Nulls, Constraints and Transactions in SQL

**Incomplete information**

Employee(empId#, name, age, mgrId#, phone, spouse, sex)

- Some attribute values may be missing: age
- Some attribute value is not applicable: spouse of an unmarried person
- Sometimes it is not clear which of the above: Has a phone been assigned? Is the phone number unlisted?

**DATABASE SOLUTION: NULL VALUES**

`INSERT INTO Employee(empId#, name, mgrId#, sex) VALUES (2331, 'joel gray', 4445, 'M')`

causes NULLs to appear in other fields (unless schema specified default values for fields)

**The “calculus” of NULL**

- What is the effect of operations with null?
  - $(\text{NULL} + 5)$ has value `null`

- What about selection condition “age=20” when age is null?

- In SQL, use 3 truth values: `true`, `false`, `unknown`
  - $(\text{NULL} = 20)$ has value `unknown`
  - `unknown AND true = unknown`
  - `unknown OR true = true`, ...

Logical operations can be explained concisely by equating `true=1`, `false=0`, `unknown=0.5`, and defining functions

- $\text{and}(x,y) = \min(x,y)$
- $\text{or}(x,y) = \max(x,y)$
- $\text{not}(x) = 1 - x$.

**Queries with NULL**

- SELECT e.name FROM Emp e
  WHERE e.age > 18 // known to be adult

- SELECT e.name FROM Emp e
  WHERE e.age > 18 OR (e.age IS NULL) // possibly adult

- SELECT e.name FROM Emp e
  WHERE e.age > 20 OR e.age <= 20 //? \(\equiv (\text{e.age is not null})\)

- Same for (e.age=e.age), (e.age=e.age),
- Duplicate elimination: <2,null> and <2,null> are duplicates, but <2,null> and <2,3> are not! So “duplicate” is not defined using '='.

- Aggregates functions discard null values before computing value. (If no value left, return null.)

  count(age) vs. count(*)
Is there a logic of nulls? (unofficial material)

- “Bob’s salary is unspecified” \( \exists y. \text{Emp}(\text{‘bob’}, y) \)
- “Bob has no wife” \( \neg \exists p. \text{spouse}(\text{‘bob’}, p) \)
- “Either Bob has no phone or it is unlisted” \( \neg \exists p. \text{phbook}(\text{‘bob’}, p) \lor \exists p. \text{phbook}(\text{‘bob’}, p) \)

Logically this is a tautology (always true) so toss it out. Has same info content as

\( \neg \exists p. \text{phbook}(\text{‘santa’}, p) \lor \exists p. \text{phbook}(\text{‘santa’}, p) \)

But how can db record that ‘bob’ exists but ‘santa’ does not?

- “Bob’s salary cannot be null”:
  \( \exists y. \text{Emp}(\text{‘bob’}, y) \) allows db without tuple \( \text{Emp}(\text{‘bob’}, _) \)

Solution [Reiter] Epistemic logic

\( \exists x. K \text{emp}(\text{‘bob’}, x) \) -- there is an x that is known to be the value of bob’s salary

\( \mathcal{L}_{\text{constrain in SQL}} \)

- Domain constraints
  CREATE TABLE \textit{Emp}(
    \textit{age} \text{ INT},
    \textit{sex} \text{ CHAR(1)},
  )
  Used to define fixed record format - very useful in efficient file/disk access

- Values must be known
  CREATE TABLE \textit{Emp}(
    \textit{empId#} \text{ INT},
    \textit{name} \text{ VARCHAR(30)} \text{ NOT NULL},
  )

Constraints(2): Identifying attribute(s)

- Have unique combination of values for the relation
  e.g., \textit{empId#} for Employee
  (\textit{name}, \textit{phone}) for Employee
  (\textit{sid},\textit{bid},\textit{day}) for Reservation

i). Primary Key

  \text{PRIMARY KEY} (\text{sid},\textit{bid},\textit{day})
  - exactly one per relation (at most 1 in SQL )
  - no nulls allowed (for implem. reasons)

ii). Key

  \text{UNIQUE (name,phone)}
  - 0 or more per relation; may be treated exactly as key, or just be used for indexing; may contain null

Integrity Constraints

Capture “regularities” in the application domain or our manipulation of it:

- \$ age is an integer
- \$ human ages are between 0 and 130; sex is the char M or F
- \$ we must know the name before entering information about the employee
- \$ the manager of an employee must also be an employee

Purpose:

- \$ help maintain validity of the data (rather than relying on every application program to check it, before/after an update)
- \$ used for more efficient data storage and manipulation
**Constraints over 1 Table**

- Can use query fragments to express constraints
- Attribute and Tuple constraints relatively efficient to check - on insert and update of tuple

```sql
CREATE TABLE Sailors
  ( sid INTEGER, sname CHAR(10), rating INTEGER, age REAL, sex CHAR(1) CHECK (sex IN ('M','F')), PRIMARY KEY (sid), CHECK (sex='F' OR (sname NOT LIKE 'Ms%')) )
  //"name begins with ‘Ms’ implies female"
```

**More general constraints over 1 table**

- Can use full query language
- Constraint can be named.
- BUT the above form is only checked when table is updated!

```sql
CREATE TABLE Employee
  ( empId# INT PRIMARY KEY, salary REAL, mngr# INT, CHECK (salary <= (SELECT m.salary FROM Employee m WHERE m.empId#=mngr#)) )
  // "must make less than manager"
CREATE TABLE Boat
  CONSTRAINT notTooMany
  CHECK ( (SELECT COUNT(*) FROM Boats) <= (SELECT COUNT(*) FROM Sailors) )
```

**Constraints(3): referential integrity**

- **Foreign key**: Set of fields in one relation that is used to `refer` to a tuple in another relation. Must correspond to primary key of the second relation. (Like a `logical pointer`. Makes up for the absence of “object identity” in relational data model.)
- E.g. `bid` in Reservations is a foreign key referring to Boats

If all foreign key constraints are enforced, **referential integrity** is achieved, i.e., no dangling references.

```sql
CREATE TABLE Reservations
  (sailid CHAR(20), boatid CHAR(20), day DATE, PRIMARY KEY (sailid,boatid,day), FOREIGN KEY (boatid) REFERENCES Boats)
```

**Syntax for Foreign Keys**

```sql
CREATE TABLE Reservations
  (sailid CHAR(20), boatid CHAR(20), day DATE, PRIMARY KEY (sailid,boatid,day), FOREIGN KEY (boatid) REFERENCES Boats)
  FOREIGN KEY (sailid) REFERENCES Sailors)
```

```sql
CREATE TABLE Boats
  (bid CHAR(20) PRIMARY KEY, name CHAR(20), color CHAR(6))
```
**Assertions: the most general constraints**

CREATE ASSERTION notTooMany
CHECK ( (SELECT COUNT(*) FROM Boats) <= ( SELECT COUNT(*) FROM Sailors) )

DBMS chooses all the updates after which this needs to be checked!

**Transactions - motivation**

Can group sequences of primitive modifications into “procedures/methods” (described by PL)

P1:

{y=SELECT balance FROM account WHERE id=445;
 w=SELECT balance FROM account WHERE id=297;
 println(y); y=y-30; w=w+30; ask for confirmation ...
 UPDATE account SET balance=y WHERE id=445;
 UPDATE account SET balance=w WHERE id=297; }
Potential problems

- Suppose that telephone connection is broken part way through.
- Concurrency control problem: a withdrawal attempted from same bank account at another ATM:
  \( z = \text{SELECT balance FROM account WHERE id=445;}
  \text{println(z); z=z-55; ...}
  \text{UPDATE account SET balance=z WHERE id=445;} \)

Transactions and Integrity Maintenance

- Verification of constraint
  - check when change occurs vs. when transaction ends
  - what kinds of changes cause constraint checking:
    - any change in any relation
    - update/insert of a tuple in a relation
    - update of specific attribute in a relation
  - how extensive is re-checking of constraint?
- Repair
  - abort transaction
  - make other changes to restore consistency
    - use nulls

Transactions: ACID property

- Atomic: whole process or none
- Consistent: integrity constraints over database preserved at end of transaction
- Isolation: gives appearance of procedure having run alone, without anyone else around
- Durable: effects not lost once committed to them

Achieved by having notions of

- begin/end transaction
- transaction reads, writes
- commit at end of transaction
- rollback, abort
Example: Referential Integrity

e.g.,
TABLE Enrolled(sid REFERENCES Students,cid...)
Can be restated as general constraint
not exists (select * from Enrolled where sid not in
(select sid from Students) )

Checking:

- Only changes to Enrolled and Students, not to
  Courses, can affect correctness of this constraint. In
  fact only delete Students, update Students.sid, insert Enrolled,
  update Enrolled.sid
- Even when such a change occurs, there is no need to
  recheck the whole constraint: rely on the fact that
  constraint held before, and check only the new sid

Triggers

- Constraints: How does one carry out other kinds of
  repairs? Or repairs for other kinds of constraints?
  How can one check efficiently when general
  constraints are violated? Or check dynamic integrity
  constraints: “Salaries do not decrease”?  

Example: Referential Integrity

TABLE Enrolled(sid REFERENCES Students(sid),cid...)

Violation handling:

- What should be done if an Enrolled tuple with a non-
  existent student id is inserted? Reject it! (abort)
- What should be done if a Students tuple is deleted?
  - Reject such a deletion if some Enrolled refers to it
  - Also delete all Enrolled tuples that refer to it. (“cascade”)
  - Set sid in Enrolled tuples that refer to it to a default sid.
  - Set sid in Enrolled tuples that refer to it to null
- Similar if primary key of Students tuple is updated.
Triggers

- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)

Issues:
- Event specification language: primitive changes to table (ins/del/upd), time alarms, “event patterns”
- Action
  - must refers to data in event and condition
  - carried out for each tuple change vs entire update statement
  - when carried out: immediate, end of transaction (deferred), separate transaction (asynchronous)

SQL’99 Trigger example

CREATE TRIGGER dynamicSalaryCheck
AFTER UPDATE OF salary ON Employee
REFERENCING
  OLD ROW AS oldTuple
  NEW ROW AS newTuple
FOR EACH ROW
WHEN (oldTuple.salary > newTuple.salary)
  INSERT INTO WarningTable(empId#, oldSal, newSal)
  VALUES(oldTuple.empid#, oldTuple.salary, newTuple.salary)