Toy Story	1995-10-30 00:00:00.000	81	en	30000000	373554033
...	...	...	...	...	...	...

Movies

Movie Lens

Ratings

Movie_ID	Rating
605	8
603	6
605	10
605	6
...	...

Genres

Genre_ID	Genre_Name
28	Action
878	Science Fiction
16	Animation
35	Comedy
...	...

Movie_Genres

Movie_ID	Genre_ID
862	16
862	35
603	28
603	878
...	...

Foreign Key

Foreign key *--- a primary key stored in another table. FK is what allows data to be* ***joined*** *back together.*

Movie_ID	Title	Release_Date				
603	The Matrix	1999-03-30 00:00:00.000	136		0000000	463517383
862	Toy Story	1995-10-30 00:00:00.000		en	30000000	373554033
...	...	...	...	...	...	...

Movies

Movie Lens

Ratings

Movie_ID	Rating
605	8
603	6
605	10
605	6
...	...

No need to add Primary Key everywhere for the tables, which are not individually referenced.

Genres

Genre_ID	Genre_Name
28	Action
878	Science Fiction
16	Animation
35	Comedy
...	...

Movie_Genres

Movie_ID	Genre_ID
862	16
862	35
603	28
603	878
...	...

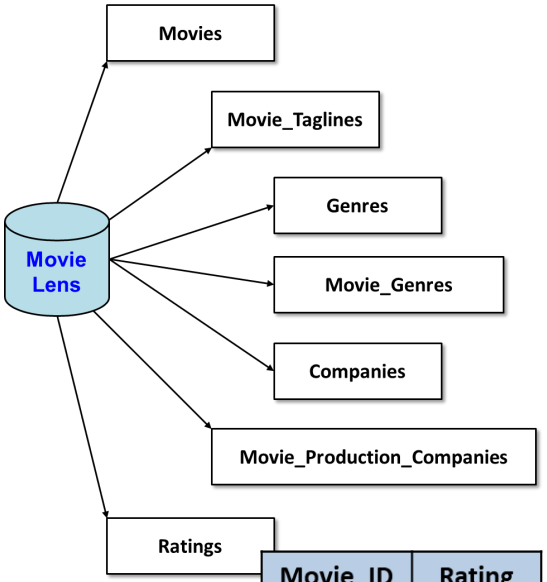
GenreID : Foreign Key for the Genres and Movie Genres Table

Creating MovieLens using DDL

- **Databases are created using SQL…**

```
CREATE DATABASE MovieLens;

CREATE TABLE Movies(
    Movie_ID           INT NOT NULL,
    Release_Date       TEXT NOT NULL,
    Runtime            INT NOT NULL,
    Original_Language  TEXT NOT NULL,
    Budget             INT NOT NULL,
    Revenue            INT NOT NULL,
    Title              TEXT NOT NULL,
    PRIMARY KEY (Movie_ID)
);

CREATE TABLE Ratings(
    Movie_ID  INT NOT NULL,
    Rating    INT NOT NULL

    -- no primary key --
    FOREIGN KEY (Movie_ID) REFERENCES Movies (Movie_ID)
);
```

Movie_ID	Title	Release_Date	Runtime	Original_Language	Budget	Revenue
603	The Matrix	1999-03-30 00:00:00.000	136	en	63000000	463517383
862	Toy Story	1995-10-30 00:00:00.000	81	en	30000000	373554033
…	…	…	…	…	…	…

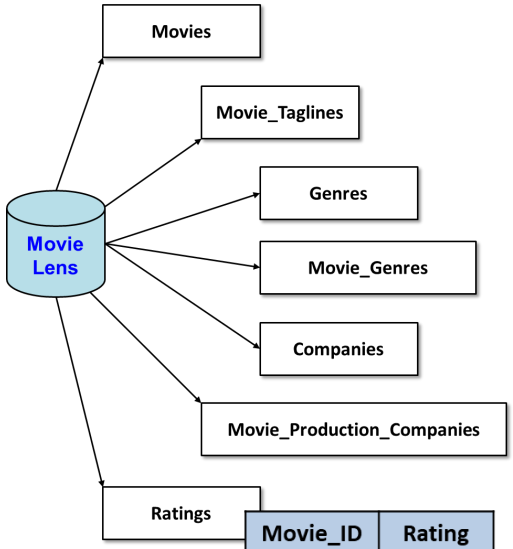


Movie_ID	Rating
605	8
603	6
605	10
605	6
…	…

Constraints

- **Databases allow constraints on data – use them!**

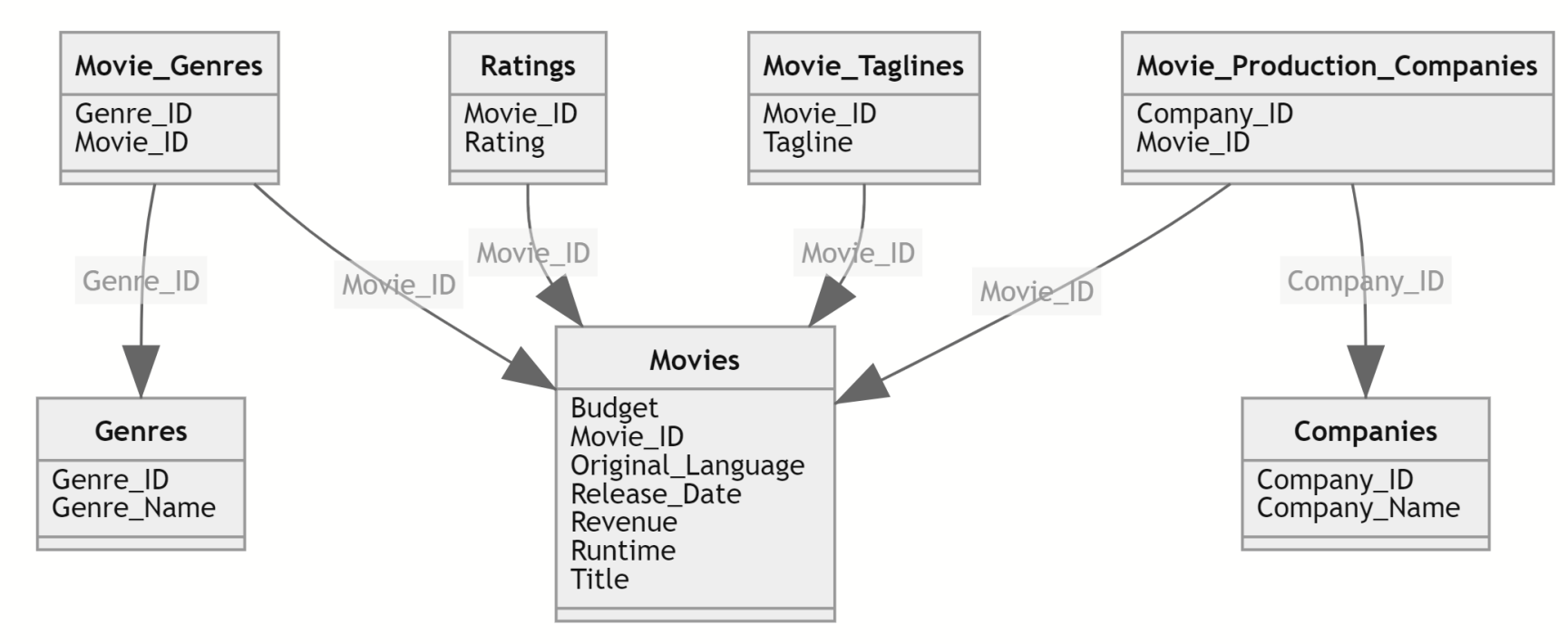
```
CREATE DATABASE MovieLens;

CREATE TABLE Movies(
    Movie_ID          INT NOT NULL,
    Release_Date      TEXT NOT NULL,
    Runtime           INT NOT NULL,
    Original_Language TEXT NOT NULL,
    Budget            INT NOT NULL,
    Revenue           INT NOT NULL,
    Title             TEXT NOT NULL,
    PRIMARY KEY (Movie_ID)
);

CREATE TABLE Ratings(
    Movie_ID  INT NOT NULL,
    Rating    INT NOT NULL CHECK(Rating >= 0
                             AND Rating <= 10),

    -- no primary key --
    FOREIGN KEY (Movie_ID) REFERENCES Movies (Movie_ID)
);
```



Movie_ID	Rating
605	8
603	6
605	10
605	6
…	…

Database schema (design)

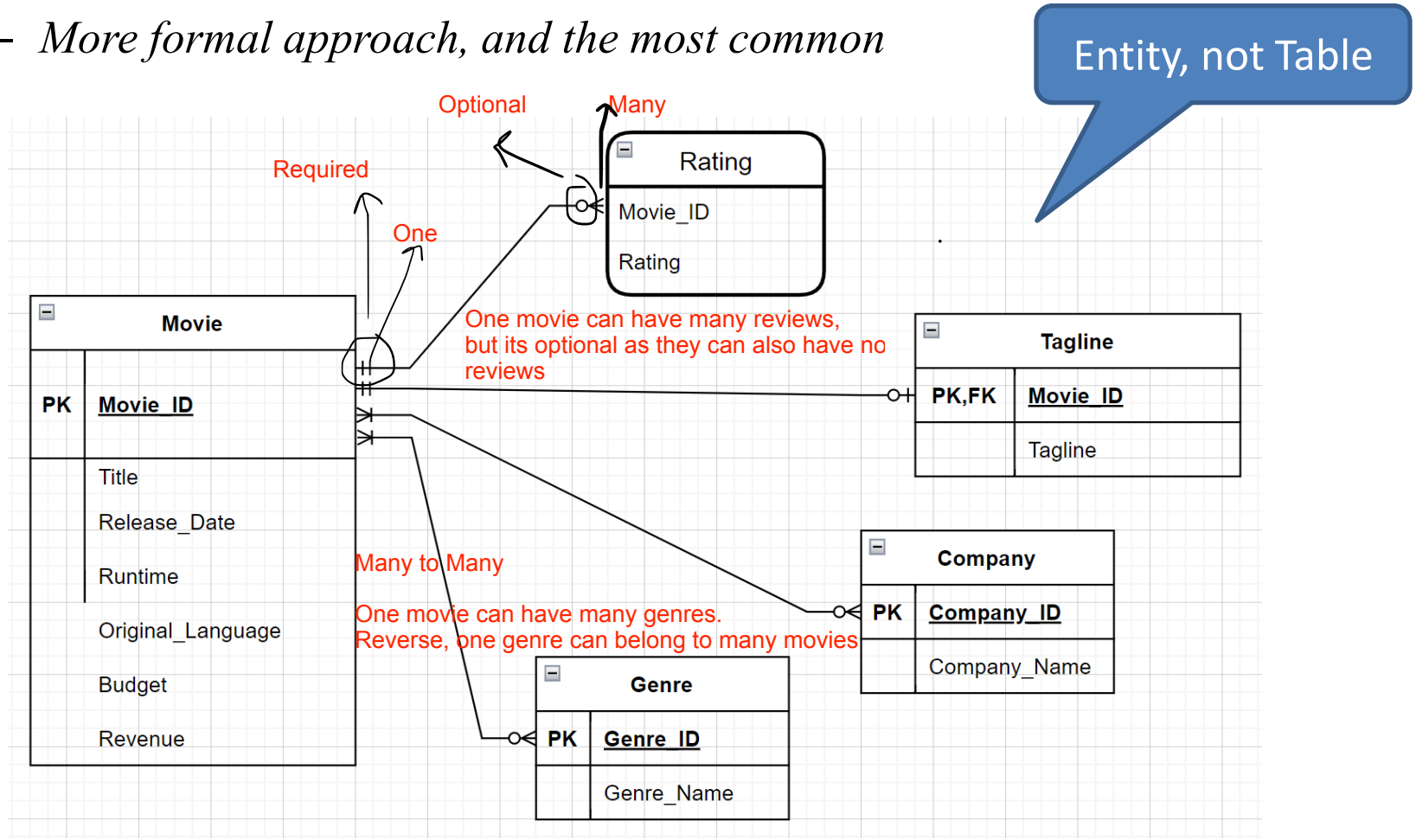
- One view is the SQL used to create the database
- Another view is some sort of visual diagram…



Produced by Azure Data Studio via "Schema Visualization" extension

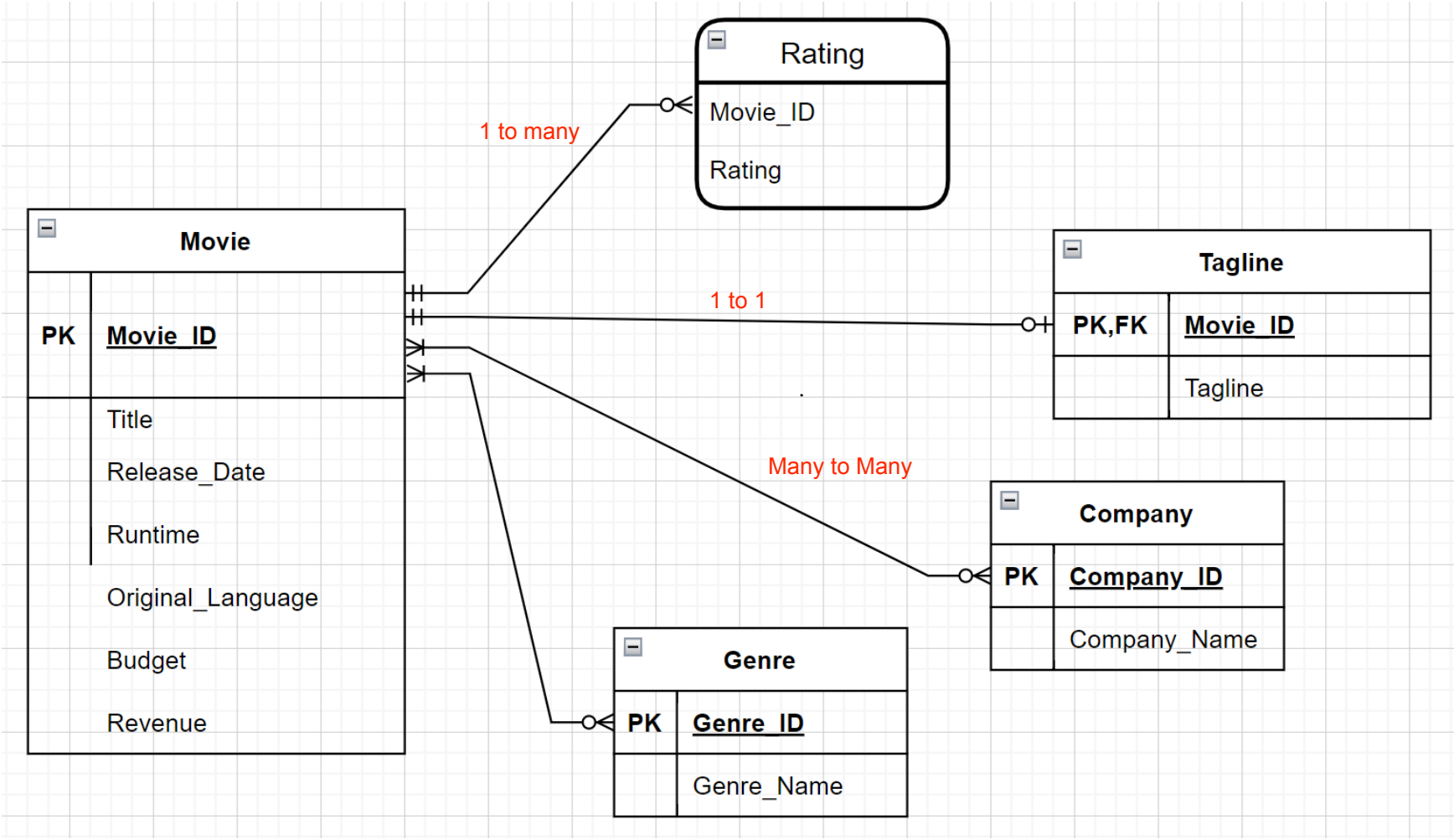
ER Diagram

- **Entity Relationship diagram…**
  - *More formal approach, and the most common*



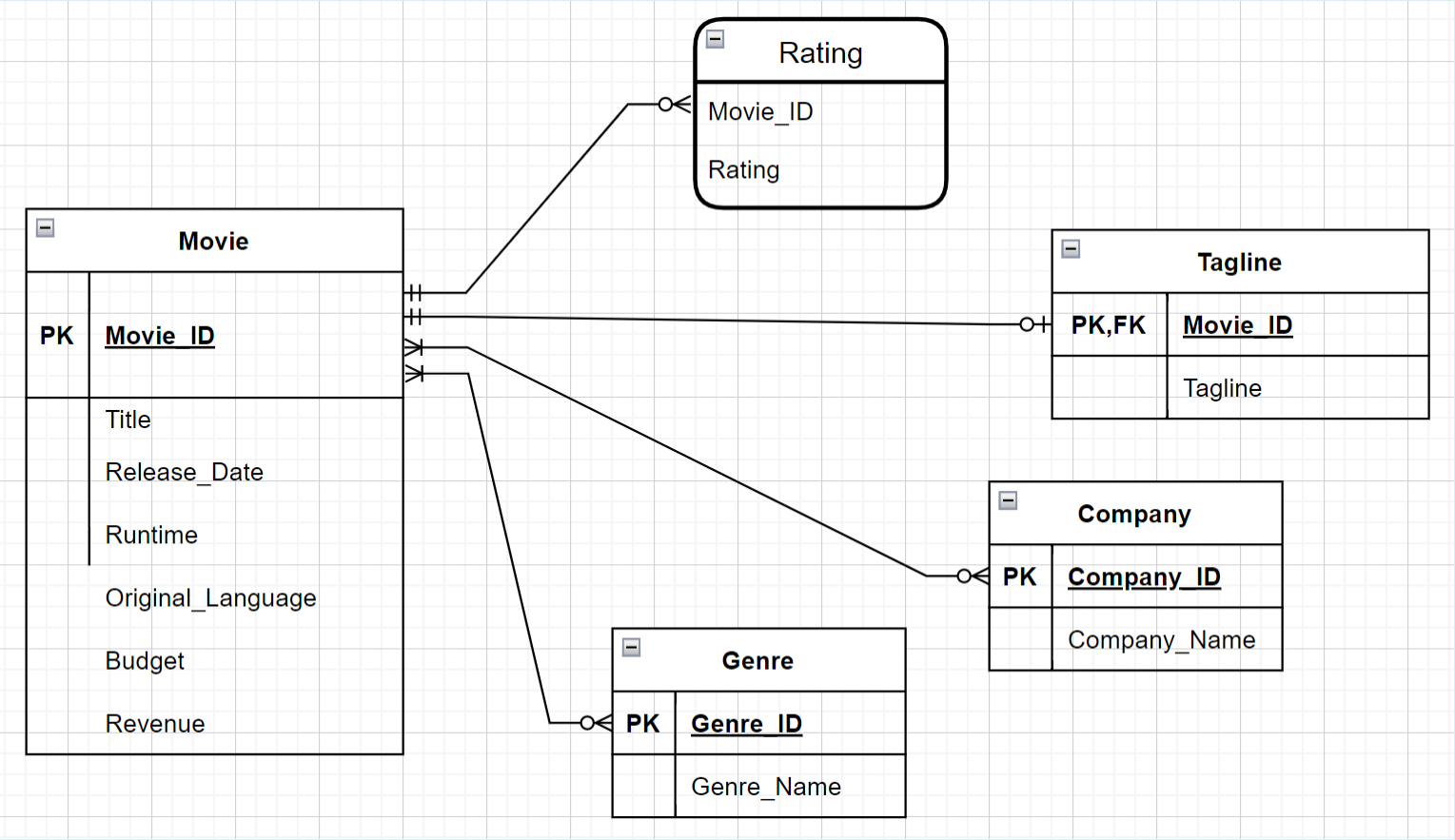
Drawn manually using free tool: https://app.diagrams.net/

- **Types of relationships**
  - *1-to-1*
  - *1-to-many*
  - *many-to-many*



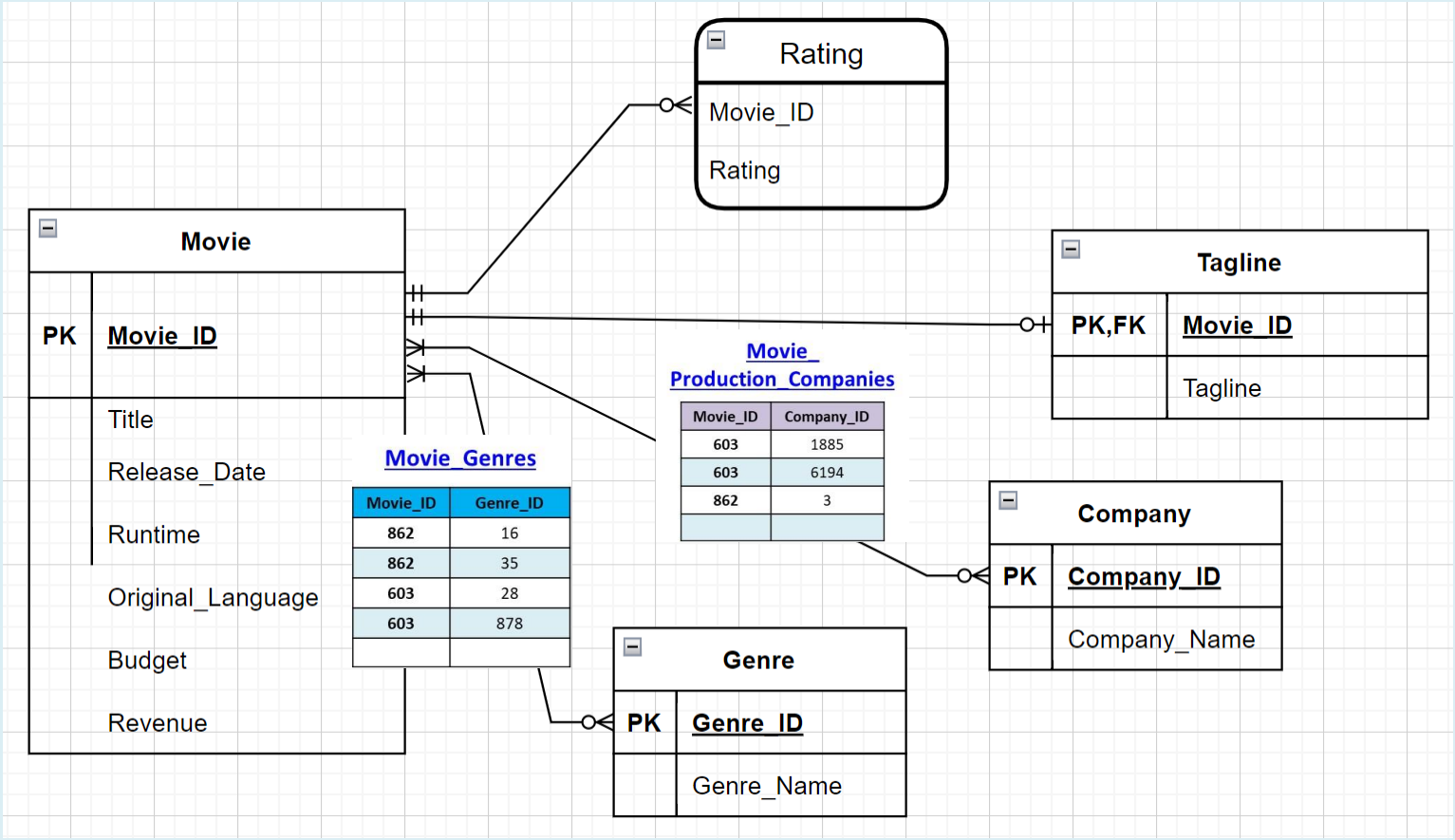
Question

- **Interestingly, what low-level detail is *missing* from the ER diagram?**



Answer

- **Many-to-many relationships imply a linking or bridge table to implement the connection…**

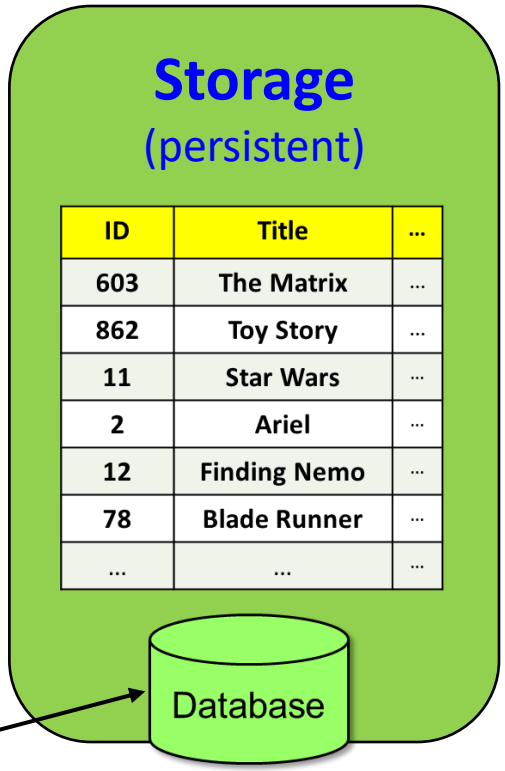


Performance: Scan vs. Seek

- **By default, DBMS has to "scan" when searching**
  - *Row by row search of all movies in persistent store (file sys)*

This is usually slow as it scans all rows of the database to fine the answer. To improve the searching, we can use seek, which works indexing.

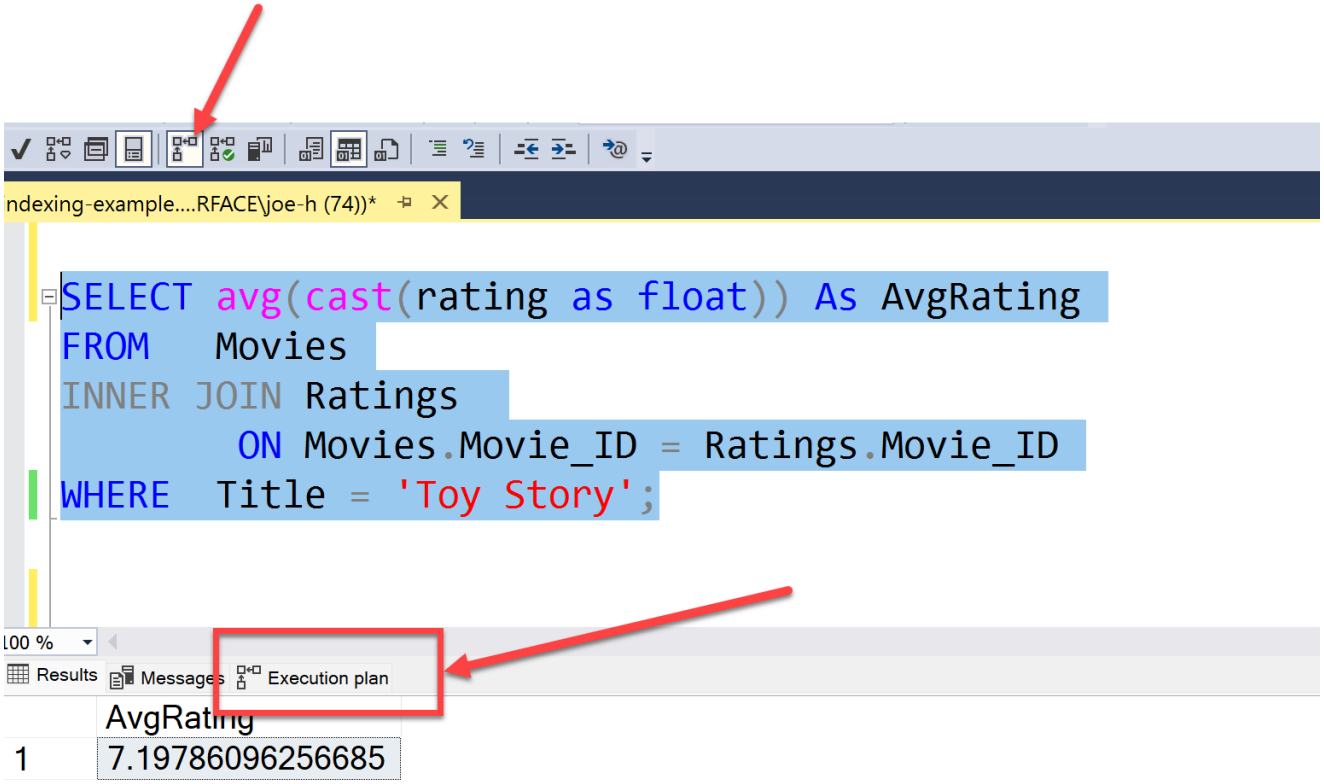
- **Example:**
  - *Searching for a movie by title*




```
SELECT *
FROM Movies
WHERE Title = 'Blade Runner';
```

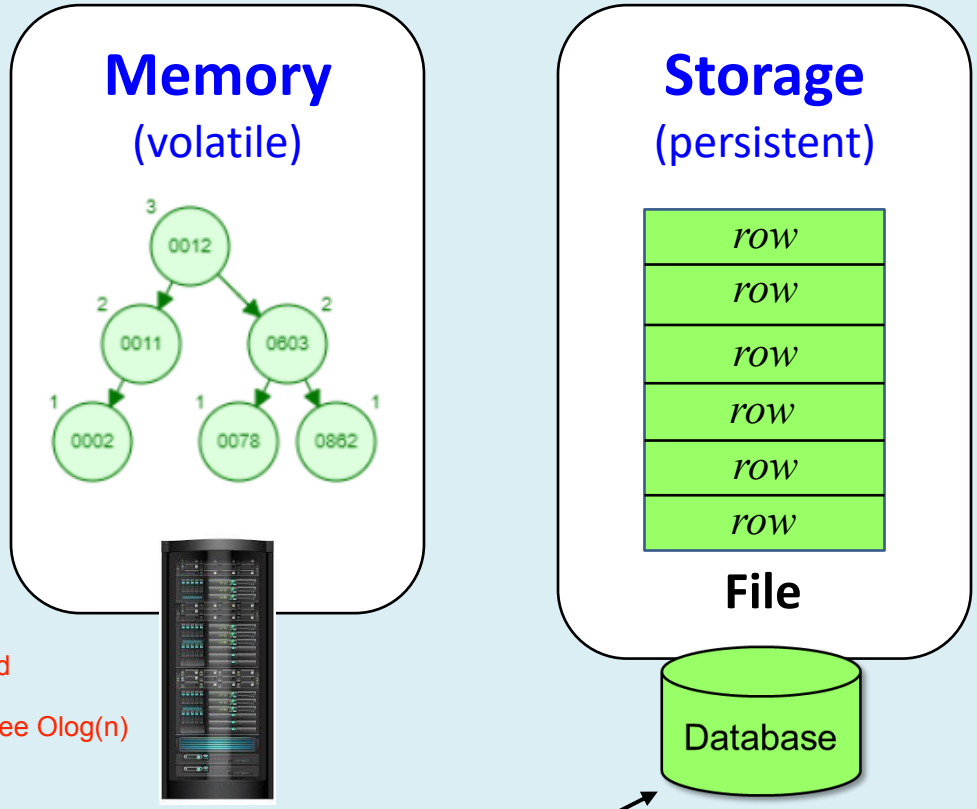
Query plan

- **Query plan** is how DBMS plans to execute query
- Most database tools can display plan…



Indexing

- **With an index, DBMS can "seek"**
  - *Search of volatile memory + **1 access** to persistent store*
  - *Primary key always indexed*
- **DBMS builds @ startup**
  - *Indexes data residing in persistent storage*



Whenever a database is indexed, a binary tree like structure is created in the RAM, and the indexes point to the actual data in the Hard Disk. Whenever we query something, we need to do a search on indexed tree Olog(n) and then 1 access to HDD.

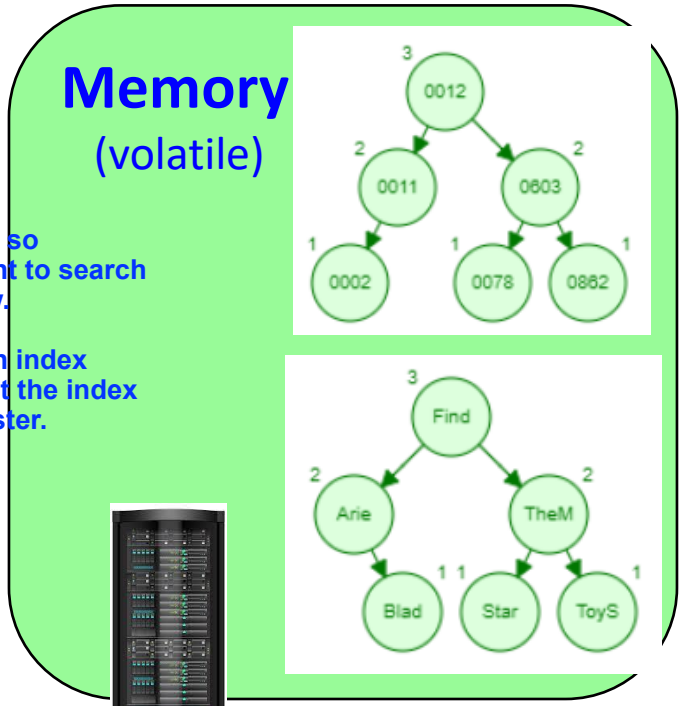
```
SELECT Title
FROM   Movies
WHERE  Movie_ID = 78;
```

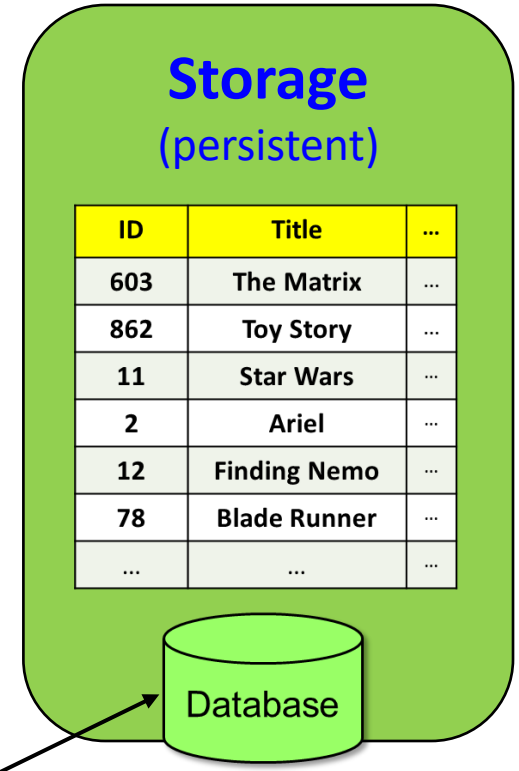
Example #1

- Searching by movie id is fast (primary key is indexed)
- Searching by movie title is slow --- add index!

By default primary key is always indexed, so search on it is always faster, but if we want to search on any other column, it will be pretty slow.

The way to improve the speed is create an index on that column. For eg. Here we have built the index over Title which have made the search faster.






```
SELECT Title
FROM Movies
WHERE Title = 'Blade Runner';
```

DBMS

```
CREATE INDEX Title_Index
ON Movies(Title);
```

Example #2

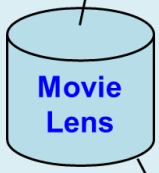
- There are 26MB reviews
- Computation of avg rating…

The problem here is Movie ID from Movies table is indexed so, we are able to find the movies pretty quickly but Ratings table is not indexed , so at the time of joining Movies and Rating, it has to go through each entry in Ratings (scan) and hence is slow

The solution is to index the Movie_ID from Ratings, this will speed up the query.

Movie_ID	Title	Release_Date	Runtime	Original_Language	Budget	Revenue
603	The Matrix	1999-03-30 00:00:00.000	136	en	63000000	463517383
862	Toy Story	1995-10-30 00:00:00.000	81	en	30000000	373554033

Movies



Ratings

Movie_ID	Rating
605	8
603	6
605	10
605	6

```
SELECT Avg(cast(Rating as float))
FROM   Movies
INNER JOIN Ratings
        ON Movies.Movie_ID = Ratings.Movie_ID
WHERE  Title = 'Toy Story';
```

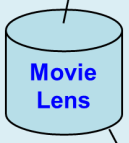
Demo

- **Without index, query takes 4-5 seconds…**
  - *Not bad, but we can do much better…*

```
SELECT Avg(cast(Rating) as float)
FROM   Movies
INNER JOIN Ratings
        ON Movies.Movie_ID = Ratings.Movie_ID
WHERE  Title = 'Toy Story';
```

Movie_ID	Title	Release_Date	Runtime	O…		
603	The Matrix	1999-03-30 00:00:00.000	136			
862	Toy Story	1995-10-30 00:00:00.000	81	en	30000000	373554033

Movies



Ratings

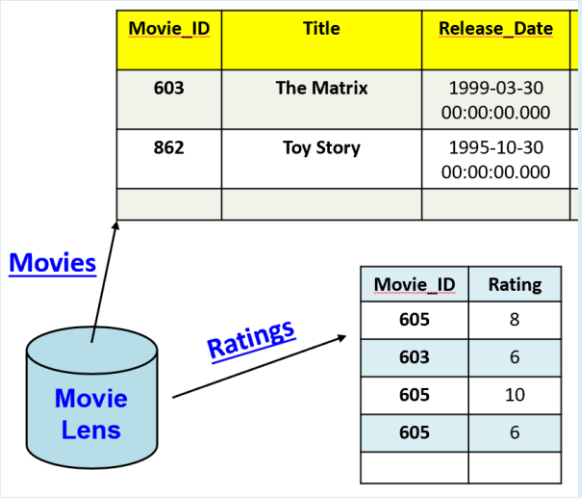
Movie_ID	Rating
605	8
603	6
605	10
605	6

Solution : index Ratings by Movie_ID

- With correct index, query takes 1/10th of a second

```
CREATE INDEX Ratings_Index ON Ratings(Movie_ID);
```

```
SELECT Avg(cast(Rating) as float)
FROM   Movies
INNER JOIN Ratings
        ON Movies.Movie_ID = Ratings.Movie_ID
WHERE  Title = 'Toy Story';
```

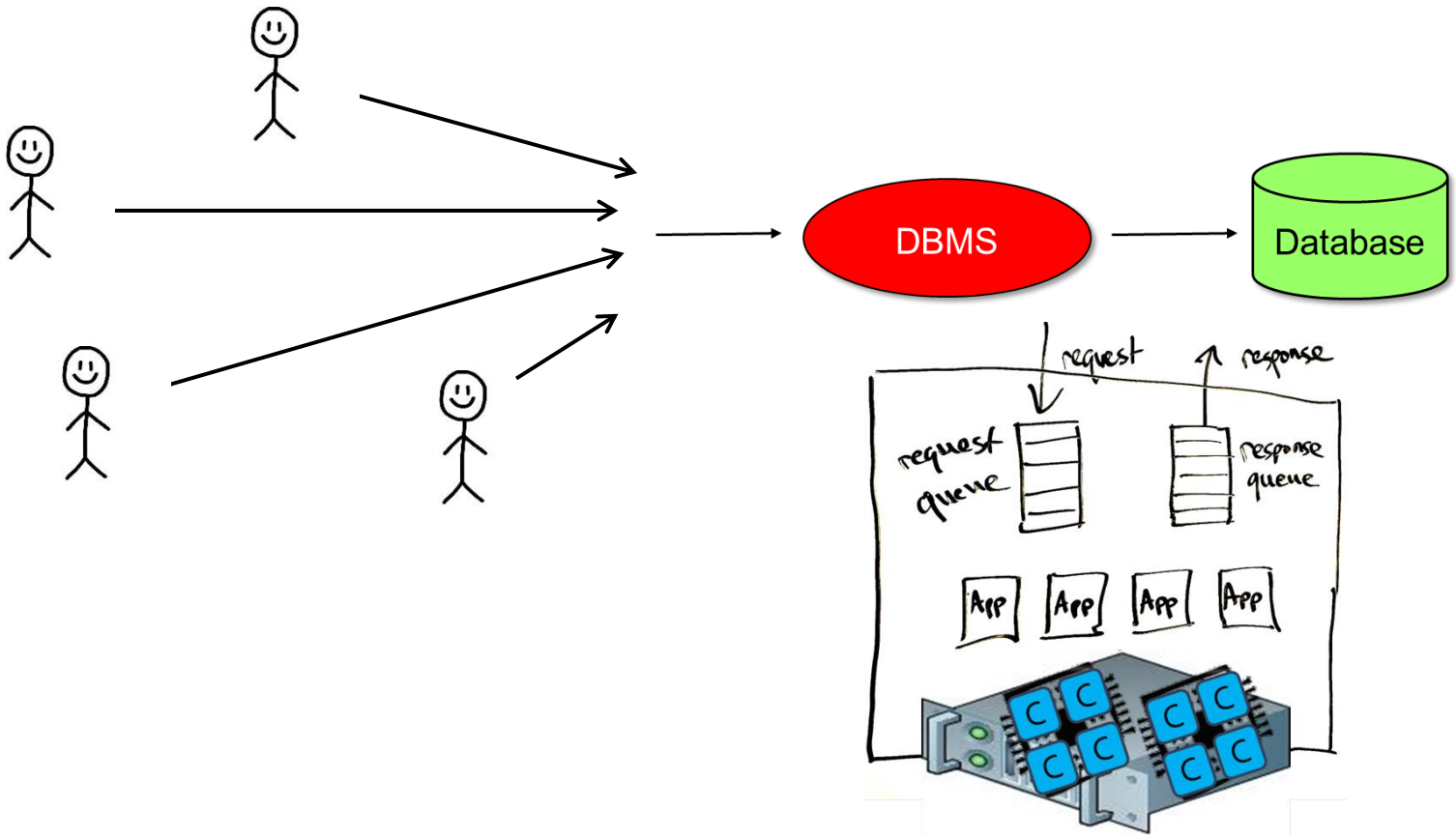


Trade-offs

- **Why not create an index on every column?**
  - *Takes more server memory for underlying data structures*
  - *Slows down inserts / updates / deletes since have to maintain underyling data structures* becuase these structures are needed to built again with each modifcation.

- **General approach is to create indexes on the obvious columns, then deploy and monitor**
  - *See how DB is used, and add more indexes as needed*

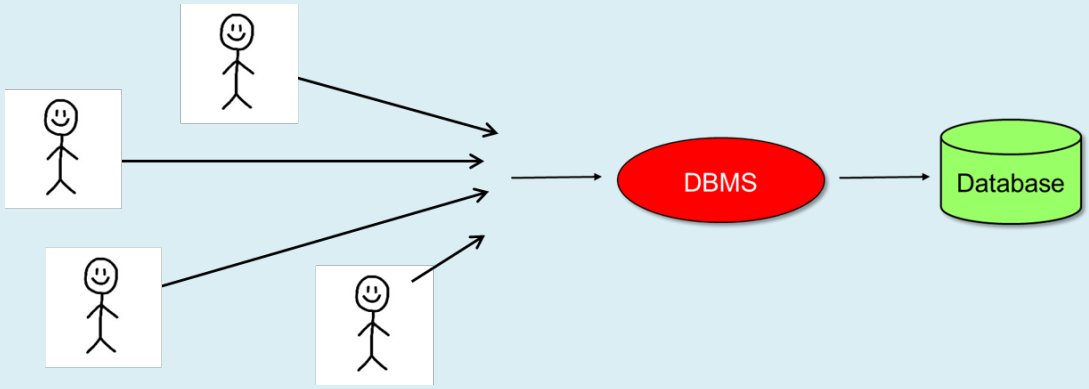
Databases are powerful systems



- Can handle thousands of concurrent users
- Handle simultaneous insert / update / delete
- Take advantage of multi-core hardware

Question

- **We are building a ticket selling app**
  - *everyone is trying to buy 2 tickets*



This is unsafe in a multi-user scenario. How to fix?

Here, since all users will access the Database at the same time and try to perfrom the same action, it can be unsafe and cause unwanted results. The query will make the seats go from 2 -> 0 -> -2 -> -4 etc...

```
sql = 'Select num_avail From Tickets Where eventid=1234;'
dbCursor.execute(sql);
row = dbCursor.fetchone()
numAvail = int(row[0])

if numAvail >= 2:
  success = True
  sql = "Update Tickets Set num_avail = num_avail-2 Where eventid=1234;'
  dbCursor.execute(sql)
else:
  success = False
```

Solution: Transactions

- Transaction isolate operations from other users
- Transactions make a series of operation atomic

```
BEGIN TRANSACTION;

  -- withdraw from checking
  UPDATE Accounts
  SET    Balance = Balance - 100.00
  WHERE  Account = 22197;

  -- deposit into savings
  UPDATE Accounts
  SET    Balance = Balance + 100.00
  WHERE  Account = 43992;

COMMIT;
```

Take this example, if power goes off after withdraw, then the data is lost, no way to resume and perform deposit.

Therefore, wrap the whole thing under one transaction object. Until the transaction is commited the database will not be permanently updated.

If anything goes wrong during the transaction, any temporary changes will be reverted back.

Beware

- **For better performance, transactions default to "not-quite-perfect" isolation between users**
- **Solutions**

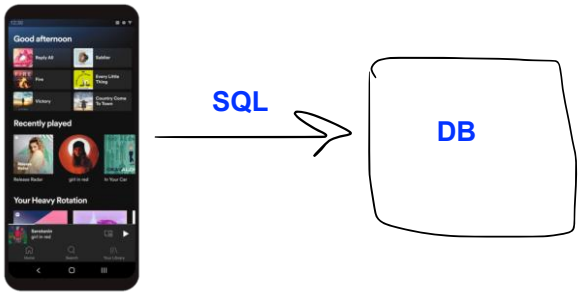
(1) isolation level = Serializable

*(2) default isolation level + "Select … **For Update**;"*

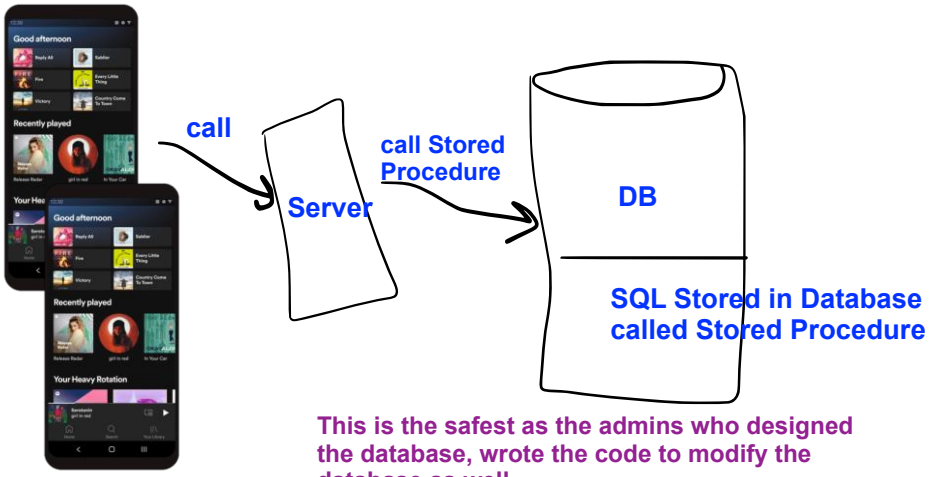
```
BEGIN TRANSACTION
sql = 'Select num_avail From Tickets Where eventid=1234 For Update;'
dbCursor.execute(sql);
row = dbCursor.fetchone()
numAvail = int(row[0])

if numAvail >= 2:
  success = True
  sql = "Update Tickets Set num_avail = num_avail-2 Where eventid=1234;'
  dbCursor.execute(sql)
  COMMIT
else:
  success = False
  ROLLBACK
```

This level of understanding is why you want DBAs writing stored procedures

Design: who executes the SQL?

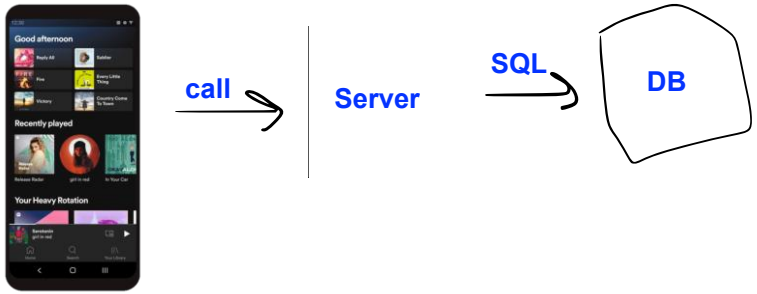


(1) Client
(project 01)



This is the safest as the admins who designed the database, wrote the code to modify the database as well.

(3) Stored Procedures



(2) Server
(project 02)

Stored Procedures

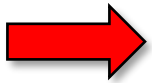
- **Functions stored in the database**
  - *Support parameters and nearly all features of SQL*
  - *Can retrieve, insert, update, and delete data*

```
EXEC AddMovieReview 603, 10;
```

```
--
-- inserts a review into the database
--
Create Procedure AddMovieReview
  @id int,
  @rating tinyint
As
Begin
  BEGIN TRANSACTION;
  Insert Into Ratings(Movie_ID, Rating) Values(@id, @rating);
  COMMIT;
  RETURN @@ROWCOUNT;  -- # of rows modified
End;
```

Top N movies with Min ratings

```
EXEC TopN 3, 500;
```



Movie_ID	Title	NumRatings	AvgRating
318	The Million Dollar Hotel	768	8.87890625
527	Once Were Warriors	559	8.554561717352415
296	Terminator 3: Rise of the Ma…	755	8.512582781456954

```
--
-- retrieves top N movies with at least Min ratings
--
Create Procedure TopN
  @N int,    -- retrieve N movies
  @Min int   -- must have >= Min ratings
As
Begin
  Select Top (@N)
         Movies.Movie_ID,
         Title,
         Count(Rating) As NumRatings,
         Avg(Cast(Rating as real)) As AvgRating
  From  Movies
  Inner Join Ratings on Movies.Movie_ID = Ratings.Movie_ID
  Group By Movies.Movie_ID, Movies.Title
  Having Count(Rating) >= @Min
  Order By AvgRating DESC, Title ASC;
End;
```

Why stored procedures?

- **Faster**
- **More trustworthy!**

- **Typically written by database designer(s)**
  - *Experts in SQL*
  - *Understand details of database design*

That's it, thank you!