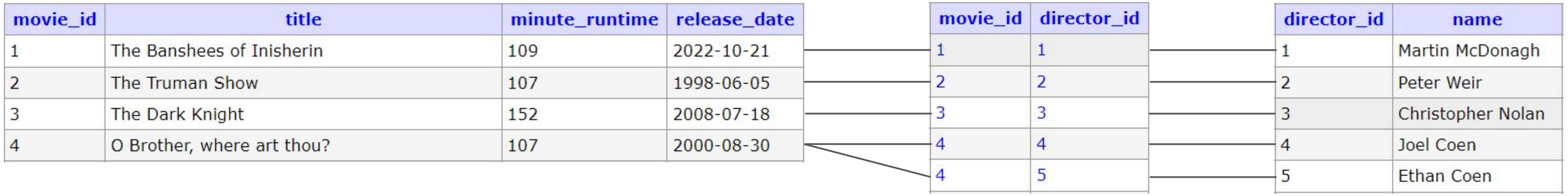
Week 9

Views & Triggers

Views / Example

- To output movies with their directors, we need to join 3 tables ***each time***



movie_id	title	minute_runtime	release_date
1	The Banshees of Inisherin	109	2022-10-21
2	The Truman Show	107	1998-06-05
3	The Dark Knight	152	2008-07-18
4	O Brother, where art thou?	107	2000-08-30

movie_id	director_id
1	1
2	2
3	3
4	4
4	5

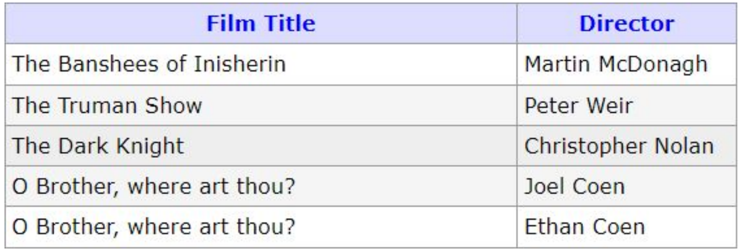
director_id	name
1	Martin McDonagh
2	Peter Weir
3	Christopher Nolan
4	Joel Coen
5	Ethan Coen

```
SELECT m.title AS “Film Title”, d.name AS “Director” FROM movie m INNER JOIN movie_director md ON m.movie_id = md.movie_id
LEFT JOIN director d ON md.director_id = d.director_id;
```

- Or we **CREATE** and name a view that will save the 3 table join query
- Once created we can **SELECT** the view that has the join query saved

```
CREATE VIEW movie_with_director AS
    SELECT m.title AS “Film Title”, d.name AS “Director”
    FROM movie m
    INNER JOIN movie_director md ON m.movie_id = md.movie_id
    LEFT JOIN director d ON md.director_id = d.director_id;

SELECT * FROM movie_with_director;
```



Film Title	Director
The Banshees of Inisherin	Martin McDonagh
The Truman Show	Peter Weir
The Dark Knight	Christopher Nolan
O Brother, where art thou?	Joel Coen
O Brother, where art thou?	Ethan Coen

Views

- A **view** is a virtual table from the **result-set** of a SQL statement
- Views are known as **“virtual tables”** because a view **does not form part of the schema**
- **Views are not stored data – just a stored query**
    - A view is a shortcut to run a query and get back a results table
- The virtual table is **computed each time** access to that view is requested
- Views are particularly useful to show frequently used and complex outputs that may request fields from multiple tables
    - For example, the output table of a complex join statement
- Therefore, **views are used to answer questions that are asked often** by your data users

CREATE VIEW[1]

Example: Create a view of all movies coming soon

```
CREATE VIEW movie_coming_soon AS
    SELECT m.title, m.release_date, d.name
    FROM movie m
    INNER JOIN movie_director md ON m.movie_id = md.movie_id
    LEFT JOIN director d ON md.director_id = d.director_id
    WHERE DATE(m.release_date) > NOW();
```

Syntax

```
    CREATE VIEW view_name AS
        SELECT column1, column2, ...
        FROM table_name WHERE condition;
```

[1] W3Schools: CREATE VIEW

SELECT View

Example: Select view of all movies coming soon, or movies coming soon in 2024

```
SELECT * FROM movie_coming_soon;

SELECT * FROM movie_coming_soon
    WHERE release_date LIKE '%2024%';
```

- Views should be **subsets of data** from other tables
- Any of the select strategies can be used when querying your view:
    - Aggregate functions: `SUM()`, `MIN()`, `MAX()`, `COUNT()`
    - Keywords: `DISTINCT`, `WHERE`, `ORDER BY`, `GROUP BY`, `LIMIT` etc.
    - Subqueries
- These select strategies can be **useful to reduce the view result set**
- Just like select statements, specific queries of views improve query speed

Views / Advantages

- Simplifies complex queries
- Reusability: Save frequently used queries as views for easy reference
- Protect data: An extra layer of security to limit or restrict data access

Views / Limitations

- `UPDATE`, `INSERT` and `DELETE` can be used with some views but not recommended
- It is sometimes recommended to *not* use aggregate functions, `GROUP BY`, `HAVING`, and `DISTINCT` with views for these reasons:
  - Performance issues: High resource consumption & Inefficiencies without indexes
  - Maintenance complexity: Difficult to debug and update
  - Limiting the flexibility of queries to the view

ALTER VIEW

- A view can use **ALTER** keyword to change the views query

Syntax

```
ALTER VIEW view_name AS
    SELECT column1, column2, ...
    FROM table_name WHERE condition;
```

DROP VIEW

- A view can be deleted with the **DROP** keyword

Syntax

```
DROP VIEW view_name;
```

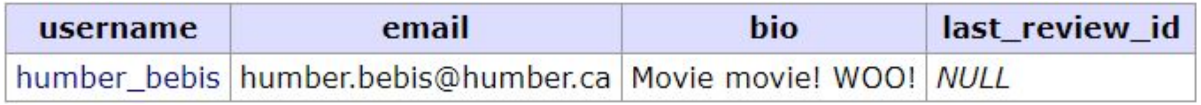
Triggers

Syntax

```
CREATE TRIGGER trigger_name
    [ BEFORE | AFTER ] [ INSERT | UPDATE | DELETE ]
    ON table_name
    FOR EACH ROW
    -- Trigger Body, AKA SQL statements to run when event triggered;
```

Triggers

Example: When a user leaves a review for a movie, update their last_review_id column to hold the id of the most recent review by the user



username	email	bio	last_review_id
humber_bebis	humber.bebis@humber.ca	Movie movie! WOO!	*NULL*

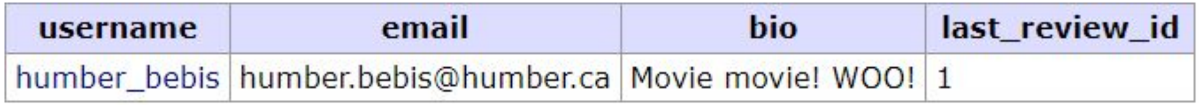
```
INSERT INTO review (username, rating, movie_id, content)
    VALUES ('humber_bebis',4.5,1,"This movie rocks!");
```

Insert Trigger Event

AFTER INSERT

```
UPDATE user
    SET last_review_id = NEW.review_id
    WHERE username=NEW.username;
```

run trigger script



username	email	bio	last_review_id
humber_bebis	humber.bebis@humber.ca	Movie movie! WOO!	1

Triggers

Example: When a user leaves a review for a movie, update their last_review_id column to hold the id of the most recent review by the user

```
CREATE TRIGGER update_after_latest_review
    AFTER INSERT ON review
    FOR EACH ROW
        UPDATE user
        SET last_review_id = NEW.review_id
        WHERE username=NEW.username;
```

- Trigger Event Type
- Trigger body run after each row is inserted

Triggers

- Triggers are **SQL scripts that run when a DML** (Data Manipulation Language) **event occurs**
- These **DML events** include changes in table structure or when data is manipulated using **`INSERT`**, **`UPDATE`** or **`DELETE`** statements
- Triggers execute in response to a specific event on a specified table
    - For example, trigger a SQL script to run after a new value is inserted into a table
- Using SQL, DB developers can create both the trigger script and determine when the trigger script should run
- *Similar to JavaScript html element events*
    - *An event occurs* -> *code runs in response to the event*

Trigger Events

Trigger Event	OLD	NEW
INSERT	No	Yes
UPDATE	Yes	Yes
DELETE	Yes	No

- For an `INSERT` trigger
  - **OLD** contains no values
  - **NEW** contains the new values being inserted
- For an `UPDATE` trigger
  - **OLD** contains the old values that are being replaced
  - **NEW** contains the new values that are replacing the old values
- For a `DELETE` trigger
  - **OLD** contains the old values that are being deleted
  - **NEW** contains no values

Triggers / BEFORE & AFTER

- Before triggers can be used to validate or change before they are inserted in the database tables
  - Like checking more complicated inputs
  - Or changing values to meet a specific format required in the table

- After triggers are used when data modifications need to be completed and available in tables first
  - Like if you want to reference the new data in a different table using a foreign key

Triggers / Multiple Statements - DELIMITER

Example: When a user leaves a review for a movie, update their last_review_id column to hold the id of the most recent review by the user AND update the review with a timestamp

```
DELIMITER //
CREATE TRIGGER update_after_latest_review
    AFTER INSERT ON review
    FOR EACH ROW
    BEGIN
        UPDATE user
        SET last_review_id = NEW.review_id
        WHERE username=NEW.username;
        UPDATE review
        SET last_update = NOW()
        WHERE review_id=NEW.review_id;
    END//
DELIMITER ;
```

Trigger Event Type

Trigger body can run Multiple statements after each row is inserted

To use multiple statement we must change delimiter from ; to // allowing us to use the semicolon delimiter in our trigger body

Triggers

Example: When a user leaves a review for a movie, update their last_review_id column to hold the id of the most recent review by the user

```
CREATE TRIGGER update_after_latest_review
  AFTER INSERT ON review
  FOR EACH ROW
    UPDATE user
    SET last_review_id = NEW.review_id
    WHERE username=NEW.username;
```

username	email	bio	last_review_id
humber_bebis	humber.bebis@humber.ca	Movie movie! WOO!	*NULL*

```
INSERT INTO review (username, rating, movie_id, content)
      VALUES ('humber_bebis',4.5,1,"This movie rocks!");
```

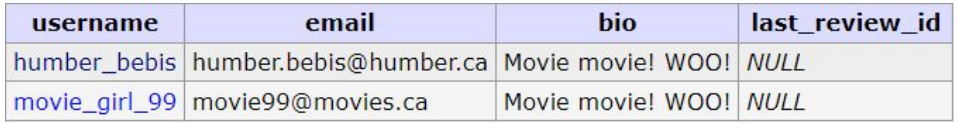
```
NEW.review_id = 1
NEW.username = 'humber_bebis'
```

```
UPDATE user
      SET last_review_id = NEW.review_id
      WHERE username=NEW.username;
```

username	email	bio	last_review_id
humber_bebis	humber.bebis@humber.ca	Movie movie! WOO!	1

Triggers / FOR EACH ROW

Example: When a user leaves a review for a movie, update their last_review_id column to hold the id of the most recent review by the user



username	email	bio	last_review_id
humber_bebis	humber.bebis@humber.ca	Movie movie! WOO!	*NULL*
movie_girl_99	movie99@movies.ca	Movie movie! WOO!	*NULL*

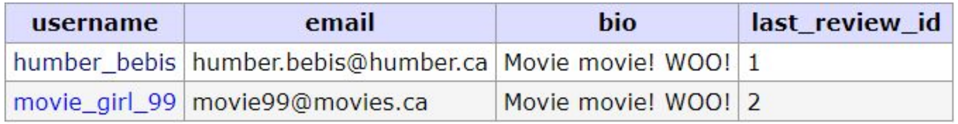
```
INSERT INTO review (username, rating, movie_id, content)
    VALUES ('humber_bebis',4.5,1,"This movie rocks!"),
    (‘movie_girl_99’,0.5,3,"This movie sucks!");
```

Insert Trigger Event

AFTER INSERT

```
UPDATE user
    SET last_review_id = NEW.review_id
    WHERE username=NEW.username;
```

run trigger script



username	email	bio	last_review_id
humber_bebis	humber.bebis@humber.ca	Movie movie! WOO!	1
movie_girl_99	movie99@movies.ca	Movie movie! WOO!	2

Triggers / Advantages

- Database integrity: Triggers provide an extra layer of checks for data integrity
  - Example: Invalid data will be prevented from being inserted or updated into the database
- Auditing data changes
  - Triggers are useful for logging any changes in table data
- Scheduled tasks
  - Triggers can automatically run scheduled tasks

Triggers / Disadvantages

- Triggers do not provide any feedback to the database user, so you will not be notified if there is an issue, Triggers do things quietly
- Difficult to troubleshoot because triggers run automatically

DROP TRIGGER

- A trigger can be deleted with the **DROP** keyword

Syntax

```
DROP TRIGGER trigger_name;
```

- To alter a trigger it is recommended to drop the trigger and recreate it, or use this syntax:

Syntax

```
CREATE OR REPLACE TRIGGER trigger_name
    [ BEFORE | AFTER ] [ INSERT | UPDATE | DELETE ]
    ON table_name
    FOR EACH ROW
    -- Trigger Body, AKA SQL statements to run when event triggered;
```