Today

Data Modeling in RDBMS

Real World Entity

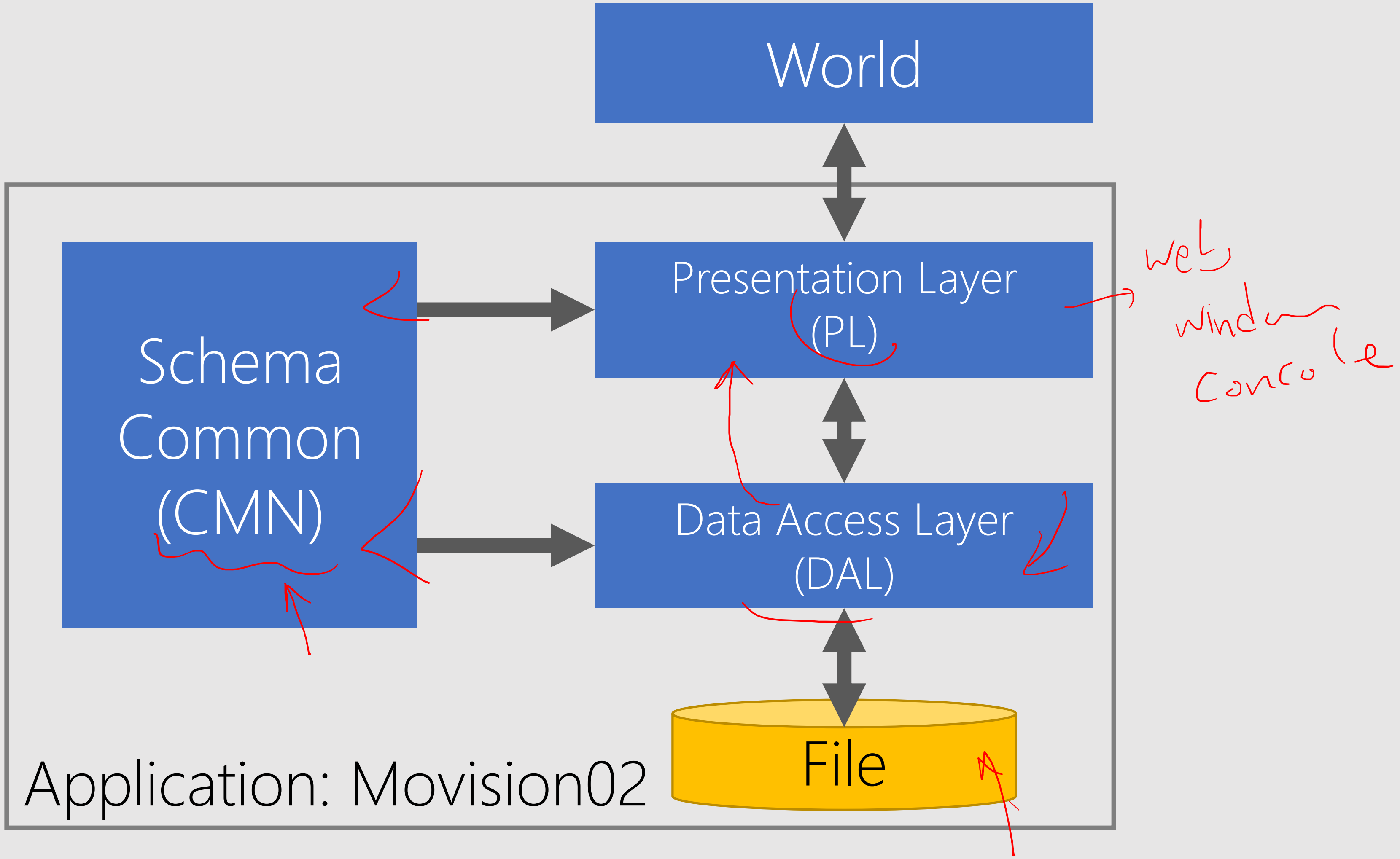
Conceptual Level | Entity-Relationship Model (E/R)

| Logical Level | Relational Model

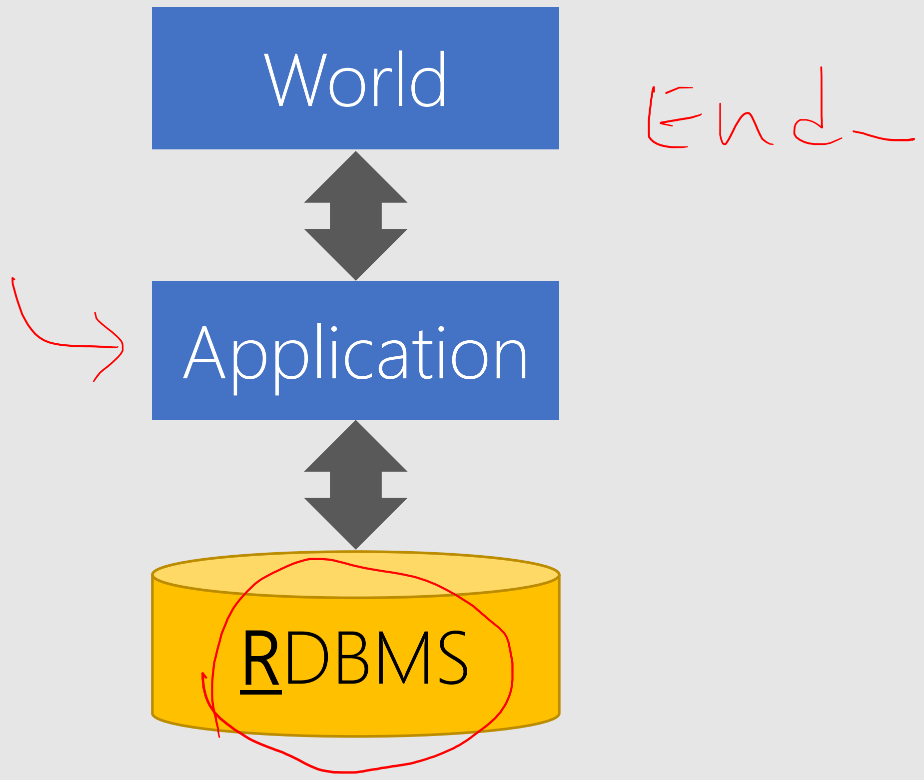
| Physical Level | SQL

Computable Entity

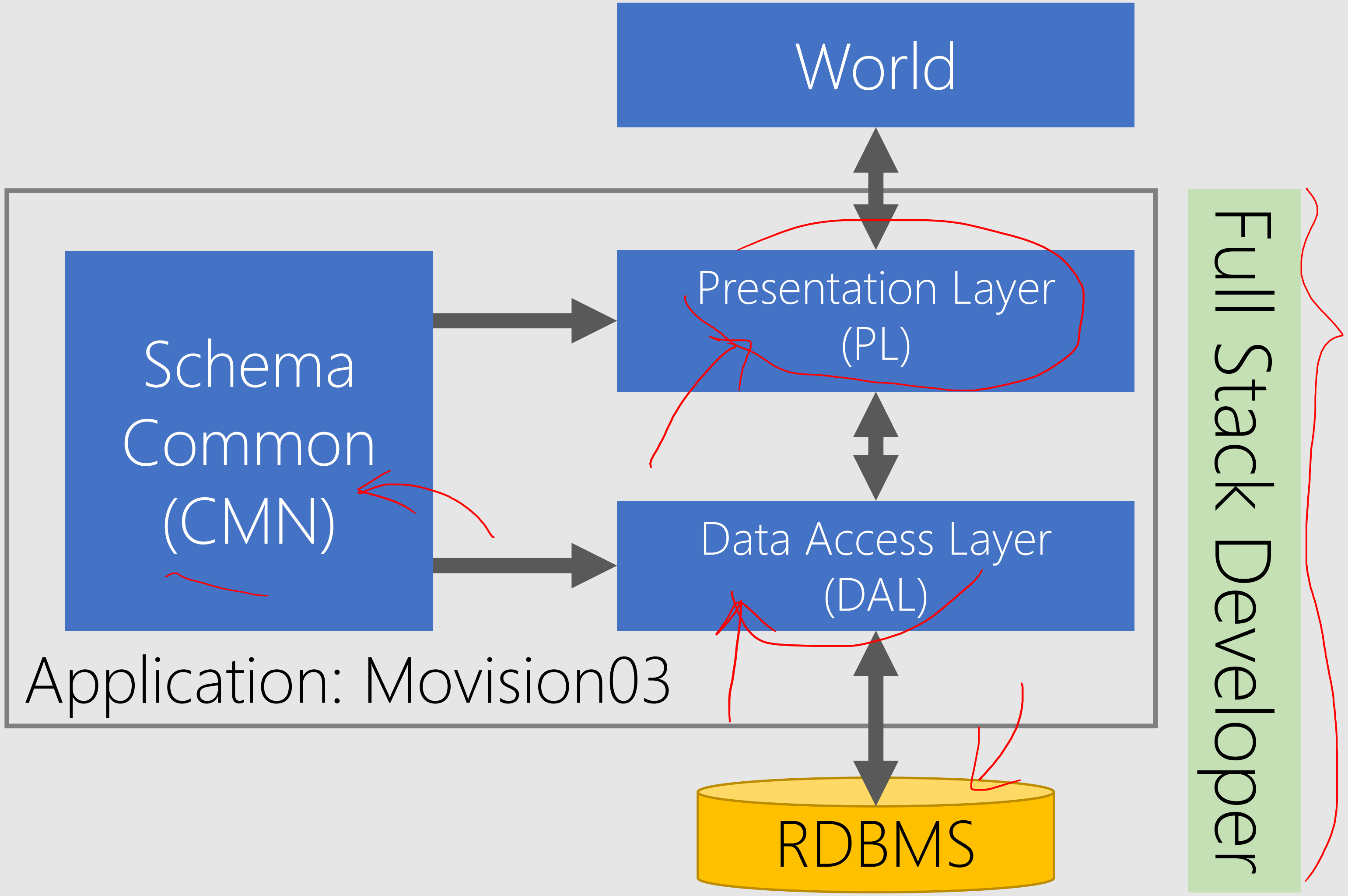
Physical Level × File



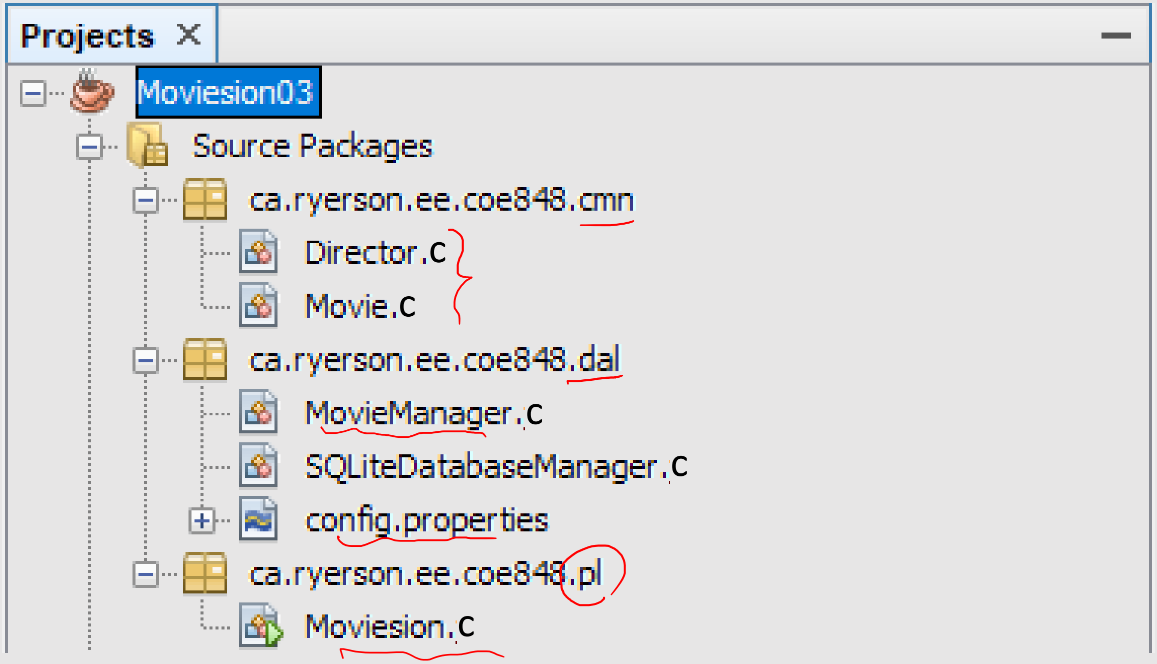
Physical Level × RDBMS



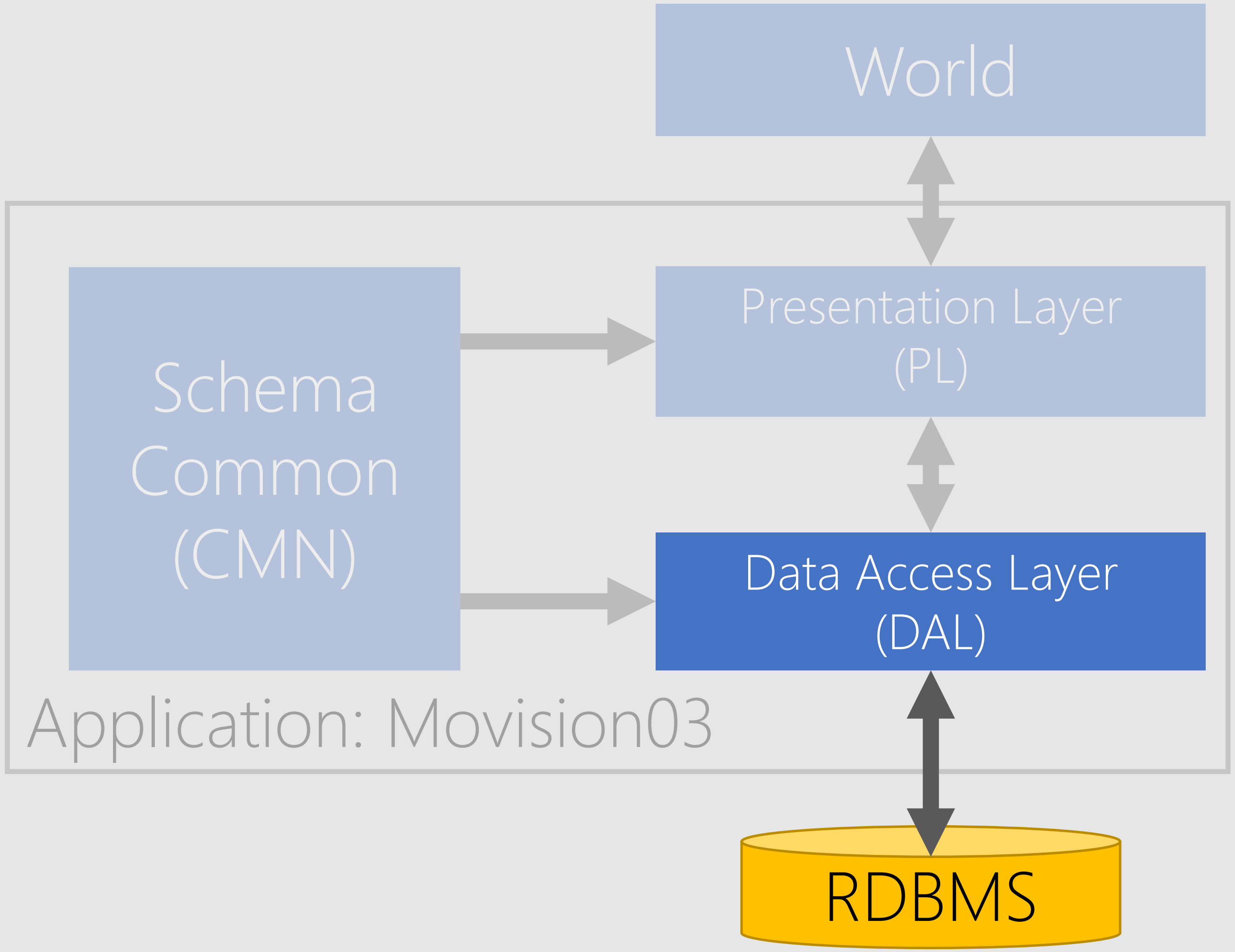
Physical Level × RDBMS



Physical Level × RDBMS 5



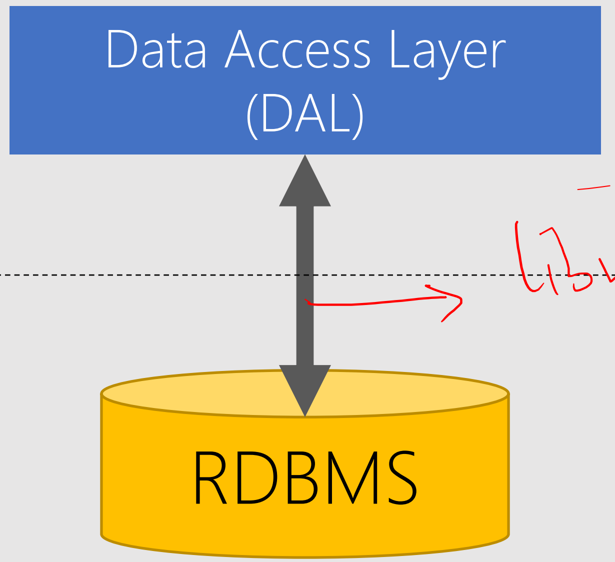
Physical Level × ORM



Physical Level × ORM

Object Relational Mapping (ORM)

Object | struct



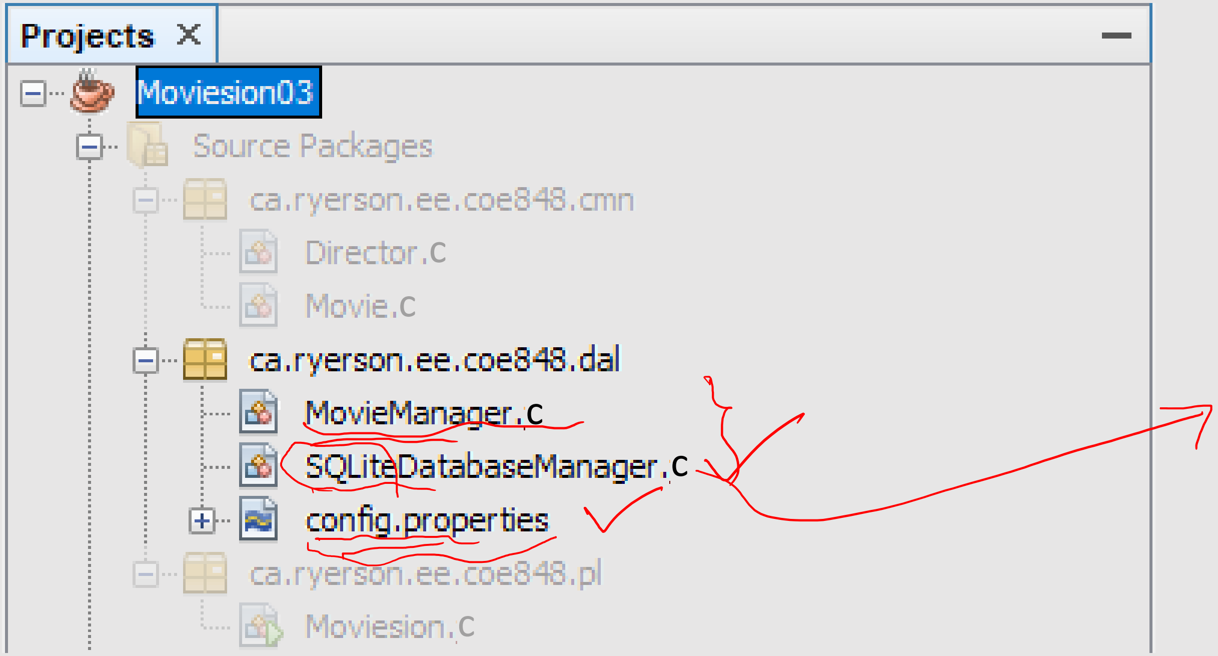
Function

lib sqlite3.h

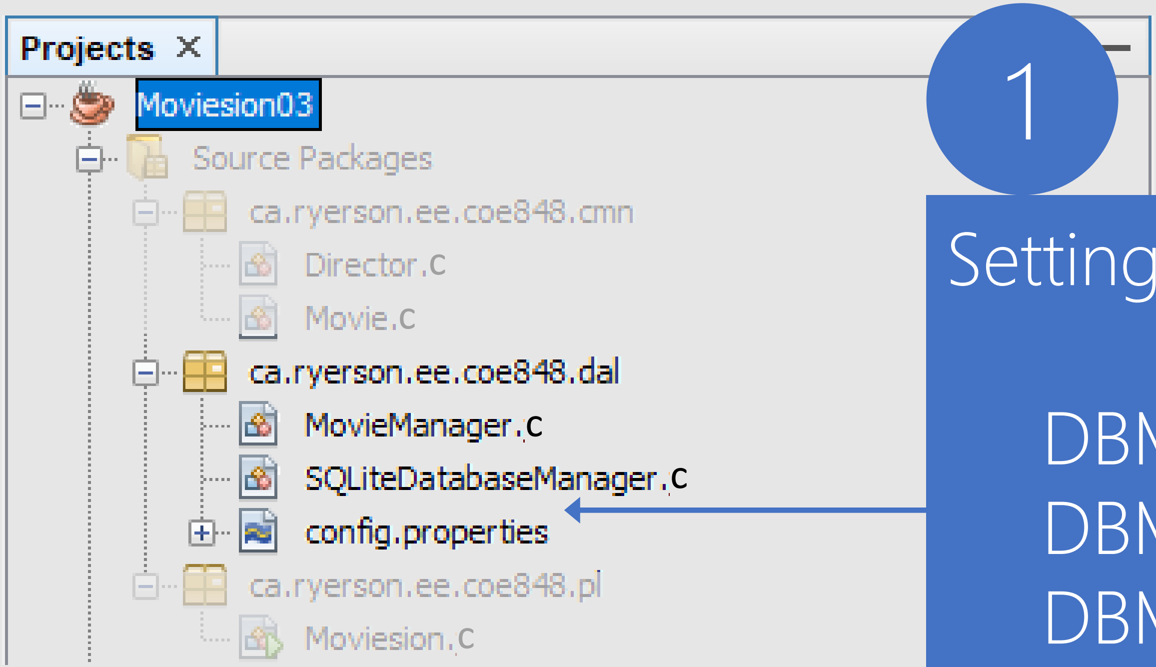
Relation
(Table)

SQL
(DML | DDL)

Physical Level × ORM



Physical Level × ORM

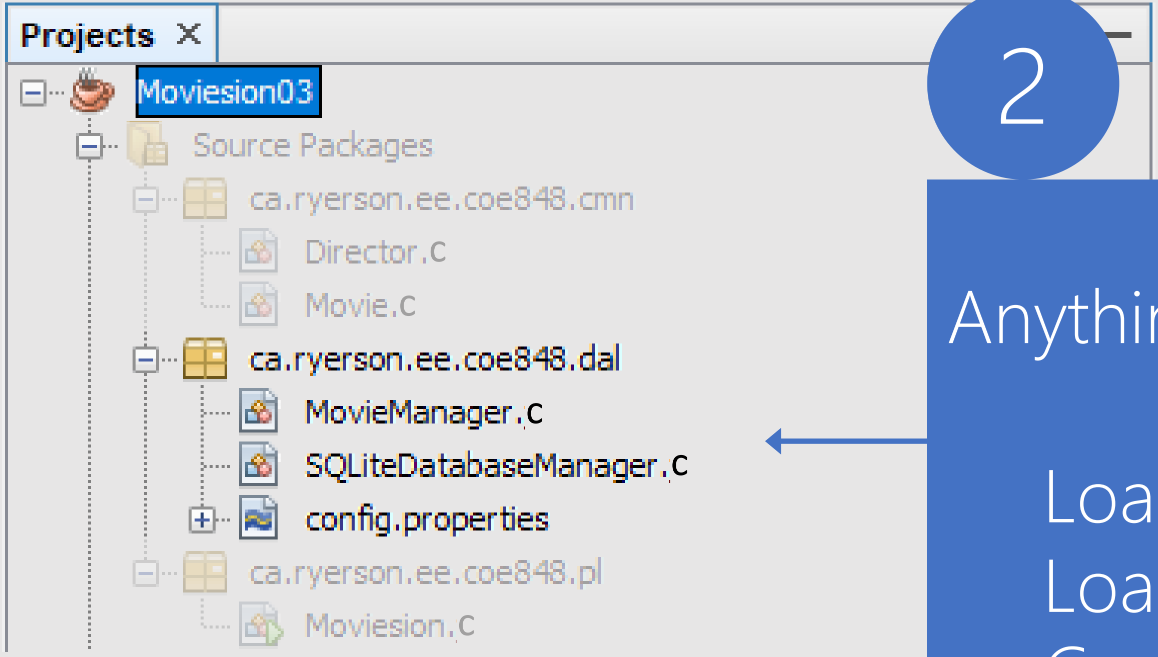


1

Settings to connect to DBMS:

- DBMS Location
- DBMS Username
- DBMS Password
- DB Name

Physical Level × ORM

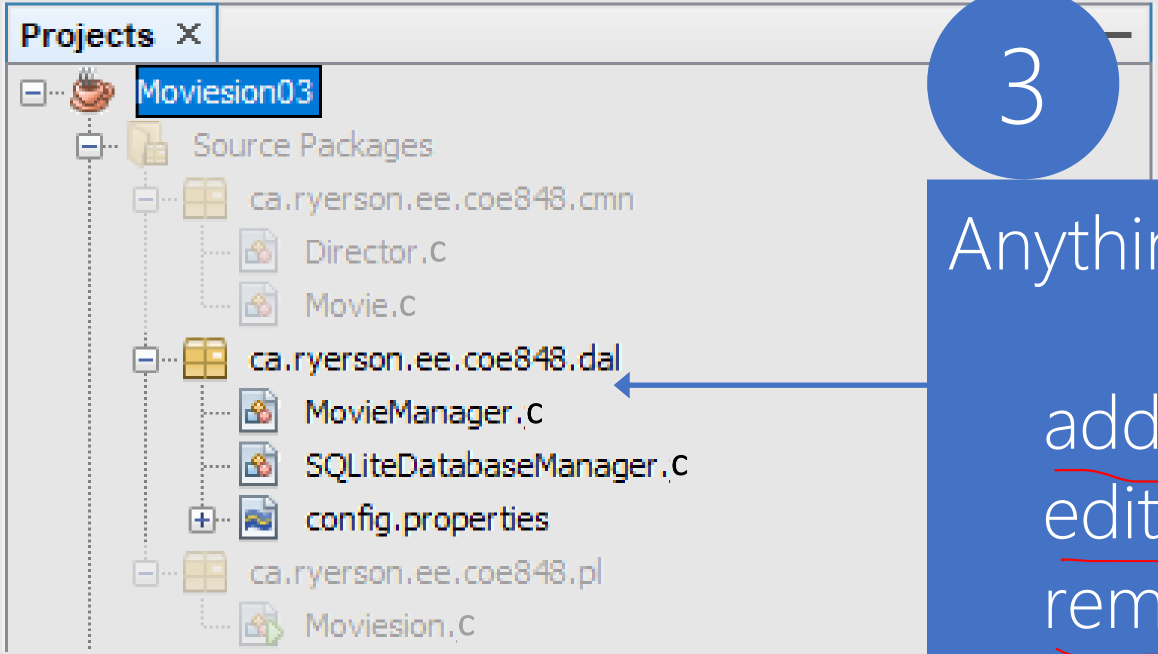


2

Anything related to DBMS connection:

- Loading SQLite library
- Loading settings from config file
- Create a connection

Physical Level × ORM



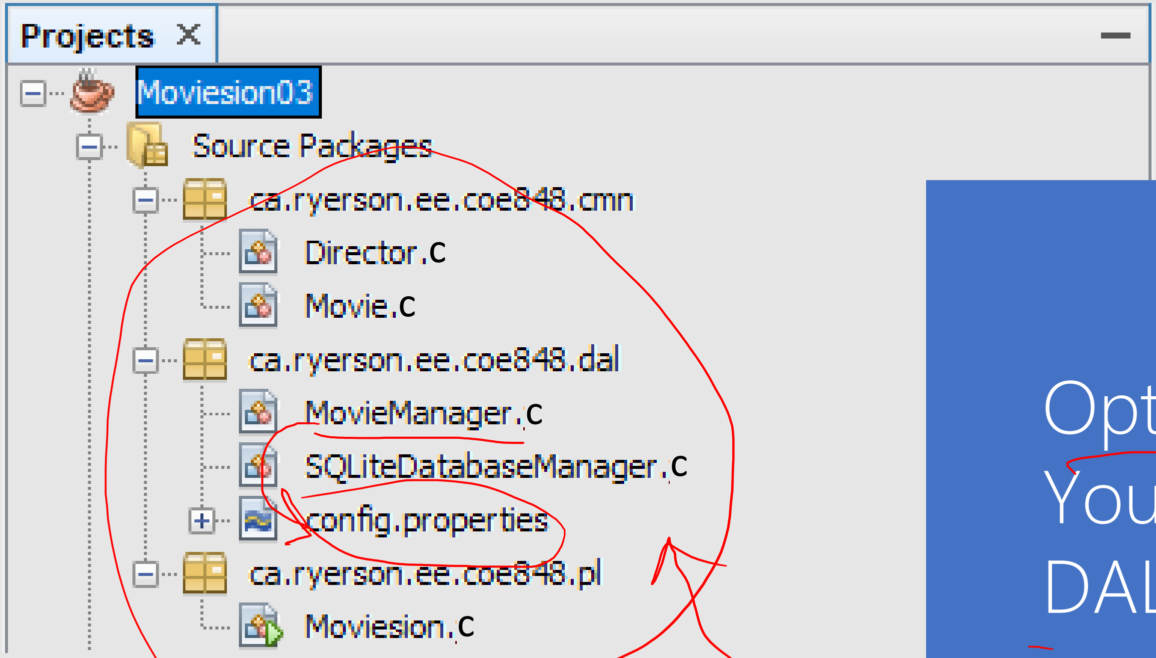
3

Anything related to Movie (ORM):

addMovie()	INSERT
editMovie()	UPDATE
removeMovie()	DELETE
getMovie()	SELECT

by Id (int)

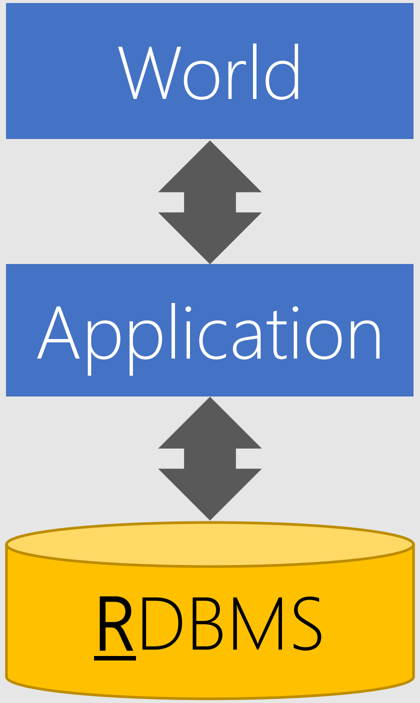
Physical Level × ORM



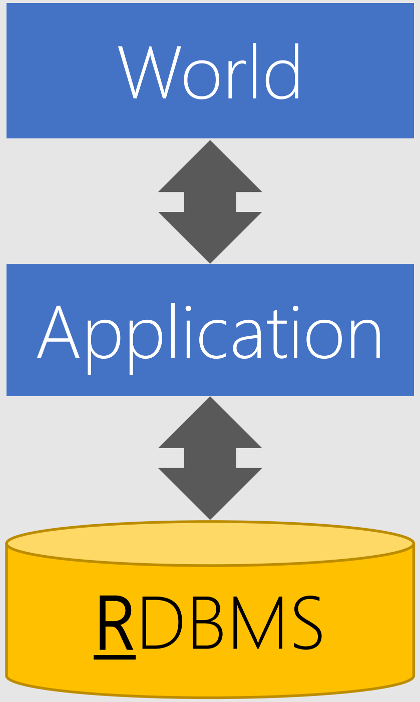
Optional:
You can follow 3-layered architecture (PL, DAL, DB) for your Lab5 project.

DB vs. APP Level Processing

How many movies do we have?



DB Level Processing



How many movies do we have?

In SQL: what is the movie count?

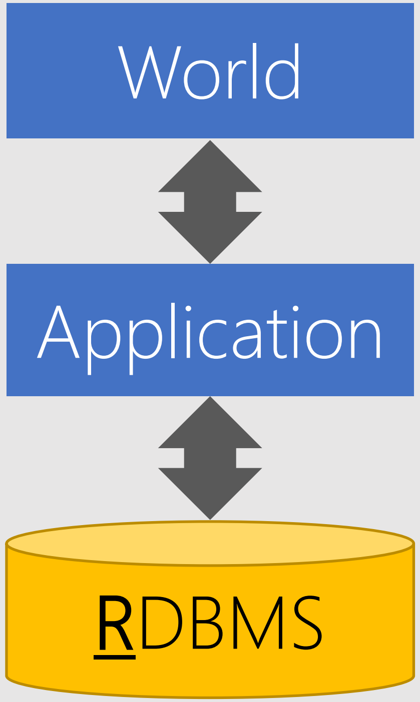
I) Count the movies
II) Return a single number

lo: # movie?

CountMovie(){

return

APP Level Processing



How many movies do we have?

I) Get all movies
II) Count the movies

Return all movies

DB vs. APP Level Processing

APP Level

+:

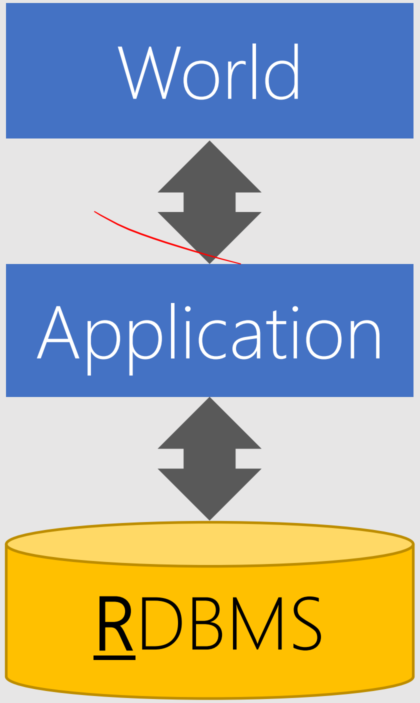
Only simple SQL

Able to do very complex tasks

–:

Slow, moving all data to app. level

Waste of network bandwidth



DB Level

+:

Fast, no need to move data

Fast, DBMS is a powerful machine

–:

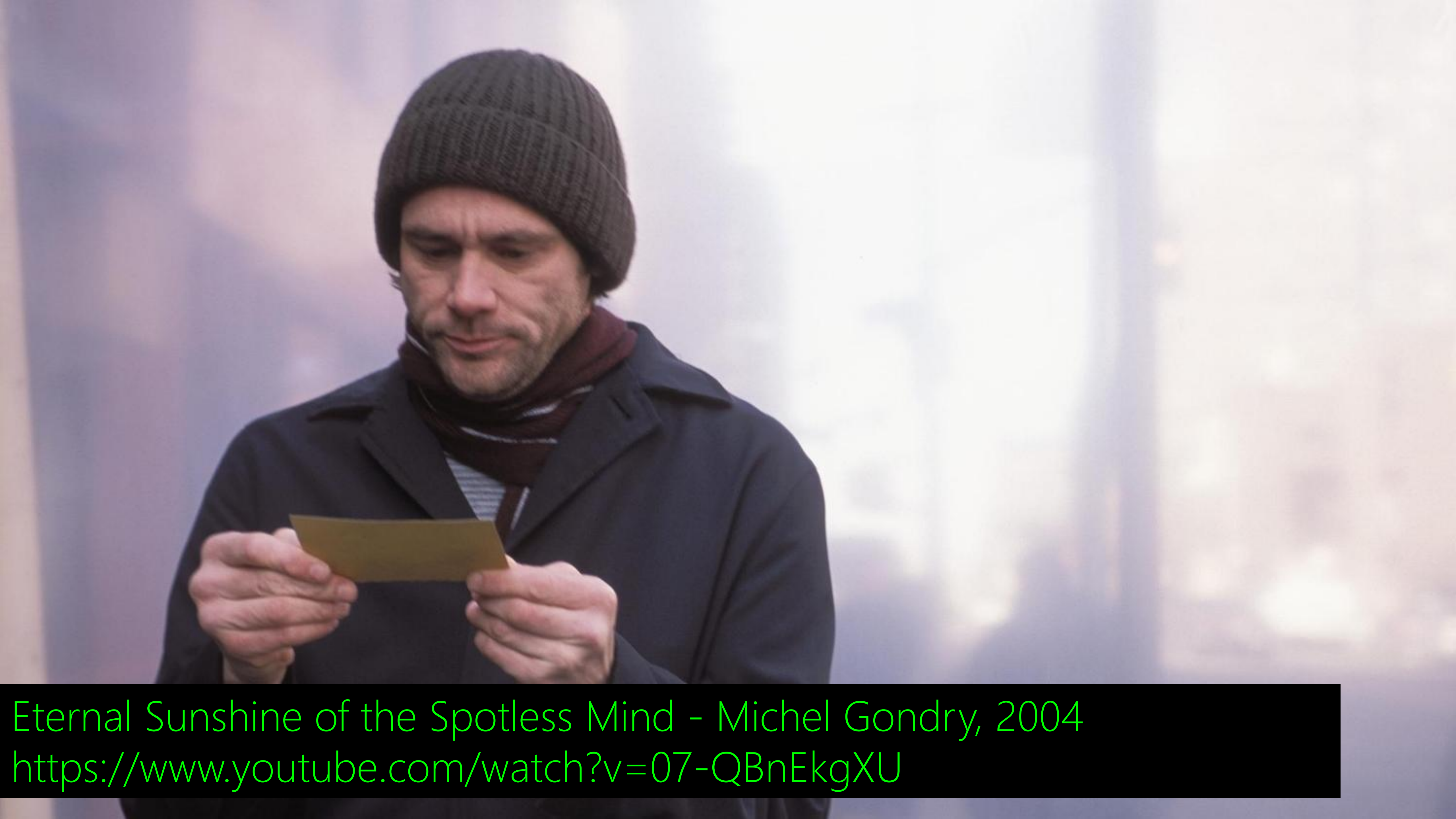
Master SQL language

Not able to do very complex tasks

Ad hoc SQL Query

Ad hoc query is created to obtain info as need arises,
e.g., which director has made the most movies?

Contrast with a query that is predefined & routinely processed,
e.g., INSERT, UPDATE, DELETE, SELECT by Id



Eternal Sunshine of the Spotless Mind - Michel Gondry, 2004
https://www.youtube.com/watch?v=07-QBnEkgXU

1

Indexing is finding the whole information quicker using only part of it.

Part of information is called Search Key.
It points to the whole information.

Index

~~Primary Key~~
~~Surrogate Key~~
~~Candidate Key~~
~~Foreign Key~~

Index

How to find a webpage in WWW?

Search Key = 'UWindsor'
Whole Information = https://www.uwindsor.ca

> 10^{10} seconds by no index, traverse all webpages
< 0.31 seconds by Google
~ 0 seconds by ?

SQL × INDEX


```
SELECT * FROM Director WHERE Id=1
SELECT * FROM Director WHERE LastName='Kubrick'
SELECT * FROM Director WHERE LastName='Kubrick' AND FirstName = 'Stanley'
```

SQL × INDEX

$\frac{c_1}{c_2}$

```
CREATE [UNIQUE] INDEX IndexName ON TableName (c1, c2, ...);
```

Could be any name, but by convention we follow this:
IX_ColumnName1_ColumnName2_...
UIX_ColumnName1_ColumnName2_...

UNIQUE INDEX does not allow duplicate in indexed columns.
A way to create a candidate key set of columns in a table.

SQL × INDEX

O([illegible])

```
SELECT * FROM Director WHERE Id = 1
CREATE UNIQUE INDEX UIX_Id ON Director(Id)
```

O(1)

By default, most DBMSs CREATE UNIQUE INDEX on primary key set of a table.

SQL × INDEX

What other columns of a table should to be indexed?

- Those columns of table that appears a lot in WHERE clause.
- The search key of the table to find a single or range of rows.

It's a tuning task: → DBA

- After the DB goes under heavy load DB designer need to increase retrieval speed.
- Recently is done automatically by DBMS

SQL × INDEX


```
SELECT * FROM Director WHERE LastName = 'Kubrick'
CREATE INDEX IX_LastName ON Director(LastName)
```

SQL × INDEX


```
ALTER TABLE TableName ADD [UNIQUE] INDEX IndexName ON  (c1, c2, ...);
ALTER TABLE TableName DROP INDEX IndexName;
```

SQL × INDEX


```
SELECT * FROM Director WHERE LastName = 'Kubrick' AND FirstName = 'Stanley'
```

Which one?

A) `CREATE INDEX IX_LastName_FirstName ON Director(LastName, FirstName)`
B) `CREATE INDEX IX_FirstName_LastName ON Director(FirstName, LastName)`
C) `CREATE INDEX IX_FirstName ON Director(FirstName)`
D) `CREATE INDEX IX_LastName ON Director(LastName)`
E) All
F) A & B are the same

DBMS × INDEX

As far as DB designer is concerned, knowing how to CREATE | ADD | DROP INDEX in SQL is more than enough.

However, knowing the implementation details inside DBMS helps DB designer with right decisions about indexing.

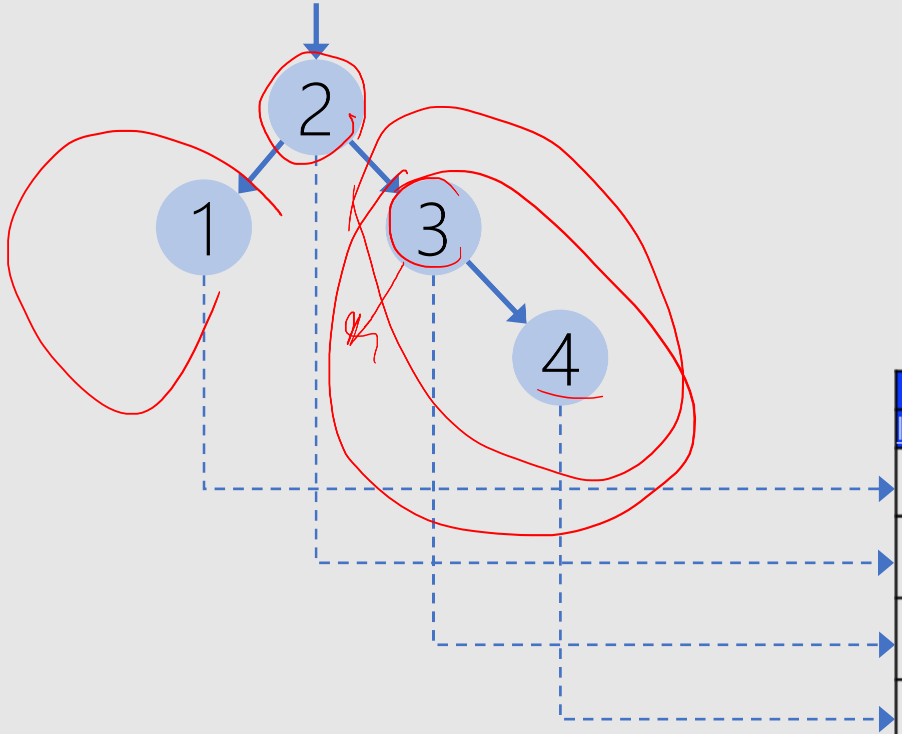
DBMS × INDEX × Binary Search Tree

Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
2	*Rosemary's Baby*	*English*	*1968*	NULL
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

SELECT * FROM Movie WHERE Id = 1

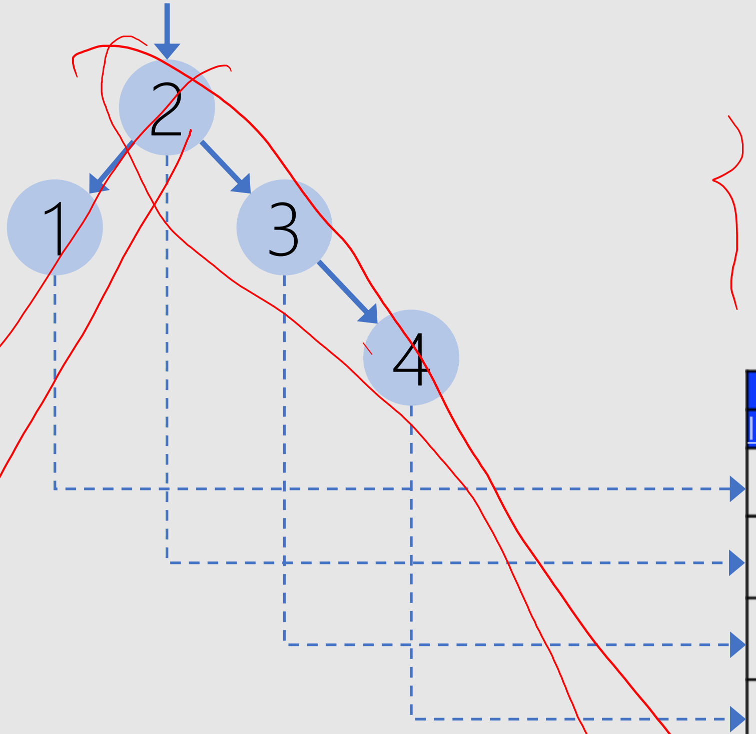
A. Sequential search, check all movies' Id with the given Id, i.e., 1
B. Binary search, after sorting elements in the list by Id → $\log_2 n$
C. Having an index structure:
   A. Creating a Binary Search Tree (BST)

DBMS × INDEX × Binary Search Tree



Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
2	*Rosemary's Baby*	*English*	*1968*	NULL
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

DBMS × INDEX × Binary Search Tree



Average

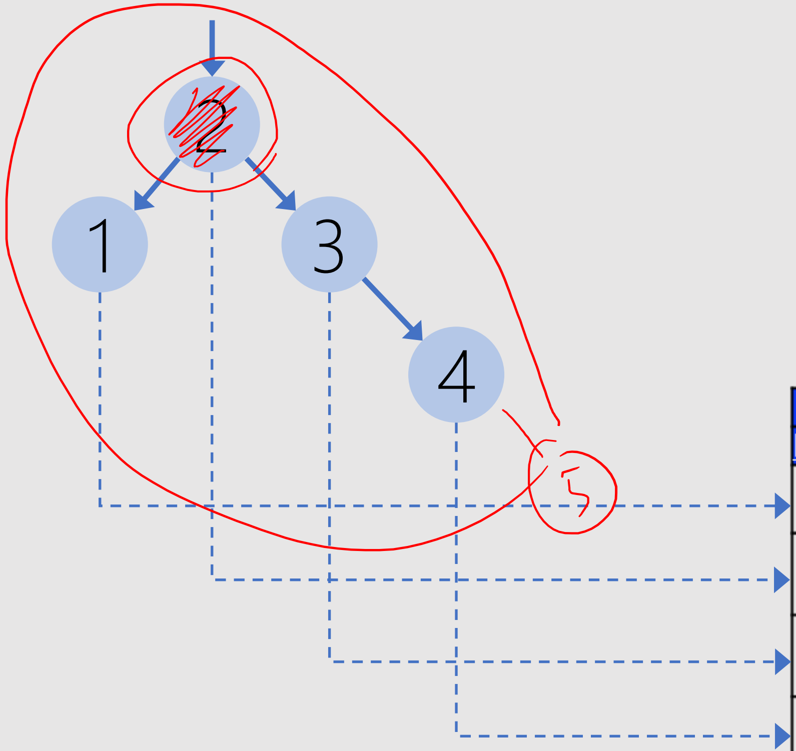
Search O(log n)
Insert O(log n)
Delete O(log n)

Worst (When?)

Search O(n)
Insert O(n)
Delete O(n)

Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
2	*Rosemary's Baby*	*English*	*1968*	NULL
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

DBMS × INDEX × Binary Search Tree



Average	Worst (When?)
Search O(log n)	Search O(n)
Insert O(log n)	Insert O(n)
Delete O(log n)	Delete O(n)

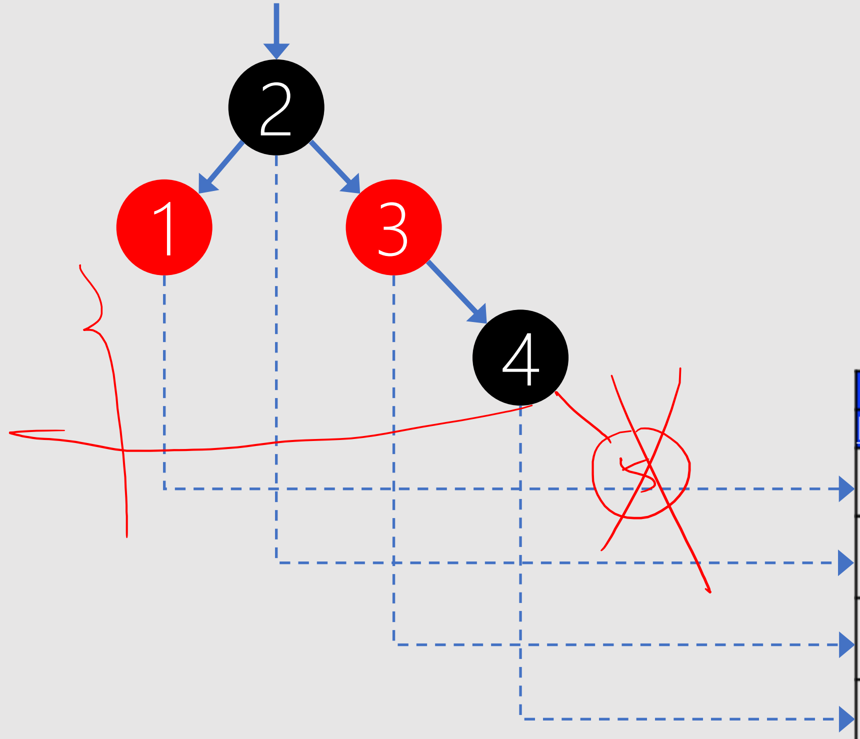
Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
~~*2*~~	~~*Rosemary's Baby*~~	~~*English*~~	~~*1968*~~	~~NULL~~
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

5

Overhead:

Each DML on the table needs additional DML on indexes of the table by DBMS

DBMS × INDEX × Balanced Binary Tree



Average		Worst	
Search	O(log n)	Search	O(log n)
Insert	O(log n)	Insert	O(log n)
Delete	O(log n)	Delete	O(log n)

Movie				
Id	Title	Language	ReleaseDate	RunningTime
1	*2001: A Space Odyssey*	*English*	*1968*	*142*
2	*Rosemary's Baby*	*English*	*1968*	NULL
3	*The Birds*	*English*	*1963*	*119*
4	*Planet of the Apes*	*EN*	*1968*	*112*

Overhead:

Each DML on the table needs additional DML on indexes of the table by DBMS

DBMS × INDEX × B-tree

Balanced Multi-way Tree

Bayer, R.; McCreight, E. (1972)
Organization and Maintenance of Large Ordered Indexes
Acta Informatica, 1 (3): 173–189

@Boeing

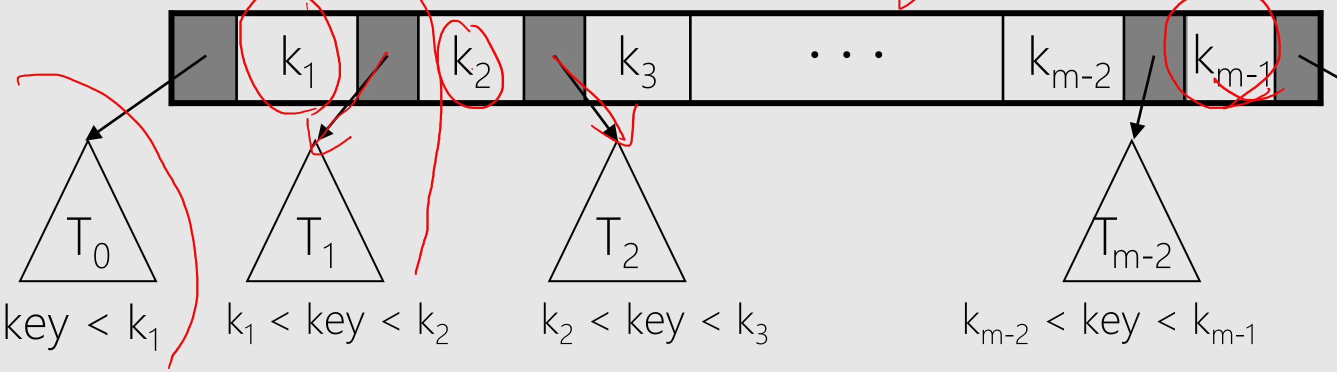
DBMS × INDEX × B-tree

B-tree of order m (branching factor) is a tree in which:

- $0 \le$ #keys in root $\le (m-1)$ → m
- $0 \le$ #subtrees in root $\le m$
- $(\frac{1}{2}m) \le$ #keys in other nodes $\le (m-1)$
- $1+(\frac{1}{2}m) \le$ #subtrees in other nodes $\le m$

- The keys in each node are sorted.
- It is balanced!

$m \le 200$



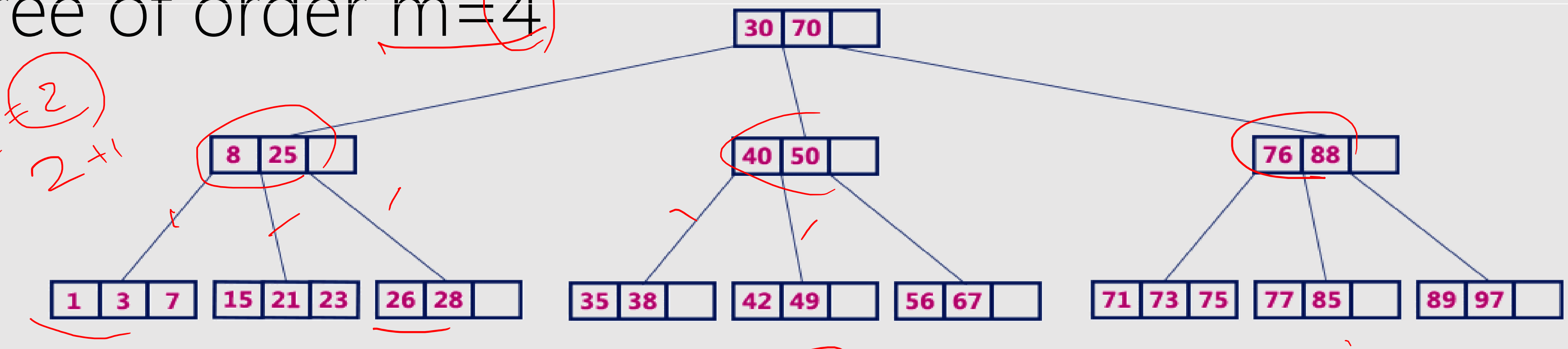
m subtr

DBMS × INDEX × B-tree

B-tree of order m=4

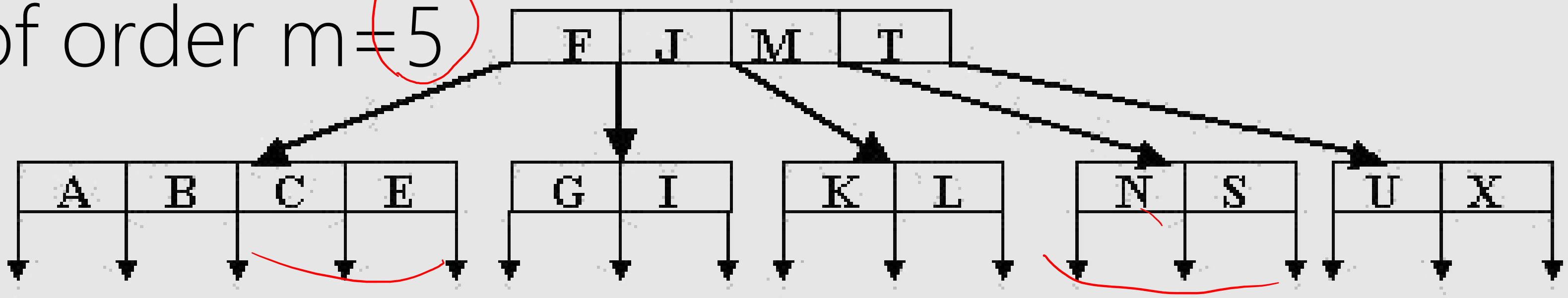
0 → 4
0 → 4-1

$m/2 = 4/2 = 2$
$2+1$



$5/2 = 2 + 1$

B-tree of order m=5



DBMS × INDEX × B-tree

The height h of a B-tree of order m, with a total of n keys:

$$log_m^{(n+1)} \leq h \leq 1 + log_{\lceil m/2 \rceil}^{(n+1/2)}$$

150

If m=300 and n = 16,000,000 then h ≈ 4.

i.e., the worst case finding a key in such B-tree requires ? accesses.