1 Views

Relations that are defined with `CREATE TABLE` actually exist in the database, so SQL system stores the tables at some physical memory location and these tables persist. A (virtual) view on the other hand is a relation whose rows are not explicitly stored in the database but are computed as needed from a *view definition*. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used.

1.1 Usefulness of views

- *logical data independence*
  - If the schema of a stored relation is changed, we can define a view with the old schema and applications that expect to see the old schema can now use this view
  - May wish to create a personalized collection of relations that is better matched to a certain user's intuition than is the logical model.
- *security* - security considerations may require that certain data be hidden from users
  - Can define views that give a group of users access to just the information they are allowed to see. For instance we can define a view that allows students to see other student's name and age but not their GPA, and allow all students to see this view but not the underlying `Students` table.

2 Views in SQL

Views are defined in SQL using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view.

`CREATE VIEW` v `AS <query expression>`

Consider the clerk who needs to access all data in the `Instructor` table, except *salary*. A view relation $faculty$ can be made available to the clerk as follows:

```
CREATE VIEW faculty AS
SELECT ID, name, dept_name
FROM instructor;
```

The view relation conceptually contains the tuples in the query result, but is not precomputed and stored. Instead, the database system stores the query expression associated with the view relation. Whenever the view relation is accessed, its tuples are created by computing the query result. Thus, the view relation is created whenever needed, on demand

2.1 Using Views in SQL Queries

Suppose we have created a view that list all course sections offered by the Physics department in the Fall 2020 quarter with the building and room number of each section:

```
CREATE VIEW physics_fall_2020 AS
    SELECT course.course_id, sec_id, building, room_number
    FROM course, section
    WHERE course.course_id = section.course_id
    AND course_dept_name = 'Physics'
    AND section.quarter = 'Fall'
    AND section.year = '2020'
```

Using view *physics_fall_2020*, we can find all physics courses offered in the fall of 2020 in the Annenburg Hall as follows:

```
SELECT course_id
FROM physics_fall_2020
WHERE building='Annenburg Hall'
```

The attribute names of a view can be specified explicitly as follows:

```
CREATE VIEW derpartments_total_salary(dept_name, total_salary) AS
SELECT dept_name, SUM(salary)
FROM instructor
GROUP BY dept_name
```

The preceding view gives each department the sum of the salaries of all the instructors at that department. Since the expression `SUM(salary)` does not have a name, the attribute name is specified explicitly in the view definition.

3 WITH clause and Views

The `WITH` clause defines a temporary view with scope limited to the query

The following query finds the departments with the maximum budget

```
WITH max_budget(value) AS
    (SELECT MAX(budget)
    FROM department)
SELECT budget
FROM department, max_budget
WHERE department.budget = max_budget.value;
```

The `WITH` clause defines the temporary relation *max_budget*, which is used in the immediately following query

Obviously, we could have written the query using a nested sub-query. However, the nesting would have made the query harder to read. In addition to making the query logic clearer; it also *permits a view definition to be used at multiple places within the same query.*

Suppose we want to find all departments where the total salary is greater than the average of the total salary of all departments:

```
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
```

The equivalent query without the `WITH` clause would be complicated.

4 Recursion and Hierarchies

The *Prereq* relation

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-201	CS-101
CS-301	CS-201
CS-347	CS-301
EE-181	PHY-101

The *Prereq* table shows info about various courses offered at a university and the prerequisite for each course. Suppose we want to find out which courses are a prerequisite whether directly or indirectly, for a specific course – say, CS-347. That is, we wish to find a course that is a direct prerequisite for CS-347, or is a prerequisite for a course that is a prerequisite for CS-347, and so on. Thus, if CS-301 is a prerequisite for CS-347, and CS-201 is a prerequisite for CS-301, and CS-101 is a prerequisite for CS-201, then {CS-301, CS-201,CS-101} all prerequisites for CS-347.

The **transitive closure** of the relation *Prereq* is a relation that contains all pairs $(course\_id, prereq\_id)$ such that *prereq_id* is a direct or indirect prerequisite of *course_id*. Common examples are transitive closures on **hierarchies**.

- Organizations typically consist of several levels of hierarchies. Who are the people that Becky from accounting reports to, both directly and indirectly?
  - supervisor → manager → regional manager → vice president → senior vice president → president → assistant director → director → regional director → CEO.
- Machines consist of parts that in turn have subparts, and so on
  - A bicycle may have subparts such as wheels and pedals, which in turn have subparts such as tires, rims, and spokes.

Transitive closure can be used on such hierarchies to find, for example, all parts in a bicycle.

4.1 Recursion Example

Consider the task of finding all prerequisites for the course CS-347. The courses that are prerequisites (directly or indirectly) of CS-347 are:

1. Courses that are prerequisites for CS-347
2. Courses that are prerequisites for the above courses
3. Courses that are prerequisites for the above courses
4. ...

The case is recursive, since we are asking the same question from case 1 again except on the set of courses that are prerequisites of CS-347, instead of for CS-347 itself.
SQL supports a limited form of recursion, using the `WITH RECURSIVE` clause, where a view is expressed in terms of it-selfs. Recall that the `WITH` clause is used to define a temporary view whose definition is available only to the query in which it is defined. The additional keyword `RECURSIVE` specifies that the view is recursive. Any recursive view must be defined as the union of two sub-queries: a `base query` that is non-recursive and a `recursive query` that uses the recursive view

Find every pair $(course\_id, prereq\_id)$ such that $prereq\_id$ is directly or indirectly a prerequisite for course $course\_id$

```
WITH RECURSIVE C_prereq(course_id, prereq_id) AS (
    SELECT course_id, prereq_id
    FROM Prereq

    UNION

    SELECT P.course_id, P.prereq_id
    FROM Prereq P, C_prereq C
    WHERE P.course_id = C.prereq_id )
SELECT *
FROM C_prereq
```

In this example the base query is the select on $Prereq$ while the recursive query computes the join of $Prereq$ and C_prereq

The process of a recursive view is understood as follows.

1. First, compute the base query and add all the resultant tuples to the recursively defined view relation C_prereq (which is initially empty) – giving us the first layer.
2. Next, compute the recursive query using the current contents of the view relation, and add all the resulting tuples back to the view relation – giving us another layer.
3. Repeating (step 2) until no new tuples are added to the view relation.

Applying this logic to our example:

- We first find all direct prerequisites of each course by executing the base query.
- The recursive query adds one more level of courses in each iteration, until the maximum depth of the course-prereq relationship is reached.

Note that `UNION` eliminates duplicates so that we are not stuck in an infinite loop.

4.2 More Examples

4.2.1 Example 1: Employees

Advanced Topics

The *Emp* relation

employee	manager
Beck	James
Ben	James
Chris	Beck
Joseph	Beck
Nancy	Ben
Mike	Ben
John	Chris
Milton	Chris
Harris	Joseph
Paul	Joseph
Dhana	Nancy
Joan	Nancy
Trevor	Mike
David	Mike
Laura	Paul
Edwin	Paul

Consider a table called Emp as shown above. Each row contains the employee name and his/her manager name. Find all people under manager Beck either directly (or) indirectly as shown below:

- Chris and Joseph (reporting to Beck)
- John, Milton, Harris, Paul (reporting to Chris and Joseph)
- Laura and Edwin reporting to Paul

Obtain all people under manager Beck either directly (or) indirectly

```
WITH RECURSIVE Emp_cte (emp_name, mgr_name) AS (
    SELECT employee, manager
    FROM Emp
    WHERE manager = 'Beck'

    UNION

    SELECT A.employee, A.manager
    FROM Emp A, Emp_cte B
    WHERE A.manager = B.employee)
SELECT emp_name, mgr_name
FROM Emp_cte;
```

4.2.2 Example 2: Airlines

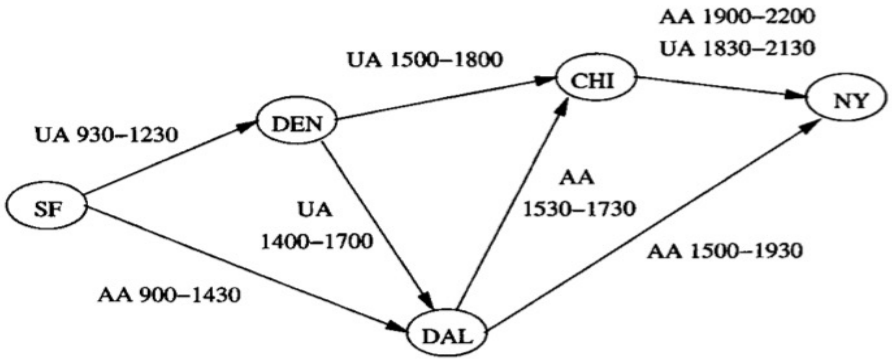

Figure 10.9: A map of some airline flights

airline	*from*	*to*	*departs*	*arrives*
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Recursion can often be understood with the study of paths in a graph. The figure above shows a graph representing some flights of two hypothetical airlines - *Uniqlo Airlines* (UA) and *Americana Airlines* (AA). The relations of the graph is as follows:

```
Flights(airline, from, to, departs, arrives)
```

For what pairs of cities (x,y) is it possible to get from city x to city y by taking one or more flights:

```
WITH RECURSIVE Reaches(from, to) AS (
        SELECT from, to
    FROM Flights

    UNION

    SELECT R.from, F.to
    FROM Flights F, Reaches R
    WHERE R.to = F.from)
SELECT *
FROM Reaches;
```

Reaches

from	to
SF	DEN
SF	DAL
SF	CHI
SF	NY
DEN	CHI
DEN	DAL
DEN	NY
DAL	CHI
DAL	NY
CHI	NY

5 Existential Set Operations

5.1 SOME and ANY

`SOME` and `ANY` are equivalent so we will just focus on `SOME`. `SOME` must match at least one row in the sub-query and must be preceded by comparison operators. So suppose we using greater than with some, `> SOME`, this means 'greater than at least one value'.

Find the names of all instructors whose salary is greater than at least one instructor in the Biology department:
WITHOUT SOME:

```
SELECT DISTINCT T.name
FROM Instructor as T, Instructor as S
WHERE T.salary > S.salary AND S.dept_name = 'Biology';
```

The alternative to the preceding query uses the clause `SOME` and it resembles closely our formulation of the query in English.

Find the names of all instructors whose salary is greater than at least one instructor in the Biology department:
WITH SOME:

```
SELECT name
FROM Instructor
WHERE salary > SOME (SELECT SALARY
                     FROM Instructor
                     WHERE dept_name = 'Biology');
```

The `> SOME` comparison in the `WHERE` clause of the outer `SELECT` is true if the salary value of the tuple is greater than at least one member of the set of all salary values for instructors

in Biology.

SQL allows `> SOME`, `<= SOME`, `>= SOME`, `= SOME`, and `<> SOME` comparisons. Note that `= SOME` is equivalent to `IN`.

5.2 ALL

`ALL` must match every row in the sub-query and must be proceeded by comparison operators. The construct `> ALL` corresponds to the phrases 'greater than all'

Find the names of all instructors that have a salary greater than that of any instructor in the Biology department

```
SELECT name
FROM Instructor
WHERE salary > ALL (SELECT SALARY
                    FROM Instructor
                    WHERE dept_name = 'Biology');
```

SQL allows `> ALL`, `<= ALL`, `>= ALL`, `= ALL`, and `<> ALL` comparisons. Note that `<> ALL` is equivalent to `NOT IN`.

5.3 EXISTS

SQL includes a feature for testing whether a sub-query has any tuples in its results. The `EXISTS` construct returns value *true* if the argument sub-query is nonempty.

Find all courses taught in both the Fall 2020 quarter and in the Winter 2021 quarter

```
SELECT course_id
FROM section as S
WHERE semester = 'Fall' AND year = 2020 AND
    EXISTS (SELECT *
            FROM section as T
            WHERE semester = 'Winter' AND year=2021 AND
                  S.course_id = T.course_id);
```

6 Exercise

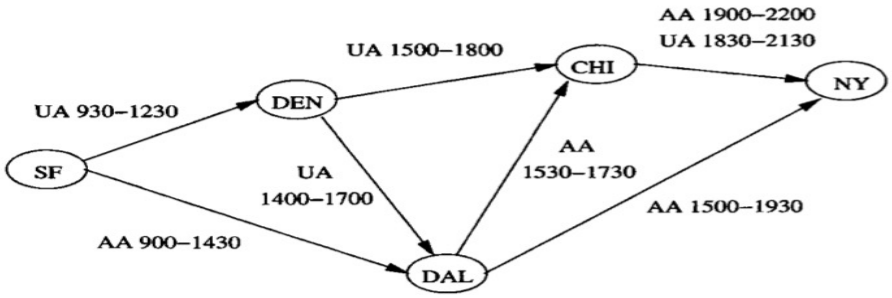

Figure 10.9: A map of some airline flights

airline	*from*	*to*	*departs*	*arrives*
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

The relation (From Example 2)

`Flights(airline,from,to,departs,arrives)`

has arrival- and departure-time information that we did not consider. Suppose we are interested not only in whether it is possible to reach one city from another, but whether the journey has reasonable connections. That is, when using more than one flight, each flight must arrive at least an hour before the next flight departs. You may assume that no journey takes place over more than one day, so it is not necessary to worry about arrival close to midnight followed by a departure early in the morning

Write the recursion in SQL.

```
WITH RECURSIVE Reaches(from, to) AS (
       SELECT from, to FROM Flights

       UNION

       SELECT R.from, F.to
       FROM Flights F, Reaches R
       WHERE R.to = T.from AND
              F.departs - R.arrives >= 100)
SELECT * FROM Reaches;
```