Databases

Database Objects

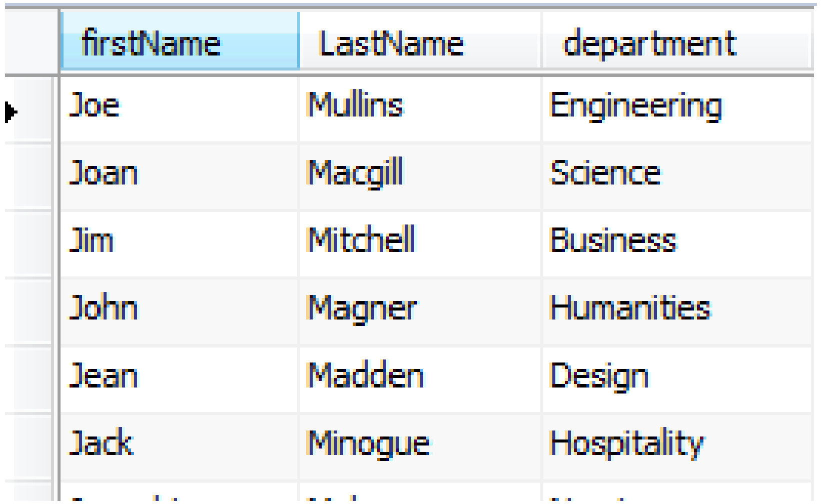
Introduction to views

- a **view** is a virtual table that is created by querying data from one or more tables
- A view consists of a SELECT statement that’s stored as an object in the database
- Use the CREATE VIEW statement

```
CREATE VIEW  names_department AS
SELECT firstName, LastName, department
FROM details;

    select * from names_department;
```

viewed table

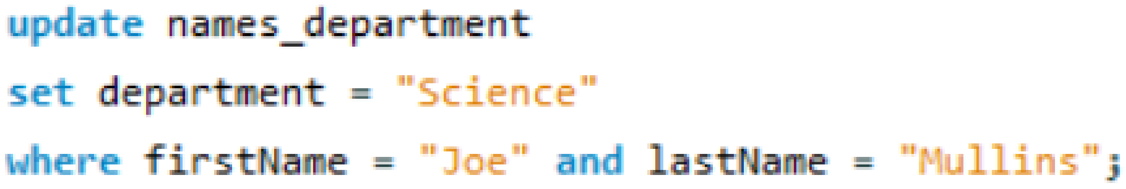
- Although a view behaves like a virtual table, it doesn't store any data. Instead, a view always refers back to its base tables.



firstName	LastName	department
Joe	Mullins	Engineering
Joan	Macgill	Science
Jim	Mitchell	Business
John	Magner	Humanities
Jean	Madden	Design
Jack	Minogue	Hospitality

Update example

- When you create a view, you can refer to the view anywhere you would normally use a table in a SELECT, INSERT, UPDATE or DELETE statement



```
update names_department
set department = "Science"
where firstName = "Joe" and lastName = "Mullins";
```

- This will update Joe Mullins to Science department in the details table.

- To drop a view :
  - DROP VIEW view_name;

Benefits of using a view

1. Design Independence
2. Data Security
3. Simplified queries
4. Updatability

Design Independence

- Views can limit the exposure of tables to external users and applications.

- As a result, if the design of the table changes, you can modify the view as necessary so users who query the view don’t need to be aware of the change, and applications that use the view don’t need to be modified.

Data Security

- Views can restrict access to data in a table by using the SELECT clause to include only selected columns of a table or by using the WHERE clause to include only selected rows in a table.

Simplified Queries

- Views can be used to hide the complexity of retrieval operations. Then the data can be retrieved using simple SELECT statements that specify a view in the FROM clause.

- You can also expand on the view with Where clauses etc.

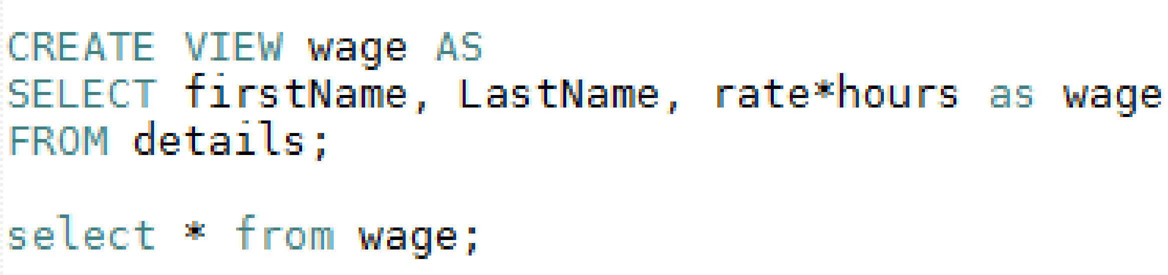
Updatability

- With certain restrictions, views can be used to update, insert, and delete data from a base table

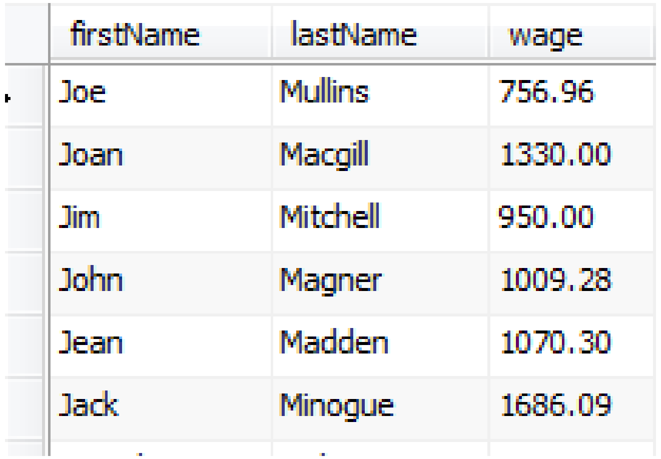
Working with views

- You can create a view by joining two tables

- If a view contains calculated columns, you will want to name that column



```
CREATE VIEW wage AS
SELECT firstName, LastName, rate*hours as wage
FROM details;

select * from wage;
```



firstName	lastName	wage
Joe	Mullins	756.96
Joan	Macgill	1330.00
Jim	Mitchell	950.00
John	Magner	1009.28
Jean	Madden	1070.30
Jack	Minogue	1686.09

Creating an Updatable view

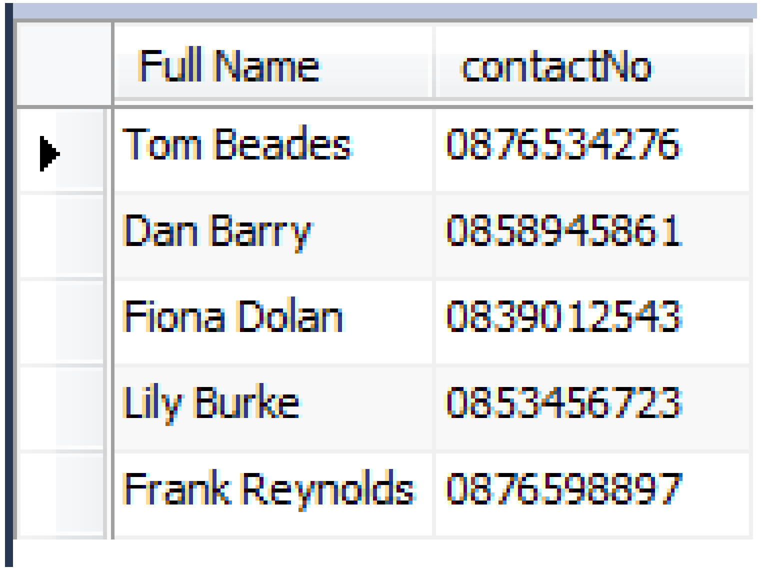
- Once you create a view, you can refer to it in a SELECT statement.
- You can also refer to it in INSERT, UPDATE and DELETE statements, but to do this the view must be updatable.
- If a view isn't updatable, it's called a **read-only view**
- The requirements for coding updatable views are more restrictive than for coding read-only views. That's because MySQL must be able to unambiguously determine which base tables and columns are affected

Requirements for creating an updateable view

- The select list can’t include a DISTINCT clause
- The select list can’t include aggregate functions
- The SELECT statement can’t include a GROUP BY or HAVING clause
- The view can’t include the UNION operator

Class Exercise

- In theDocs database create the following view (hint: look up the concat function)



	Full Name	contactNo
▶	Tom Beades	0876534276
	Dan Barry	0858945861
	Fiona Dolan	0839012543
	Lily Burke	0853456723
	Frank Reynolds	0876598897

Stored Programs

- Stored Programs can include procedural code to control the flow of execution
- 4 types of Stored Program:
    1. Stored Procedure
    2. Stored Function
    3. Trigger
    4. Event

Stored Procedure

- A Stored Procedure is a database object that contains a block of procedural SQL code. You can use SPs to execute an INSERT, UPDATE , or DELETE statement

Stored Procedure

The Syntax of the CREATE PROCEDURE Statement

```
CREATE PROCEDURE procedure_name
(
   parameter_name_1     data_type,
   parameter_name_2   data_type
)
Begin
           execution code
End
```

Delimiter

The delimiter marks the end of one SQL command and the beginning of another.   “;”

```
CREATE PROCEDURE procedure_name
(
   parameter_name_1    data_type,
   parameter_name_2   data_type
)
Begin
           SQL Statement 1;
           SQL Statement 2;
           SQL Statement 3;
End
```

SP that updates a table

Changes default delimiter of the semicolon (;) to //
This is necessary as the ;semicolan is used within the CREATE PROCEDURE statement and it allows you to use the // to identify the end of the CREATE PROCEDURE statement

```
DELIMITER //

CREATE PROCEDURE update_invoices_credit_total
(
 in invoice_id_param      INT,
 in credit_total_param    DECIMAL(9,2)
)
BEGIN
        UPDATE invoices
        SET credit_total = credit_total_param
        WHERE invoice_id = invoice_id_param;
END //
Delimiter ;

CALL update_invoices_credit_total(56, 300);
```

Parameters used to pass values to the SP from a calling program

<- marks the end of the procedure

<- restores delimiter to semi-colon

Input & Output parameters

- Input parameters accept values that are passed from the calling program.
- These values cannot be changed by the body of the SP. By default, parameters are identified as input parameters. As a result, the IN keyword is optional for identifying input parameters.

- Output parameters store values that are passed back to the calling program. These values must be set by the body of the SP. To identify an output parameter, you must code the OUT keyword.

- CALL update_invoices_credit_total(56, 300);

Exercise

- Write a SP named “update_details_rate” that will increase the rate for an employee by a given amount

- call update_details_rate (6, 10.10);

- Id and rate increase are passed in as parameters

Stored Function

- The code for creating a Stored Function works similarly to the code for creating a Stored Procedure.

- However, there are 2 primary differences
  1. A MySQL function always returns a single value
  2. A function can’t make changes to the database such as executing an INSERT, UPDATE or DELETE statement.

Stored Functions

- To identify the data type that’s returned by a function, you use the RETURNS keyword in the declaration for the function. Then, in the body of the function, you use the RETURN keyword to specify the value that’s returned

- A function can accept input parameters that work like the input parameters for a SP
- To call a SF, you can use it in any expression just like a built-in function.

A function that calculates salary

```
Drop  Function if exists calculate_salary;
DELIMITER //
CREATE Function calculate_salary
(
  id_param        INT
)
RETURNS DECIMAL(9,2)
BEGIN
  DECLARE salary_var DECIMAL(9,2);
                select sum(rate*hours)
                into salary_var
                from details
                where id = id_param;
 RETURN salary_var;
END//
select calculate_salary(3);
```

Run this example

→ Calling the Function

DROP Function

- DROP FUNCTION IF EXISTS calculate_salary;

Triggers

- A trigger is a named block of code that executes in response to an insert update or delete statement
- You can fire a trigger **before** or **after** an insert, update or delete statement is executed on a table.
- You must specify a FOR EACH ROW clause. This creates a *row-level trigger* that fires once for each row that's modified.
- MySQL only supports row-level triggers

BEFORE TRIGGER

```
DELIMITER //
CREATE TRIGGER details_before_update

BEFORE UPDATE ON details
  FOR EACH ROW
BEGIN
  SET NEW.department = UPPER(NEW.department);
END//
```

Run this example

- An UPDATE statement that fires the trigger

```
update details
set department = “Science"
where id = 5;
```

AFTER TRIGGER

- You can use an AFTER trigger to insert rows into an audit table
- Example
- Create a table that stores information about actions that occurred on the orders table

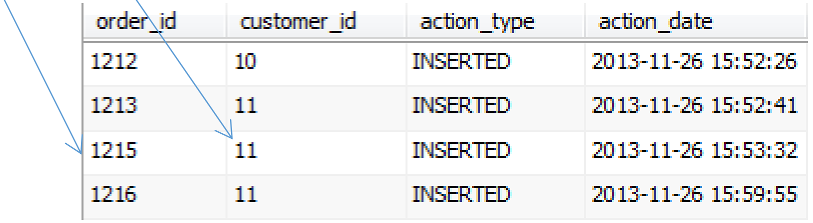
Create an Audit table

```
#CREATE TABLE

use om;
DROP TABLE IF EXISTS orders_audit;
CREATE TABLE orders_audit
 (
    order_id        INT     NOT NULL,
    customer_id     INT     NOT NULL,
    action_type     VARCHAR(50),
    action_date     DATETIME NOT NULL
 )
```

Trigger that inserts rows into the audit table (After Insert)

```
DROP TRIGGER IF EXISTS orders_after_insert;
DELIMITER //
CREATE  TRIGGER orders_after_insert
    AFTER INSERT on orders
    FOR EACH ROW
BEGIN
    INSERT INTO orders_audit VALUES
    (NEW.order_id, NEW.customer_id, "INSERTED", NOW());

END//
INSERT INTO orders VALUES (1215, 11, '2009-11-23', '2009-11-28');
SELECT * from orders_audit;
```

Use OM database to run this example

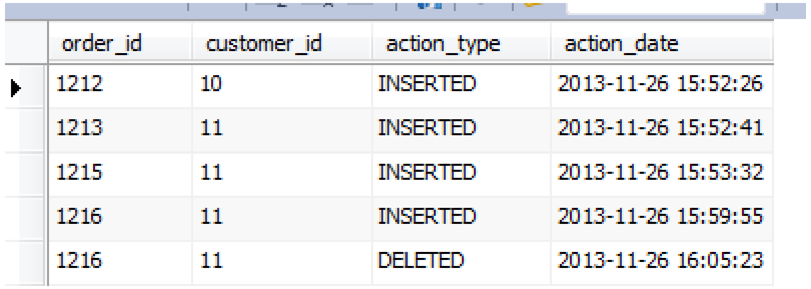


order_id	customer_id	action_type	action_date
1212	10	INSERTED	2013-11-26 15:52:26
1213	11	INSERTED	2013-11-26 15:52:41
1215	11	INSERTED	2013-11-26 15:53:32
1216	11	INSERTED	2013-11-26 15:59:55

Triggers that insert rows into the audit table (AFTER DELETE)

```
DROP TRIGGER IF EXISTS orders_after_delete;
DELIMITER //

CREATE  TRIGGER orders_after_delete
   AFTER DELETE on orders
   FOR EACH ROW
BEGIN
   INSERT INTO orders_audit VALUES
   (OLD.order_id, OLD.customer_id, "DELETED", NOW());
END//

DELETE FROM  orders WHERE order_id = 1216;

    SELECT * from orders_audit;
```

Use OM database to run this example



order_id	customer_id	action_type	action_date
1212	10	INSERTED	2013-11-26 15:52:26
1213	11	INSERTED	2013-11-26 15:52:41
1215	11	INSERTED	2013-11-26 15:53:32
1216	11	INSERTED	2013-11-26 15:59:55
1216	11	DELETED	2013-11-26 16:05:23

Show Triggers/ DROP TRIGGERS

- SHOW TRIGGERS;

or

- SHOW TRIGERS IN om;

- MySQL does not provide a way to alter TRIGGERS, you have use the DROP TRIGGER statement and CREATE a new TRIGGER

Triggers – some considerations

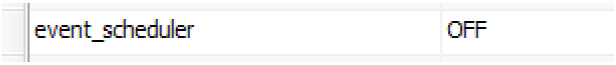
- Using triggers can slow down processing if there are a lot of data inserts, e.g. an over night job that populates a warehouse.

- Using Triggers can make maintenance of code more difficult as they are not directly visible.

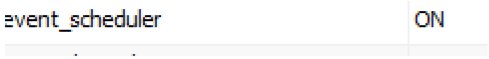
Events

- An event, or scheduled event, is a named block of code that executes, or fires according to the event scheduler.
- By default the event scheduler is OFF
- SHOW VARIABLES;

- SET GLOBAL event_scheduler = ON;



event_scheduler	OFF

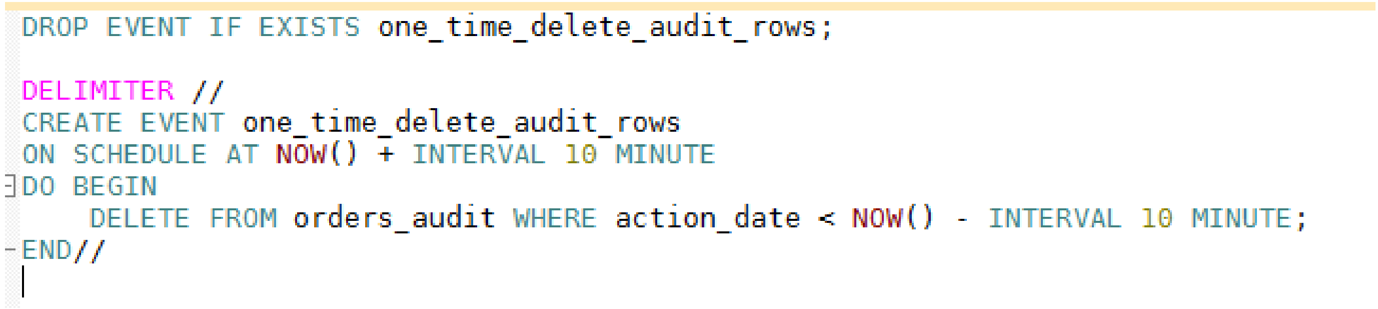


event_scheduler	ON

One-time Event

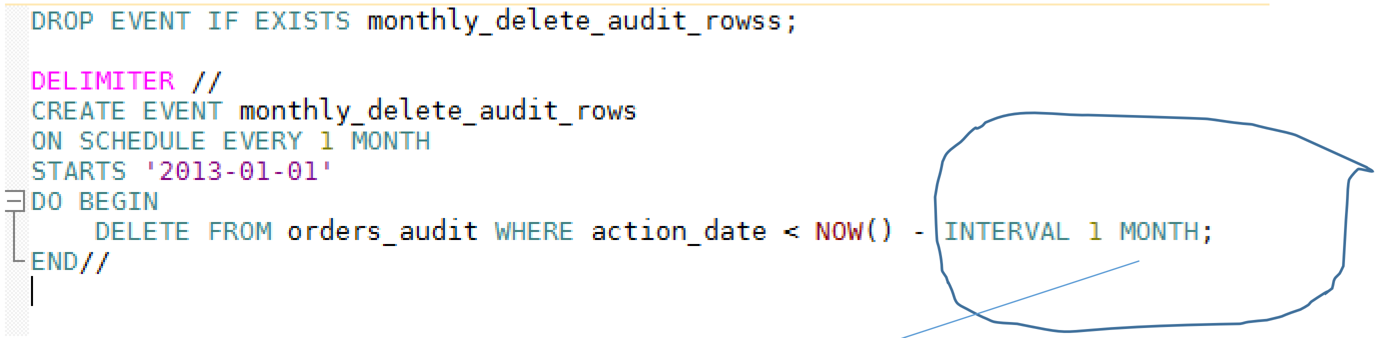
- An event can be a *one-time event* that occurs once or a *recurring event* that occurs regularly at a specified interval.

Use OM database to run this example



```
DROP EVENT IF EXISTS one_time_delete_audit_rows;

DELIMITER //
CREATE EVENT one_time_delete_audit_rows
ON SCHEDULE AT NOW() + INTERVAL 10 MINUTE
DO BEGIN
    DELETE FROM orders_audit WHERE action_date < NOW() - INTERVAL 10 MINUTE;
END//
```

Recurring Event



```
DROP EVENT IF EXISTS monthly_delete_audit_rowss;

DELIMITER //
CREATE EVENT monthly_delete_audit_rows
ON SCHEDULE EVERY 1 MONTH
STARTS '2013-01-01'
DO BEGIN
    DELETE FROM orders_audit WHERE action_date < NOW() - INTERVAL 1 MONTH;
END//
```

You can use MINUTE, HOUR, DAY, WEEK, MONTH or YEAR

View, Alter or Drop Events

- SHOW EVENTS
- SHOW EVENTS IN om;
- To enable or disable an event:
- ALTER EVENT monthly_delete_audit_rows DISABLE/ENABLE

- DROP EVENT IF EXISTS monthly_delete_audit_rows

Union

- The UNION operator is used to combine the result-set of two or more SELECT statements.
- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

Union

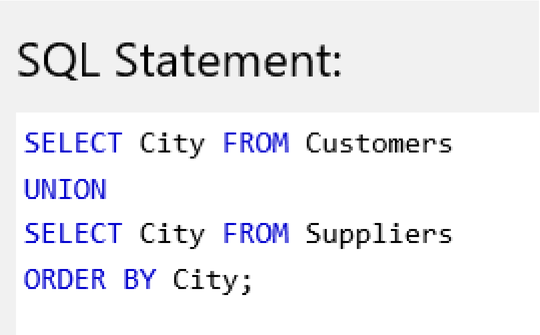
- SELECT *column_name(s)* FROM *table1*
  UNION
  SELECT *column_name(s)* FROM *table2*;

Union

- Try some examples from W3schools

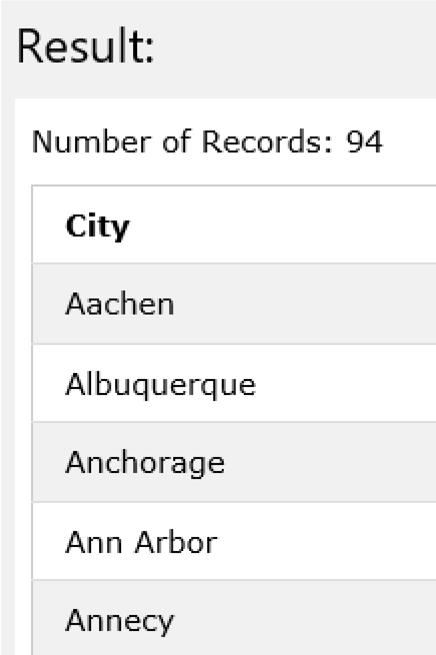
https://www.w3schools.com/sql/sql_union.asp

W3schools example



SQL Statement:

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```



Result:

Number of Records: 94

City
Aachen
Albuquerque
Anchorage
Ann Arbor
Annecy