CHAPTER 8: DATABASE APPLICATION DEVELOPMENT

Modern Database Management

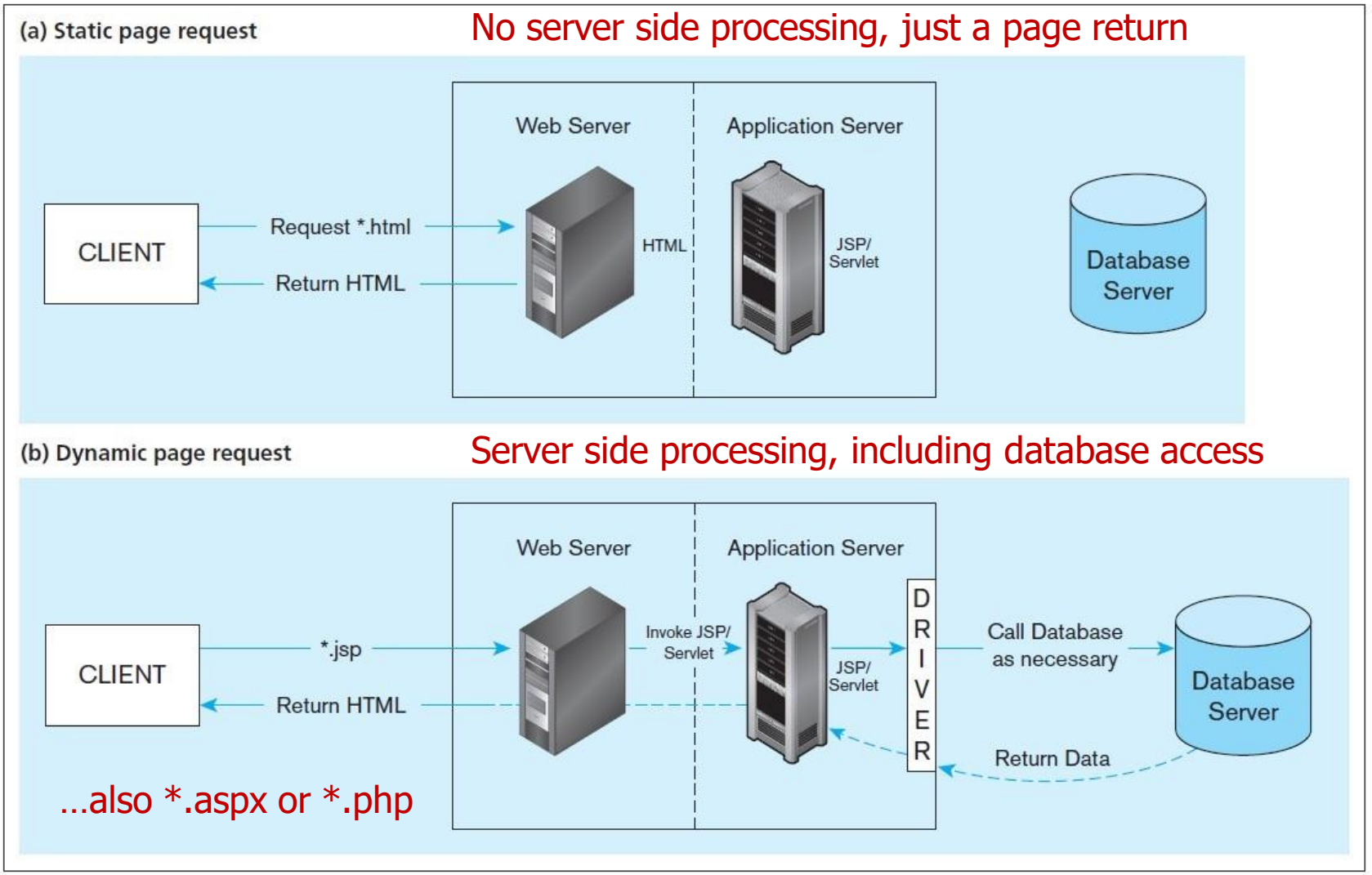
12th Edition

Jeff Hoffer, Ramesh Venkataraman, Heikki Topi

CLIENT/SERVER ARCHITECTURES

- Networked computing model
- Processes distributed between clients and servers
- Client–Workstation (PC, smartphone, tablet) that requests and uses a service
- Server– Powerful computer (PC/mini/mainframe) that provides a service
- For DBMS, server is a database server
- For the Internet, server is a web server

Information flow in a three tier architecture



CONSIDERATIONS IN 3-TIER APPLICATIONS

Stored procedures

- Code logic embedded in DBMS
- Improve performance, but proprietary

Transactions

- Involve many database updates
- Either all must succeed, or none should occur

Database connections

- Maintaining an open connection is resource-intensive
- Use of connection pooling

BENEFITS OF THREE-TIER ARCHITECTURES

- Scalability
- Technological flexibility
- Long-term cost reduction
- Better match of systems to business needs

ADVANTAGES AND DISADVANTAGES OF STORED PROCEDURES

- Advantages
- Performance improves for compiled SQL statements
- Reduced network traffic
- Improved security
- Improved data integrity
- Thinner clients

- Disadvantages
- Programming takes more time
- Proprietary, so algorithms are not portable

PL/SQL

- **TSQL** is a proprietary procedural language used by Microsoft SQL Server.
- **PL/SQL** is a proprietary procedural language with embedded SQL used by Oracle.
- **PL/pgSQL** is a procedural language used by PostgreSQL.
- MySQL supports stored routines (procedures and functions). A stored routine is a set of SQL statements that can be stored in the server.

Stored Procedures & Functions

- Stored Procedures & Functions - SQL statements stored inside the database
- A stored procedure consists of:
  - Procedure name
  - Parameter list
  - SQL statement(s)

```
DELIMITER $$

CREATE PROCEDURE sp_name()

BEGIN
  -- statements
END $$

DELIMITER ;
```

- Most RDBMS support stored procedures
- However, the syntax is a little different among them

- For MySQL, refer to https://dev.mysql.com/doc/refman/8.0/en/stored-routines.html

Advantages & Disadvantages of Stored Procedures

- Advantages:
  - Performance – stored procedures are fast (often cached)
  - Centralize business logic in the database
  - Reduce SQL statement traffic over network
- Disadvantages:
  - Maintenance considerations
  - Difficulty in debugging & troubleshooting

Altering Stored Procedures

- MySQL doesn't support altering stored procedures
- Use **`DROP PROCEDURE`** and **`CREATE PROCEDURE`** instead

Listing All Stored Procedures

- Check store procedures and their statuses using SHOW PROCEDURE:

```
SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE
condition];
```

- Can also list procedures using **information_schema** on MySQL:

```
SELECT
    routine_name
FROM
    information_schema.routines
WHERE
    routine_type = 'PROCEDURE'
        AND routine_schema = '<schema_name>';
```

Stored Procedure Example

- Creating stored procedures
- Executing stored procedures
- Deleting/dropping stored procedures
- Listing stored procedures

```
DELIMITER $$

CREATE PROCEDURE GetAllProducts()

BEGIN
  SELECT * FROM PRODUCT;
END $$

DELIMITER ;
```

```
CALL GetAllProducts();
```

```
DROP PROCEDURE IF EXISTS GetAllProducts;
```

```
SHOW PROCEDURE STATUS LIKE 'GetAll%'
```

Repeat Until Example

```
DELIMITER //
CREATE FUNCTION CalcIncome ( starting_value INT )
RETURNS INT
BEGIN
        DECLARE income INT;
        SET income = 0;
        label1: REPEAT
                SET income = income + starting_value;
                UNTIL income >= 4000
        END REPEAT label1;
        RETURN income;
END; //
DELIMITER ;
```

Using Parameters & Loops in Stored Procedures

- Stored procedures support input & output parameters

```
DELIMITER $$

CREATE PROCEDURE GetAllProducts(
    IN VENDOR_CODE INT(11)
)

BEGIN
  SELECT * FROM PRODUCT WHERE V_CODE = VENDOR_CODE;
END $$

DELIMITER ;
```

```
CALL GetAllProducts(21344);
```

P_CODE	P_DESCRIPT	P_INDATE	P_QOH	P_MIN	P_PRICE	P_DISCOUNT	V_CODE
13-Q2/P2	7.25-in. pwr. saw blade	2017-12-13	32	15	14.99	0.05	21344
14-Q1/L3	9.00-in. pwr. saw blade	2017-11-13	18	12	17.49	0.00	21344
54778-2T	Rat-tail file, 1/8-in. fine	2017-12-15	43	20	4.99	0.00	21344

- Also support `IF THEN-ELSEIF THEN-ELSE` statement & `WHILE` loops

Stored Function

- Unlike stored procedures, stored functions can be used in any SQL statement wherever an expression is used (without the CALL keyword)

```
DELIMITER $$

CREATE FUNCTION function_name(
    param1,
    param2,…
)
RETURNS datatype
BEGIN
  -- statements
END $$

DELIMITER ;
```

Stored Function Example

- Define a stored function to return the product count for a specified vendor code:

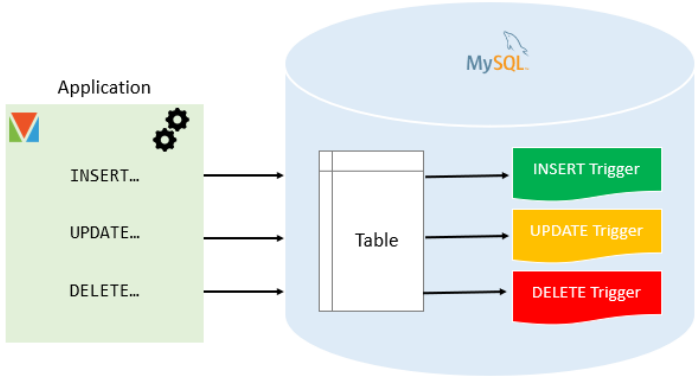
```
DELIMITER $$

CREATE FUNCTION GetVendorCode(
    PROD_CODE VARCHAR(11)
)
RETURNS INT
BEGIN
    DECLARE VENDOR_CODE INT DEFAULT 0;

    SELECT V_CODE INTO VENDOR_CODE FROM PRODUCT WHERE P_CODE = PROD_CODE;

    RETURN (VENDOR_CODE);

END $$

DELIMITER ;
```

```
SELECT GetVendorCode("54778-2T") AS VENDOR_CODE;
```

VENDOR_CODE
21344

Triggers

- Triggers are stored programs that are invoked automatically by RDBMS when given data manipulation event occurs (**INSERT**, **UPDATE** & **DELETE**)
- Standard SQL defines 2 types of triggers:
  - Row-level triggers
  - Statement-level triggers



- MySQL only supports row-level triggers
- For MySQL, refer to https://dev.mysql.com/doc/refman/8.0/en/stored-routines.html

Advantages & Disadvantages of Triggers

- Advantages:
  - Provide another way to check/enforce data integrity
  - Handle errors from the database layer
  - Run scheduled tasks
  - Help audit data changes in tables
- Disadvantages:
  - Difficult to troubleshoot and debug (automatically executed & invisible to client applications)
  - Increase overhead on database server

Creating Triggers

- **`CREATE TRIGGER`** statement creates a new trigger:
  - Before INSERT, UPDATE or DELETE
  - After INSERT, UPDATE or DELETE

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```

```
DELIMITER $$

CREATE TRIGGER trigger_name
    BEFORE INSERT
    ON table_name FOR EACH ROW
BEGIN
    -- statements
END$$

DELIMITER ;
```

Trigger Commands

- **SHOW TRIGGER** trigger_name command

```
SHOW TRIGGERS [{FROM | IN} database_name] [LIKE 'pattern' | WHERE search_condition];
```

- **DROP TRIGGER** - deletes a trigger without deleting the table

```
DROP TRIGGER [IF EXISTS] trigger_name;
```

The Special Values of a Trigger

- While in the body of a trigger, there are potentially two sets of column values available to you, with special syntax for denoting them.
  - **old**.<column name> will give you the value of the column before the DML statement executed.
  - **new**.<column name> will give you the value of that column **after** the DML statement executed.
- Insert triggers have no old values available, and delete triggers have no new values available for obvious reasons. Only update triggers have both the old and the new values available.
- Only triggers can access these values this way.

Trigger Example

- For example, a database trigger fires whenever salaries in the EMP table are updated

```
CREATE TRIGGER audit_sal
        AFTER UPDATE OF salary ON EMP
        FOR EACH ROW
        WHEN NEW.salary - OLD.salary < 1000
BEGIN
        INSERT INTO emp_audit VALUES
                (:NEW.enum, :NEW.salary)
END;
```

Trigger Example

- WHEN clause can be included as boolean expression which must evaluate to TRUE for the trigger to fire.
- The special variables NEW and OLD are available to refer to new and old tuples respectively.
- Statement-level triggers fire once per statement while row triggers fire once for each row affected by the SQL statement.

Viewing Your Triggers

- MySQL has a schema that has tables for all of the information that is needed to define and run the data in the database. This is meta data.
- you can use the “show triggers” command (this is not SQL) that will display a report of your triggers from the default schema.

```
mysql> show triggers;
```

PARTITION BY vs GROUP BY

- The **GROUP BY** clause allows users to apply aggregate functions and reduces the number of rows returned (by the query)

TherapistID	TotalSessionLength	AverageSessionLength	NumPatients
AS648	180	60.0000	3
BM273	105	35.0000	3
JR085	60	30.0000	2
SN852	135	67.5000	2
SW124	90	45.0000	2

- The **PARTITION BY** clause allows users to apply window functions over a subset of rows, but the number of rows returned aren't reduced.

TherapistID	SessionNum	PatientNum	LengthOfSession	TotalSessionLength	AverageSessionLength
AS648	28	1016	30	180	60.0000
AS648	31	1016	90	180	60.0000
AS648	38	1021	60	180	60.0000
SN852	33	1017	60	135	67.5000
SN852	36	1019	75	135	67.5000
BM273	30	1013	30	105	35.0000
BM273	34	1015	45	105	35.0000
BM273	37	1020	30	105	35.0000
SW124	29	1014	60	90	45.0000
SW124	35	1010	30	90	45.0000
JR085	27	1011	45	60	30.0000
JR085	32	1018	15	60	30.0000

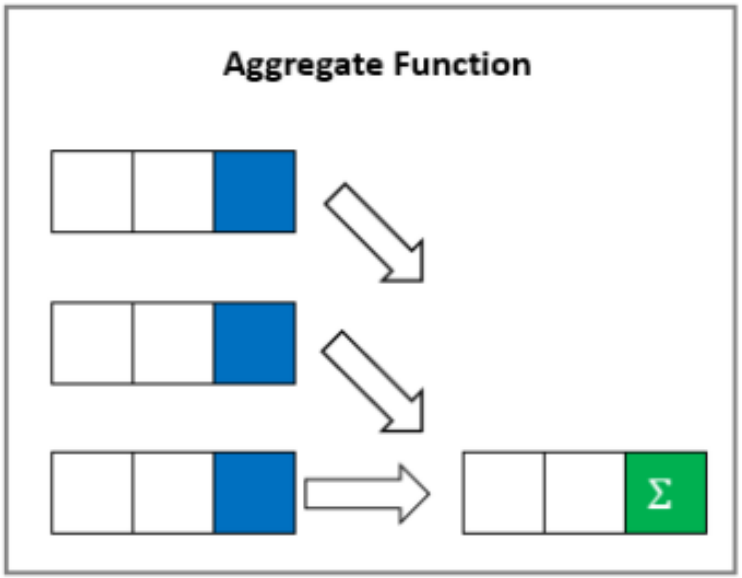
Window Functions

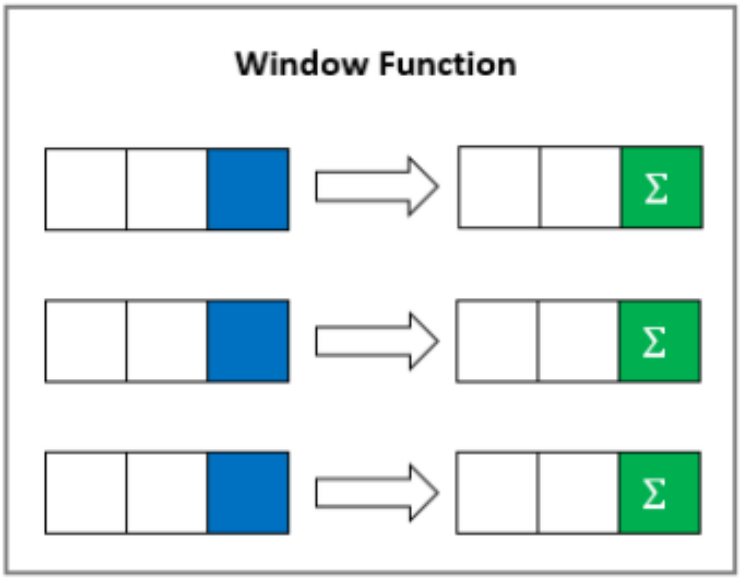
Introduction to SQL Window Functions

https://www.sqltutorial.org/sql-window-functions/

- Window functions operate on a set of rows and return a single value for each row from the underlying query. The term window describes the set of rows on which the function operates. The window is defined by the OVER clause.
  https://dev.mysql.com/doc/refman/8.0/en/window-functions.html

Difference between aggregate functions vs window functions





Different types of Window Functions

- Aggregate Functions: MAX(), MIN(), AVG(), SUM(), COUNT(), etc.
- Ranking Functions: ROW_NUMBER(), RANK(), DENSE_RANK(), NTILE(), PERCENT_RANK(), CUME_DIST() etc.
- Analytic Functions: LEAD(), LAG(), FIRST_VALUE(), LAST_VALUE(), etc.

Window Functions

- Window Function Usage Syntax:

```
WINDOW_FUNCTION_NAME ( EXPRESSION ) OVER ( PARTITION
  BY …
ORDER BY … FRAME_CLAUSE
)
```

- Aggregate window functions from before: **`AVG()`**, **`COUNT()`**, **`MAX()`**, **`MIN()`**, **`SUM()`**
- https://dev.mysql.com/doc/refman/8.0/en/window-functions.html

Example

Employee_Name	Department	Salary
John Roberts	Finance	2300
Peter Hudson	Marketing	1800
Sue Gibson	Finance	2000
Melinda Bishop	Marketing	1500
Nancy Hudson	IT	1950

```
SELECT Department,
       avg(salary) as average,
       max(salary) as top_salary
FROM employee
GROUP BY department
```

Department	average	top_salary
Marketing	1650	1800
Finance	2150	2300
IT	1950	1950

Example

- Windows functions allow to apply functions on a group of records while still leaving the records accessaible.

```
SELECT  employee_name,
department,
salary,
max(salary) OVER (PARTITION BY department) as top_salary
FROM        employee
```

Employee_Name	Department	salary	top_salary
John Roberts	Finance	2300	2300
Peter Hudson	Marketing	1800	1800
Sue Gibson	Finance	2000	2300
Melinda Bishop	Marketing	1500	1800
Nancy Hudson	IT	1950	1950

Additional Window Functions

Name	Description
CUME_DIST()	Cumulative distribution value
DENSE_RANK()	Rank of current row within its partition, without gaps
FIRST_VALUE()	Value of argument from first row of window frame
LAG()	Value of argument from row lagging current row within partition
LAST_VALUE()	Value of argument from last row of window frame
LEAD()	Value of argument from row leading current row within partition
NTH_VALUE()	Value of argument from N-th row of window frame
NTILE()	Bucket number of current row within its partition.
PERCENT_RANK()	Percentage rank value
RANK()	Rank of current row within its partition, with gaps
ROW_NUMBER()	Number of current row within its partition

ranking window functions

value window functions

ranking window functions

https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html

Example

year	month	make	model	type	quantity	revenue
2021	01	Ford	F100	PickUp	40	2500000
2021	01	Ford	Mustang	Car	9	1010000
2021	01	Renault	Fuego	Car	20	9000000
2021	02	Renault	Fuego	Car	50	23000000
2021	02	Ford	F100	PickUp	20	1200000
2021	02	Ford	Mustang	Car	10	1050000
2021	03	Renault	Megane	Car	50	20000000
2021	03	Renault	Koleos	Car	15	1004000
2021	03	Ford	Mustang	Car	20	2080000
2021	04	Renault	Megane	Car	50	20000000
2021	04	Renault	Koleos	Car	15	1004000
2021	04	Ford	Mustang	Car	25	2520000

LAG() is a window function that provides access to a row at a specified physical offset which comes before the current row. return any column of the previous month

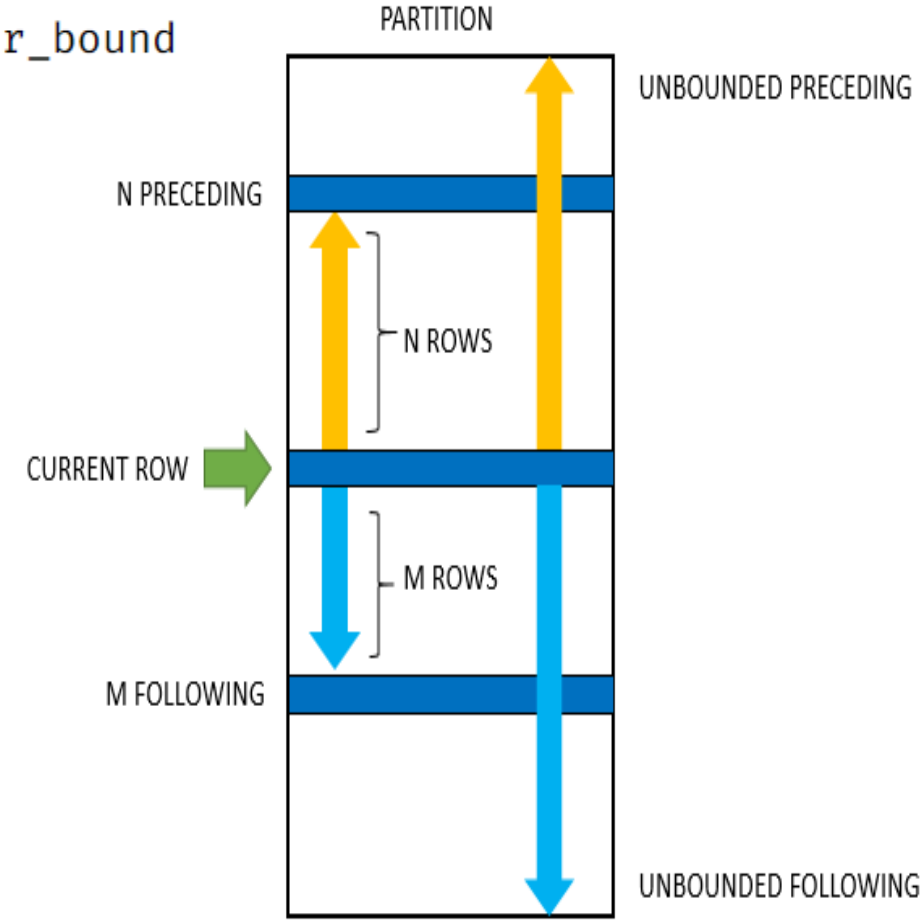
```
SELECT make,
       model,
       month,
       revenue AS current_month_revenue,
       LAG(revenue) OVER ( ORDER BY month) AS previous_month_revenue,
     revenue - LAG (revenue) OVER (ORDER BY month) AS delta_revenue
FROM monthly_car_sales
WHERE year = 2021
  AND model = 'Mustang'
```

make	Model	Month	Current Month Revenue	Previous Month Revenue	Delta Revenue
Ford	Mustang	1	1010000	NULL	NULL
Ford	Mustang	2	1050000	1010000	4000
Ford	Mustang	3	2080000	1050000	103000
Ford	Mustang	4	2520000	2080000	440000

Window Function Frames

```
ROWS BETWEEN lower_bound AND upper_bound
```

- Frame Start
  - N PRECEDING
  - UNBOUNDED PRECEDING
  - CURRENT ROW
- Frame End
  - CURRENT ROW
  - UNBOUNDED FOLLOWING
  - N FOLLOWING




```
MAX(revenue) OVER (PARTITION BY make, model ORDER BY month ROWS BETWEEN
UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS max_year_revenue
```

a revenue report by make for the current month and for each of the last three months

```
SELECT make,
       model,
       month,
     revenue AS model_revenue_current_month,
       SUM(revenue) OVER ( PARTITION BY make
                           ORDER BY month
                           RANGE BETWEEN 0 PRECEDING AND CURRENT ROW
                         ) AS make_current_month,
     SUM(revenue) OVER (PARTITION BY make
                           ORDER BY month
                           RANGE BETWEEN 1 PRECEDING AND CURRENT ROW
                        ) AS make_last_2_months,
       SUM(revenue) OVER (PARTITION BY make
                            ORDER BY month
                            RANGE BETWEEN 2 PRECEDING AND CURRENT ROW
                        ) AS make_last_3_months
FROM monthly_car_sales
WHERE year = 2021
```

Summary

- Procedural SQL can be used to create triggers, stored procedures, and functions.
- A stored procedure/function is a named collection of SQL statements.
- Window function together with partition by clause can be used to perform analytics on the data.

Exercise

- create table employee(emp_ID int, emp_NAME varchar(50), DEPT_NAME varchar(50), SALARY int);

1. Get max salary using window function
2. Find row number in each department using window function
3. Fetch the first 2 employees from each department to join the company.
4. Fetch the top 3 employees in each department earning the max salary.

Exercise

5. Fetch a query to display if the salary of an employee is higher, lower or equal to the previous employee.

6. fetch a query to display the salary of an previous and forward employees using lag() and lead().

- Think of more exercises such as third highest salary, percentage rank of salary, your own formula, etc.

Embedded SQL

- Embedded SQL is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL.
- Embedded SQL statements are SQL statements written inline with the program source code, of the host language.
- https://en.wikipedia.org/wiki/Embedded_SQL

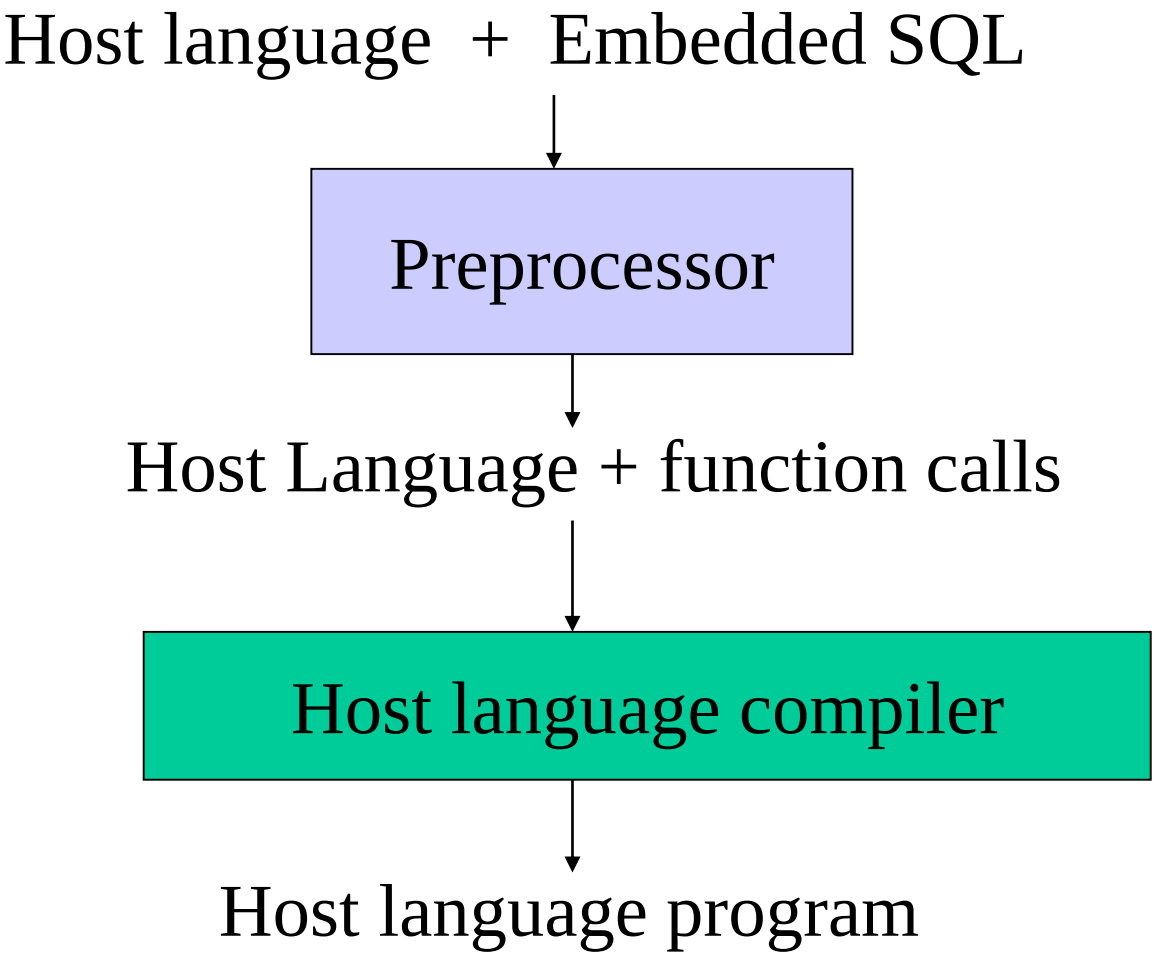
- It allows to execute any SQL statement from an application program.

Host Languages

- Any conventional language can be a *host language*, that is, a language in which SQL calls are embedded.
- The use of a host/SQL combination allows us to do anything computable, yet still get the very-high-level SQL interface to the database

1. *Embedded SQL* is a standard for combining SQL with seven languages.
2. CLI (*Call-Level Interface*) is a different approach to connecting C to an SQL database.
3. JDBC (*Java Database Connectivity*) is a way to connect Java with an SQL database.

Programs with SQL



Embedded SQL

- Use host language as front-end. Use SQL to access statements to access database.
- Preprocessor translates SQL statements to lib function calls.
- Values get passed through shared variables.
- Colons precede shared variables when they occur within the SQL statements.
- **EXEC SQL:** precedes every SQL statement in the host language.
- A special variable **SQLSTATE** provides error messages and status reports (e.g., 00000 says that the operation completed with noproblem).

SQL Syntax

- All SQL statements are embedded as
  EXEC SQL … ;
- Example)

```
int a;
…
EXEC SQL SELECT salary INTO :a
        FROM employee
        WHERE  ssn = ‘111111111’;
printf(“The salary is %d\n”, a);
```

Embedded SQL: Queries

- Updates can be done as shown in previous example
- SQL Queries (i.e. Select) ➔ Impedance mismatch

- Two cases:
  - Query returns a single row
  - Query returns multiple rows

Cursors

- **A cursor is a tuple-variable that ranges over all tuples in a table or the result of a query.**

- **Declare cursor:**

EXEC SQL DECLARE cursor_name CURSOR FOR
<Query>/<table>;

- **To use cursor, must issue command:**

EXEC SQL OPEN cursor_name;

- **When finished with cursor, close it:**

EXEC SQL CLOSE cursor_name;

JDBC

- JDBC is a Java API for executing SLQ statements.
- JDBC provides a standard way to integrate relational database into JAVA systems.
- JDBC does the three things:
  - Establish a connection with a database
  - Send SQL statements
  - Process the results
- Advantage: potable

Example

- **import java.sql.*;**
  class jdbctest {
  try {
  class.forName(“oracle.jdbc.driver.OracleDriver”);
  connection conn =
  driverManager.getConnection(“jdbc:oracle:thin:username
  /password@hostname:port:database”);
  statement stmt = conn.createStatement();
  // Query the employee names
  resultSet rset = stmt.executeQuery(“select ename from
  emp”);
  while (rset.next())
  { system.out.println(rset.getString(1)); }
  rset.close(); stmt.close(); conn.close();
  } }

Accessing the ResultSet

- An object of type ResultSet is like a cursor
- Method Next() advances the “cursor” to the next tuple.
- The first time Next() is applied, it gets the first tuple. Subsequently, it get the next tuple in order.
- If there are no more tuples, Next() returns False

Connecting to MySQL Using Connector/Python

- https://dev.mysql.com/doc/connector-python/en/connector-python-example-connecting.html

- import mysql.connector
- cnx = mysql.connector.connect(user='scott', password='*password*', host='127.0.0.1', database='employees’)
- cnx.close()

- import mysql.connector

```
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Prepared Statement

- Executing a statement takes time for database server
  - Parses SQL statement and looks for syntax errors
  - Determines optimal way to execute statement
    - Particularly for statements involving loading multiple tables
- Most database statements are similar in form
  - Example: Adding books to database
    - Thousands of statements executed
    - All statements of form:
    - **"SELECT * FROM books WHERE productCode = ____“**

Prepared Statement

```
check = connection.prepareStatement("SELECT * FROM books WHERE productCode = ?");
check.setString(1, productCode);
books = check.executeQuery();
if (books.next()) {
  RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/AddError.jsp");
  dispatcher.forward(request, response);
  return;
  }
}
catch (SQLException e) { System.out.println("BAD QUERY"); }
```

SQL Injection

- Username = ' or 1=1 --
  - The original statement looked like:
    'select * from users where username = '" +
    username + '" and password = '" + password + "'
    The result =
    select * from users where username = '' or 1=1 --'
    and password = ''

SQL Injection

- is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed.

DB Encryption

- DB Encryption can be divided into Data-in-transit and Data-at-rest
- Encryption is useful as a last layer of defense (defense in depth). Should never be used as an alternative solution
- Encryption should be used only when needed
- Key Management is Key

Auditing

- Oracle-supplied auditing using AUDIT command. Results go to AUD$
- Trigger-based DML auditing
- Either way, DBA must monitor auditing table.

When to audit

- When should we audit users ?
  - Basic set of auditing measures all the time
  - Capture user access, use of system privileges, changes to the db schema (DDL)

    If company handles sensitive data (financial market, military, etc.) OR

    If there are suspicious activities concerning the DB or a user, specific actions should be done.

How to optimize Queries?

- Less work → Faster query

- What is work for a query?
  - I/O — How many bytes did you read?
  - Shuffle — How many bytes did you pass to the next stage?
    - Grouping — How many bytes do you pass to each group?
  - Materialization — How many bytes did you write?
  - CPU work — User-defined functions (UDFs), functions

Don’t project unnecessary columns

- On how many columns are you operating?
- Excess columns incur wasted I/O and materialization
- Don’t SELECT * unless you need every field

Filter early and often using WHERE clauses

- On how many rows (or partitions) are you operating?
- Excess rows incur “waste” similar to excess columns

Do the biggest joins first (if you have to)

- Joins — In what order are you merging data?
  - Guideline — Biggest, Smallest, Decreasing Size Thereafter
  - Avoid self-join if you can, since it squares the number of rows processed
- Consider your JOIN order, try to filter the sets pre-JOIN

Low Cardinality GROUP BYs are faster

- Grouping — How much data are we grouping per-key for aggregation?
- Guideline — Low-cardinality keys/groups → fast, high-cardinality → slower
- However, higher key cardinality (more groups) leads to more shuffling; key skew can lead to increased tail latency
- Note: Get a count of your groups when trying to understand performance

Built-in functions are faster than JavaScript UDFs

- Functions — What work are we doing on the data?
- Guideline — Some operators are faster than others; all are faster than JavaScript® UDFs
- Example — Exact COUNT(DISTINCT) is very costly, but APPROX_COUNT_DISTINCT is very fast
- **Use SQL analytic functions -** The analytic functions can do multiple aggregations (e.g. rollup by cube) with a single pass through the tables, making them very fast for reporting SQL

ORDER on the outermost query

- Sorting—How many values do you need to sort?
- Filtering first reduces the number of values you need to sort
- Ordering first forces you to sort the world

Other Minor SQL Tuning

- **Avoid the LIKE predicate** = Always replace a "like" with an equality, when appropriate.
- **Use those aliases -** Always use table aliases when referencing columns
- **Use minus instead of EXISTS subqueries -** Some say that using the minus operator instead of NOT IN and NOT Exists will result in a faster execution plan.

- Sometimes you may have more than one subqueries in your main query. Try to minimize the number of subquery block in your query.

Minor SQL Tuning

- The sql query becomes faster if you use the actual columns names in SELECT statement instead of than '*'.
- **Example:** Write the query as
- SELECT id, first_name, last_name, age, subject FROM
  student_details;
- Instead of: SELECT * FROM student_details;

Minor SQL Tuning

- Use operator EXISTS, IN and table joins appropriately in your query.
  a) Usually IN has the slowest performance.

  Example: Write the query as
- Select * from product p
  where EXISTS (select * from order_items o where o.product_id = p.product_id)
- Instead of: Select * from product p where product_id IN (select product_id from order_items)

Minor SQL Tuning

- Be careful while using conditions in WHERE clause. **For Example:** Write the query as
- SELECT id, first_name, age FROM student_details WHERE age > 10;
- Instead of: SELECT id, first_name, age FROM student_details WHERE age != 10;

Databases Security – General Strategies

- Principle of Least Privilege!
- Stay up-to-date on patches
- Remove/disable unneeded default accounts
- Firewalling/Access Control
- Running Database processes under dedicated non-privileged account.
- Password Security
- Disable unneeded components

Principle of Least Privilege

- If X service doesn't need access to all tables in Y database… then don't give it access to all tables.
    - Example: A web application that reads a list of people from a database and lists them on a website. The database also contains sensitive information about those people.
- Do not give accounts privileges that aren't needed
  - Unneeded privileges to accounts allow more opportunity for privilege escalation attacks.

Stored Procedures, Triggers

- Stored Procedures and Triggers can lead to privilege escalation and compromise. Be sure to be thinking about security implications when allowing the creation of, and creating these.

- insert into employee values(101, 'Mohan', 'Admin', 4000);insert into employee values(102, 'Rajkumar', 'HR', 3000);
- insert into employee values(103, 'Akbar', 'IT', 4000);insert into employee values(104, 'Dorvin', 'Finance', 6500);
- insert into employee values(105, 'Rohit', 'HR', 3000);
- insert into employee values(106, 'Rajesh', 'Finance', 5000);
- insert into employee values(107, 'Preet', 'HR', 7000)
- ;insert into employee values(108, 'Maryam', 'Admin', 4000);
- insert into employee values(109, 'Sanjay', 'IT', 6500);
- insert into employee values(110, 'Vasudha', 'IT', 7000);
- insert into employee values(111, 'Melinda', 'IT', 8000);
- insert into employee values(112, 'Komal', 'IT', 10000);
- insert into employee values(113, 'Gautham', 'Admin', 2000)
- ;insert into employee values(114, 'Manisha', 'HR', 3000);
- insert into employee values(115, 'Chandni', 'IT', 4500);
- insert into employee values(116, 'Satya', 'Finance', 6500);
- insert into employee values(117, 'Adarsh', 'HR', 3500);
- insert into employee values(118, 'Tejaswi', 'Finance', 5500);

Exercise Answer key

1. select e.*,max(salary) over(partition by dept_name) as max_salary
   from employee e;
2. select e.*,row_number() over(partition by dept_name) as rn
   from employee e;
3. select * from (
   select e.*, row_number() over(partition by
   dept_name order by emp_id) as rn from
   employee e) x
   where x.rn < 3;
4. select * from (
   select e.*,
   rank() over(partition by dept_name order by salary desc) as rnk
   from employee e) x
   where x.rnk < 4;

Exercise Answer key

5. select e.*,

lag(salary) over(partition by dept_name order by emp_id) as prev_empl_sal,

case when e.salary > lag(salary) over(partition by dept_name order by emp_id) then 'Higher than previous employee'

when e.salary < lag(salary) over(partition by dept_name order by emp_id) then 'Lower than previous employee’

when e.salary = lag(salary) over(partition by dept_name order by emp_id) then 'Same than previous employee' end as sal_range

from employee e;

6. select e.*,
lag(salary) over(partition by dept_name order by emp_id) as prev_empl_sal,
lead(salary) over(partition by dept_name order by emp_id) as next_empl_sal
from employee e;