COL 362 & COL 632

SQL Recursion and Constraints

27 Jan 2023

relvars

With Clause

max_budget

value
max(budget)

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

Also called Common Table Expressions (CTEs)

department(dept_name, building, budget)

```
with max_budget (value) as
     (select max(budget) from department)
select department.name
from department, max_budget
where department.budget = max_budget.value;
```

max_budget

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

***instructor*(ID, *name*, *dept_name*, *salary*)**

```
with dept_total (dept_name, value) as
        (select dept_name, sum(salary)
         from instructor
         group by dept_name),
     dept_total_avg (value) as
        (select avg(value)
         from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

SQL - Recursion

Fix point

prereq (course_id, prereqCourse_id)

find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
      select course_id, prereqCourse_id
      from prereq
   union
      select rec_prereq.course_id, ~~prereq~~.prereq_id   prereqCourse_id
      from rec_prereq, prereq
      where rec_prereq.prereq_id = prereq.course_id
)
Select *
from rec_prereq;
```

prereqs for a course X?

1. X's prereqs (i.e., from the *prereq* relation)
2. Courses that are prereqs of all courses that are the prereqs found in step 1

SQL - Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.

- **Intuition**: *Without recursion*, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
  - This can give only a fixed number of levels of prerequisites
  - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
  - **Alternative**: write a procedure to iterate as many times as required

Restrictions on Recursion

- **query must be monotonic**
  - Without this there is no guarantee of termination
  - Termination – when you reach **a fix-point**

- SQL prohibits the use of following in recursive query definition:
  - Aggregation on the recursive view.
  - **not exists** on a subquery that uses the recursive view
  - Set difference (**except**) whose right-hand side uses the recursive view.
- Each of these make the query size to *reduce* during the recursive computation

A Small Homework

- Design a relational schema and SQL queries to
  i. Store the basis functional dependencies
  ii. Check if a functional dependency is part of the closure of basis set of functional dependencies stored in your relation(s) designed above

Constraints

Constraints for DB Consistency

- Security Contraints
  - Access control mechanisms
  - Restricted views
  - Prevent unauthorized access to relations and attributes
  - *Eg. A student may not be allowed to see records of another student,*
- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency
  - *A current account must have a monthly balance greater than Rs. 5,000*
  - *Minimum salary of an employee with graduate degree in Delhi should be Rs. 21,000*
  - *Every faculty member must have a (non-null) phone number*

Constraints on Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

Specified as part of the relation declaration

Constraints on Single Relation

- **not null**
  - Declare name and budget to be not null (as part of the relation defn.)

```
name varchar(20) not null
budget numeric(12,2) not null
```

- **primary key**
- **unique**
- **check** (P), where P is a predicate

Constraints on Single Relation

- **not null**
- **primary key**
- **Unique**
  unique (A1, A2, …, Am)
  - The unique specification states that the attributes A1, A2, …, Am form **a candidate key**
  - Candidate keys are permitted to be null (in contrast to primary keys).
- **check** (P), where P is a predicate

Constraints on Single Relation

- **check** (P), where P is a predicate

The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.

Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
     (course_id varchar (8),
      sec_id varchar (8),
      semester varchar (6),
      year numeric (4,0),
      building varchar (15),
      room_number varchar (7),
      time slot id varchar (4),
      primary key (course_id, sec_id, semester, year),
      check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

Referential Integrity (1/2)

ActedIn

Name	Movie
PC	Don2
AH	Thor: R
KS	Eclipse

Movies

Movie	Year
Thor: R	2011
Eclipse	2008
Don2	2011
DKR	2012

Foreign key

Key or Unique constraint

```
CREATE TABLE ActedIn (
     Name varchar(30), Movie varchar(30),
     CONSTRAINT actedIn_fkey
         FOREIGN KEY (Movie) REFERENCES Movies (Movie));
```

Referential Integrity (2/2)

- Reject, Cascade, Set-null

Reject modifications which violate constraints

"Transfer" modifications

Set attribute(s) to null if needed

ActedIn

Name	Movie
PC	Don2
AH	Thor: R
KS	Eclipse
CB	DKR

Movie

Movie	Year
Thor: R	2011
Eclipse	2008
Don2	2011
DKR	2012

insert into actedIn: ('CB', 'DKR')
insert into actedIn: ('AH', 'Thor: Ragnarok')

delete from actedIn: ('KS', 'Eclipse')
delete from Movie: ('DKR', 2012)
delete from Movie: ('Eclipse', 2008)

update Movie: ('DKR', 2012) to ('DK', 2011)

```
CREATE TABLE ActedIn (
     Name varchar(30), Movie varchar(30),
     CONSTRAINT actedIn_fkey
          FOREIGN KEY (Movie) REFERENCES Movie (movieid)
          ON DELETE CASCADE)
```

“Cyclic” Constraints (1/3)

Actors

Name	Age	Addr	Famous_ Movie
Priyanka Chopra	36	Mumbai	Don-II
Anthony Hopkins	81	LA	Thor: R
Bill Nighy	69	LA	Valkyrie
Abhishek Bachchan	42	Mumbai	Raavan

Movies

Name	Year	Title
Priyanka Chopra	2011	Don-II
Anthony Hopkins	2011	Thor: R
Bill Nighy	2009	Valkyrie
Abhishek Bachchan	2010	Raavan
Anthony Hopkins	2003	TLS

insert into Actors: (‘Kristen Stewart’, 23, ‘LA’, ‘Breaking Dawn’);
insert into Movies: (‘Kristen Stewart’, 2011, ‘Breaking Dawn’);

“Cyclic” Constraints (2/3)

- Notion of a “Transaction”
  - An atomic unit of execution
    The state of the database is *consistent* before and after a successful completion of a transaction
  - Currently, the two inserts together form a single transaction

```
insert into Actors: (‘Kristen Stewart’, 23, ‘LA’, ‘Breaking Dawn’);
insert into Movies: (‘Kristen Stewart’,  2011, ‘Breaking Dawn’);
```

- **Defer constraint checking until after transaction**

“Cyclic” Constraints (3/3)

Actors

Name	Age	Addr	Famous _Movie
Priyanka Chopra	36	Mumbai	Don-II
Anthony Hopkins	81	LA	Thor: R
Bill Nighy	69	LA	Valkyrie
Abhishek Bachchan	42	Mumbai	Raavan

Movies

Name	Year	Title
Priyanka Chopra	2011	Don-II
Anthony Hopkins	2011	Thor: R
Bill Nighy	2009	Valkyrie
Abhishek Bachchan	2010	Raavan
Anthony Hopkins	2003	TLS

```
CREATE TABLE Actors (
   Name varchar(30), Age int, Addr varchar (30), Famous_Movie varchar(30),
   CONSTRAINT actors_fkey
        FOREIGN KEY (Famous_Movie) REFERENCES Movies (Title)
        DEFERRABLE INITIALLY DEFERRED)
```

Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a “unit” of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work**. The updates performed by the transaction become permanent in the database.
  - **Rollback work**. All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions

We will study txns later in the course