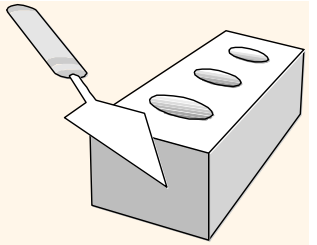
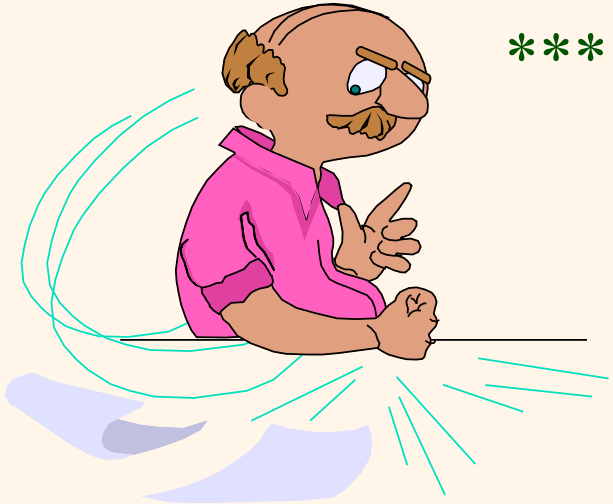


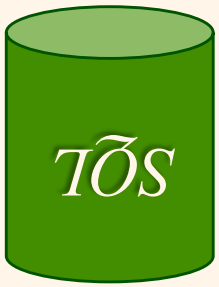
Introduction to Data Management



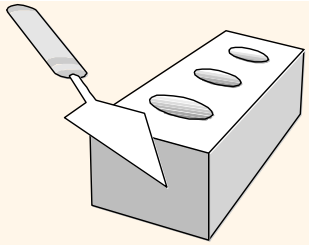
**** The "Flipped" Edition ****

Lecture #16
(Advanced SQL I)

Instructor: Mike Carey
mjcarey@ics.uci.edu



Announcements

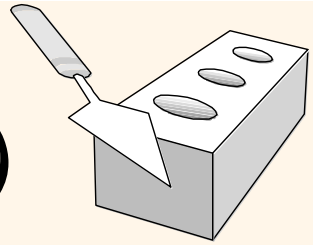


- You're over half-way through...!
  - You can ***do*** this....! ☺
- Roadmap reminder:

Relational Algebra	Ch. 2.5-2.7
Relational Calculus	Wikipedia: Tuple relational calculus
SQL Basics (SPJ and Nested Queries)	Ch. 3.3-3.5
SQL Analytics: Aggregation, Nulls, and Outer Joins	Ch. 3.6-3.9, 4.1
Advanced SQL: Constraints, Triggers, Views, and Security	Ch. 4.2, 4.4-4.5, 4.7
Midterm Exam 2	***Mon, Nov 15*** (during lecture time)

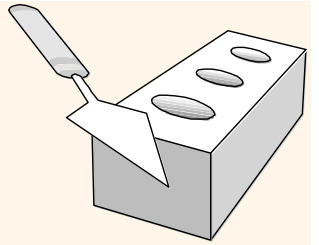
- HW#5 is in flight (and again: we're in "*Friday 6PM mode*")
  - First of the series of ***SQL-based*** HW assignments
  - Be sure to resolve any last PostgreSQL issues *immediately*!

SQL Data Integrity (Largely Review)



- An *integrity constraint* describes a condition that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key, *bid* refers to a known boat) or prevent inconsistencies (e.g., *sname* has to be a string, integer *age* must be < 120)
- *Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, unique constraints, not null constraints, general constraints.
  - *Domain constraints*: Field values must be of the right type (i.e., per the schema specification). *Always enforced!*

SQL Data Integrity (Cont.)

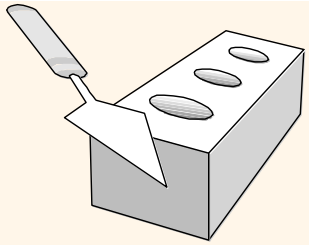


- So far we have made good use of:
  - PRIMARY KEY
  - UNIQUE
  - NOT NULL
  - FOREIGN KEY

Trivia Note: MySQL will permit a "foreign key" to reference any *indexed* column(s)... (This enables "inclusion dependencies".)

- Other features for ensuring field value integrity:
  - DEFAULT (instead of NULL for missing INSERT values)
  - CHECK (can access anything in the current row)
- More powerful integrity features include
  - ~~ASSERTION~~ (book cites as unsupported, correctly ☺)
  - TRIGGER (a sledge hammer to use when all else fails!)

Some Integrity Related Examples

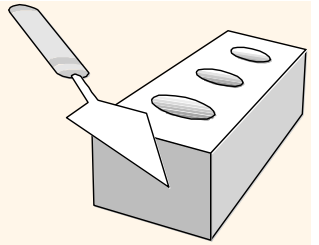


- CHECK is useful when more *general* ICs than just keys are involved.
- Could use SQL subqueries to express richer constraints (*if supported...* ☺).
- Constraints can be *named* (to manage them).

```
CREATE TABLE Sailors
        ( sid INTEGER,
        sname CHAR(10),
        rating INTEGER,
        age REAL DEFAULT 18.0,
        PRIMARY KEY (sid),
        CHECK ( rating >= 1
                AND rating <= 10 ) )
```

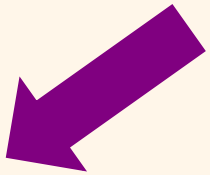
```
CREATE TABLE Reserves
        ( sid INTEGER
        bid INTEGER,
        day DATE,
        PRIMARY KEY (bid, day),
        CONSTRAINT noInterlakeRes
        CHECK (`Interlake' <>
                ( SELECT B.bname
                FROM Boats B
                WHERE B.bid=bid)) )
```

☹



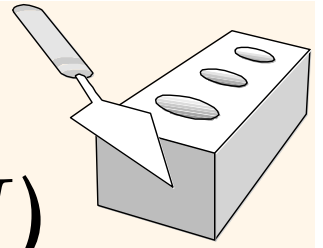
Quick PostgreSQL Examples

```
ALTER TABLE Sailors
ALTER COLUMN age SET DEFAULT 18.0;

ALTER TABLE Sailors
ADD CONSTRAINT ValidRatingConstraint
CHECK (rating >= 1 AND rating <= 10);

ALTER TABLE Sailors
ADD CONSTRAINT AliveConstraint
CHECK (age > 0);
```

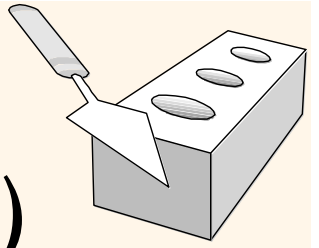


```
INSERT INTO SAILORS (sid, sname, rating, age)
VALUES (110, 'Mike', 9, 0.0);
```

Enforcing Referential Integrity (RI)



- Consider Sailors and Reserves; *sid* in Reserves is a foreign key (FK) that references Sailors.
- What should be done if a Reserves tuple with a non-existent sailor id is ***inserted***? (**A:** *Reject it!*)
- What should be done if a Sailors tuple is ***deleted***?
  - **Also delete all Reserves tuples that refer to it, or**
  - **Disallow deletion of a Sailor that's being referred to, or**
  - Set sid in Reserves tuples that refer to it to some ***default*** *sid*.
  - Could opt to set sid in Reserves tuples that refer to it to *null*, but this would not be a great idea (or allowed if key).
- Similar issues if a Sailor's primary key is ***updated***!



RI Enforcement in SQL (Reminder)

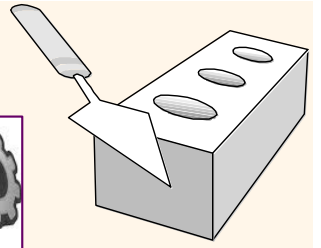
- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is NO ACTION (delete/update is rejected)
  - CASCADE (also delete all tuples that refer to the deleted tuple)
  - SET NULL / SET DEFAULT (set foreign key value of referencing tuple)

Ex:

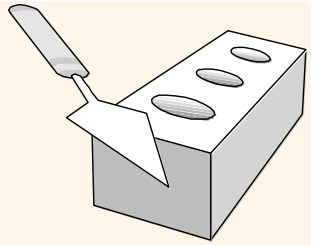
```
CREATE TABLE Reserves
   (sid INTEGER,
    bid INTEGER,
    date DATE,
    ....
    FOREIGN KEY (sid)
    REFERENCES Sailors
          ON DELETE CASCADE
          ON UPDATE SET NULL)
```

Note: *An odd combo; just illustrating some of what's possible…*

Stored Procedures (and Functions) in SQL

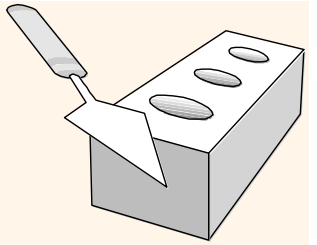


- What is a stored procedure?
  - A program executed via a single SQL statement
  - Executes in the process space of the **server**
- Advantages:
  - Can encapsulate application logic while staying "close" to the data
  - Supports the reuse (sharing) of application logic by different users
  - Can be used to help secure database applications, as we will see a bit later on



Stored Procedures: More Detail

- A ***stored procedure*** is a function or procedure written in a *general-purpose* programming language that executes *within* the DBMS.
- These can perform computations that *cannot* be expressed in SQL – i.e., they go *beyond* the limits of relational completeness.
- Procedure execution is requested through a single SQL statement (***CALL***).
- Executes on the (usually *remote*!) DBMS server.
- SQL ***PSM*** (*Persistent Stored Modules*) extends SQL with concepts from general-purpose PLs.



Stored Procedures: Functions

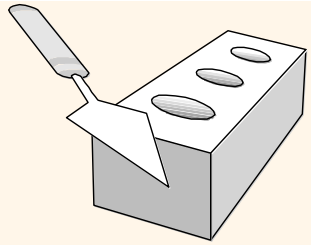
Ex: Let's define a simple function that we might want:

```
CREATE FUNCTION ShowNumReservations(boat_id int)
RETURNS TABLE (sid int, sname text, cnt bigint)
AS $$
    SELECT S.sid, S.sname, COUNT(*) AS cnt
    FROM Sailors S, Reserves R
    WHERE S.sid = R.sid AND R.bid = boat_id
    GROUP BY S.sid;
$$
LANGUAGE SQL;
```

Q: *What does this **function** do?*

First: INSERT INTO Reserves VALUES – need more data
(22, 101, '2021-10-31'), (22, 104, '2021-10-31');

Then: SELECT * FROM ShowNumReservations (101);



Stored Procedures: Procedures

Ex: Let's define a procedure that might be useful:

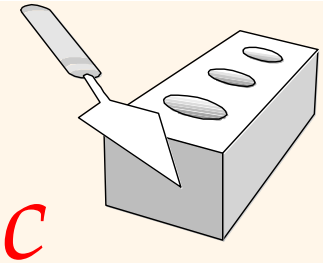
(Possible modes for parameters: IN, OUT, INOUT)

```
CREATE PROCEDURE IncreaseRating (
                IN sailor_sid int, IN increase int)
AS $$
  UPDATE Sailors
  SET rating = rating + increase
  WHERE sid = sailor_sid;
$$
LANGUAGE SQL;
```

Q: *How is this **procedure** different?*

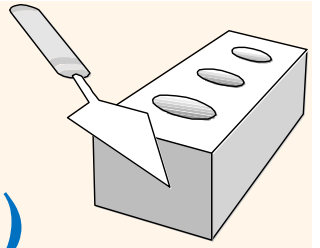
Then: `CALL IncreaseRating(95,1);`

Stored Procedures: External Logic



Stored procedures can also be written outside of the SQL language:

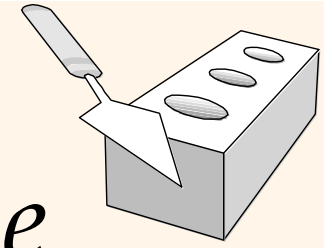
```
CREATE PROCEDURE RecklessSailors ( )
AS 'DIRECTORY/funcs', 'reckless_sailors'
LANGUAGE C;
```



Main SQL/PSM Constructs (FYI)

- Supports FUNCTIONs and PROCEDUREs
- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
  - IF (condition) THEN statements; ELSEIF (condition) statements; … ELSE statements; END IF;
  - LOOP statements; END LOOP
- Queries can be parts of expressions
- Cursors available to iterate over query results

Note: *SQL PSM is the SQL standard language for S.P.'s; **not** supported by all vendors – and **not** in PostgreSQL – due to the relative **lateness** of its standardization...!)*

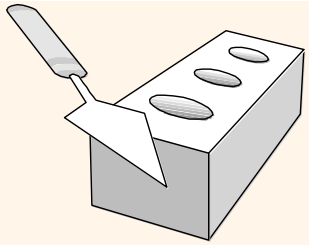


A (random ☺) SQL/PSM Example

```
CREATE FUNCTION ResvRateSailor (IN sailorId INT)
    RETURNS INT
BEGIN
    DECLARE resvRating INT
    DECLARE numResv INT
    SET numResv = (SELECT COUNT(*)
                   FROM Reserves R
                   WHERE R.sid = sailorId)
    IF (numResv > 10) THEN resvRating = 1;
                      ELSE resvRating = 0;
    END IF;
    RETURN resvRating;
END;
```

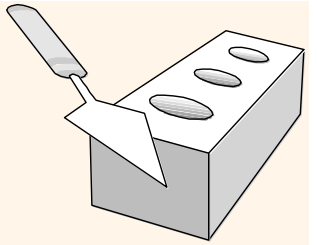
Note: *See your chosen RDBMS's docs for info about its procedural extension to SQL...*

Triggers in SQL



- Trigger: a procedure that runs automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests if the trigger should run)
  - Action (what happens if the trigger runs)
- Can be used to do "whatever"!
  - In PostgreSQL, "whatever" is a stored function call; it can also cause the current update to bail out.
  - Details vary ***WIDELY*** from vendor to vendor (!)
  - Major source of "vendor lock-in", along with their *stored procedure language* (= trigger action language)

Trigger Syntax (PostgreSQL)



CREATE TRIGGER

CREATE TRIGGER — define a new trigger

Synopsis

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

```
CREATE TRIGGER trigger_name
   {BEFORE | AFTER} { event }
   ON table_name
   [FOR [EACH] { ROW | STATEMENT }]
       EXECUTE PROCEDURE trigger_function
```

(See https://www.postgresql.org/docs/14/sql-createtrigger.html)

Trigger Example (PostgreSQL)

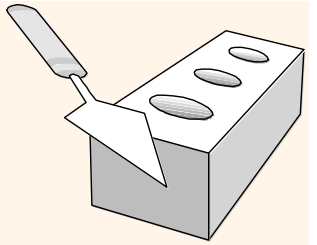
```
-- First create the logic, i.e., the "whatever" part

CREATE FUNCTION AddYoungSailor ( )
RETURNS Trigger
AS $$
  BEGIN
    INSERT INTO YoungSailors (sid, sname, age, rating)
      VALUES (NEW.sid, NEW.sname, NEW.age, NEW.rating);
    RETURN NEW;
  END;
$$
LANGUAGE PLPGSQL;
```

Trigger Example (PostgreSQL, cont.)

```
-- Now create the trigger itself

CREATE TRIGGER YoungSailorLogger
AFTER INSERT ON Sailors
FOR EACH ROW
WHEN (NEW.age < 18)
EXECUTE FUNCTION AddYoungSailor ( );
```

Note: ***FOR EACH ROW*** provides less power than ***FOR EACH STATEMENT*** (e.g., can't compute average new age)

Trigger Example (PostgreSQL, cont.)



FIRST: CREATE TABLE YoungSailors (LIKE Sailors);

And now let's try some INSERTs:

☐ INSERT INTO Sailors(sid, sname, rating, age)
VALUES (777, 'Lucky', 7, 77);

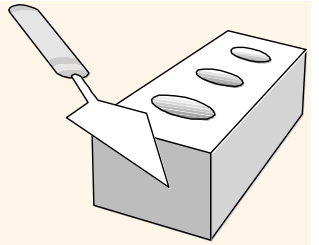
☑ INSERT INTO Sailors(sid, sname, rating, age)
VALUES (778, 'Lucky Jr', 7, 17);

*(NOTE: Look at **YoungSailors** table content after each one!)*

Trigger Example (PostgreSQL, cont.)

```
-- Let's implement a poor man's CHECK constraint!

CREATE FUNCTION BlockYoungSailor ( ) RETURNS Trigger AS $$
  BEGIN
    RAISE 'Sailors must be at least 10';
  END;
$$ LANGUAGE PLPGSQL;

CREATE TRIGGER SailorAgeEnforcer
BEFORE INSERT ON Sailors
FOR EACH ROW
WHEN (NEW.age < 10)
EXECUTE FUNCTION BlockYoungSailor ( );
```

```
INSERT INTO Sailors (sid, sname, rating, age)
VALUES (800, 'Baby Face', 10, 1);
```



To Be Continued...