CMSC424: Database Design

Instructor: Amol Deshpande
amol@cs.umd.edu
Databases

- Data Models
  - Conceptual representation of the data

- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions

- Data Storage
  - How/where to store data, how to access it

- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies
Today’s Plan

- Discuss reading assignment topics
  - Duplicates in SQL
  - Embedded/Dynamic SQL, Functions, Recursion

- New topics to discuss
  - Views, Integrity Constraints, Triggers, Transactions, Roles
  - Writing SQL Queries

- Other things
  - Computing environment setup
  - Small change in the late days policy (max 4 days for any assignment instead of 5, so I can discuss in class on Wed)
  - DSS: Let me know soon
Outline

- More SQL
  - Views
  - Integrity Constraints
  - Triggers
  - Transactions
  - Authorization
Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

\[
\text{create view } v \text{ as } \langle \text{query expression} \rangle
\]

where:

- \(\langle \text{query expression} \rangle\) is any legal expression
- The view name is represented by \(v\)

- Can be used in any place a normal table can be used
- For users, there is no distinction in terms of using it
Example Queries

- A view of instructors without their salary
  ```sql
  create view faculty as
  select ID, name, dept_name
  from instructor
  ```

- Find all instructors in the Biology department
  ```sql
  select name
  from faculty
  where dept_name = 'Biology'
  ```

- Create a view of department salary totals
  ```sql
  create view departments_total_salary(dept_name, total_salary) as
  select dept_name, sum(salary)
  from instructor
  group by dept_name;
  ```
Views

- Is it different from DBMS’s side?
  - Yes; a view may or may not be *materialized*
  - Pros/Cons?

- Updates into views have to be treated differently
  - In most cases, disallowed.
# Views vs Tables

<table>
<thead>
<tr>
<th>Creating</th>
<th>Create view V as (select * from A, B where ...)</th>
<th>Create table T as (select * from A, B where ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used</td>
<td>In any select query. Only some update queries.</td>
<td>It's a new table. You can do what you want.</td>
</tr>
<tr>
<td>Maintained as</td>
<td>1. Evaluate the query and store it on disk as if a table. 2. Don’t store. Substitute in queries when referenced.</td>
<td>It’s a new table. Stored on disk.</td>
</tr>
<tr>
<td>What if a tuple inserted in A?</td>
<td>1. If stored on disk, the stored table is automatically updated to be accurate. 2. If we are just substituting, there is no need to do anything.</td>
<td>T is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger.</td>
</tr>
</tbody>
</table>
Views vs Tables

- Views strictly supercede “create a table and define a trigger to keep it updated”

- Two main reasons for using them:
  - Security/authorization
  - Ease of writing queries

- Perhaps the only reason to create a table is to force the DBMS to choose the option of “materializing”
  - That has efficiency advantages in some cases
  - Especially if the underlying tables don’t change
Update of a View

- Add a new tuple to faculty
  
  \textbf{insert into} \textit{faculty values} (’30765’, ’Green’, ’Music’);

- This insertion must be represented by the insertion of the tuple
  into the \textit{instructor} relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed, e.g.:
  - \textbf{create view} \textit{instructor\_info as}
    \begin{verbatim}
    select ID, name, building 
    from instructor, department 
    where instructor.dept\_name = department.dept\_name;
    \end{verbatim}
  - \textbf{insert into} \textit{instructor\_info values} (’69987’, ’White’, ’Taylor’);

- Many SQL implementations allow updates only on simple views (without aggregates) defined on a single relation
Outline

- More SQL
  - Views
  - Integrity Constraints
  - Triggers
  - Transactions
  - Authorization
Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - **commit work**: makes all updates of the transaction permanent in the database
    - **rollback work**: undoes all updates performed by the transaction.

Motivating example
- Transfer of money from one account to another involves two steps:
  - deduct from one account and credit to another
- If one steps succeeds and the other fails, database is in an inconsistent state
- Therefore, either both steps should succeed or neither should

- If any step of a transaction fails, all work done by the transaction can be undone by rollback work.
- Rollback of incomplete transactions is done automatically, in case of system failures
In most database systems, each SQL statement that executes successfully is automatically committed.

- Each transaction would then consist of only a single statement.
- Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system.
- Another option in SQL:1999: enclose statements within `begin atomic
  
  ...

  end`
More SQL
- Views
- Integrity Constraints
- Triggers
- Transactions
- Authorization
A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
- 1. setting the account balance to zero
- 2. creating a loan in the amount of the overdraft
- 3. giving this loan a loan number identical to the account number of the overdrawn account
create trigger overdraft-trigger after update on account
  referencing new row as nrow
  for each row
  when nrow.balance < 0
  begin atomic
    actions to be taken
  end
create trigger overdraft-trigger after update on account
  referencing new row as nrow
  for each row
  when nrow.balance < 0
begin atomic
  insert into borrower
    (select customer-name, account-number
     from depositor
     where nrow.account-number = depositor.account-number);
  insert into loan values
    (nrow.account-number, nrow.branch-name, nrow.balance);
  update account set balance = 0
    where account.account-number = nrow.account-number
end
Triggers...

- External World Actions
  - How does the DB *order* something if the inventory is low?

- Syntax
  - Every system has its own syntax

- Careful with triggers
  - Cascading triggers, Infinite Sequences...

- More Info/Examples:
  - Google: “create trigger” oracle download-uk
Outline

- More SQL
  - Views
  - Integrity Constraints
  - Triggers
  - Transactions
  - Authorization
IC’s

- Predicates on the database
- Must always be true (checked whenever db gets updated)

There are the following 4 types of IC’s:

- **Key constraints** (1 table)
  
  e.g., 2 accts can’t share the same acct_no

- **Attribute constraints** (1 table)
  
  e.g., accts must have nonnegative balance

- **Referential Integrity constraints** (2 tables)
  
  E.g. bnames associated w/ loans must be names of real branches

- **Global Constraints** (n tables)
  
  E.g., all loans must be carried by at least 1 customer with a savings acct
Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. **Primary Key:**

   ```sql
   CREATE TABLE branch(
     bname  CHAR(15)  PRIMARY KEY,
     bcity      CHAR(20),
     assets    INT);
   ```

   or

   ```sql
   CREATE TABLE depositor(
     cname   CHAR(15),
     acct_no  CHAR(5),
     PRIMARY KEY(cname, acct_no));
   ```

2. **Candidate Keys:**

   ```sql
   CREATE TABLE customer (  
     ssn     CHAR(9)    PRIMARY KEY,  
     cname  CHAR(15),  
     address CHAR(30),  
     city          CHAR(10),  
     UNIQUE (cname, address, city));
   ```
Key Constraints

Effect of SQL Key declarations
  PRIMARY (A1, A2, .., An) or
  UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any inserted tuple. If found, reject insertion

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)
  1. 1 primary key per table, several unique keys allowed.
  2. Only primary key can be referenced by “foreign key” (ref integrity)
  3. DBMS may treat primary key differently (e.g.: create an index on PK)

How would you implement something like this?
Attribute Constraints

- **Idea:**
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)

- **In SQL:**
  1. **NOT NULL**
     - e.g.: `CREATE TABLE branch(
               bname  CHAR(15)  NOT NULL,
               ....
          )`
     - Note: declaring `bname` as primary key also prevents null values

  2. **CHECK**
     - e.g.: `CREATE TABLE depositor(
               ....
               balance int NOT NULL,
               CHECK(  balance >= 0),
               ....
          )`
     - affect insertions, update in affected columns
Attribute Constraints

Domains: can associate constraints with DOMAINS rather than attributes

e.g. instead of:

```sql
CREATE TABLE depositor(
  ....
  balance INT NOT NULL,
  CHECK (balance >= 0)
)
```

One can write:

```sql
CREATE DOMAIN bank-balance INT (
  CONSTRAINT not-overdrawn CHECK (value >= 0),
  CONSTRAINT not-null-value CHECK( value NOT NULL));

CREATE TABLE depositor (  
  ....
  balance   bank-balance,
)
```

Advantages?
Attribute Constraints

Advantage of associating constraints with domains:

1. can avoid repeating specification of same constraint for multiple columns

2. can name constraints
   e.g.: CREATE DOMAIN bank-balance INT (CONSTRAINT not-overdrawn CHECK (value >= 0), CONSTRAINT not-null-value CHECK( value NOT NULL));

allows one to:

1. add or remove:
   ALTER DOMAIN bank-balance ADD CONSTRAINT capped CHECK( value <= 10000)

2. report better errors (know which constraint violated)
Referential Integrity Constraints

Idea: prevent “dangling tuples” (e.g.: a loan with a bname, *Kenmore*, when no *Kenmore* tuple in branch)

Ref Integrity:
ensure that:
foreign key value $\rightarrow$ primary key value

(note: don’t need to ensure $\leftarrow$, i.e., not all branches have to have loans)
Referential Integrity Constraints

In SQL:

```
CREATE TABLE branch(
    bname CHAR(15) PRIMARY KEY
....)

CREATE TABLE loan ( 
    ........
    FOREIGN KEY bname REFERENCES branch);
```

Affects:
1) Insertions, updates of referencing relation
2) Deletions, updates of referenced relation
what happens when we try to delete this tuple?

Ans: 3 possibilities
1) reject deletion/ update

2) set $t_i[c], t_j[c] = \text{NULL}$

3) propagate deletion/update
   
   DELETE: delete $t_i, t_j$
   
   UPDATE: set $t_i[c], t_j[c]$ to updated values
Referential Integrity Constraints

**CREATE TABLE A ( .....**

FOREIGN KEY c REFERENCES B **action**

**......... )**

**Action: 1) left blank (deletion/update rejected)**

**2) ON DELETE SET NULL/ ON UPDATE SET NULL**

sets $ ti[c] = NULL$, $ tj[c] = NULL$

**3) ON DELETE CASCADE**

deletes $ti$, $tj$

**ON UPDATE CASCADE**

sets $ ti[c]$, $ tj[c]$ to new key values

what happens when we try to delete this tuple?
Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)
   - E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

2) multiple relations: CREATE ASSERTION

SQL examples:

1) single relation: All BkLn branches must have assets > 5M

   CREATE TABLE branch (  
       ............
       bcity CHAR(15),
       assets INT,
       CHECK (NOT(bcity = 'BkLn') OR assets > 5M))

Affects:

insertions into branch
updates of bcity or assets in branch
Global Constraints

SQL example:
2) Multiple relations: every loan has a borrower with a savings account

CHECK (NOT EXISTS ( 
    SELECT * 
    FROM loan AS L 
    WHERE NOT EXISTS( 
        SELECT * 
        FROM borrower B, depositor D, account A 
        WHERE B.cname = D.cname AND 
        D.acct_no = A.acct_no AND 
        L.lno = B.lno) 
))

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:
    CREATE ASSERTION loan-constraint
    CHECK( ..... )

Checked with EVERY DB update! very expensive.....
## Summary: Integrity Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Where declared</th>
<th>Affects...</th>
<th>Expense</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Constraints</strong></td>
<td>CREATE TABLE (PRIMARY KEY, UNIQUE)</td>
<td>Insertions, Updates</td>
<td>Moderate</td>
</tr>
<tr>
<td><strong>Attribute Constraints</strong></td>
<td>CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)</td>
<td>Insertions, Updates</td>
<td>Cheap</td>
</tr>
<tr>
<td><strong>Referential Integrity</strong></td>
<td>Table Tag (FOREIGN KEY .... REFERENCES ....)</td>
<td>1. Insertions into referencing rel’n</td>
<td>1,2: like key constraints. Another reason to index/sort on the primary keys</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Updates of referencing rel’n of relevant attrs</td>
<td>3,4: depends on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Deletions from referenced rel’n</td>
<td>a. update/delete policy chosen</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. Update of referenced rel’n</td>
<td>b. existence of indexes on foreign key</td>
</tr>
<tr>
<td><strong>Global Constraints</strong></td>
<td>Table Tag (CHECK) or outside table (CREATE ASSERTION)</td>
<td>1. For single rel’n constraint, with insertion, deletion of relevant attrs</td>
<td>1. cheap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. For assertions w/ every db modification</td>
<td>2. very expensive</td>
</tr>
</tbody>
</table>
Outline

- More SQL
  - Views
  - Integrity Constraints
  - Triggers
  - Transactions
  - Authorization
Authorization and Roles

- Forms of authorization on parts of the database:
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.

- The **grant** statement is used to confer authorization
  
  \[
  \text{grant} \ <\text{privilege list}> \\
  \text{on} \ <\text{relation name or view name}> \ \text{to} \ <\text{user list}> \\
  \]

- Examples from PostgreSQL manual:
  - GRANT INSERT ON films TO PUBLIC;
  - GRANT ALL PRIVILEGES ON kinds TO manuel;
Is that it?
  ◦ Unfortunately No
  ◦ SQL 3 standard is several hundreds of pages (if not several thousands)
  ◦ And expensive too..

We will discuss a few more constructs along the way

  E.g. Embedded SQL, creating indexes etc

Again, this is what the reference books are for; you just need to know where to look in the reference book
Examples of Complex SQL Queries

- How to approach a complex query?
  - Try to break up the task into smaller pieces, by figuring out some “intermediate” tables that might help
0. Find student-instructor pairs s.t. the student has taken multiple courses with the instructor.
1. Find students who took a course with the instructor they are advised by

```
section
- course_id
- sec_id
- semester
- year
- building
- room_no
- time_slot_id

section
- time_slot_id
- day
- start_time
- end_time

room
- building
- room_no

classroom
- building
- room_no
- capacity

prereq
- course_id
- prerequisite_id

course
- course_id
- title
- dept_name
- credits

department
- dept_name
- building
- budget

advisor
- s_id
- i_id

instructor
- ID
- name
- dept_name
- salary

takes
- ID
- course_id
- sec_id
- semester
- year

student
- ID
- name
- dept_name
- tot_cred
```
2. Find classrooms that have hosted most classes so far
3. Identify students who took a course but not one of its pre-req
4. Find classrooms that were over capacity, i.e., had more students enrolled than the capacity.