

## LEC-9: SQL in 1-Video

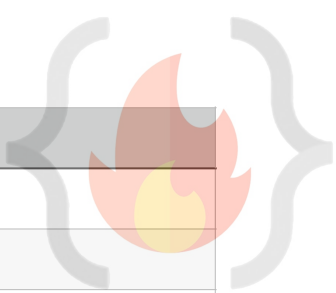


1. **SQL:** Structured Query Language, used to access and manipulate data.
2. SQL used **CRUD** operations to communicate with DB.
  1. **CREATE** - execute INSERT statements to insert new tuple into the relation.
  2. **READ** - Read data already in the relations.
  3. **UPDATE** - Modify already inserted data in the relation.
  4. **DELETE** - Delete specific data point/tuple/row or multiple rows.
3. **SQL is not DB, is a query language.**
4. What is **RDBMS**? (Relational Database Management System)
  1. Software that enable us to implement designed relational model.
  2. e.g., MySQL, MS SQL, Oracle, IBM etc.
  3. Table/Relation is the simplest form of data storage object in R-DB.
  4. **MySQL** is open-source RDBMS, and it uses SQL for all CRUD operations
5. **MySQL** used client-server model, where client is CLI or frontend that used services provided by MySQL server.
6. **Difference between SQL and MySQL**
  1. SQL is Structured Query language used to perform CRUD operations in R-DB, while MySQL is a RDBMS used to store, manage and administrate DB (provided by itself) using SQL.

**SQL DATA TYPES** (Ref: [https://www.w3schools.com/sql/sql\\_datatypes.asp](https://www.w3schools.com/sql/sql_datatypes.asp))

1. In SQL DB, data is stored in the form of tables.
2. Data can be of different types, like INT, CHAR etc.

DATATYPE	Description
CHAR	string(0-255), string with size = (0, 255], e.g., CHAR(251)
VARCHAR	string(0-255)
TINYTEXT	String(0-255)
TEXT	string(0-65535)
BLOB	string(0-65535)
MEDIUMTEXT	string(0-16777215)
MEDIUMBLOB	string(0-16777215)
LONGTEXT	string(0-4294967295)
LOBLOB	string(0-4294967295)
TINYINT	integer(-128 to 127)
SMALLINT	integer(-32768 to 32767)
MEDIUMINT	integer(-8388608 to 8388607)
INT	integer(-2147483648 to 2147483647)
BIGINT	integer (-9223372036854775808 to 9223372036854775807)
FLOAT	Decimal with precision to 23 digits
DOUBLE	Decimal with 24 to 53 digits



DATATYPE	Description
DECIMAL	Double stored as string
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYYMMDDHHMMSS
TIME	HH:MM:SS
ENUM	One of the preset values
SET	One or many of the preset values
BOOLEAN	0/1
BIT	e.g., BIT(n), n upto 64, store values in bits.

3. Size: **TINY < SMALL < MEDIUM < INT < BIGINT.**
4. **Variable length Data types** e.g., VARCHAR, are better to use as they occupy space equal to the actual data size.
5. Values can also be unsigned e.g., INT UNSIGNED.
6. **Types of SQL commands:**
  1. **DDL** (data definition language): defining relation schema.
    1. **CREATE:** create table, DB, view.
    2. **ALTER TABLE:** modification in table structure. e.g, change column datatype or add/remove columns.
    3. **DROP:** delete table, DB, view.
    4. **TRUNCATE:** remove all the tuples from the table.
    5. **RENAME:** rename DB name, table name, column name etc.
  2. **DRL/DQL** (data retrieval language / data query language): retrieve data from the tables.
    1. **SELECT**
  3. **DML** (data modification language): use to perform modifications in the DB
    1. **INSERT:** insert data into a relation
    2. **UPDATE:** update relation data.
    3. **DELETE:** delete row(s) from the relation.
  4. **DCL** (Data Control language): grant or revoke authorities from user.
    1. **GRANT:** access privileges to the DB
    2. **REVOKE:** revoke user access privileges.
  5. **TCL** (Transaction control language): to manage transactions done in the DB
    1. **START TRANSACTION:** begin a transaction
    2. **COMMIT:** apply all the changes and end transaction
    3. **ROLLBACK:** discard changes and end transaction
    4. **SAVEPOINT:** checkout within the group of transactions in which to rollback.

## MANAGING DB (DDL)

1. **Creation of DB**
  1. **CREATE DATABASE IF NOT EXISTS db-name;**
  2. **USE db-name;** //need to execute to choose on which DB CREATE TABLE etc commands will be executed.  
//make switching between DBs possible.
  3. **DROP DATABASE IF EXISTS db-name;** //dropping database.
  4. **SHOW DATABASES;** //list all the DBs in the server.
  5. **SHOW TABLES;** //list tables in the selected DB.



## DATA RETRIEVAL LANGUAGE (DRL)

1. Syntax: `SELECT <set of column names> FROM <table_name>;`
2. Order of execution from RIGHT to LEFT.
3. Q. Can we use SELECT keyword without using FROM clause?
  1. Yes, using DUAL Tables.
  2. Dual tables are dummy tables created by MySQL, help users to do certain obvious actions without referring to user defined tables.
  3. e.g., `SELECT 55 + 11;`  
`SELECT now();`  
`SELECT ucase();` etc.
4. **WHERE**
  1. Reduce rows based on given conditions.
  2. E.g., `SELECT * FROM customer WHERE age > 18;`
5. **BETWEEN**
  1. `SELECT * FROM customer WHERE age between 0 AND 100;`
  2. In the above e.g., 0 and 100 are inclusive.
6. **IN**
  1. Reduces **OR** conditions;
  2. e.g., `SELECT * FROM officers WHERE officer_name IN ('Lakshay', 'Maharana Pratap', 'Deepika');`
7. **AND/OR/NOT**
  1. **AND**: `WHERE cond1 AND cond2`
  2. **OR**: `WHERE cond1 OR cond2`
  3. **NOT**: `WHERE col_name NOT IN (1,2,3,4);`
8. **IS NULL**
  1. e.g., `SELECT * FROM customer WHERE prime_status is NULL;`
9. **Pattern Searching / Wildcard ('%', '\_')**
  1. '%', any number of character from 0 to n. Similar to '\*' asterisk in regex.
  2. '\_', only one character.
  3. `SELECT * FROM customer WHERE name LIKE '%p_';`
10. **ORDER BY**
  1. Sorting the data retrieved using **WHERE** clause.
  2. `ORDER BY <column-name> DESC;`
  3. **DESC** = Descending and **ASC** = Ascending
  4. e.g., `SELECT * FROM customer ORDER BY name DESC;`
11. **GROUP BY**
  1. **GROUP BY** Clause is used to collect data from multiple records and group the result by one or more column. It is generally used in a **SELECT** statement.
  2. Groups into category based on column given.
  3. `SELECT c1, c2, c3 FROM sample_table WHERE cond GROUP BY c1, c2, c3.`
  4. All the column names mentioned after **SELECT** statement shall be repeated in **GROUP BY**, in order to successfully execute the query.
  5. Used with aggregation functions to perform various actions.
    1. `COUNT()`
    2. `SUM()`
    3. `AVG()`
    4. `MIN()`
    5. `MAX()`
12. **DISTINCT**
  1. Find distinct values in the table.
  2. `SELECT DISTINCT(col_name) FROM table_name;`
  3. **GROUP BY** can also be used for the same
    1. "Select col\_name from table GROUP BY col\_name;" same output as above **DISTINCT** query.

2. SQL is smart enough to realise that if you are using GROUP BY and not using any aggregation function, then you mean "DISTINCT".

### 13. GROUP BY HAVING

1. Out of the categories made by GROUP BY, we would like to know only particular thing (cond).
2. Similar to WHERE.
3. Select COUNT(cust\_id), country from customer GROUP BY country HAVING COUNT(cust\_id) > 50;
4. WHERE vs HAVING
  1. Both have same function of filtering the row base on certain conditions.
  2. WHERE clause is used to filter the rows from the table based on specified condition
  3. HAVING clause is used to filter the rows from the groups based on the specified condition.
  4. HAVING is used after GROUP BY while WHERE is used before GROUP BY clause.
  5. If you are using HAVING, GROUP BY is necessary.
  6. WHERE can be used with SELECT, UPDATE & DELETE keywords while GROUP BY used with SELECT.

### CONSTRAINTS (DDL)

#### 1. Primary Key

```
CREATE TABLE Customer
(
  id INT PRIMARY KEY,
  branch_id INT,
  First name CHAR(50),
  Last name CHAR(50),
  DOB DATE,
  Gender CHAR(6),
  PRIMARY KEY (id)
);
```

Choose one of the two ways.

1. PK is not null, unique and only one per table.

#### 2. Foreign Key

1. FK refers to PK of other table.
2. Each relation can having any number of FK.
3. CREATE TABLE ORDER (  
id INT PRIMARY KEY,  
delivery\_date DATE,  
order\_placed\_date DATE,  
cust\_id INT,  
FOREIGN KEY (cust\_id) REFERENCES customer(id)  
);

#### 3. UNIQUE

1. Unique, can be null, table can have multiple unique attributes.
2. CREATE TABLE customer (  
...  
email VARCHAR(1024) UNIQUE,  
...  
);

#### 4. CHECK

1. CREATE TABLE customer (  
...  
CONSTRAINT age\_check CHECK (age > 12),  
...  
);  
2. "age\_check", can also avoid this, MySQL generates name of constraint automatically.



## 5. DEFAULT

1. Set default value of the column.
2. CREATE TABLE account (  
...  
savings-rate DOUBLE NOT NULL DEFAULT 4.25,  
...  
);

6. An attribute can be **PK and FK both** in a table.

## 7. ALTER OPERATIONS

1. Changes schema
2. **ADD**
  1. **Add new column.**
  2. ALTER TABLE table\_name ADD new\_col\_name datatype ADD new\_col\_name\_2 datatype;
  3. e.g., ALTER TABLE customer ADD age INT NOT NULL;
3. **MODIFY**
  1. **Change datatype of an attribute.**
  2. ALTER TABLE table-name MODIFY col-name col-datatype;
  3. E.g., VARCHAR TO CHAR  
ALTER TABLE customer MODIFY name CHAR(1024);
4. **CHANGE COLUMN**
  1. **Rename column name.**
  2. ALTER TABLE table-name CHANGE COLUMN old-col-name new-col-name new-col-datatype;
  3. e.g., ALTER TABLE customer CHANGE COLUMN name customer-name VARCHAR(1024);
5. **DROP COLUMN**
  1. **Drop a column completely.**
  2. ALTER TABLE table-name DROP COLUMN col-name;
  3. e.g., ALTER TABLE customer DROP COLUMN middle-name;
6. **RENAME**
  1. **Rename table name itself.**
  2. ALTER TABLE table-name RENAME TO new-table-name;
  3. e.g., ALTER TABLE customer RENAME TO customer-details;

## DATA MANIPULATION LANGUAGE (DML)

### 1. INSERT

1. INSERT INTO table-name(col1, col2, col3) VALUES (v1, v2, v3), (val1, val2, val3);

### 2. UPDATE

1. UPDATE table-name SET col1 = 1, col2 = 'abc' WHERE id = 1;
2. Update multiple rows e.g.,
  1. UPDATE student SET standard = standard + 1;

### 3. ON UPDATE CASCADE

1. Can be added to the table while creating constraints. Suppose there is a situation where we have two tables such that primary key of one table is the foreign key for another table. if we update the primary key of the first table then using the ON UPDATE CASCADE foreign key of the second table automatically get updated.

### 3. DELETE

1. DELETE FROM table-name WHERE id = 1;
2. DELETE FROM table-name; //all rows will be deleted.
3. **DELETE CASCADE - (to overcome DELETE constraint of Referential constraints)**
  1. What would happen to child entry if parent table's entry is deleted?
  2. CREATE TABLE ORDER (  
order\_id int PRIMARY KEY,  
delivery\_date DATE,  
cust\_id INT,



```
FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE CASCADE
);
```

### 3. ON DELETE NULL - (can FK have null values?)

```
1. CREATE TABLE ORDER (
    order_id int PRIMARY KEY,
    delivery_date DATE,
    cust_id INT,
    FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE SET NULL
);
```

### 4. REPLACE

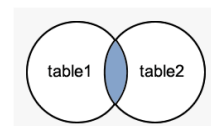
1. Primarily used for already present tuple in a table.
2. As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
3. As INSERT, if there is no duplicate data new tuple will be inserted.
4. REPLACE INTO student (id, class) VALUES(4, 3);
5. REPLACE INTO table SET col1 = val1, col2 = val2;

## JOINING TABLES

1. All **RDBMS** are relational in nature, we refer to other tables to get meaningful outcomes.
2. FK are used to do reference to other table.

### 3. INNER JOIN

1. Returns a resultant table that has matching values from both the tables or all the tables.
2. SELECT column-list FROM table1 INNER JOIN table2 ON condition1  
INNER JOIN table3 ON condition2  
...;



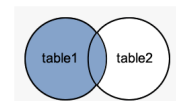
### 3. Alias in MySQL (AS)

1. Aliases in MySQL is used to give a temporary name to a table or a column in a table for the purpose of a particular query. It works as a nickname for expressing the tables or column names. It makes the query short and neat.
2. SELECT col\_name AS alias\_name FROM table\_name;
3. SELECT col\_name1, col\_name2,... FROM table\_name AS alias\_name;

### 4. OUTER JOIN

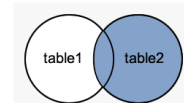
#### 1. LEFT JOIN

1. This returns a resulting table that all the data from left table and the matched data from the right table.
2. SELECT columns FROM table LEFT JOIN table2 ON Join\_Condition;



#### 2. RIGHT JOIN

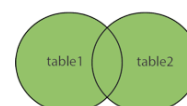
1. This returns a resulting table that all the data from right table and the matched data from the left table.
2. SELECT columns FROM table RIGHT JOIN table2 ON join\_cond;



#### 3. FULL JOIN

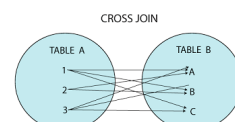
1. This returns a resulting table that contains all data when there is a match on left or right table data.
2. **Emulated** in MySQL using LEFT and RIGHT JOIN.
3. LEFT JOIN UNION RIGHT JOIN.
4. SELECT columns FROM table1 as t1 LEFT JOIN table2 as t2 ON t1.id = t2.id  
UNION  
SELECT columns FROM table1 as t1 RIGHT JOIN table2 as t2 ON t1.id = t2.id;
5. UNION ALL, can also be used this will duplicate values as well while UNION gives unique values.

FULL OUTER JOIN



### 5. CROSS JOIN

1. This returns all the cartesian products of the data present in both tables. Hence, all possible variations are reflected in the output.
2. Used rarely in practical purpose.
3. Table-1 has 10 rows and table-2 has 5, then resultant would have 50 rows.
4. SELECT column-lists FROM table1 CROSS JOIN table2;



### 6. SELF JOIN



1. It is used to get the output from a particular table when the same table is joined to itself.
2. Used very less.
3. Emulated using INNER JOIN.
4. `SELECT columns FROM table as t1 INNER JOIN table as t2 ON t1.id = t2.id;`

#### 7. Join without using join keywords.

1. `SELECT * FROM table1, table2 WHERE condition;`
2. e.g., `SELECT artist_name, album_name, year_recorded FROM artist, album WHERE artist.id = album.artist_id;`

### SET OPERATIONS

1. Used to combine multiple select statements.
2. Always gives distinct rows.

JOIN	SET Operations
Combines multiple tables based on matching condition.	Combination is resulting set from two or more SELECT statements.
Column wise combination.	Row wise combination.
Data types of two tables can be different.	Datatypes of corresponding columns from each table should be the same.
Can generate both distinct or duplicate rows.	Generate distinct rows.
The number of column(s) selected may or may not be the same from each table.	The number of column(s) selected must be the same from each table.
Combines results horizontally.	Combines results vertically.

#### 3. UNION

1. Combines two or more SELECT statements.
2. `SELECT * FROM table1  
UNION  
SELECT * FROM table2;`
3. Number of column, order of column must be same for table1 and table2.

#### 4. INTERSECT

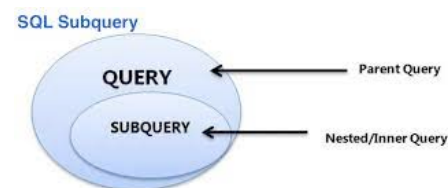
1. Returns common values of the tables.
2. Emulated.
3. `SELECT DISTINCT column-list FROM table-1 INNER JOIN table-2 USING(join_cond);`
4. `SELECT DISTINCT * FROM table1 INNER JOIN table2 ON USING(id);`

#### 5. MINUS

1. This operator returns the distinct row from the first table that does not occur in the second table.
2. Emulated.
3. `SELECT column_list FROM table1 LEFT JOIN table2 ON condition WHERE table2.column_name IS NULL;`
4. e.g., `SELECT id FROM table-1 LEFT JOIN table-2 USING(id) WHERE table-2.id IS NULL;`

### SUB QUERIES

1. Outer query depends on inner query.
2. Alternative to joins.
3. Nested queries.
4. `SELECT column_list(s) FROM table_name WHERE column_name OPERATOR (SELECT column_list(s) FROM table_name [WHERE]);`
5. e.g., `SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table1);`
6. Sub queries exist mainly in 3 clauses
  1. Inside a WHERE clause.







2. Inside a FROM clause.
3. Inside a SELECT clause.

#### 7. Subquery using FROM clause

1. SELECT MAX(rating) FROM (SELECT \* FROM movie WHERE country = 'India') as temp;

#### 8. Subquery using SELECT

1. SELECT (SELECT column\_list(s) FROM T\_name WHERE condition), columnList(s) FROM T2\_name WHERE condition;

#### 9. Derived Subquery

1. SELECT columnLists(s) FROM (SELECT columnLists(s) FROM table\_name WHERE [condition]) as new\_table\_name;

#### 10. Co-related sub-queries

1. With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

```
SELECT column1, column2, ....  
FROM table1 as outer  
WHERE column1 operator  
      (SELECT column1, column2  
        FROM table2  
        WHERE expr1 =  
              outer.expr2);
```

### JOIN VS SUB-QUERIES

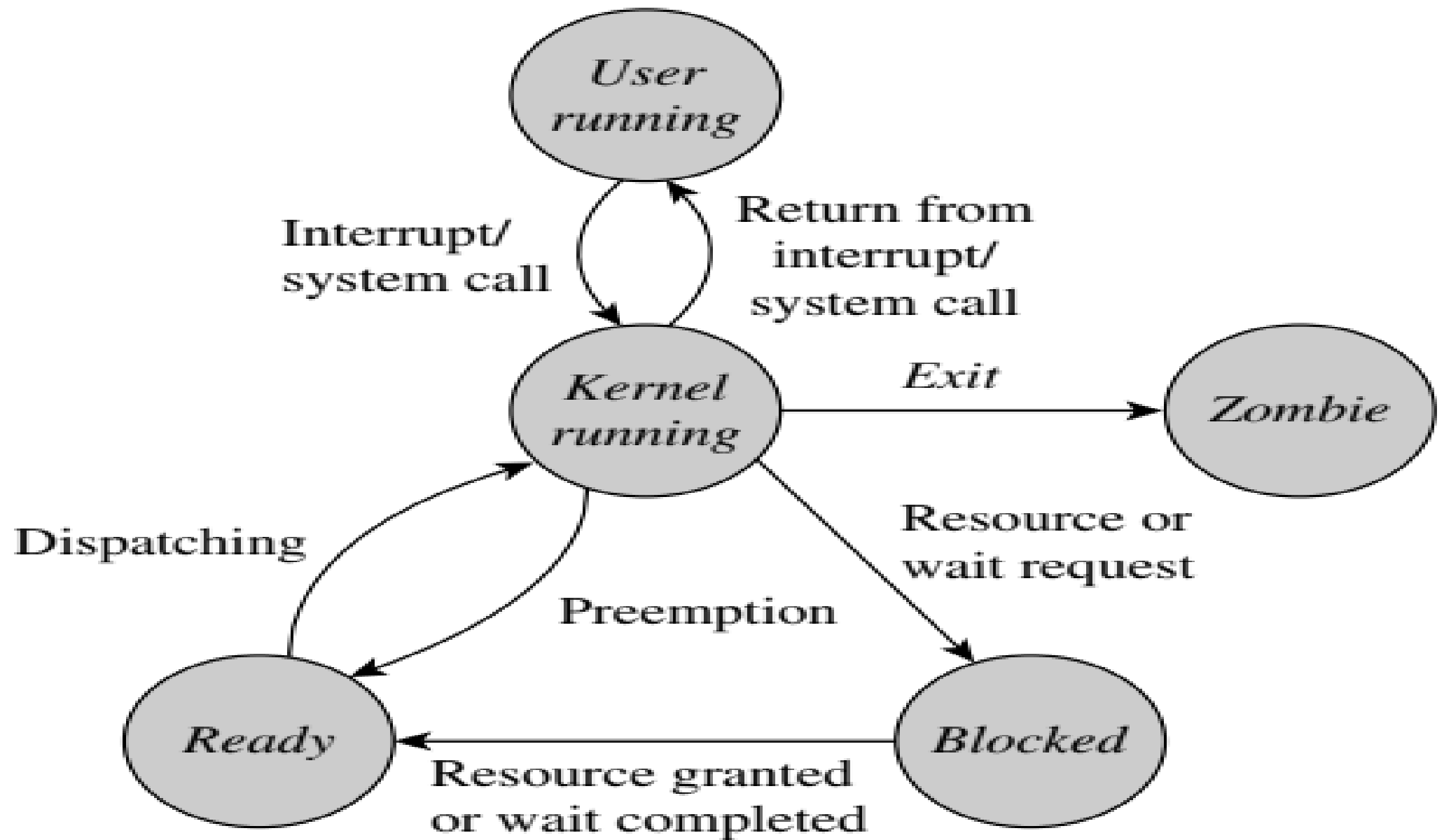
JOINS	SUBQUERIES
Faster	Slower
Joins maximise calculation burden on DBMS	Keeps responsibility of calculation on user.
Complex, difficult to understand and implement	Comparatively easy to understand and implement.
Choosing optimal join for optimal use case is difficult	Easy.

### MySQL VIEWS

1. A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table.
2. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own.
3. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.
4. CREATE VIEW view\_name AS SELECT columns FROM tables [WHERE conditions];
5. ALTER VIEW view\_name AS SELECT columns FROM table WHERE conditions;
6. DROP VIEW IF EXISTS view\_name;
7. CREATE VIEW Trainer AS SELECT c.course\_name, c.trainer, t.email FROM courses c, contact t WHERE c.id = t.id; (View using Join clause).

NOTE: We can also import/export table schema from files (.csv or json).





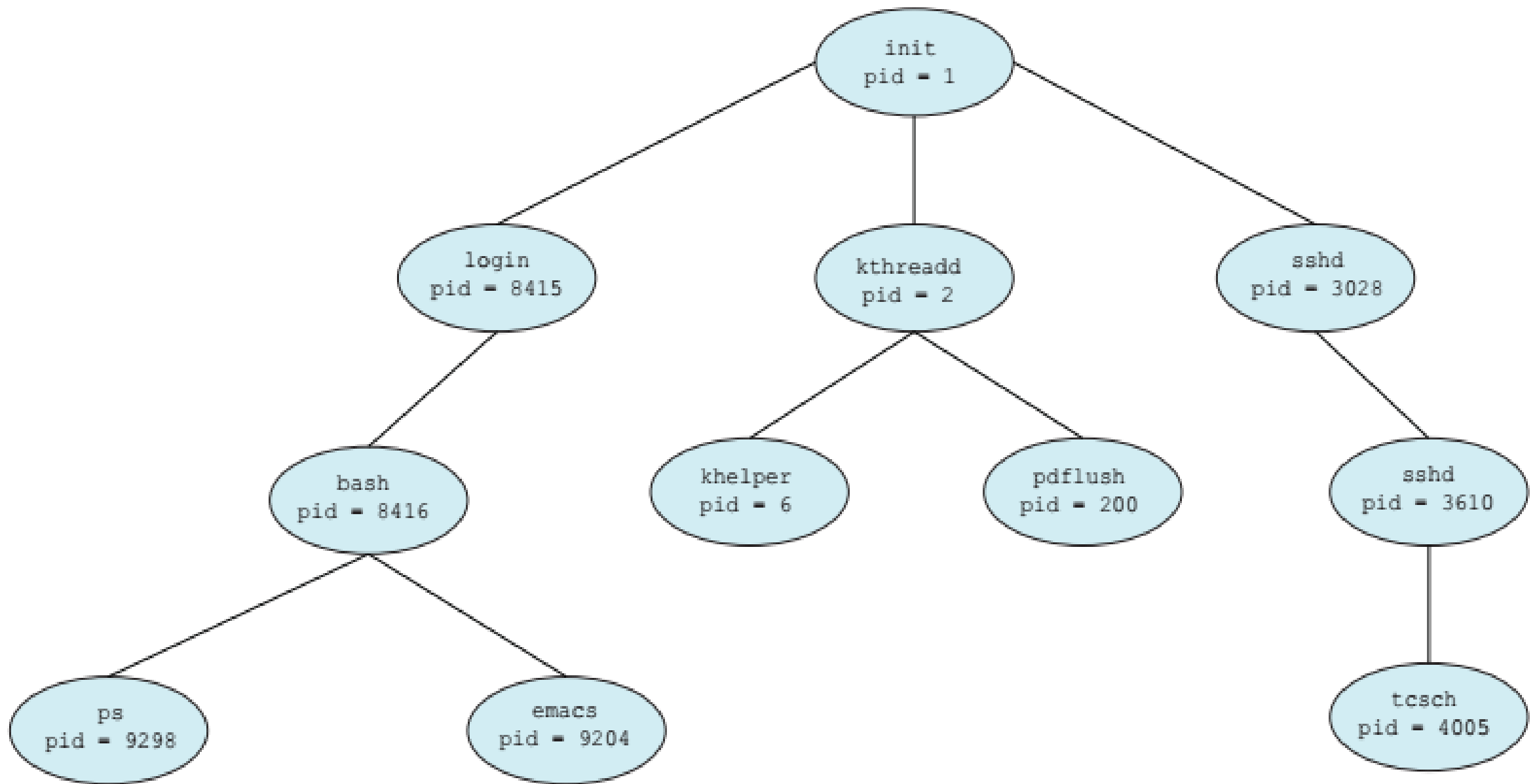
Process state transitions in Unix.

# Process States and State Transitions

- There is one conceptual difference between the process model described earlier and that used in Unix.
- In the previous model, a process in the *running* state is put in the *ready* state the moment its execution is interrupted.
- A system process then handles the event that caused the interrupt.
- If the running process had itself caused a software interrupt by executing an *<SI\_instrn>*, its state may further change to *blocked* if its request cannot be granted immediately.
- In this model a user process executes only user code; it does not need any special privileges.
- A system process may have to use privileged instructions like I/O initiation and setting of memory protection information, so the system process executes with the CPU in the kernel mode.

- Processes behave differently in the Unix model.
- When a process makes a system call, the process itself proceeds to execute the kernel code meant to handle the system call.
- To ensure that it has the necessary privileges, it needs to execute with the CPU in the kernel mode.
- A mode change is thus necessary every time a system call is made.
- The opposite mode change is necessary after processing a system call.
- Similar mode changes are needed when a process starts executing the interrupt servicing code in the kernel because of an interrupt, and when it returns after servicing an interrupt.
- Unix uses two distinct *running* states.
- These states are called *user running* and *kernel running* states.
- A user process executes user code while in the *user running* state, and kernel code while in the *kernel running* state.

- It makes the transition from *user running* to *kernel running* when it makes a system call, or when an interrupt occurs.
- It may get blocked while in the *kernel running* state because of an I/O operation or non availability of a resource.
- When the I/O operation completes or its resource request is granted, the process returns to the *kernel running* state and completes the execution of the kernel code that it was executing.
- It now leaves the kernel mode and returns to the user mode.
- Accordingly, its state is changed from *kernel running* to *user running*.
- Because of this arrangement, a process does not get blocked or preempted in the *user running* state—it first makes a transition to the *kernel running* state and then gets blocked or preempted.
- In fact, *user running* → *kernel running* is the only transition out of the *user running* state.
- Process termination occurs when a process is in the *kernel running* state. This happens because the process executes the system call *exit* while in the *user running* state.
- This call changes its state to *kernel running*. The process actually terminates and becomes a *zombie* process as a result of processing this call.



# Process Creation

- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.
- The init process (which always has a pid of 1) serves as the root parent process for all user processes.
- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like.
- In Figure there are two children of init—kthreadd and sshd.
- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel
- The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell***).
- The login process is responsible for managing clients that directly log onto the system.

- When a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process.
- Alternatively, some operating systems pass resources to child processes.
- When a process creates a new process, two possibilities for execution exist:
  1. The parent continues to execute concurrently with its children.
  2. The parent waits until some or all of its children have terminated.
- There are also two address-space possibilities for the new process:
  1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
  2. The child process has a new program loaded into it.



# Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call.
- At that point, the process may return a status value (typically an integer) to its parent process (via the `wait()` system call).
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- Termination can occur in other circumstances as well. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.
- Note that a parent needs to know the identities of its children if it is to terminate them.
- Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The child has exceeded its usage of some of the resources that it has been allocated.
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

# Process Synchronization in Unix

- Unix system V provides a kernel-level implementation of semaphores.
- The name of a semaphore is called a *key*.
- The key is actually associated with an array of semaphores, and individual semaphores in the array are distinguished with the help of subscripts.
- Processes share a semaphore by using the same key.
- A process wishing to use a semaphore obtains access to it by making a *semget* system call with a key as a parameter.
- If a semaphore array with matching key already exists, the kernel makes that array accessible to the process making the *semget* call; otherwise, it creates a new semaphore array, assigns the key to it and makes it accessible to the process.
- The kernel provides a single system call *semop* for *wait* and *signal* operations.

# Scheduling in Unix

- Unix is a pure time-sharing operating system.
- It uses a multilevel adaptive scheduling policy in which process priorities are varied to ensure good system performance and also to provide good user service.
- Processes are allocated numerical priorities, where a larger numerical value implies a lower effective priority.
- In Unix 4.3 BSD, the priorities are in the range 0 to 127.
- Processes in the user mode have priorities between 50 and 127, while those in the kernel mode have priorities between 0 and 49.
- When a process is blocked in a system call, its priority is changed to a value in the range 0–49, depending on the cause of blocking.
- When it becomes active again, it executes the remainder of the system call with this priority.
- This arrangement ensures that the process would be scheduled as soon as possible, complete the task it was performing in the kernel mode and release kernel resources.
- When it exits the kernel mode, its priority reverts to its previous value, which was in the range 50–127.

- Unix uses the following formula to vary the priority of a process:  

$$\text{Process priority} = \text{base priority for user processes} + f(\text{CPU time used recently}) + \text{nice value (i)}$$
- The scheduler maintains the CPU time used by a process in its process table entry.
- This field is initialized to 0.
- The real-time clock raises an interrupt 60 times a second, and the clock handler increments the count in the CPU usage field of the running process.
- The scheduler re computes process priorities every second in a loop.
- For each process, it divides the value in the CPU usage field by 2, stores it back, and also uses it as the value of  $f$ .
- A large numerical value implies a lower effective priority, so the second factor in Eq. (i) lowers the priority of a process.
- The division by 2 ensures that the effect of CPU time used by a process *decays*; i.e., it wears off over a period of time, to avoid the problem of starvation faced in the least completed next (LCN) policy.
- A process can vary its own priority through the last factor in Eq. (i).
- The system call “*nice(<priority value>);*” sets the *nice value* of a user process.
- It takes a zero or positive value as its argument. Thus, a process can only decrease its effective priority to be nice to other processes. It would typically do this when it enters a CPU-bound phase.

# Fair Share Scheduling

- To ensure a fair share of CPU time to groups of processes, Unix schedulers add the term  $f$  (CPU time used by processes in the group) to Eq. (i).
- Thus, priorities of all processes in a group reduce when any of them consumes CPU time.
- This feature ensures that processes of a group would receive favored treatment if none of them has consumed much CPU time recently.
- The effect of the new factor also decays over time.