Intermediate- sql-2

Materialized Views

- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the view

- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Syntax of MV

```
Create materialized view View_Name
 Build [Immediate/Deffered]
 Refresh [Fast/Complete/Force]
on [Commit/Demand]
as Select .....;
```

example

- Create or Replace Materialized view MV_Employee
- as
- Select E.Employee_num,E.Employee_name,D.Department_Name
- from Employee E , Department D where E.Dept_no=D.Dept_no
- Refresh auto on commit select * from Department;

Difference Between Materialized View And View :

View	Materialized Views(Snapshots)
1.View is nothing but the logical structure of the table which will retrieve data from 1 or more table.	1.Materialized views(Snapshots) are also logical structure but data is physically stored in database.
2.You need to have Create view privileges to create simple or complex view	2.You need to have create materialized view 's privileges to create Materialized views
3.Data access is not as fast as materialized views	3.Data retrieval is fast as compare to simple view because data is accessed from directly physical location
4.There are 2 types of views: 1.Simple View 2.Complex view	4.There are following types of Materialized views: 1.Refresh on Auto 2.Refresh on demand
5.In Application level views are used to restrict data from database	5.Materialized Views are used in Data Warehousing.

Basics of Transactions

- A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Transactions begin implicitly
- Ended by **commit work** : commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database
- **rollback work :** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction.
- **(note: detail discussion on transaction will revisit later)**

Integrity Constraints

- 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.
- FOR EXAMPLE:
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

- Integrity constraints are usually identified as part of the database schema design process, and declared as part of the **create table** command used to create relations.
- However, integrity constraints can also be added to an existing relation by using the command **alter table** *table-name* **add** *constraint*, where *constraint* can be any constraint on the relation.
- When such a command is executed, the system first ensures that the relation satisfies the specified constraint.
- If it does, the constraint is added to the relation; if not, the command is rejected.

Integrity Constraints on a Single Relation

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

Not Null and Unique Constraints

- **Primary Key Constraint**: When this constraint is associated with the column of a table it will not allow NULL values into the column and it will maintain unique values as part of the table.
- **not null**
  - Declare *name* and *budget* to be **not null**

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

- **unique** (A_1, A_2, ..., A_m)
- Unique Constraint: When this constraint is associated with the column(s) of a table it will not allow us to store a repetition of data/values in the column, but Unique constraint allows ONE NULL value. More than one NULL value is also considered as repetition, hence it does not store a repetition of NULL values also.
  - 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).
- A constraint that is added immediately next to the column definition is known as a Column-level constraint. All constraints that are added after the columns definition of the table is completed are known as Table-level constraints.

- CREATE TABLE employee
  (id number(5) PRIMARY KEY,
  name char(20),
  dept char(10),
  age number(2),
  salary number(10),
  location char(10) UNIQUE
  );
- A constraint that is added immediately next to the column definition is known as a Column-level constraint. All constraints that are added after the columns definition of the table is completed are known as Table-level constraints.

The check clause

- When applied to a relation declaration, the clause **check**(*P*) specifies a predicate *P* that must be satisfied by every tuple in a relation.
- **check** (P), where P is a predicate

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

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.

- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if **for any values of A appearing in R these values also appear in S**.

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (that is, the transaction performing the update action is rolled back).
- However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint

Cascading Actions in Referential Integrity

- **create table** *course* (
  ...
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department*
  on delete cascade
  on update cascade ,
  . . .
)

- Because of the clause **on delete cascade** associated with the foreign-key declaration, **if a delete of a tuple in *department*** results in this referential-integrity constraint being violated, the system does not reject the delete.
- Instead, the **delete "cascades"** to the *course* relation, deleting the tuple that refers to the department that was deleted.
- Similarly, the system does not reject **an update** to a field referenced by the constraint if it violates the constraint; instead, the system updates the field *dept name* in the referencing tuples in *course* to the new value as well.
- alternative actions to cascade: **set null**, **set default**

```
CREATE TABLE buildings (
    building_no INT PRIMARY KEY AUTO_INCREMENT,
    building_name VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL
);
```

Step 2. Create the `rooms` table:

```
CREATE TABLE rooms (
    room_no INT PRIMARY KEY AUTO_INCREMENT,
    room_name VARCHAR(255) NOT NULL,
    building_no INT NOT NULL,
    FOREIGN KEY (building_no)
        REFERENCES buildings (building_no)
        ON DELETE CASCADE
);
```

```
SELECT * FROM buildings;
```

building_no	building_name	address
1	ACME Headquaters	3950 North 1st Street CA 95134
2	ACME Sales	5000 North 1st Street CA 95134

```
SELECT * FROM rooms;
```

room_no	room_name	building_no
1	Amazon	1
2	War Room	1
3	Office of CEO	1
4	Marketing	2
5	Showroom	2

```
DELETE FROM buildings
WHERE building_no = 2;
```

Step 8. Query data from `rooms` table:

```
SELECT * FROM rooms;
```

	room_no	room_name	building_no
▶	1	Amazon	1
	2	War Room	1
	3	Office of CEO	1

Integrity Constraint Violation During Transactions

- E.g.

  create table *person* (
  ID **char**(10),
  name **char**(40),
  mother **char**(10),
  father **char**(10),
  primary key *ID,*
  foreign key *father* **references** *person,*
  foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation ?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking (next slide)

Complex check clause

- As defined by the SQL standard, the predicate in the **check** clause can be an arbitrary predicate, which can include a subquery.
- **check** (*time slot id* **in** (**select** *time slot id* **from** *time slot*))
- The **check** condition verifies that the *time slot id* in each tuple in the *section* relation is actually the identifier of a time slot in the *time slot* relation.
- Thus, the condition has to be checked not only when a tuple is inserted or modified in *section*, but also when the relation *time slot* changes (in this case, when a tuple is deleted or modified in relation *time slot*)

Complex Check Clauses

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions
- **create assertion** <assertion-name> **check** <predicate>;

- *This SQL statement creates an assertion to demand that there's no more than a single president among the employees* :

```
create assertion AT_MOST_ONE_PRESIDENT as CHECK
((select count(*)
   from EMP e
  where e.JOB = 'PRESIDENT') <= 1
)
```

- *This SQL statement creates an assertion to demand that Boston based departments do not employ trainers*:

```
create assertion NO_TRAINERS_IN_BOSTON as CHECK
  (not exists
   (select 'trainer in Boston'
     from EMP e, DEPT d
    where e.DEPTNO = d.DEPTNO
      and e.JOB    = 'TRAINER'
      and d.LOC    = 'BOSTON')
  )
```

- *This SQL statement creates an assertion to demand that vacation records cannot be outside of one's employment period*:

```
create assertion VACATION_DURING_EMPLOYMENT as CHECK
(not exists
    (select 'vacation outside employment'
      from EMP e
        ,EMP_VACATION ev
     where e.EMPNO = ev.EMPNO
      and (ev.FIRST_DATE < e.HIRE_DATE or
          ev.LAST_DATE > e.TERMINATION_DATE))
  )
```

- Make sure that supervisee's salary is less than his Supervisor's salary

```
Create assertion Emp_sal_chek
check(not exists (select * from Employee E1, Employee E2
        where E1.Mgr_Id=E2.Emp_Id and
        E1.salary>E2.salary  ) );
```

- ***For each tuple in the student relation, the value of the attribute tot cred must equal the sum of credits of courses that the student has completed successfully .***

create assertion *credits earned constraint* **check**
(**not exists** (**select** ID **from** *student*
where *tot cred* <> (**select sum**(*credits*)
from *takes* **natural join** *course*
where *student.ID= takes.ID*
and *grade* **is not null and** *grade*<> 'F')

Built-in Data Types in SQL

- **date**: Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time**: Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval**: period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Index Creation

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- Indices are data structures used to speed up access to records with specified values for index attributes
  - e.g. **select ***
    from *student*
    where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

  More on indices in Chapter 11

User-Defined Types

- **create type** construct in SQL creates user-defined type

create type *Dollars* **as numeric (12,2)**

- **create table** *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself.