Recap

- Data-savviness is the future!
- Notion of a DBMS
- The relational data model and algebra: bags and sets
- SQL Queries
- SQL Modifications
- SQL DDL
- Database Design
- Next: views

Views

- In addition to database design, you can also specify views

- A view is a "virtual table/relation", one that is defined in terms of contents of other "base" tables/relations, or other views

- Declare by:

  CREATE VIEW <name> AS <query>;

  e.g., CREATE VIEW FilmActorJointRelation AS

  ```
  SELECT * FROM Film, Actor, FilmActor
  WHERE Actor.actor_id = FilmActor.actor_id
  AND Film.film_id = FilmActor.film_id
  ```

- Views are not stored in the database but can be queried as if they existed
  - Will talk about an exception later

Why Should I Declare Views?

Multiple Reasons:

- Convenience: If I repeatedly use a subquery, I might want to give it a name
- Access control: I might want to provide you a view with only the attributes you should be allowed to query
- Efficiency: in some cases, you can materialize (pre-populate, or cache) the view result, so that you can return it efficiently

Once Declared…

- Can treat view like any other relation

```
CREATE VIEW FilmActorJointRelation AS
        SELECT * FROM Film, Actor, FilmActor
        WHERE Actor.actor_id = FilmActor.actor_id
        AND Film.film_id = FilmActor.film_id

SELECT * FROM FilmActorJointRelation
SELECT AVG (Film.Rating), Actor.name FROM FilmActorJointRelation
GROUP BY Actor.name
```

Updating a View

- One can update a view, but only if the modification "makes sense" as a modification of the base tables
- Basically, cases where multiple tuples in the base tables are mapped to a single tuple in the view (e.g., via a projection, aggregation) are problematic
  - Why?
  - Hard to map modification to view tuple to underlying base table tuples
- Not something we will worry about in this class
  - Look into the "view update" problem

Special Case of Materialized Views

- The results of a view designated as a MATERIALIZED VIEW are actually stored and maintained (as opposed to simply storing the query)
  - CREATE MATERIALIZED VIEW <name> AS <query>

- Keeping a view up-to-date as the base tables change is an efficiency bottleneck
  - So only use if it is worth the payoff, e.g., when the materialized view is queried much more often than the base tables change.

Recap

- Data-savviness is the future!
- Notion of a DBMS
- The relational data model and algebra: bags and sets
- SQL Queries
- SQL Modifications
- SQL DDL
- Database Design
- Views
- Next: constraints and triggers

Constraints and Triggers

- These are used to make sure that the data in the database "makes sense": important real-world properties are kept valid
  - Via continuous maintenance of "assertions": constraints
  - Via book-keeping done by special commands: triggers
- Constraints are relationships among data elements that the DBMS is required to enforce
  - e.g., a key constraint
- Triggers are "triggered" into execution when some action happens, e.g., insertion of a tuple
  - Easier to implement (within the DBMS) than many constraints

Kinds of Constraints

- Key (via PRIMARY KEY or UNIQUE)
- Referential integrity constraints (remember semi-circles in ER?) as FOREIGN KEY constraints

- Attribute based constraints
- Tuple-based constraints but we won't cover them
- Other constraints (any SQL expression) but we won't cover them

Example

- Film_actor (actor_id, film_id, last_update),
- Actor (actor_id, first_name, last_name, last_update),
- Film (film_id, …)

- We might expect that actor_id in Film_actor must be present in Actor, and likewise, film_id in Film_actor must be present in Film

- Such a constraint is called a **foreign key** constraint
- Referenced attributes must be declared UNIQUE or PRIMARY KEY (i.e., a key), why?

Declaring a FK Constraint

Using REFERENCES

```
CREATE TABLE Actor (actor_id INTEGER, name VARCHAR (20),
PRIMARY KEY actor_id);

CREATE TABLE FilmActor (actor_id INTEGER, film_id INTEGER,
FOREIGN KEY (actor_id) REFERENCES Actor (actor_id),
FOREIGN KEY (film_id) REFERENCES Film (film_id));
```


Enforcing Foreign Key Constraints

- If there is a FK constraint from attributes of R to the keys of S, two violations are possible:
  - An insert/update to R introduces values not found in S
  - A deletion from S causes a tuple in R to "dangle"
- Q: Why are other cases not important?

Enforcing Foreign Key Constraints

- Suppose R = FilmActor, S = Actor
- Referencing relation changes:
  - An insert or update to R that introduces a non-existent actor will be rejected
- Referenced relation changes:
  - A deletion or update to S that causes some tuples in R to "dangle" can be handled in three ways:
    - Default: reject the modification
    - CASCADE: make the same change to R (FilmActor) as in S
      - Deleted Actor: delete the FilmActor tuples that refer to it
      - Updated Actor: update the FilmActor tuples that refer to it
    - SET NULL: change the actor_id in FilmActor be NULL

Example

```
CREATE TABLE FilmActor (actor_id INTEGER, film_id INTEGER,
        FOREIGN KEY (actor_id) REFERENCES Actor (actor_id)
        ON DELETE SET NULL ON UPDATE CASCADE,
        FOREIGN KEY (film_id) REFERENCES Film (film_id)
        ON DELETE SET NULL);
```

Q: what if we

- Delete a tuple from Actor corresponding to actor_id = 10?
- Change a tuple in Actor from actor_id = 10 to actor_id = 20?
- Delete a tuple from Film corresponding to film_id = 123
- Change a tuple in Film from film_id = 121 to film_id = 212?

Attribute Based Constraints

- NOT NULL is one such constraint
- Example: ensure that actor_ids are four digit integers.

```
CREATE TABLE FilmActor (
    actor_id INTEGER
        CHECK (actor_id >= 1000 AND actor_id <= 9999),
    film_id INTEGER,);
```

- Checked when a new actor_id is added or an existing one is updated to FilmActor, not when deleted (why?)
- For this reason, we can't use CHECKs to enforce referential integrity
  - Even if we use CHECK (actor_id IN (SELECT * FROM Actor)) - see homework!
  - We will miss changes to the referenced relation (in this case Actor)

Kinds of Constraints

- Key (via PRIMARY KEY or UNIQUE)
- Referential integrity constraints (remember semi-circles in ER?) as FOREIGN KEY constraints
- Attribute based constraints

- Tuple-based constraints
  - **Constraints across multiple attributes in a tuple**
  - but we won't cover them (still via **CHECK**)
- Other constraints (any SQL expression)
  - **Constraints across multiple relations (via CREATE ASSERTION)**
  - but we won't cover them

Triggers

- Tuple and General Assertion-based constraints often become very expensive to check
- And, in some cases, we don't want to simply reject the modification, we may want to "fix" it in some way

- Enter triggers
  - Triggers allow users to specify when the check happens
    - Can support a general assertion-like condition
  - Can also support arbitrary database modifications

Syntax Differs Across Databases…

- But here is the generic SQL standard format
- Event-Condition-Action

- What do you think this trigger does?

```
CREATE TRIGGER updatedAgeActors
BEFORE UPDATE OF age ON Actors
FOR EACH ROW
WHEN (OLD.age + 1.0 < NEW.age)
INSERT INTO possibleFakeAges VALUES (OLD.name, OLD.age, NEW.age)
```


Syntax Differs Across Databases…

- Event-Condition-Action

```
CREATE TRIGGER updatedAgeActors
BEFORE UPDATE OF age ON Actors
FOR EACH ROW
WHEN (OLD.age + 1.0 < NEW.age)
INSERT INTO possibleFakeAges
VALUES (OLD.name, OLD.age, NEW.age)
```

- Event
  - BEFORE can be AFTER or INSTEAD OF
  - UPDATE can be INSERT/DELETE
- Condition
  - In some cases, we need to declare explicit OLD/NEW references via REFERENCING
- Action
  - Can skip "FOR EACH ROW" if one global action needed
  - If we need more actions, can place within BEGIN/END blocks
  - For PostgreSQL, this is a "stored procedure" (generic function call)

View Updates via Triggers

- Can capture updates to views via INSTEAD OF

```
CREATE TRIGGER catchViewUpdate
INSTEAD OF UPDATE ON FilmActorJointRelation
FOR EACH ROW
BEGIN
  UPDATE ….
  UPDATE ….
END
```


On Constraints, Triggers, and Materialized Views

- While constraints and triggers allow us to add more "semantics" to the database system, …
  - And therefore, they help us avoid mistakes
  - However, they also sometimes make the database more sluggish, due to unnecessary checks during updates, or triggered changes
- Materialized views are great for avoiding repeated computation
  - And therefore allow for quick access for popular query patterns
  - However, they need to be maintained during updates to base tables, or will end up becoming stale (and therefore useless)
- So we should be careful about using them!
- Views, on the other hand, are convenient without downsides: just