# SQL: Structured Query Language
# Views

## Database Views

**Database View.** A database view is a single table *that is derived* from other tables in the database, *and whose content changes together with the changes in the tables it is derived from.*

**Defining database views.** A database view $V$ is typically associated with a single query $Q_V$ over the database. When discussed theoretically, we assume that the query is rendered in *relational algebra*. When working with actual DBMS, the query is rendered in SQL in the view definition command.

**Why?** There are a number of reasons why database views are useful.

- Security. A view may yield a subset of a database that is made acessible to a specific person or a specific application that is otherwise not authorized to access an entire database.
- Convenience. Views can be used in subsequent SQL tables anywhere where a SQL table name expected. If a view is defined by a particularly tricky query, this may help avoid very complex SQL expressions in subsequent queries. They can effectively be viewed as convenient *temp tables.*
- Dynamic updates. One of the key features of a view, that makes it a preferred way of creating *temp tables* (to actually *creating* relational tables) is that a view $V$ views are innately tied to the tables referenced in its query $Q_V$. Whenever new tuples are added to any of those tables, if any new tuples can be placed into $V$ they automatically are.

## Views in SQL

SQL syntax for views is straightforward.

**View creation.** SQL uses CREATE VIEW command for creating a new view:

```
CREATE VIEW <Name> AS
     <SQLQuery>;
```

Here <Name> is the name of the view, and <SQLQuery> is a SQL expression defining the view.

**Example.** Consider a database consisting of the following three tables:

```
Students(Id INT, Name String, Major String)
Courses(Id String, Name String)
Rosters(Student INT, Course String, Quarter String, Year INT, Grade String)
```

with primary keys Students.Id, Courses.Id and (Rosters.Student, Rosters.Course, Rosters.Quarter, Rosters.Year), and foreign keys Rosters.Student referencing Students.Id and Rosters.Course referencing Courses.Id.

Suppose we want to have a handy list of of all students who took CSC 365. The view definition command for this view is

```
CREATE VIEW csc365 AS
    SELECT s.Name, s.Major, r.Quarter, r.Year, r.Grade
    FROM Students s, Roster r
    WHERE r.Student = s.Id and r.course = 'CSC 365'
;
```

**Querying views.** After they are created, view name is added to the list of table names available in the database. SQL queries can use the name of the view where they otherwise can use the name of a relational table.

**Example.** After the CREATE VIEW command above, we can check the contents of the csc365 view using the name of the view as if it is a regular relational table[1]:

```
mysql> select *
    -> from csc365
    -> limit 5;
+------------------+--------------+---------+------+-------+
| Name             | Major        | quarter | year | grade |
+------------------+--------------+---------+------+-------+
| Rudolph Attia    | Mathematics  | Fall    | 2016 | B     |
| Golda Singer     | Software Eng | Fall    | 2016 | C     |
| Francina Dorie   | Mathematics  | Winter  | 2016 | B     |
| Carola Davis     | English      | Winter  | 2016 | D     |
| Kathaleen Copass | English      | Fall    | 2016 | A     |
+------------------+--------------+---------+------+-------+
5 rows in set (0.00 sec)
```

Note, that the view is now part of the list of tables:

[1]The MySQL interactions below use a small database created for the purpose of showing how views work.

```
mysql> show tables;
+--------------------+
| Tables_in_viewdemo |
+--------------------+
| Courses            |
| Rosters            |
| Students           |
| csc365             |
+--------------------+
4 rows in set (0.00 sec)
```

**Dynamic nature of views.** If we add more data to the underlying tables that satisfies the conditions found in the query $Q_V$ defining the view $V$, the new data appears as part of the view.

Consider the following view:

```
CREATE VIEW mimiSchedule AS
     SELECT Course, Quarter, Year, Grade
     FROM Rosters
     WHERE Student = (
                SELECT Id
                FROM Students
                WHERE Name = 'Mimi Pallazzo'
                )
 ;
```

This view shows the list of courses `Mimi Pallazzo` took. Consider the following sequence of SQL commands executed after the view has been created[2].

```
mysql> select * from mimiSchedule;
+----------+---------+------+-------+
| Course   | Quarter | Year | Grade |
+----------+---------+------+-------+
| CSC 365  | Winter  | 2016 | C     |
| CSC 453  | Winter  | 2016 | D     |
| MATH 206 | Fall    | 2016 | B     |
| MATH 248 | Winter  | 2016 | B     |
+----------+---------+------+-------+
4 rows in set (0.00 sec)

mysql> insert into Rosters VALUES(9,'CSC 453', 'Spring', 2016,'B');
Query OK, 1 row affected (0.00 sec)

mysql> select * from mimiSchedule;
+----------+---------+------+-------+
| Course   | Quarter | Year | Grade |
+----------+---------+------+-------+
| CSC 365  | Winter  | 2016 | C     |
| CSC 453  | Spring  | 2016 | B     |
| CSC 453  | Winter  | 2016 | D     |
| MATH 206 | Fall    | 2016 | B     |
| MATH 248 | Winter  | 2016 | B     |
+----------+---------+------+-------+
5 rows in set (0.00 sec)
```

We have inserted information about `Mimi Pallazzo` retaking `CSC 453` in Spring of 2016 into the `Rosters` table. As soon as this information appears in the `Rosters` table, the contents of the `mimiSchedule` table change, and the new entry appears there as well.

[2] In our test database, Mimi's student Id is 9.

**Deleting views.** After views are no longer necessary, they can be successfully deleted using the `DROP VIEW` command:

```
DROP VIEW <ViewName>;
```

where the `<ViewName>` is the name of the view to be dropped.

### View Modifications

In some specific circumstances, DBMS allow for direct view modifications. Views can be updated using standard DML `INSERT`, `UPDATE` and `DELETE` commands.

Key things to remember are:

- Changes to the contents of the view **yield** changes to the contents of the underlying tables.
- Because of the above, views can be modified **only when** it is possible to establish a direct **one-to-one correspondence** between a tuple modified in the view and an underlying tuple in the base table(s).
- DBMS restrict this even more. Direct views are essentially allowed when:
  - the `FROM` clause of the view definition query $Q_V$ contains only one table $R$.
  - there is no *duplicate elimination* in the query $Q_V$ , i.e., the query is `SELECT`, **not** `SELECT DISTINCT`.
  - There are no subqueries in the `WHERE` clause of $Q_V$ that involve $R$.
  - There are enough attributes in the `SELECT` clause of $Q_V$ to uniquely determine a new tuple from table $R$ (if the remaining attributes are filled with `NULL` values).

**Example.** As seen from above, this a very restrictive set of requirements.

Consider the following view:

```
CREATE VIEW cs AS
    SELECT *
    FROM Students
    WHERE Major = 'Computer Science'
;
```

After this view is created, consider the following set of commands:

```
mysql> select * from cs;
+----+------------------+------------------+
| Id | Name             | Major            |
+----+------------------+------------------+
| 24 | Matha Kimball    | Computer Science |
| 25 | Cinderella Bread | Computer Science |
| 32 | Jeanie Kleekamp  | Computer Science |
+----+------------------+------------------+
3 rows in set (0.01 sec)

mysql> INSERT INTO cs VALUES(42, 'Grant Huffman', 'Computer Science');
```

```
Query OK, 1 row affected (0.00 sec)

mysql> select * from cs;
+----+------------------+------------------+
| Id | Name             | Major            |
+----+------------------+------------------+
| 24 | Matha Kimball    | Computer Science |
| 25 | Cinderella Bread | Computer Science |
| 32 | Jeanie Kleekamp  | Computer Science |
| 42 | Grant Huffman    | Computer Science |
+----+------------------+------------------+
4 rows in set (0.00 sec)

mysql> select * from Students where major = 'Computer Science';
+----+------------------+------------------+
| Id | Name             | Major            |
+----+------------------+------------------+
| 24 | Matha Kimball    | Computer Science |
| 25 | Cinderella Bread | Computer Science |
| 32 | Jeanie Kleekamp  | Computer Science |
| 42 | Grant Huffman    | Computer Science |
+----+------------------+------------------+
4 rows in set (0.00 sec)
```

As seen here, inserting information about `Grant Huffman` into the `cs` view propagates the insertion of the tuple about Grant into the `Students` table. Similarly, we may correct spelling errors:

```
mysql> update cs
    -> set Name = 'Grant Hoffman' where Id = 42;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from cs;
+----+------------------+------------------+
| Id | Name             | Major            |
+----+------------------+------------------+
| 24 | Matha Kimball    | Computer Science |
| 25 | Cinderella Bread | Computer Science |
| 32 | Jeanie Kleekamp  | Computer Science |
| 42 | Grant Hoffman    | Computer Science |
+----+------------------+------------------+
4 rows in set (0.00 sec)

mysql> select * from Students where major='Computer Science';
+----+------------------+------------------+
| Id | Name             | Major            |
+----+------------------+------------------+
| 24 | Matha Kimball    | Computer Science |
| 25 | Cinderella Bread | Computer Science |
| 32 | Jeanie Kleekamp  | Computer Science |
| 42 | Grant Hoffman    | Computer Science |
+----+------------------+------------------+
4 rows in set (0.00 sec)
```

## View Implementation in DBMS

Views can be implemented in a number of different ways.

**On-the-fly views.** A view exists in the DBMS as a query. Each time a view $V$ is referenced in a SQL query, its defining query $Q_V$ is executed, and the results are used in the query. The prepared results are **not stored** after the SQL query is completed.

**Materialized views.** A materialized view is a view that exists, *as a list of tuples* in the DBMS memory (either on disk or in main memory). Materialized views have advantages and disadvantages.

Advantages:

- `SELECT` queries are much faster, as the current state of the view does not need to be computed on the fly.
- The information represented by the view is made persistent and will not disappear if there is a crash.

Disadvantages:

- *Updates* are expensive. There are two strategies.

  The first strategy is to materialize the view anew each time an update is made to the underlying tables. This strategy is feasible only when there are relatively few updates to the underlying tables take place.

  The second strategy is called *incremental view update.* Under this strategy, each time an update is made to the base tables, a decision procedure is run to determine whether any new tuples need to be added to the view, or whether any other modifications to the view need to be performed. These modifications are then performed as individual insertions, deletions and updates. This strategy works fine for simple views, but in complex views that depend on many tables incremental view update becomes computationally very expensive.

**MySQL.** MySQL does not offer *materialized view updates.* All MySQL views are maintained on the *on-the-fly* basis.