**Connection Object:**

Class.forName("com.mysql.jdbc.Driver");

System.out.println("Connecting to database...");

conn = DriverManager.getConnection(DB\_URL,USER,PASS);

System.out.println("Creating statement...");

stmt = conn.createStatement();

.........

stmt.close();

conn.close();

How costly is opening and closing of a DB connection?

The server maintains several buffers for each client connection. One is used as a communications buffer for exchanging information with the client. Other buffers are maintained per client for reading tables and performing join and sort operations.

What settings govern the per-connection buffers?

join\_buffer\_size

sort\_buffer\_size

read\_buffer\_size

read\_rnd\_buffer\_size

tmp\_table\_size / max\_heap\_table\_size

net\_buffer\_length / max\_allowed\_packet

thread\_stack

It takes time to allocate and deallocate these buffers when a connection comes into being. Don't forget to multiple the sum of those values by max\_connections.

In heavy-read, heavy-write environment, such as OLTP, this would be expensive in terms of RAM usage and possible inhibition due to swapping in the OS.

if you are performing a loop to do some interaction with the database, you are going to be far better off re-using the same active connection than opening and closing within the loop (an anti-pattern that I see quite often).

So while creating and destroying database connections can be expensive, keeping them open while they're not being used can also be expensive.

there are typically a limited number of simultaneous connections permitted to the database.

connection pooling feature must address above points.

**Connection Pool**

Connection Pooling is a technique to allow multiple clients to make use of a cached set of shared and reusable connection objects providing access to a database.

why do we need it?

Opening/Closing database connections is an expensive and time-consuming operation. Creating connections over and over in complex environments (where Transactions are generally short-lived, and user sessions are often limited in time) is simply too expensive.

A connection pool operates by performing the work of creating connections ahead of time, and hence connection pools improve the performance of execution of commands on a database for which we maintain connection objects in the pool.

In the case of a JDBC connection pool, a pool of Connection objects is created at the time the application server (or some other server) starts.

It facilitates reuse of the same connection object to serve a number of client requests.

* Every time a client request is received, the pool is searched for an available connection object and if it gets the connection object is returned.
* Otherwise, either the incoming requests are queued or a new connection object is created and added to the pool.
  + Depending upon how many connections are supported.
* As soon as a request finishes using a connection object, the object is given back to the pool from where it's assigned to one of the queued requests.

Since most of the requests are served using existing connection objects only so the connection pooling approach brings down the average time required for the users to wait for establishing the connection to the database.

Connection pooling has become the standard for middleware database drivers.

These objects are then managed by a pool manager that disperses connections as they are requested by clients and returns them to the pool when it determines the client is finished with the Connection object. A great deal of housekeeping is involved in managing these connections.

When the connection pool server starts, it creates a predetermined number of Connection objects.

A client application would then perform a JNDI lookup to retrieve a reference to a DataSource object that implements the ConnectionPoolDataSource interface. The client application would not make any special provisions to use the pooled data source;

When the client application requests a connetion from the ConnetionPoolDataSource, the data source implementation would retrieve a physical connection to the client application.

the ConnectionPoolDataSource would return a Connection object that implemented the PooledConnection interface.

The PooledConnection interface dictates the use of event listeners.

These event listeners allow the connection pool manager to capture important connection events, such as attempts by the client application to close the connection.

When the driver traps a close-connection event, it intercedes and performs a pseudo-close operation that merely takes the Connection object, returns it to the pool of available connection, and performs any housekeeping that is necessary.

The operation of the connection pool should be completely transparent to the client application.

The triggering of connection events, the manipulation of the object pool, and the creation and destruction of physical connections are all managed by the pool manager.

The activities of the connection pool are, however, configurable by the application developer or the application deployer.

Steps :

Keep two list

availableConnections = total available coonection in pool;

busyConnections = total connection in use.

so total connections = availableConnections.size + busyConnections.size

If client request for connection do the following

if connection is available in availableConnections then remove the conncetion and add it to busyConnections and provide the connection.

else (if max limit is not reached) create new connection and add it to availableConnections pool and wait till connection is made and on notify retry above steps.

When connection is released simply remove it from busyConnections and put it into availableConnections.