Administrivia

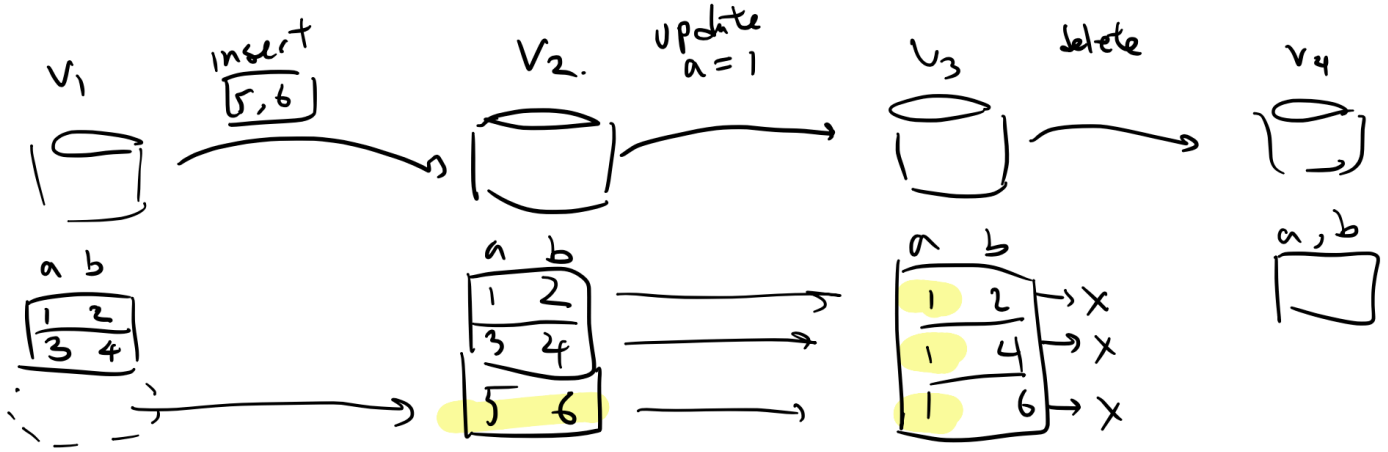
Midterm next Thursday

- In class
- 1 cheat sheet 8x11 page both sides
- practice exams online
- student-created study guide on scribenotes

Project 1 Part 1 grades out tonight/early weds

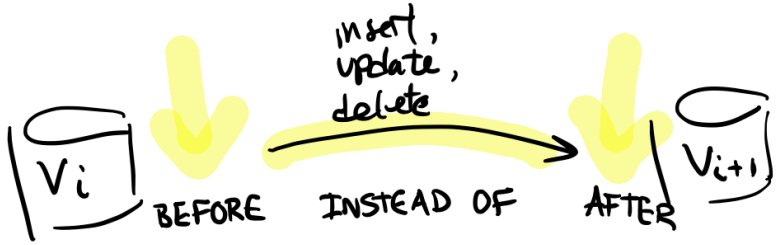
Triggers (background)

- Recall that a database instance is the database schema + the specific records
- Changing a DB instance essentially creates a new DB instance because the records are different. Let's call each instance a "version" of the DB
- Let's say we made 3 separate changes

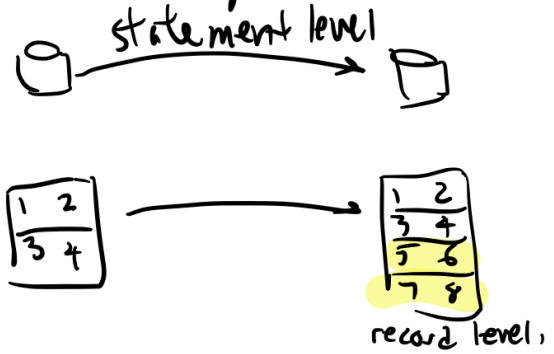


Triggers (background)

- When/where can we add trigger logic?



- At what granularity?



Does a SELECT query create a new version?

No

Triggers (logical)

def: procedure that runs automatically if specified changes in DBMS happen

```
CREATE TRIGGER name
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Triggers (logical)

def: procedure that runs automatically if specified changes in DBMS happen

```
CREATE TRIGGER name
    [BEFORE | AFTER | INSTEAD OF] event_list
    ON table
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Triggers (logical)

def: procedure that runs automatically if specified changes in DBMS happen

```
CREATE TRIGGER name
    [BEFORE | AFTER | INSTEAD OF] event_list
    ON table

    WHEN trigger_qualifications
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Triggers (logical)

def: procedure that runs automatically if specified changes in DBMS happen

```
CREATE TRIGGER name
    [BEFORE | AFTER | INSTEAD OF] event_list
    ON table
    [FOR EACH ROW]
    WHEN trigger_qualifications
    procedure
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Copy new young sailors into special table

(logical)

```
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewInserts
FOR EACH STATEMENT
    INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM NewInserts N
        WHERE N.age <= 18
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Copy new young sailors into special table

(logical)

```
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON SAILORS
FOR EACH ROW
    WHEN NEW.age <= 18
    INSERT
        INTO YoungSailors (sid, name, age, rating)
        VALUES (NEW.sid, NEW.name, NEW.age, NEW.rating)
```

Event activates the trigger

Condition tests if triggers should run

Action what to do

Triggers (logical)

Can be complicated to reason about

- Triggers may (e.g., insert) cause other triggers to run
- If > 1 trigger match an action, which is run first?
- ¯_(ツ)_/¯

```
CREATE TRIGGER recursiveTrigger
    AFTER INSERT ON SAILORS
FOR EACH ROW
    INSERT INTO Sailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM Sailors S
```

Triggers vs Constraints

Constraint

- Statement about state of database
- Upheld by the database for *any* modifications
- Doesn't modify the database state

Triggers

- Operational: X should happen when Y
- Specific to statements
- *Very* flexible

Triggers (postgres)

```
CREATE TRIGGER name
    [BEFORE | AFTER | INSTEAD OF] event_list
    ON table
    FOR EACH (ROW | STATEMENT)
    WHEN trigger_qualifications
    EXECUTE PROCEDURE user_defined_function();
```

PostgreSQL only runs *trigger* UDFs

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

Trigger Example

```
CREATE FUNCTION copyrecord() RETURNS trigger
AS $$
BEGIN
    INSERT INTO blah VALUES(NEW.a);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Signature: no args, return type is trigger

Returns NULL or same record structure as modified row

Special variables: OLD, NEW

```
CREATE TRIGGER t_copyinserts BEFORE INSERT ON a
    FOR EACH ROW
        EXECUTE PROCEDURE copyrecord();
```

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

Total boats and sailors < 100

```
CREATE FUNCTION checktotal() RETURNS trigger
AS $$
BEGIN
    IF ((SELECT COUNT(*) FROM sailors) +
        (SELECT COUNT(*) FROM boats) < 100) THEN
        RETURN NEW
    ELSE
        RETURN null;
    END IF;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER t_checktotal BEFORE INSERT ON sailors
    FOR EACH ROW
        EXECUTE PROCEDURE checktotal();
```

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

You can get into trouble…

```
CREATE FUNCTION addme_bad() RETURNS trigger
AS $$
BEGIN
    INSERT INTO a VALUES (NEW.*);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER t_addme_bad BEFORE INSERT ON a
    FOR EACH ROW
        EXECUTE PROCEDURE addme_bad();
```

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

You can get into trouble…

```
CREATE FUNCTION addme_stillwrong() RETURNS trigger
AS $$
BEGIN
    IF (SELECT COUNT(*) FROM a) < 100 THEN
        INSERT INTO a VALUES (NEW.a + 1);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER t_addme_stillwrong BEFORE INSERT ON a
    FOR EACH ROW
        EXECUTE PROCEDURE addme_stillwrong();
```

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

You can get into trouble…

```
CREATE FUNCTION addme_works() RETURNS trigger
AS $$
BEGIN
    IF (SELECT COUNT(*) FROM a) < 100 THEN
        INSERT INTO a VALUES (NEW.a + 1);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER t_addme_works AFTER INSERT ON a
    FOR EACH ROW
        EXECUTE PROCEDURE addme_works();
```

http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html
http://www.postgresql.org/docs/9.1/static/plpgsql-trigger.html

WITH

```
WITH RedBoats(bid, count) AS
    (SELECT   B.bid, count(*)
     FROM     Boats B, Reserves R
     WHERE    R.bid = B.bid AND B.color = ‘red’
    GROUP BY  B.bid)
SELECT    name, count
FROM      Boats AS B,  RedBoats AS RB
WHERE     B.bid = RB.bid AND count < 2
```

Names of unpopular boats

WITH

```
WITH RedBoats(bid, count) AS
   (SELECT   B.bid, count(*)
    FROM     Boats B, Reserves R
    WHERE    R.bid = B.bid AND B.color = ‘red’
   GROUP BY  B.bid)
SELECT    name, count
FROM      Boats AS B,  RedBoats AS RB
WHERE     B.bid = RB.bid AND count < 2
```

```
WITH tablename(attr1, …) AS (select_query)
   [,tablename(attr1, …) AS (select_query)]
main_select_query
```

Recursive WITH

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Views

```
CREATE VIEW view_name
AS select_statement
```

"tables" defined as query results rather than inserted base data

- Makes development simpler
- Used for security

Not *materialized*

References to *view_name* replaced with *select_statement*

Similar to WITH, lasts longer than one query

Names of popular boats

```
CREATE VIEW boat_counts
AS  SELECT      bid, count(*)
    FROM        Reserves R
    GROUP BY    bid
    HAVING      count(*) > 10
```

Used like a normal table

```
SELECT  bname
FROM    boat_counts bc, Boats B
WHERE   bc.bid = B.bid
```

Names of popular boats

```
SELECT  bname
FROM
    (SELECT bid, count(*)
     FROM Reserves R
     GROUP BY bid
     HAVING count(*) > 10) bc,
    Boats B
WHERE   bc.bid = B.bid
```

Rewritten expanded query

CREATE TABLE

```
CREATE TABLE <table_name> AS
    <SELECT STATEMENT>
```

Guess the schema:

```
CREATE TABLE used_boats1 AS
    SELECT r.bid
    FROM   Sailors s,
           Reservations r
    WHERE  s.sid = r.sid
```

```
used_boats1(bid int)
```

```
CREATE TABLE used_boats2 AS
    SELECT r.bid as foo
    FROM   Sailors s,
           Reservations r
    WHERE  s.sid = r.sid
```

```
used_boats2(foo int)
```

How is this different than views?

- What if we insert a new record into Reservations?

Summary

SQL is pretty complex

- Superset of Relational Algebra SQL99 turing complete!
- Human readable

More than one way to skin a horse

- Many alternatives to write a query
- Optimizer (theoretically) finds most efficient plan