09 SQL Views

MySQL Views

Views in MySQL are virtual tables that don't physically exist in the database but are defined by a query. They provide a way to:

- Simplify complex queries
- Restrict access to sensitive data
- Present data in a more user-friendly format
- Encapsulate business logic

Basic View Creation

The basic syntax for creating a view is:

```SQL
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example 1: Simple Programmer Information View

```SQL
CREATE VIEW programmer_info AS
SELECT Programmer_Name, Primary_Language, Secondary_Language, Salary
FROM programmers;
```

To use this view:

```SQL
SELECT * FROM programmer_info;
```

This returns a simplified view of programmer information without exposing personal details like date of birth.

Views with Calculated Fields

Views can include calculations to transform data into more meaningful formats.

Example 2: Programmer Experience View

```SQL
CREATE VIEW programmer_experience AS
SELECT
    Programmer_Name,
    Primary_Language,
    DOJ,
    TIMESTAMPDIFF(YEAR, DOJ, CURDATE()) AS Years_Experience,
    Salary
FROM programmers;
```

Query the view:

```SQL
SELECT * FROM programmer_experience
ORDER BY Years_Experience DESC;
```

This view helps HR quickly see which programmers have the most experience without calculating it each time.

Joining Tables in Views

Views are excellent for simplifying complex joins.

Example 3: Programmer Education Profile

```SQL
CREATE VIEW programmer_education AS
SELECT
    p.Programmer_Name,
    p.Primary_Language,
    s.Institute,
    s.Course,
    s.Course_Fee
FROM programmers p
JOIN studies s ON p.Programmer_Name = s.Programmer_Name;
```

Usage:

```SQL
SELECT * FROM programmer_education
WHERE Institute = 'MIT';
```

This view connects programmer information with their educational background.

Filtered Views

Views can contain WHERE clauses to filter data.

Example 4: Senior Programmers View

```sql
CREATE VIEW senior_programmers AS
SELECT
    Programmer_Name,
    Primary_Language,
    Secondary_Language,
    TIMESTAMPDIFF(YEAR, DOJ, CURDATE()) AS Years_Experience,
    Salary
FROM programmers
WHERE TIMESTAMPDIFF(YEAR, DOJ, CURDATE()) ≥ 20;
```

Usage:

```sql
SELECT * FROM senior_programmers
ORDER BY Salary DESC;
```

This view shows only programmers with 20+ years of experience.

Views with Aggregation

Views can include aggregate functions for reporting.

Example 5: Software Success Metrics

```sql
CREATE VIEW software_metrics AS
SELECT
    Programmer_Name,
    COUNT(*) AS Number_of_Software,
    SUM(Sold) AS Total_Units_Sold,
    SUM(Software_Cost * Sold) AS Total_Revenue,
    SUM(Software_Cost * Sold - Development_Cost) AS Total_Profit
FROM software
GROUP BY Programmer_Name;
```

Usage:

```sql
SELECT * FROM software_metrics
ORDER BY Total_Profit DESC;
```

This view helps managers quickly see which programmers are creating the most profitable software.

Updatable Views

Under certain conditions, you can update data through views:

- No aggregate functions (SUM, COUNT, etc.)
- No DISTINCT, GROUP BY, or HAVING clauses
- No subqueries in the SELECT or WHERE clause
- Based on a single table

Example 6: Updatable Programmer Languages View

```SQL
CREATE VIEW programmer_languages AS
SELECT
    Programmer_Name,
    Primary_Language,
    Secondary_Language
FROM programmers;
```

Update through the view:

```SQL
UPDATE programmer_languages
SET Primary_Language = 'TypeScript'
WHERE Programmer_Name = 'Peter Parker' AND Primary_Language = 'JavaScript';
```

This would update the actual data in the programmers table.

Views with CHECK OPTION

The CHECK OPTION prevents updates that would make rows disappear from the view.

Example 7: Python Programmers View with CHECK OPTION

```SQL
CREATE VIEW python_programmers AS
SELECT *
FROM programmers
WHERE Primary_Language = 'Python' OR Secondary_Language = 'Python'
WITH CHECK OPTION;
```

If you try:

```sql
UPDATE python_programmers
SET Primary_Language = 'Java', Secondary_Language = 'Ruby'
WHERE Programmer_Name = 'Tony Stark';
```

MySQL will reject this update because it would remove Tony Stark from the view.

OR REPLACE Option

You can modify an existing view using OR REPLACE:

```sql
CREATE OR REPLACE VIEW programmer_info AS
SELECT
    Programmer_Name,
    Primary_Language,
    Secondary_Language,
    Salary,
    YEAR(DOJ) AS Year_Joined
FROM programmers;
```

This adds Year_Joined to our previous view.

Altering Views

You can also use ALTER VIEW:

```sql
ALTER VIEW programmer_info AS
SELECT
    Programmer_Name,
    Primary_Language,
    Secondary_Language,
    Salary,
    YEAR(DOJ) AS Year_Joined,
    CASE
        WHEN Salary > 15000 THEN 'High'
        WHEN Salary > 13000 THEN 'Medium'
        ELSE 'Entry Level'
    END AS Salary_Tier
FROM programmers;
```

Dropping Views

To remove a view:

```sql
DROP VIEW IF EXISTS programmer_info;
```

Complex Example: Development Department Dashboard

Let's create a comprehensive view that management could use:

```sql
CREATE VIEW department_dashboard AS
SELECT
    p.Programmer_Name,
    p.Primary_Language,
    TIMESTAMPDIFF(YEAR, p.DOJ, CURDATE()) AS Experience_Years,
    p.Salary,
    COUNT(sw.Software_Name) AS Software_Developed,
    COALESCE(SUM(sw.Sold), 0) AS Total_Units_Sold,
    COALESCE(SUM(sw.Software_Cost * sw.Sold), 0) AS Total_Revenue,
    COALESCE(SUM(sw.Software_Cost * sw.Sold - sw.Development_Cost), 0) AS
Net_Profit,
    s.Institute,
    s.Course
FROM programmers p
LEFT JOIN software sw ON p.Programmer_Name = sw.Programmer_Name
LEFT JOIN studies s ON p.Programmer_Name = s.Programmer_Name
GROUP BY p.Programmer_Name, p.Primary_Language, p.Salary, p.DOJ, s.Institute,
s.Course;
```

This comprehensive view joins all three tables and calculates revenue and profit metrics.

Using SHOW CREATE VIEW

To see how a view is defined:

```sql
SHOW CREATE VIEW department_dashboard;
```

View Information in Information Schema

To get information about all views in your database:

```sql
SELECT * FROM information_schema.views
WHERE table_schema = 'devs';
```

Real-World Use Cases

1. Data Security: Limiting Salary Information

```sql
CREATE VIEW hr_programmer_view AS
SELECT
    Programmer_Name,
    GENDER,
    Primary_Language,
    Secondary_Language,
    DOJ
FROM programmers;
```

This view could be given to team leads who need programmer information but shouldn't see salary data.

2. Simplifying Complex Reports: Software ROI Analysis

```sql
CREATE VIEW software_roi AS
SELECT
    sw.Programmer_Name,
    sw.Software_Name,
    sw.Software_Cost,
    sw.Development_Cost,
    sw.Sold,
    (sw.Software_Cost * sw.Sold) AS Revenue,
    (sw.Software_Cost * sw.Sold - sw.Development_Cost) AS Profit,
    CASE
        WHEN sw.Development_Cost > 0
        THEN ROUND(((sw.Software_Cost * sw.Sold - sw.Development_Cost) /
sw.Development_Cost) * 100, 2)
        ELSE 0
    END AS ROI_Percentage
FROM software sw;
```

Usage:

```sql
SELECT * FROM software_roi
WHERE ROI_Percentage > 100
ORDER BY ROI_Percentage DESC;
```

This identifies software with ROI greater than 100%.

3. Metadata View: Languages and Their Usage

```sql
CREATE VIEW language_usage AS
SELECT
    'Primary' AS Language_Type,
    Primary_Language AS Language,
    COUNT(*) AS Number_Of_Programmers
FROM programmers
GROUP BY Primary_Language
UNION ALL
SELECT
    'Secondary' AS Language_Type,
    Secondary_Language AS Language,
    COUNT(*) AS Number_Of_Programmers
FROM programmers
GROUP BY Secondary_Language;
```

This shows which programming languages are most used across the department.

Best Practices for Using Views

1. **Use descriptive names**: Choose view names that clearly convey their purpose
2. **Document your views**: Add comments when creating complex views
3. **Be careful with performance**: Views with multiple joins or subqueries can be slow
4. **Consider indexing**: Base tables should have appropriate indexes
5. **Limit nesting**: Avoid views that reference other views too deeply
6. **Use WITH CHECK OPTION**: When creating updatable views to maintain data integrity

Summary

MySQL views provide a powerful way to:

- Simplify complex queries
- Implement data security
- Create consistent interfaces for applications
- Encapsulate business logic
- Make reporting easier

By using views effectively, you can improve database usability, security, and maintainability in your real-world applications.