
MySQL

MySQL Overview

- MySQL, the most popular Open Source SQL database management system, is developed, distributed, and supported by MySQL AB
- The MySQL® software delivers a very fast, multi-threaded, multi-user, and robust SQL (Structured Query Language) database server.
- MySQL Server is intended for mission-critical, heavy-load production systems as well as for embedding into mass-deployed software.
- The MySQL software is Dual Licensed. Users can choose to use the MySQL software as an Open Source product under the terms of the GNU
- General Public License or can purchase a standard commercial license from MySQL AB.

MySQL Overview

SQL is language..MySQL i sRDBMS system

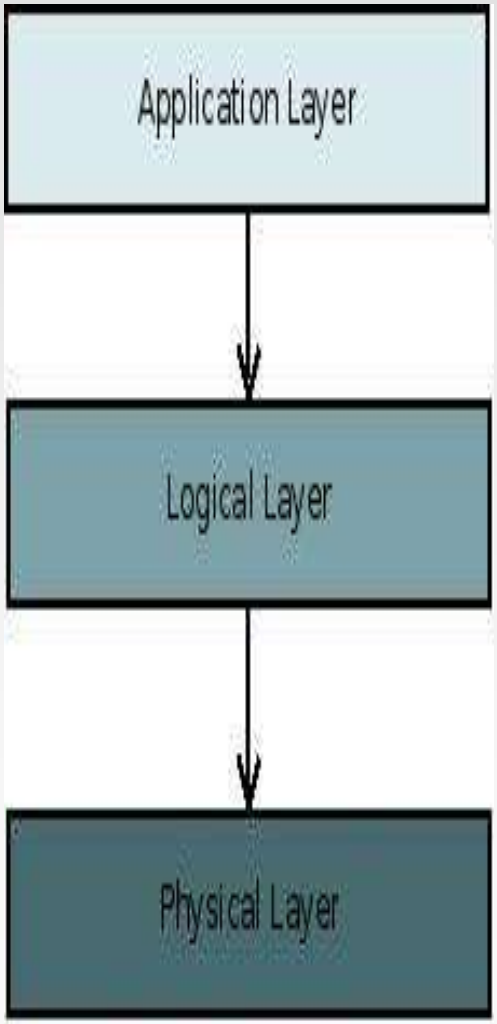
- MySQL is a relational database management system.
- MySQL software is Open Source.
- MySQL Server works in client/server or embedded systems

.

- A large amount of contributed MySQL software is available.

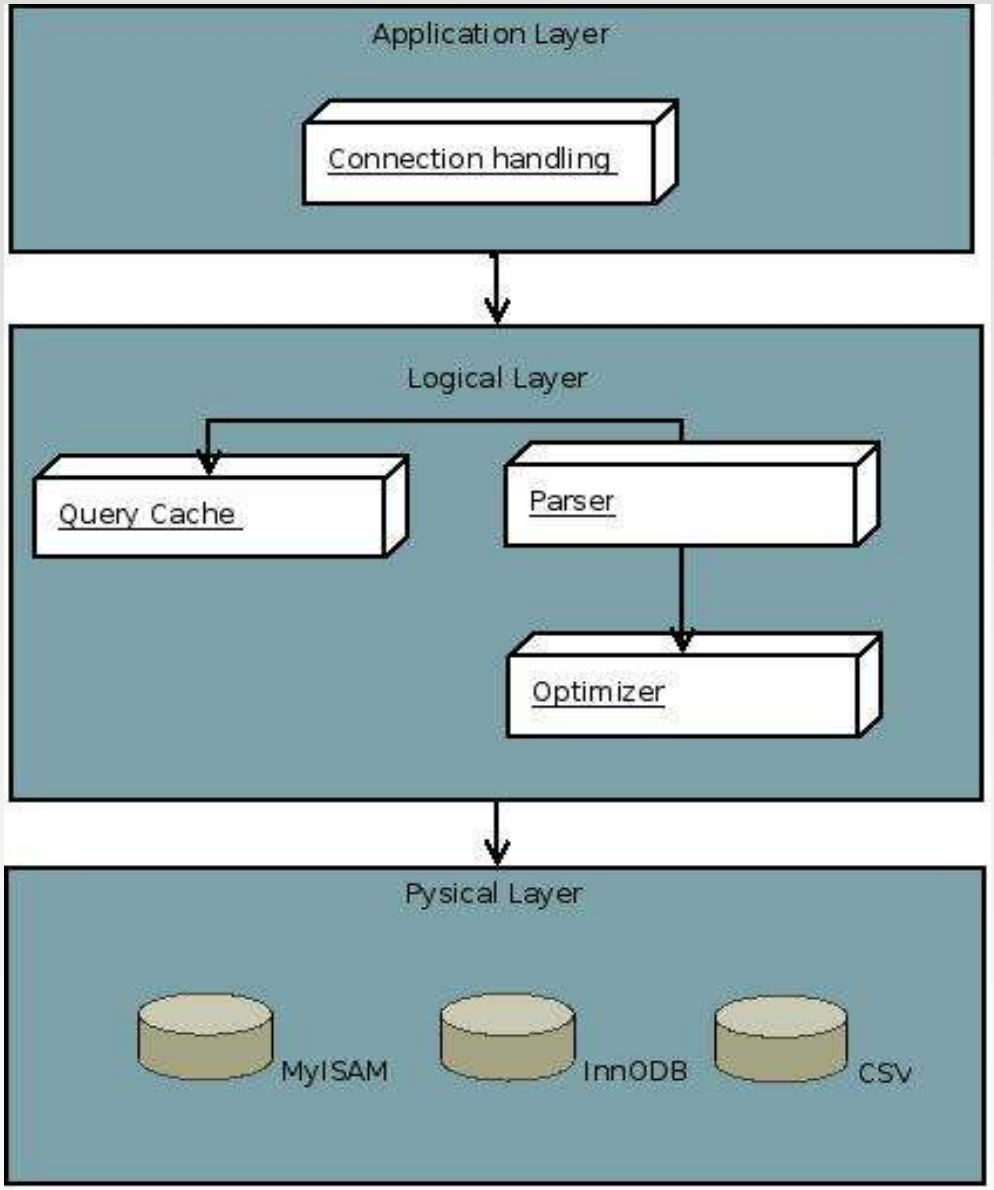
MySQL Architecture

Three layer model:



- The application layer contains common network services for connection handling, authentication and security. This layer is where different clients interact with MySQL
- The Logical Layer is where the MySQL intelligence resides, it includes functionality for query parsing, analysis, caching and all built-in functions (math, date...). This layer also provides functionality common across the storage engines.
- The Physical Layer is responsible for storing and retrieving all data stored in “MySQL”. Associated with this layer are the storage engines

MySQL Architecture



Each client connection gets its own thread within the server process.
When clients (applications) connect to the MySQL server, the server needs to authenticate them.

Before even parsing the query, though, the server consults the query cache, which only stores SELECT statements, along with their result sets.

The storage engine does affect how the server optimizes query.

MySQL is RDBMS

A Relational Data Base Management System (RDBMS) is a software that:

Enables you to implement a database with tables, columns and indexes.

Guarantees the Referential Integrity between rows of various tables.

Updates the indexes automatically.

Interprets an SQL query and combines information from various tables.

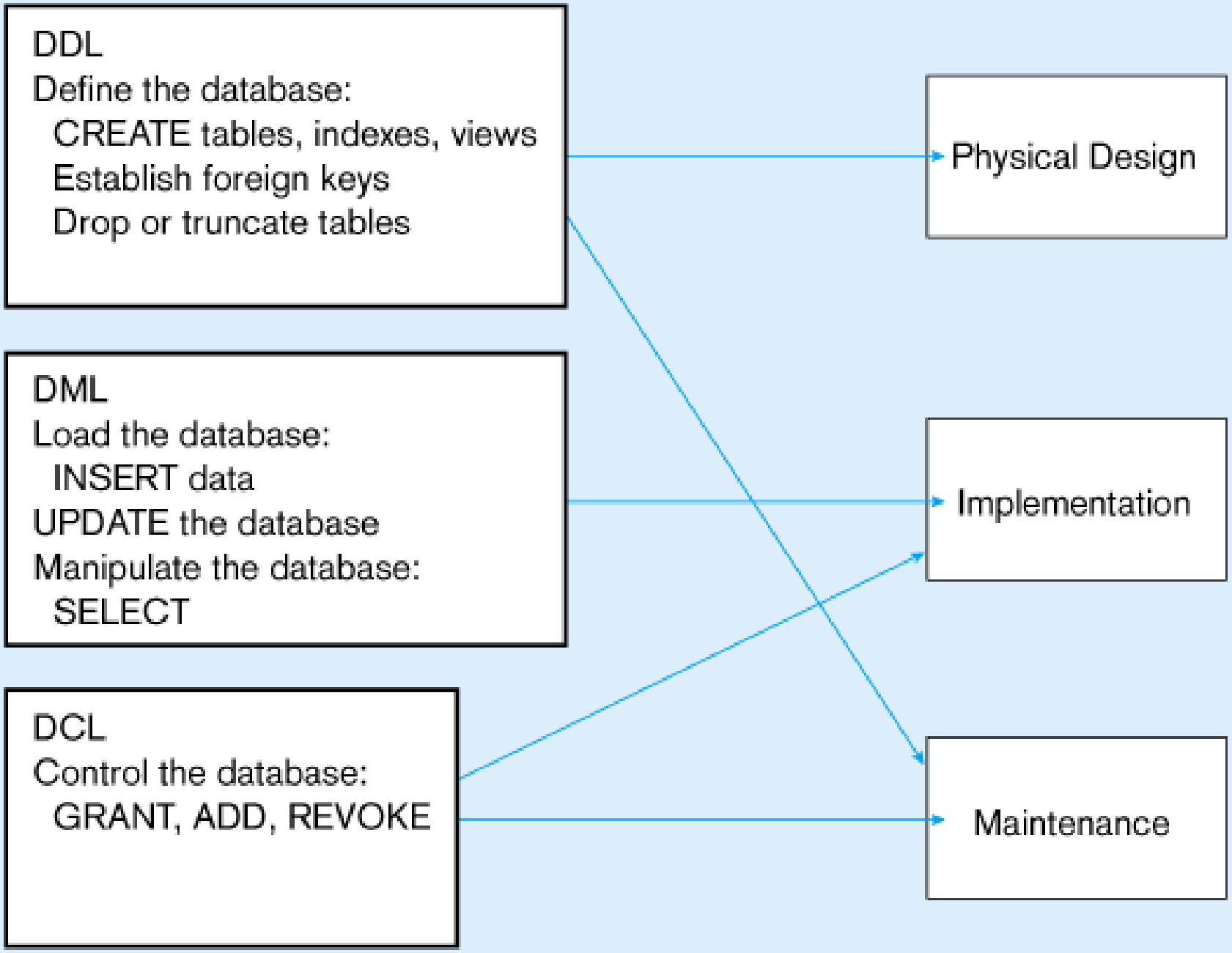
Data Definition Language

- The schema for each relation, including attribute types.
- Integrity constraints
- Authorization information for each relation.
- Non-standard SQL extensions also allow specification of
  - The set of indices to be maintained for each relations.
  - The physical storage structure of each relation on disk.

Data Manipulation Language

Data Manipulation Language (DML) statements are used for managing data within schema objects. Some examples:

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain


DDL
Define the database:
CREATE tables, indexes, views
Establish foreign keys
Drop or truncate tables
DML
Load the database:
INSERT data
UPDATE the database
Manipulate the database:
SELECT
DCL
Control the database:
GRANT, ADD, REVOKE
Physical Design
Implementation
Maintenance

Definitions related to database

Database: A database is a collection of tables, with related data.

Table: A table is a matrix with data. A table in a database looks like a simple spreadsheet.

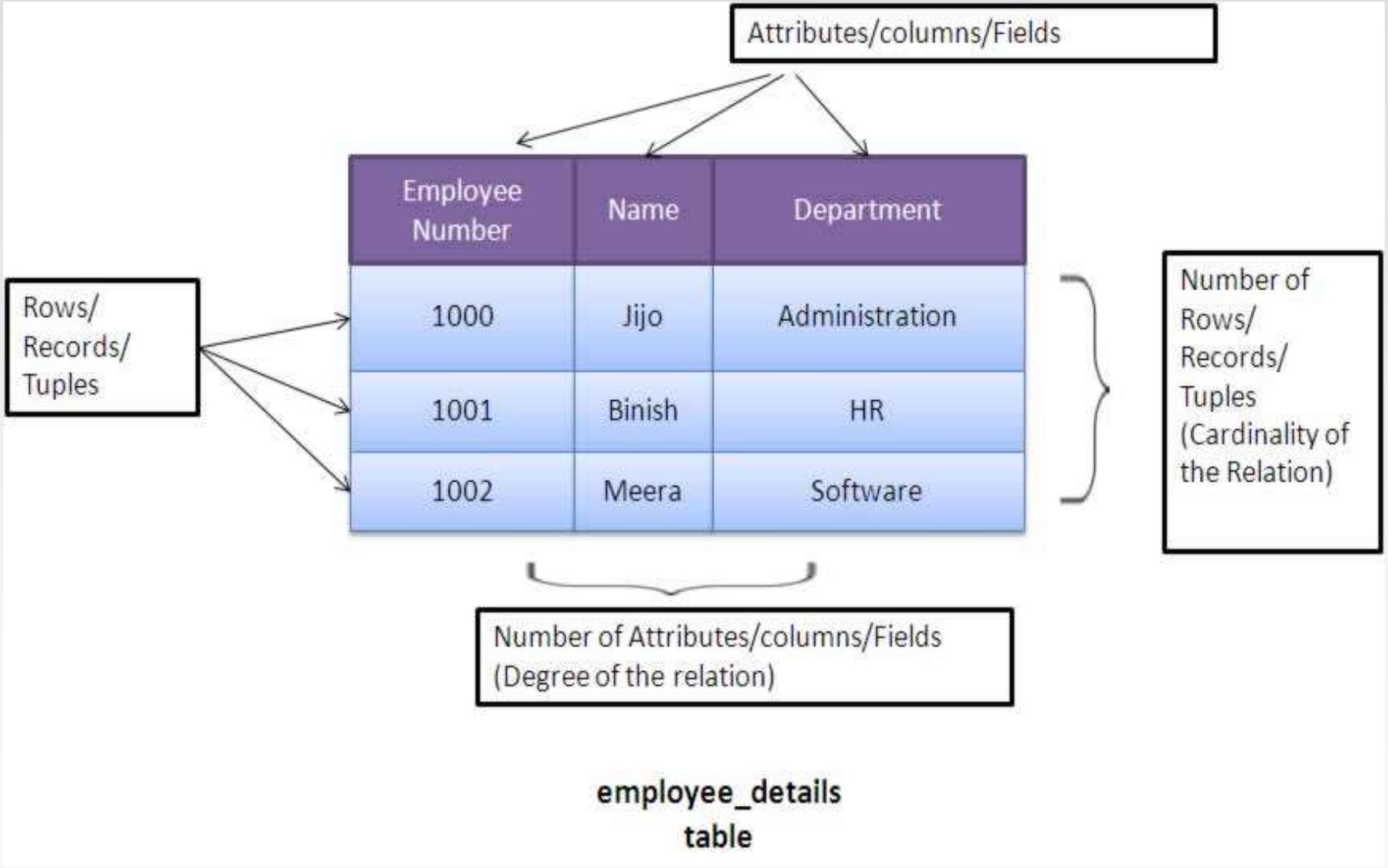
Column: One column (data element) contains data of one and the same kind, for example the column postcode.

Row: A row (= tuple, entry or record) is a group of related data, for example the data of one subscription.

Redundancy: Storing data twice, redundantly to make the system faster.

different

Table in MySQL



Definitions related to database

Primary Key: A primary key is unique. A key value cannot occur twice in one table. With a key, you can find at most one row.

Foreign Key: A foreign key is the linking pin between two tables.

Compound Key: A compound key (composite key) is a key that consists of multiple columns, because one column is not sufficiently unique.

Index: An index in a database resembles an index at the back of a book.

Referential Integrity: Referential Integrity makes sure that a foreign key value always points to an existing row.

Data Types in MySQL

- **char(n).** Fixed length character string, with user-specified length *n(0 to 255)*

.

- **varchar(n).** Variable length character strings, with user-specified maximum length *n (0 to 65535).*

- **int.** Integer (a finite subset of the integers that is machine-dependent).

- **smallint.** Small integer (a machine-dependent subset of the integer domain type).

- **Double(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.

.

- **float(n).** Floating point number, with user-specified precision of at least *n* digits.

- **Date.** A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'

MySQL Constraints

- SQL constraints are used to specify rules for the data in a table.
- Constraints are used to limit the type of data that can go into a table..
- Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

- The following constraints are commonly used in SQL:
- **NOT NULL** - Ensures that a column cannot have a NULL value
- **UNIQUE** - Ensures that all values in a column are different
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Used to create and retrieve data from the database very quickly

Primary Key

A primary key is a column or a set of columns that uniquely identifies each row in the table.The primary key follows these rules:

1. A primary key must contain unique values.
2. If the primary key consists of multiple columns, the combination of values in these columns must be unique.
3. A primary key column cannot have NULL values.

Any attempt to insert or update NULL to primary key columns will result in an error. Note that MySQL implicitly adds a NOT NULL constraint to primary key columns.

A table can have one an only one primary key.

Create table with Primary Key

- If the primary key has one column, you can use the PRIMARY KEY constraint as a column constraint:

Create table customer (CustNo INT PRIMARY KEY , CustName VARCHAR(200) , Street VARCHAR(200), City VARCHAR(200) , State CHAR(4), Zip VARCHAR(20) PRIMARY KEY (CustNo)) *;*

- When the primary key has more than one column, you must use the PRIMARY KEY constraint as a table constraint.

Create table customer (CustNo INT PRIMARY KEY , CustName VARCHAR(200) , Street VARCHAR(200), City VARCHAR(200) , State CHAR(4), Zip VARCHAR(20) PRIMARY KEY (CustNo)) *;*

Not NULL Constraint

- MySQL NOT NULL Constraint
- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column to NOT accept NULL values.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

Create table customer (CustNo INT PRIMARY KEY , CustName VARCHAR(200) NOT NULL , Street VARCHAR(200), City VARCHAR(200) ,

FOREIGN KEY Constraint

- A FOREIGN KEY is a key used to link two tables together
- A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.
- The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

CREATE TABLE PurchaseOrder (DATE, ShipDate DATE, ToStreet VARCHAR(200), ToCity VARCHAR(200), ToState CHAR(2), ToZip VARCHAR(20), PRIMARY KEY(PONo) , FOREIGN KEY fk_cust(CustNo) REFERENCES customer (CustNo)) ;

Unique Constraint

- The UNIQUE constraint ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table

.

```
CREATE TABLE Employee (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    UNIQUE (ID)
);
```

Check Constraint

- The CHECK constraint is used to limit the value range that can be placed in a column.
- If you define a CHECK constraint on a single column it allows only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

```
CREATE TABLE Employee (
    ID int NOT NULL,
    Last_Name varchar(255) NOT NULL,
    First_Name varchar(255),
    Age int ,
    CHECK (Age> =18)
);
```

CHECK constraint on multiple columns

To add constraint on multiple column table level constraint is wriiten.

Example:

```
CREATE TABLE Employee (
    Emp_ID int NOT NULL,
    Last_Name varchar(255) NOT NULL,
    First_Name varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Emp CHECK (Age>=18 AND City=‘ABC')
);
```

DEFAULT Constraint

- The DEFAULT constraint is used to provide a default value for a column.

- The default value will be added to all new records IF no other value is specified.

Example:

```
); CREATE TABLE Employee (
    Emp_ID int NOT NULL,
    Last_Name varchar(255) NOT NULL,
    First_Name varchar(255),
    Age int,
    City varchar(255) DEFAULT ‘Pune’
    )
```

AUTO Increment Field

- AUTO INCREMENT Field
- Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.
- Often this is the primary key field that we would like to be created automatically every time a new record is inserted.
  Example:

```
); CREATE TABLE Employee (
    Emp_ID int NOT NULL AUTO_INCREMENT,
    Last_Name varchar(255) NOT NULL,
    First_Name varchar(255),
    Age int,
    City varchar(255)
    )
```

Example: Person

Name	Age	Weight
Harry	34	80
Sally	28	64
George	29	70
Helena	54	54
Peter	34	80

1) `SELECT *`
`FROM person`
`WHERE age > 30;`

Name	Age	Weight
Harry	34	80
Helena	54	54
Peter	34	80

2) `SELECT weight`
`FROM person`
`WHERE age > 30;`

Weight
80
54
80

3) `SELECT` **`distinct`** `weight`
`FROM person`
`WHERE age > 30;`

Weight
80
54

Examples

```
CREATE TABLE FoodCart (
date varchar(10),
food varchar(20),
profit float
);

ALTER TABLE FoodCart (
ADD sold int
);

ALTER TABLE FoodCart(
DROP COLUMN profit
);

DROP TABLE FoodCart;
```

MySQL queries

Create a database:
create database database_name;

Display databases:
show databases;

Selecting a database
use database_name

Display tables:
show tables

Create a table:
create table create table person
(id int, person varchar (32))
engine=engine_name

Deleting a table:
Drop table table_name

Insert a value in a table:
Insert into table_name.db_name
(field1 , field 2...) Values (v1, v2, ...), (v1',v2');

View data all data in a table:
select * from table name where cond = value

Select data from two different tables:
Select field1, field2 from tbl1, tbl2 where
field3= field4;

Show database variables:
show variables;

The Banking Schema

- *branch* = (*branch_name*, *branch_city*, *assets*)
- *customer* = (*customer_id*, *customer_name*, *customer_street*, *customer_city*)
- *loan* = (*loan_number*, *amount*)
- *account* = (*account_number*, *balance*)
- *employee* = (*employee_id*. *employee_name*, *telephone_number*, *start_date*)
- *dependent_name* = (*employee_id*, *dname*)
- *account_branch* = (*account_number*, *branch_name*)
- *loan_branch* = (*loan_number*, *branch_name*)
- *borrower* = (*customer_id*, *loan_number*)
- *depositor* = (*customer_id*, *account_number*)
- *cust_banker* = (*customer_id*, *employee_id*, *type*)
- *works_for* = (*worker_employee_id*, *manager_employee_id*)
- *payment* = (*loan_number*, *payment_number*, *payment_date*, *payment_amount*)
- *savings_account* = (*account_number*, *interest_rate*)
- *checking_account* = (*account_number*, *overdraft_amount*)

Example

course	*teacher*
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	*book*
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text


MySQL

Use of MySQL

To display the version of mysql and current date

mysql> SELECT VERSION(), CURRENT_DATE;

```
+-----------------+--------------+
| VERSION()       | CURRENT_DATE |
+-----------------+--------------+
| 5.1.67-community | 2020-06-24   |
+-----------------+--------------+
1 row in set (0.00 sec)
```

;

wit
s

Use of MySQL

To display the User

```
mysql> SELECT USER (), CURRENT_DATE;

output
+---------------+--------------+
| USER ()       | CURRENT_DATE |
+---------------+--------------+
| root@localhost | 2020-06-24   |
+---------------+--------------+
1 row in set (0.00 sec)
```

wit
s

Create user

CREATE USER statement

The CREATE USER statement creates a database account that allows you to log into the MySQL database.

Syntax

CREATE USER user_name IDENTIFIED BY [PASSWORD] 'password_value';

GRANT/REVOKE Privileges

Grant/Revoke Privileges
You can GRANT and REVOKE privileges on various database objects in MySQL.

GRANT privileges ON object TO user;
Example:

GRANT SELECT, INSERT, UPDATE, DELETE ON contacts TO ‘First_user';

You can then view the privileges assigned to a user using the **Show Grants** command.

DDL in MySQL

- Data is case-sensitive, SQL commands are not.
- SQL query includes references to tuples variables and the attributes of those variables
  - `CREATE TABLE`: used to create a table.
  - `ALTER TABLE`: modifies a table after it was created.
  - `DROP TABLE`: removes a table from a database.

Use of ‘Create table’

CREATE TABLE table_name(column_list)

Example:

Create table customer (CustNo INT, CustName VARCHAR(200),Street VARCHAR(200), City VARCHAR(200),State CHAR(4), Zip VARCHAR(20));

ALTER TABLE

- ALTER TABLE table_name options[, options...]

Options for Alter table:

ADD [COLUMN] create_definition

ADD INDEX [index_name] (index_col_name,...)

ADD PRIMARY KEY (index_col_name,...)

ADD UNIQUE [index_name] (index_col_name,...)

DROP [COLUMN] col_name

DROP PRIMARY KEY

DROP INDEX index_name

Alter Table examples

- Alter table student add primary key (Roll_no);
- Alter table student add column (age int);
- Alter table student add CHECK (age >=18);
- Alter table student add Unique (Email_ID);
- Alter table student add column (license varchar(20));
- Alter table student add constraint E_ID unique (Email_id);
- Alter table student drop index U_email ;
- Alter table student drop Primary key;

DML

SELECT command

A typical SQL query has the form:

select $A_1, A_2, ..., A_n$
from $r_1, r_2, ..., r_m$
where P

A_i represents an attribute
R_i represents a relation
P is a predicate.

The result of an SQL query is a relation.

SELECT CLAUSE

1. The select clause list the attributes desired in the result of a query

2. An asterisk in the select clause denotes " all attributes

select *
from *loan*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

3. The **select** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.

select *loan_number, branch_name, amount* ∗ 100
from *loan*

The from Clause

The **from** clause lists the relations involved in the queryCorresponds to the Cartesian product operation of the relational algebra.

Find the Cartesian product ***borrower*** *X* ***loan***

select ∗
from *borrower, loan*

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

The from Clause

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

customer_name	*customer_street*	*customer_city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

select *customer_name, borrower.loan_number, amount*
from *borrower, loan ,customer*
where ***(****borrower.loan_number = loan.loan_number) **and** (customer.*
cust_id=borrower.cust.id) **and**
branch_name = 'Perryridge'

The where Clause

- The **where** clause specifies conditions that the result must satisfy

- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

  select *loan_number*
  from *loan*
  where *branch_name* = 'Perryridge' **and** *amount* > 1200

- Comparison results can be combined using the logical connectives **and, or,** and **not.**

SELECT Clause

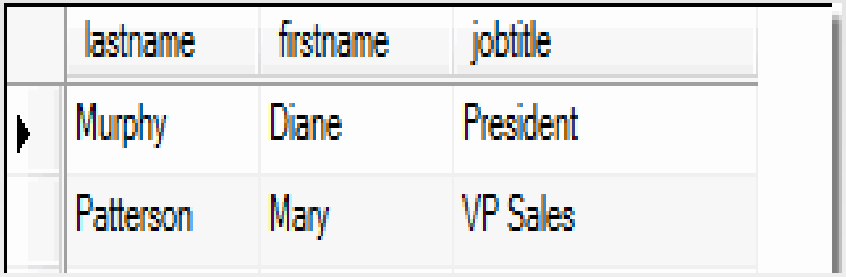
```
SELECT column_name1,column_name2...
FROM tables
[WHERE conditions]
[GROUP BY group
[HAVING group_conditions]]
[ORDER BY sort_columns]
[LIMIT limits];
```

Select operation on Employee Relation

employeeNumber	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	NULL	President
1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1	1002	VP Sales
1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1	1002	VP Marketing
1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC)
1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sale Manager (EMEA)

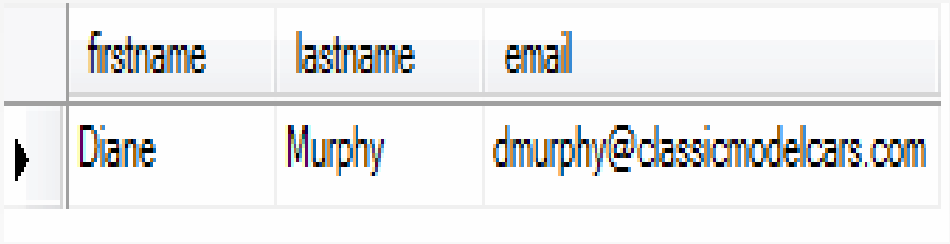
Select operation on Employee Relation

1. SELECT lastname, firstname, jobtitle FROM employees



lastname	firstname	jobtitle
Murphy	Diane	President
Patterson	Mary	VP Sales

2. SELECT firstname,lastname,email
   FROM employees
   WHERE jobtitle="president



firstname	lastname	email
Diane	Murphy	dmurphy@classicmodelcars.com

Select operation on Employee Relation

3. SELECT firstname,lastname, jobtitle
   FROM employees
   ORDER BY firstname ASC;

4. SELECT firstname,lastname, jobtitle
   FROM employees
   ORDER BY firstname DESC;

Use of LIMIT operation in select

```
SELECT * FROM table  LIMIT N;

SELECT * FROM table  LIMIT S,  N;
```

Select operation on Employee Relation

Selecting Data with SQL IN

SELECT column_list FROM table_name WHERE column IN ("list_item1","list_item2“.)

```
SELECT Employee_Name

FROM Employee

WHERE city IN (‘Pune',‘Mumbai')
```

Use of SQL BETWEEN

Retrieving Data in a Range Using SQL BETWEEN

```
SELECT column_list

FROM table_name

WHERE column_1 B ETWEEN l  ower_range    AND    upper_range
```

Example:

```
SELECT productCode,ProductName,buyPrice

FROM products

WHERE buyPrice BETWEEN 90 AND 100

ORDER BY buyPrice DESC
```

Use MySQL LIKE

MySQL scans the whole employees table to find all employees which have first name starting with character ‘a’ and followed by any number of characters

```
SELECT employeeNumber, lastName, firstName
FROM employees
WHERE firstName LIKE 'a%‘
```

Output

```
+----------------+----------+-----------+
| employeeNumber | lastName | firstName |
+----------------+----------+-----------+
|           1611 | Fixter   | Andy      |
+----------------+----------+-----------+
1 row in set (0.00 sec)
```

Use MySQL LIKE

- To search all employees which have last name ended with ‘on’ string you can perform the query as

- SELECT employeeNumber, lastName, firstName
- FROM employees
- WHERE lastName LIKE '%on‘

Output:

```
+----------------+-----------+-----------+
| employeeNumber | lastName | firstName |
+----------------+-----------+-----------+
|          1088 | Patterson | William   |
|          1216 | Patterson | Steve     |
+----------------+-----------+-----------+
2 rows in set (0.00 sec)
```

Use MySQL LIKE

- If string is embedded somewhere in a column, put the percentage wild card at the beginning and the end of it to find all possibilities.

- 1 SELECT employeeNumber, lastName, firstName
- FROM employees
- WHERE lastname LIKE '%on%

Output

```
+---------------+----------+----------+
| employeeNumber | lastName | firstName |
+---------------+----------+----------+
|          1088 | Patterson | William   |
|          1102 | Bondur   | Gerard    |
|          1216 | Patterson | Steve     |
|          1337 | Bondur   | Loui      |
|          1504 | Jones    | Barry     |
+---------------+----------+----------+
```

String Operations

SQL supports a variety of string operations such as

- concatenation
- converting upper to lower case (and vice versa)
- finding string length, extracting substrings, etc.

Concatenation

The syntax for the CONCAT function in MySQL is:

CONCAT(expression1, expression2, ... expression_n)

- If any of the *expressions* is a NULL, the CONCAT function will return a NULL value
- If *expression* is a numeric value, it will be converted by the CONCAT function to a binary string

Example:

SELECT CONCAT('The answer is ', 10+10); ***Result:*** **'The answer is 20'**

String Operations

- Upper

  SELECT Upper(Stud_name) from Student;

- Lower

  SELECT Lower(Stud_name) from Student;

- Length

  SELECT Length(Stud_name) from Student;

String Operations

Strcmp

Select Strcmp('ABC','abc')

- If *string1* and *string2* are the same, the STRCMP function will return 0.
- If *string1* is smaller than *string2*, the STRCMP function will return -1.
- If *string1* is larger than *string2*, the STRCMP function will return 1

Length

Select Length ('ABC')

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.

  Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:

  - *m* + *n* times in *r* **union all** *s*
  - min(*m,n)* times in *r* **intersect all** *s*
  - max(0, *m* – *n)* times in *r* **except all** *s*

Data for Set operation

customer_name	*account_number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Depositor

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	*null*

borrower

Set Operations

- Find all customers who have a loan, an account, or both:

  (select *cust_id* **from** *depositor*)
  union
  (select *cust_id* **from** *borrower)*

- Find all customers who have both a loan and an account.

  (**select** *cust_id* **from** *depositor*)
  intersect
  (select *cust_id* **from** *borrower)*

- Find all customers who have an account but no loan.

  (**select** *cust_id* **from** *depositor*)
  except
  (select *cust_id* **from** *borrower)*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

```
select avg (balance)
        from account
        where branch_name = 'Perryridge'
```

- Find the number of tuples in the *customer* relation.

```
select count (*)
        from customer
```

- Find the number of depositors in the bank.

```
select count (distinct customer_name)
        from depositor
```

Aggregate Functions – Group By

- Find the number of depositors for each branch.

select *branch_name*, **count (distinct** *customer_name)*
from *depositor, account*
where *depositor.account_number = account.account_number*
group by *branch_name*

Note: Attributes in **select** clause outside of aggregate functions must appear in **group by** list

Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than $1,200.

```
select branch_name, avg (balance)
        from account
        group by branch_name
        having avg (balance) > 1200
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups


MySQL

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Nested Subqueries

- A subquery is a query within a query.
- You can create subqueries within your SQL statements.
- These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as sub query expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

My SQL Joins

- INNER JOIN (or sometimes called simple join)
- LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)

Joined Relations – Datasets for Examples

- Data for Example 1
- Relation *loan* and Relation *borrower*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- Note: borrower information missing for L-260 and loan information missing for L-155
  - Data for Example 2

Supplier_id	***Supplier_Name***
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

Supplier

Order_id	***Supplier_id***	***Order_Date***
500125	10000	2013/05/12
500126	10001	2013/05/13
500127	10002	2013/08/14

Orders

Joined Relations – Examples

- *Example1:*

  *Select * from loan* **inner join** *borrower* **on** *loan.loan_number = borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *Example2*
- *SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date FROM suppliers INNER JOIN orders ON suppliers.supplier_id = orders.supplier_id;*

Supplier_id	***Supplier_Name***	***Order_id***
10000	IBM	500125
10001	Hewlett Packard	500126

Joined Relations – Examples

- *Example1:*

  *Select * from loan* **left outer join** *borrower* **on** *loan.loan_number = borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

- *Example2*
- *Select suppliers.supplier_id, suppliers.supplier_name, orders.order_date from suppliers* ***left outer join*** *orders* ***on*** *suppliers.supplier_id = orders.supplier_id;*

Supplier_id	***Supplier_Name***	***Order_id***
10000	IBM	500125
10001	Hewlett Packard	500126
10002	Microsoft	NULL
10003	NVIDIA	NULL

Joined Relations – Examples

- *Example1:*
- *Select * from loan* **natural inner join** *borrower* ***on*** *loan.loan_number=borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- *Select * from loan* **natural right outer join** *borrower* ***on*** *loan.loan_number=borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Joined Relations – Examples

- *Example2:*
- *Select suppliers.supplier_id, suppliers.supplier_name, orders.order_date from suppliers* ***right outer join*** *orders* ***on*** *suppliers.supplier_id = orders.supplier_id;*

Supplier_id	***Supplier_Name***	***Order_id***
10000	IBM	500125
10001	Hewlett Packard	500126
Null	Null	500127

- Find all customers who have either an account or a loan (but not both) at the bank.

select *customer_name*
from (*depositor* **natural full outer join** *borrower*)
where *account_number* **is null or** *loan_number* **is null**

The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- E.g. Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*

select *customer_name, borrower.loan_number* **as** *loan_id, amount*
from *borrower, loan ,customer*
where *borrower.loan_number = loan.loan_number and*
borrower.cust_id= customer.cust_id

Tuple Variables

- Tuple variables are defined in the **from** clause via the use of the **as** clause.
- Find the customer names and their loan numbers and amount for all customers having a loan at some branch.

select *customer_name, T.loan_number, S.amount*
from *borrower* **as** *T, loan* **as** *S Customer as C*
where *T.loan_number = S.loan_number and C.Cust_id = T.Cust_id*

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

select distinct *T.branch_name*
from *branch* **as** *T, branch* **as** *S*
where *T.assets* > *S.assets* **and** *S.branch_city* = 'Brooklyn'

- Keyword **as** is optional and may be omitted

borrower **as** *T* ≡ *borrower T*

  - Some database such as Oracle *require* **as** to be omitted

View Definition

- A relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

General syntax of View

- CREATE [OR REPLACE] VIEW view_name AS SELECT columns FROM tables [WHERE conditions];

OR REPLACE: Optional

If you do not specify this clause and the VIEW already exists, the CREATE VIEW statement will return an error.

View_name: The name of the VIEW that you wish to create in MySQL.

WHERE condition: Optional.

The conditions that must be met for the records to be included in the VIEW.

Advantages of Views

- Simplify query commands
- Assist with data security (but don't rely on views for security, there are more important security measures)
- Enhance programming productivity
- Contain most current base table data
- Use little storage space
- Provide customized view for user
- Establish physical data independence

Disadvantages of Views

- Use processing time each time view is referenced
- May or may not be directly updateable

Example Queries

- A view consisting of branches and their customers

```
create view all_customer as
        (select branch_name, customer_name
         from depositor, account
        where depositor.account_number =
                account.account_number )
        union
        (select branch_name, customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number )
```

- Find all customers of the Perryridge branch

```
select customer_name
        from all_customer
        where branch_name = 'Perryridge'
```

Uses of Views

- Hiding some information from some users
  - Consider a user who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount.
  - Define a view

    (**create view** *cust_loan_data* **as**
    select *customer_name, borrower.loan_number, branch_name*
    from *borrower, loan*
    where *borrower.loan_number = loan.loan_number*)
  - Grant the user permission to read *cust_loan_data,* but not *borrower or loan*
- Predefined queries to make writing of other queries easier
  - Common example: Aggregate queries used for statistical analysis of data

“In” Construct

- Find all customers who have both an account and a loan at the bank.

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                from depositor )
```

- Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer_name
                                    from depositor )
```

Example Query

- Find all customers who have both an account and a loan at the Perryridge branch

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
          branch_name = 'Perryridge'  and
          (branch_name, customer_name ) in
                (select branch_name, customer_name
                  from depositor, account
                  where depositor.account_number =
                      account.account_number )
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

“Some” Construct

- Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct  T.branch_name
        from branch as T, branch as S
        where  T.assets > S.assets and
                S.branch_city = 'Brooklyn'
```

- Same query using > **some** clause

```
select branch_name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch_city = 'Brooklyn')
```

“All” Construct

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

  delete from *account*
  where *branch_name* = 'Perryridge'

- Delete all accounts at every branch located in the city ‘Needham’.

  delete from *account*
  where *branch_name* **in** (**select** *branch_name*
  from *branch*
  where *branch_city* = 'Needham')

Example Query

- Delete the record of all accounts with balances below the average at the bank.

delete from *account*
where *balance* < (**select avg** (*balance*)
from *account*)

  - Problem: as we delete tuples from deposit, the average balance changes
  - Solution used in SQL:
    1. First, compute **avg** balance and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to *account*

  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)
  values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values ('A-777','Perryridge', *null*)

Modification of the Database – Insertion

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

```
insert into account
    select loan_number, branch_name, 200
    from loan
    where branch_name = 'Perryridge'
insert into depositor
    select customer_name, loan_number
    from loan, borrower
    where branch_name = 'Perryridge'
          and loan.account_number = borrower.account_number
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation
  - Motivation: **insert into** *table*1 **select** * **from** *table*1

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

update *account*
set *balance* = *balance* $*$ 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* $*$ 1.05
where *balance* $\leq$ 10000

  - The order is important
  - Can be done better using the **case** statement (next slide)

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the average account balance of those branches where the average account balance is greater than $1200.

```
select branch_name, avg_balance
from (select branch_name, avg (balance)
        from account
        group by branch_name )
      as branch_avg ( branch_name, avg_balance )
where avg_balance > 1200
```

Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.
  - Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

    select *loan_number*
    from *loan*
    where *amount* **is null**
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- However, aggregate functions simply ignore nulls
  - More on next slide

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Transaction Control

Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

- COMMIT - save work done
- SAVEPOINT - identify a point in a transaction to which you can later roll back
- ROLLBACK - restore database to original since the last COMMIT
- SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

JSPM'S
RAJARSHI SHAHU COLLEGE OF ENGINEERING
[An Autonomous Institute ,Affiliated to SPPU, Approved by AICTE]

Academic Year 2022-23
Course : SY B.Tech
Course Name : DBMSL

Prepared By
Prof. Archana Jadhav

DBMS Laboratory

(Assignment)

Write and execute PL/SQL

1. Stored procedure
2. Stored function

PL/ SQL

PL/ SQL : Procedural language extension to SQL

Features of PL/SQL

- **Procedural Language Capability:** It consists of procedural language constructs such as
  1. Conditional statements (if...else)
  2. Loop statements (for loop)

- **Declare variable:** It allow you to declare variable
- **Reducing network traffic**.
- **Code Re-usability:** It support code re-usability
- **Error Handling:** It Provides error handling facility

Stored Procedures in MySQL

- **A stored procedure contains a sequence of SQL commands**
- **It is stored in the database in a pre-compiled form.**
- **It can be invoked later by a program**
- **Syntax for declaring Stored procedures :**

```
Create Procedure <proc-name>
        (param_spec1, param_spec2, …, param_specn )
begin
     -- execution code
end;
```

Parameter types

param_spec is of the form:

[IN | OUT | INOUT] <param_name> <param_type>

- IN mode: allows you to pass values into the procedure,
- OUT mode: allows you to pass value back from procedure to the calling program

General syntax for Procedure

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (parameter [,parameter]) ]

   [declaration_section]

BEGIN
   executable_section

[EXCEPTION
   exception_section]

END [procedure_name];
```

Optional section for exception handling

Decision control and Looping Constructs

Decision control

```
IF <condition> then
        <statements>
END IF

IF <condition> then
        <statements>
ELSEIF <condition> then
        <statements>
END IF

IF <condition> then
        <statements>
ELSEIF <condition> then
        <statements>
ELSE
        <statements>
END IF
```

Case Statement

```
CASE <expression>
        WHEN <value> then
                <statements>
        WHEN <value> then
                <statements>
        …
        ELSE
                <statements>
END CASE;
```

Loops

```
[begin_label:] LOOP
        <statement list>
END LOOP [end_label]
```

The end_label has to = the begin_label. Both are optional

```
[begin_label:] REPEAT
        <statement list>
UNTIL <search_condition>
END REPEAT [end_label]

[begin_label:] WHILE
<condition> DO
        <statements>
END WHILE [end_label]
```

Example of Procedure

```
mysql> select * from employee;
+----+------+---------+--------+------------+------+
| id | name | superid | salary | bdate      | dno  |
+----+------+---------+--------+------------+------+
|  1 | john |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob  |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom  |       1 |  50000 | 1978-01-17 |    2 |
|  5 | bill |    NULL |   NULL | 1985-01-20 |    1 |
+----+------+---------+--------+------------+------+
```

We want to keep track of the total salaries of employees working for each department

-

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
+---------+-------------+
```

Example of Procedure

```
mysql> create procedure updateSalary (IN param1 int)
    -> begin
    ->     update deptsal
    ->     set totalsalary = (select sum(salary) from employee where dno = param1)
    ->     where dnumber = param1;
    -> end; //
Query OK, 0 rows affected (0.01 sec)
```

Stored Function in MySQL

- **A stored function contains a sequence of SQL commands stored in the database and return a single value**
- **It can be used like a built-in function**
- **Syntax for declaring Function procedures :**

```
Create Function <proc-name>
        (param_spec1, param_spec2, …, param_specn )
returns <return_type>
begin
     -- execution code
End;
```

JSPM'S
RAJARSHI SHAHU COLLEGE OF ENGINEERING
[An Autonomous Institute ,Affiliated to SPPU, Approved by AICTE]

Academic Year 2022-23
Course : SY BTech
Course Name : DBMSL

Prepared By
Prof. Archana Jadhav

DBMS Laboratory

(Assignment)

Write a PL/SQL block to implement a cursor.

MySQL Cursor

Introduction to Cursor :

- **Need of cursor:**
  - Select statement may return many records.
  - Here we need a mechanism to navigate the the result tuple by tuple.

- **Cursors are used inside the triggers, stored procedures or stored functions.**

Working with MySQL Cursor

1.Declare a cursor

DECLARE cursor_name **CURSOR FOR** SELECT_statement;

2. Open the cursor

OPEN cursor_name;

3. Use the FETCH statement

It retrieve the next row pointed by the cursor and move the cursor pointer to the next row in the result set.

FETCH cursor_name INTO variables list;

4. Finally, we close the cursor to release the memory associated with it

CLOSE cursor_name;

Working with MySQL Cursor

- **Each time you call the FETCH statement**

  --The cursor attempts to read the next row in the result set.

- **When the cursor reaches the last tuple of the result set**

  -- It will not be able to get the data

  -- Here

a condition is raised

- **The handler is used to handle this condition**

Syntax to declare a NOT FOUND handler

Example of Cursor

Relations used to demonstrate the use of cursor

1. Employee

```
mysql> select * from employee;
+----+------+---------+--------+------------+------+
| id | name | superid | salary | bdate      | dno  |
+----+------+---------+--------+------------+------+
|  1 | john |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob  |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom  |       1 |  50000 | 1978-01-17 |    2 |
|  5 | bill |    NULL |   NULL | 1985-01-20 |    1 |
+----+------+---------+--------+------------+------+
```

2. Deptsal

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
+---------+-------------+
```

Example of Cursor

```
mysql> create procedure updateSalary()
    -> begin
    ->          declare done int default 0;
    ->          declare current_dnum int;
    ->          declare dnumcur cursor for select dnumber from deptsal;
    ->          declare continue handler for not found set done = 1;
    ->
    ->          open dnumcur;
    ->
    ->          repeat
    ->                fetch dnumcur into current_dnum;
    ->                update deptsal
    ->                set totalsalary = (select sum(salary) from employee
    ->                                   where dno = current_dnum)
    ->                where dnumber = current_dnum;
    ->          until done
    ->          end repeat;
    ->
    ->          close dnumcur;
    -> end$$
Query OK, 0 rows affected (0.00 sec)
```

JSPM'S
RAJARSHI SHAHU COLLEGE OF ENGINEERING
[An Autonomous Institute ,Affiliated to SPPU, Approved by AICTE]

Academic Year 2022-23
Course : SY BTech IT
Course Name : DBMSL

Prepared By
Prof. Archana Jadhav

DBMS Laboratory

Assignment

Write and execute database triggers

Concept of trigger

- A trigger is a database object that is associated with a table and is defined to execute when a particular type of event occurs on that table.

- It provides a mean to execute a SQL statement(s) when event like insert, update or delete happens.

Concept of triggers

- A trigger is an object that belongs to a database
- Each trigger within the database must have a different name
- The events for which trigger can be defined are INSERT, DELETE, and UPDATE
- We can define multiple triggers for a table, one trigger per type of event
- Triggers can be defined to activate either BEFORE or AFTER the event

Benefits of triggers

- A trigger can examine row values to be inserted or updated.
- It can determine what values were deleted or what they were updated to.
- It can change values before they are inserted or updated into a table

Trigger types

Triggers are of two types

- **Row level**
- **Statement level**

Valid Types of Triggers

- BEFORE INSERT row
- AFTER INSERT row
- BEFORE UPDATE row
- AFTER UPDATE row
- BEFORE DELETE row
- AFTER DELETE row

Trigger Syntax

```
CREATETRIGGER trigger_name
        {BEFORE|AFTER}
        {INSERT|DELETE|UPDATE}
        ON table_name
        FOR EACH ROW
        BEGIN
          Statements;
        End;
```

Example of Trigger

```
CREATE TRIGGER Set_Amount BEFORE
   UPDATE ON account
      FOR EACH ROW
      BEGIN
         IF NEW.amount < 0 THEN
            SET NEW.amount = 0;
         END IF;
      END;
```