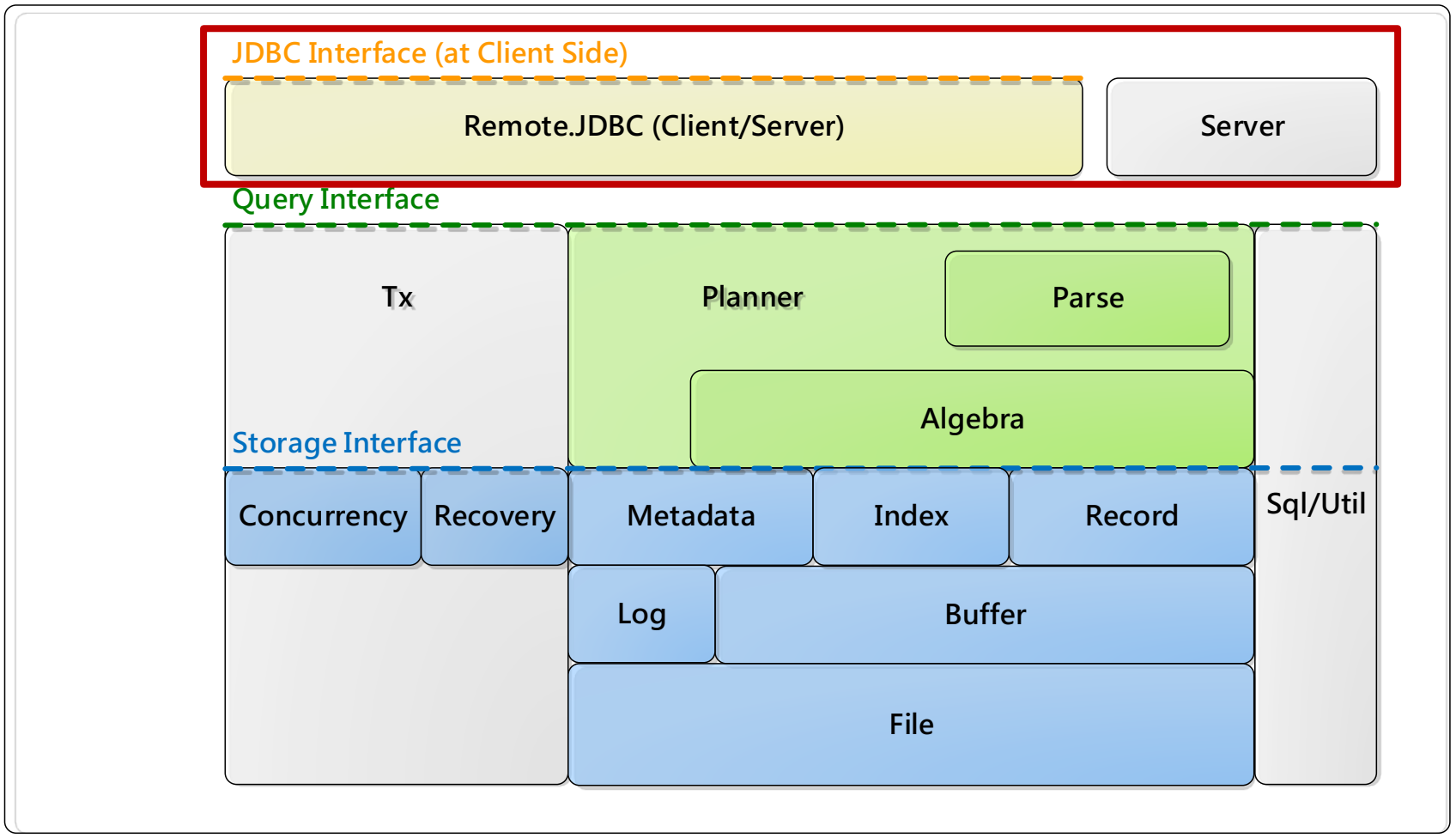
Server and Threads

Shan Hung Wu & DataLab
CS, NTHU

Where are we?

VanillaCore



Before Diving into Engines…

- How does the an RDBMS run?
  - How many processes?
  - How many threads?
  - Thread-local or thread-safe components?
  - Difference between running embedded clients and remote clients?
- Answers may influence the software architecture as well as performance

Outline

- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Outline

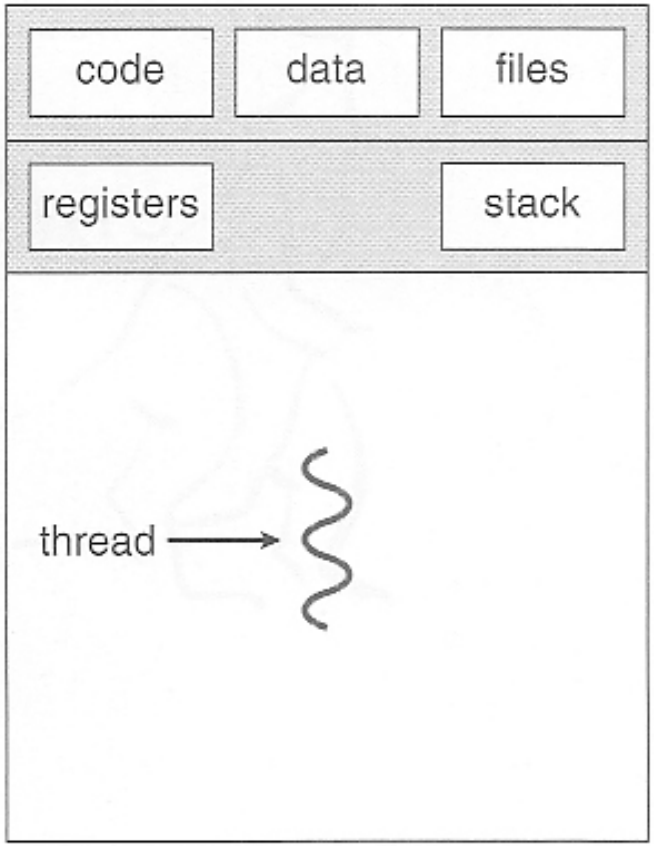
- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

What’s difference between a process and a thread?

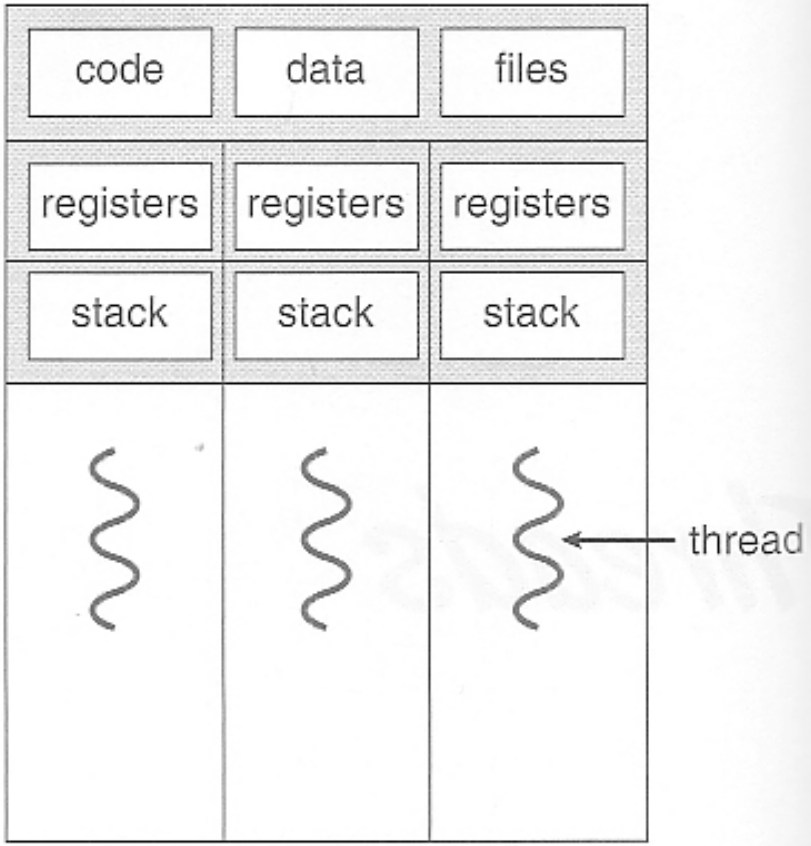
Process vs. Thread (1/2)

- Thread = a unit of CPU execution + local resources
  - E.g., program counter, registers, function call stack, etc.
- Process = threads (at least one) + global resources
  - E.g., memory space/heap, ***opened files***, etc.

Process vs. Thread (2/2)



single-threaded process

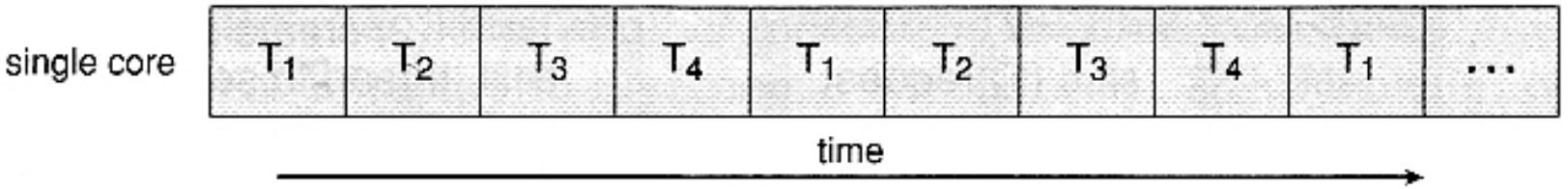


multithreaded process

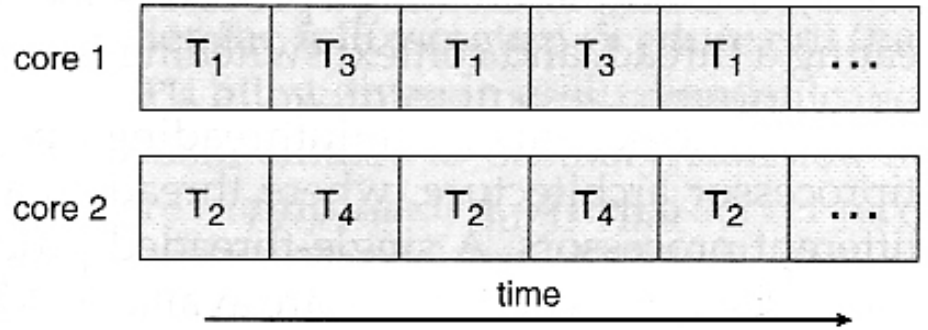
What’s difference between a kernel thread and a user thread?

Kernel Threads

- Scheduled by OS
  - On signel-core machines:



  - On multi-core machines:



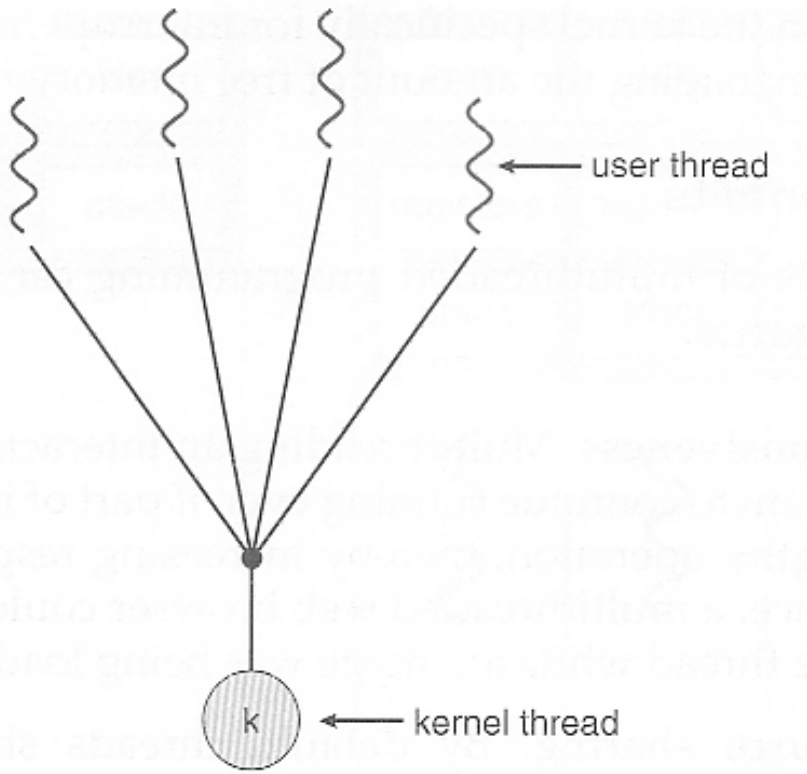
  - Examples: POSIX Pthreads (UNIX), Win32 threads

User Threads

- Scheduled by user applications (in user space above the kernel)
  - Lightweight -> faster to create/destroy
  - Examples: POSIX Pthreads (UNIX), Java threads
- Eventually mapped to kernel threads
  - How?

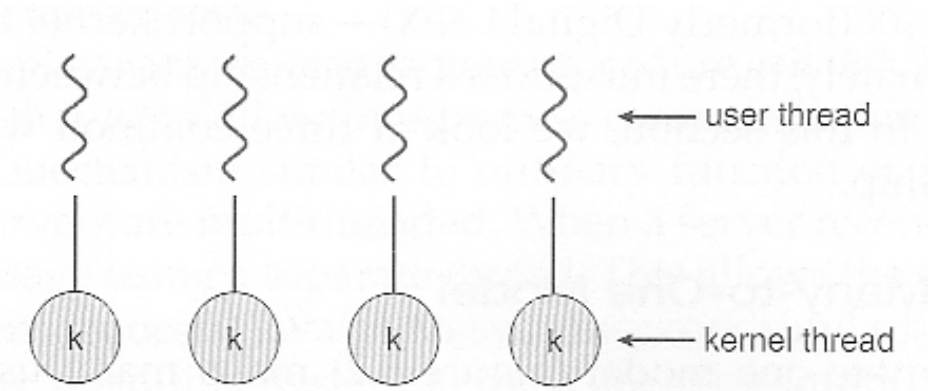
Many-to-One

- Pros:
  - Simple
  - Efficient thread mgr.
- Cons:
  - One blocking system call makes all threads halt
  - Cannot run across multiple CPU cores (each kernel thread runs on only one core)
- Examples:
  - Green threads in Solaris, seldom used in modern OS



One-to-One

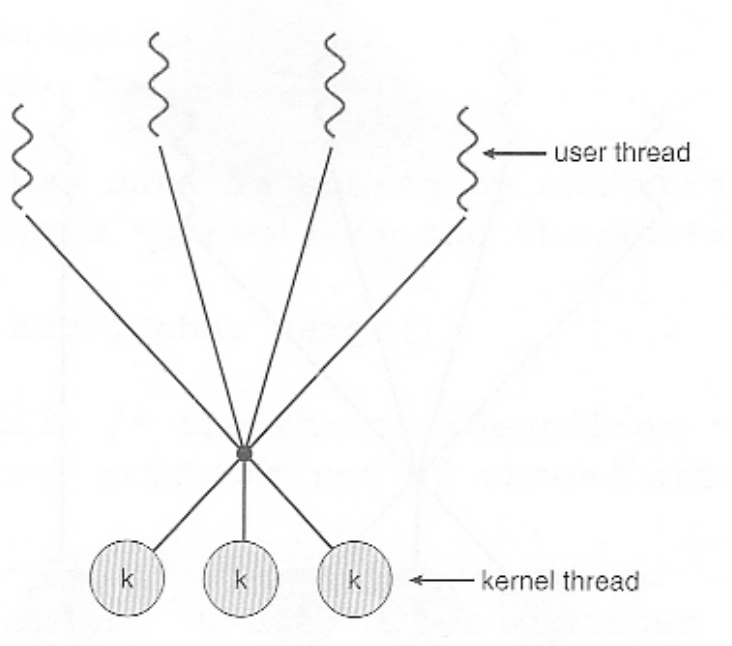
- Pros:
  - Avoid the blocking problem
- Cons:
  - Slower thread mgr.



- Most OSs limit the number of kernel threads to be mapped for a process
- Examples: Linux and Windows (from 95)

Many-to-Many

- Combining the best features of the one-to-one and many-to-one
- Allowing more kernel threads for a heavy user thread
- Examples: IRIX, HP-UX, ru64, and Solaris (prior to 9)
  - Downgradable to one-to-one



How about Java threads?

Java Threads

- Scheduled by JVM
- Mapping depends on the JVM implementation
  - But normally one-to-one mapped to Pthreads/Win32 threads on UNIX/Windows
- Pros over POSIX (one2one) threads:
  - System independent (if there's a JVM)

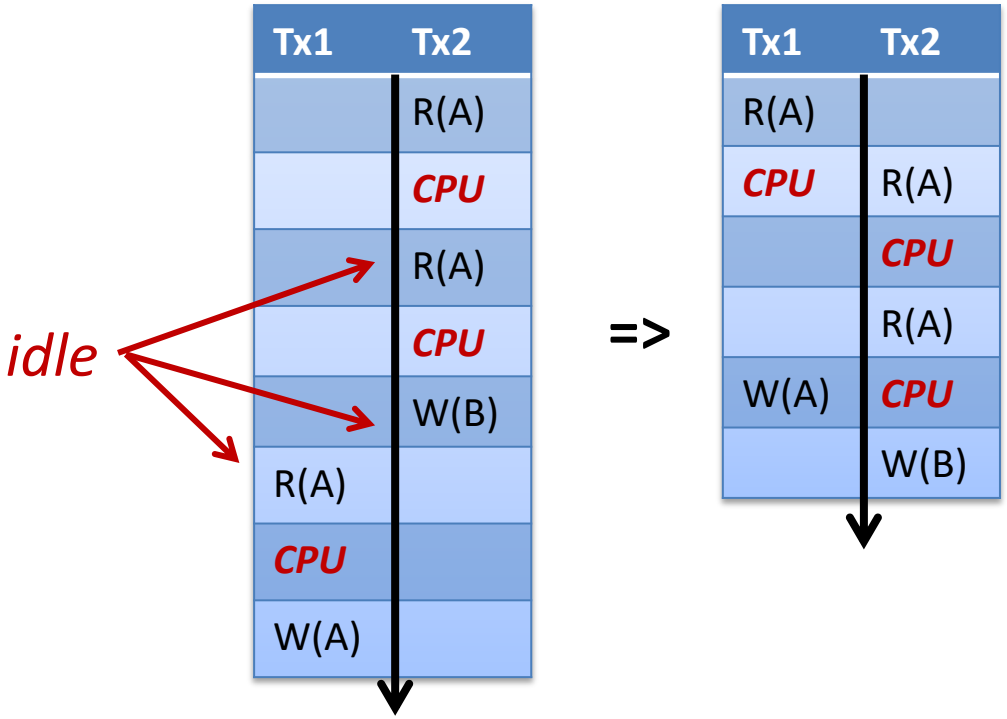
Outline

- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Why does an RDBMS support concurrent statements/txs?

Serialized or interleaved operations?

Throughput via Pipelining



- Interleaving ops increases throughput by ***pipelining CPU and I/O***

Statements run by processes or threads?

Processes vs. Threads

- DBMS is about resource management

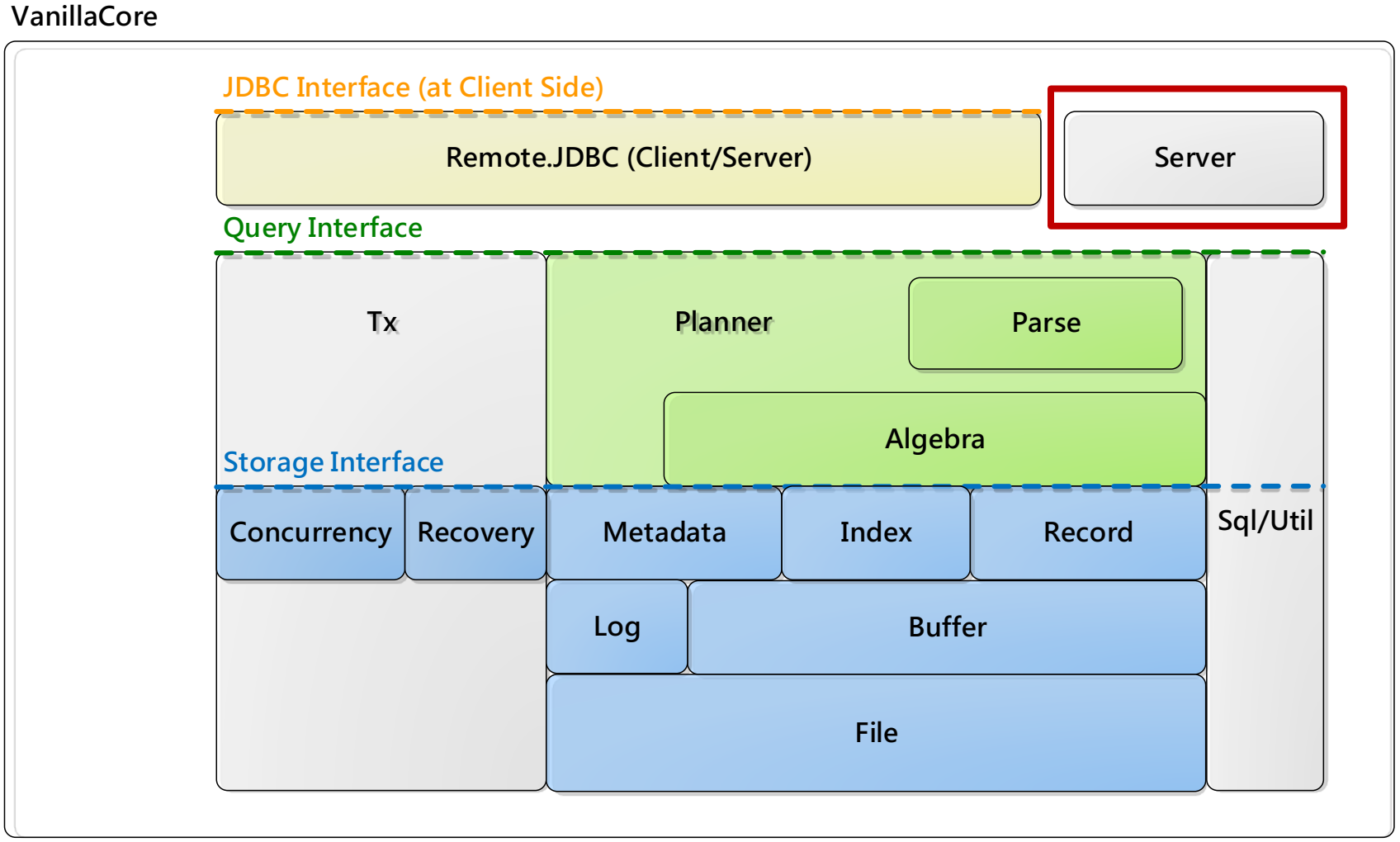
- If statements are run by process, then we need inter-process communications
  - When, e.g., two statements access the same table (file)
  - System dependent
- Threads allows global resources to be shared directly
  - E.g., through argument passing or ***static variables***

What Resources to Share?

- Opened files
- Buffers (to cache pages)
- Logs
- Locks of objects (incl. files/blocks/record locks)
- Metadata

- Example: VanillaCore

Architecture of VanillaCore



`VanillaDb` (1/2)

- Provides access to global resources:
  - `FileMgr`, `BufferMgr`, `LogMgr`, `CatalogMgr`
- Creates the new objects that access global resources:
  - `Planner` and `Transaction`

`VanillaDb` (2/2)

- Before using the VanillaCore, the `VanillaDb.init(name)` must be called
  - Initialize file, log, buffer, metadata, and tx mgrs
  - Create or recover the specified database

Outline

- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Embedded Clients

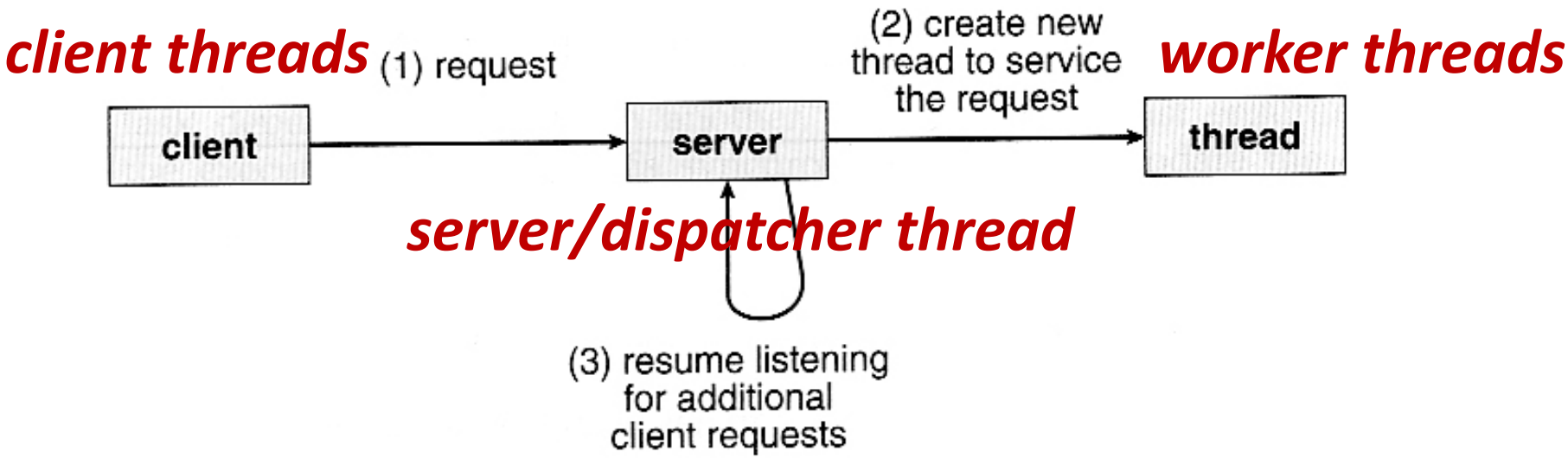
- Running on the same machine as RDBMS
- Usually single-threaded
  - E.g., sensor nodes, dictionaries, phone apps, etc.
- If you need high throughput, manage threads yourself
  - Identify causal relationship between statements
  - Run each group of causal statements in a thread
  - No causal relationship between the results outputted by different groups

Outline

- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Remote Clients

- Server (thread) creates worker threads



- One worker thread per request
- Each client can be multi-threaded
  - E.g., a web/application server

What is a request?

- An I/O operation?
- A statement?
- A transaction?
- A connection?

Request = Connection

- In VanillaDB, a worker thread handles all statements issued by the same user
- Rationale:
  - Statements issued by a user are usually in a causal order → ensure casualty in a session
  - A user may re-examine the data he/shed accessed → easier caching
- Implications:
  - All statements issued in a JDBC connection is run by a single thread at server
  - #connections = #threads

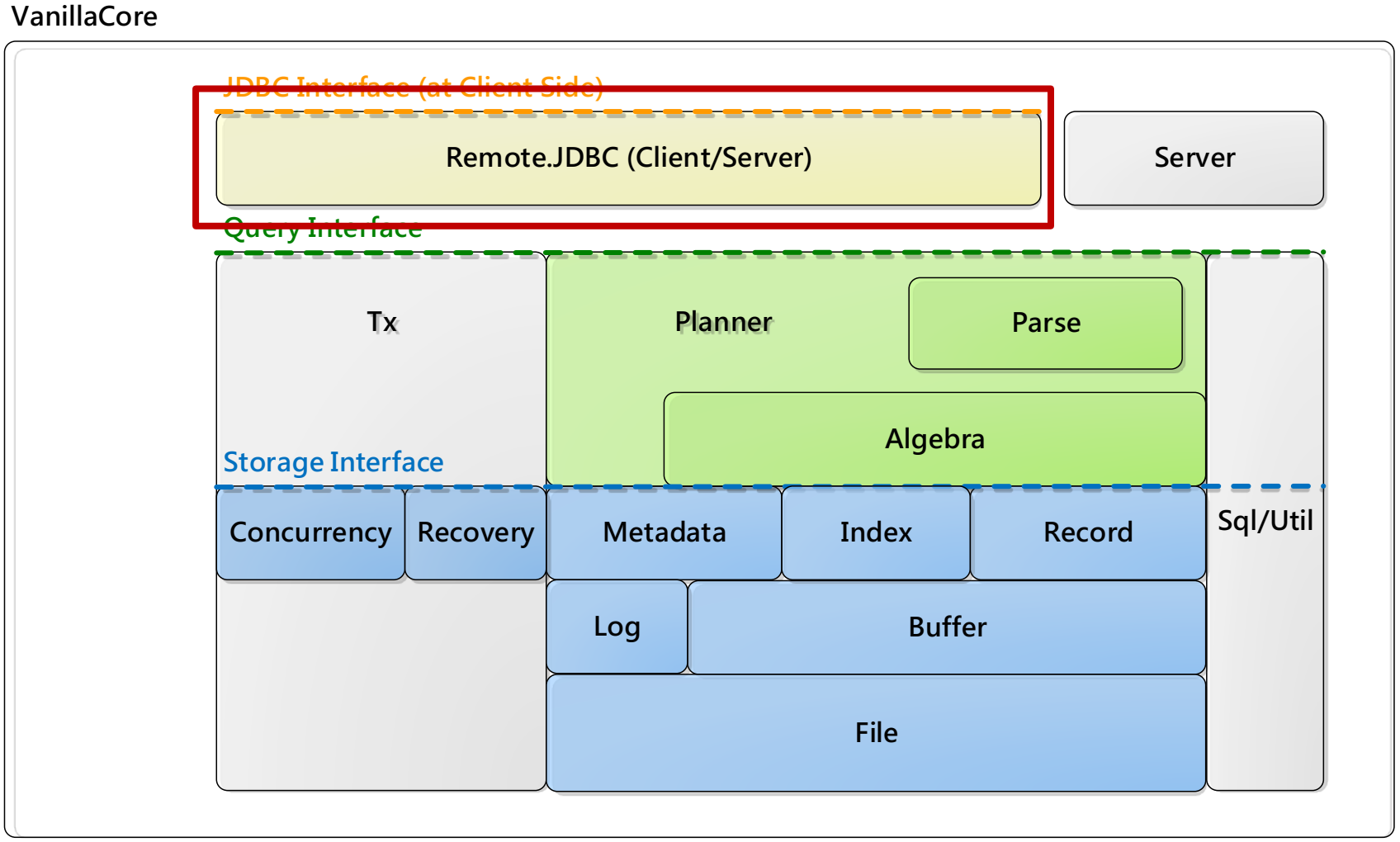
Thread Pooling

- Creating/destroying a thread each time upon connection/disconnection leads to large overhead
- To reduce this overhead, a worker ***thread pool*** is commonly used
  - Threads are allocated from the pool as needed, and returned to the pool when no longer needed
  - When no threads are available in the pool, the client may have to wait until one becomes available
- Other benefit?
- Graceful performance degradation by limiting the pool size

Outline

- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

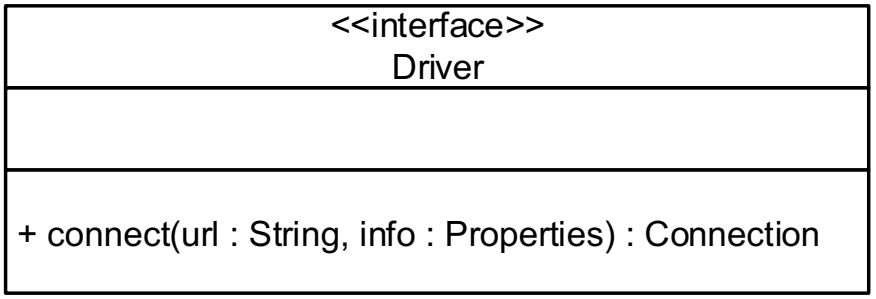
Architecture of VanillaCore

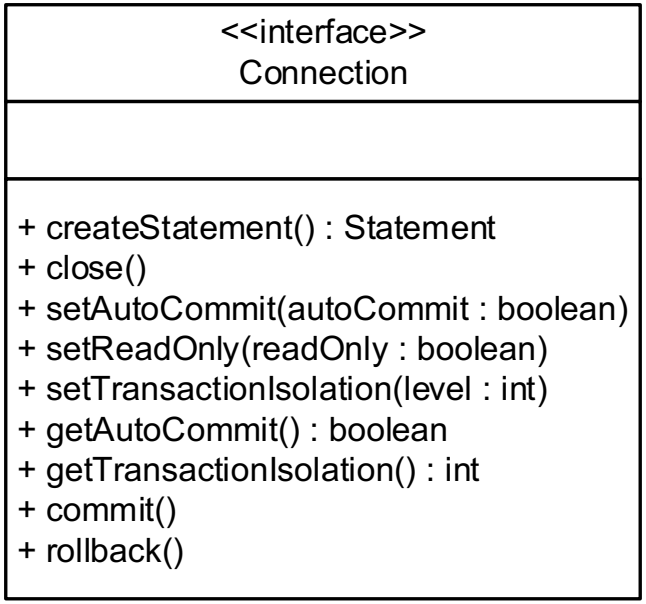


JDBC Programming

1. Connect to the server
2. Execute the desired query
3. Loop through the result set (for SELECT only)
4. ***Close*** the connection
   - A result set ties up valuable resources on the server, such as buffers and locks
   - Client should close its connection as soon as the database is no longer needed

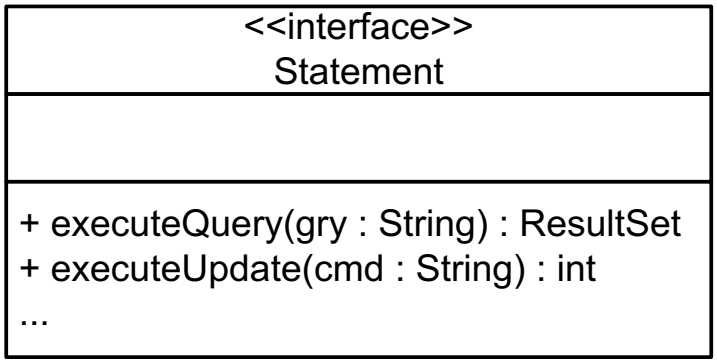
java.sql (1/2)

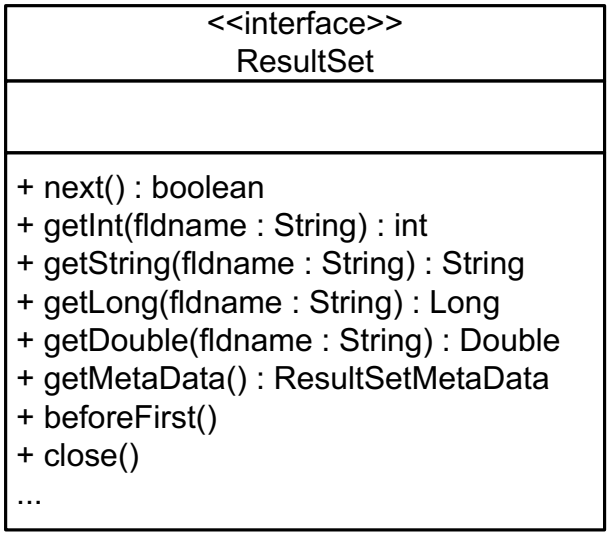




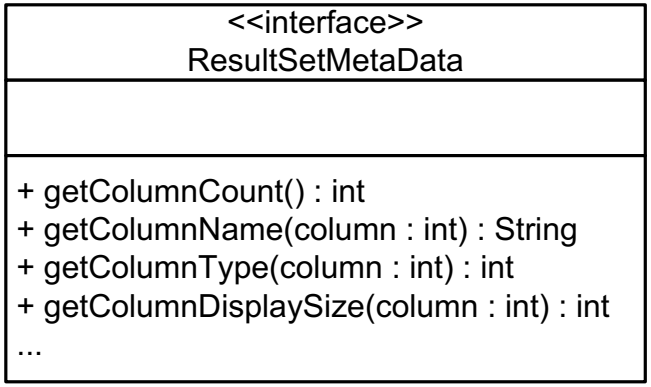
- Makes connections to the server

java.sql (2/2)





- An iterator of output records



Implementing JDBC in VanillaCore

- JDBC API is defined ***at client side***
- Needs both client- and server-side implementations
  - In `org.vanilladb.core.remote.jdbc` package
  - `JdbcXxx` are client-side classes
  - `RemoteXxx` are server-side classes
- Based on Java RMI
  - Handles server threading: dispatcher thread, worker threads, and thread pool
  - But no control to pool size
  - ***Synchronizes*** a client thread with a worker thread
    - Blocking method calls at clients

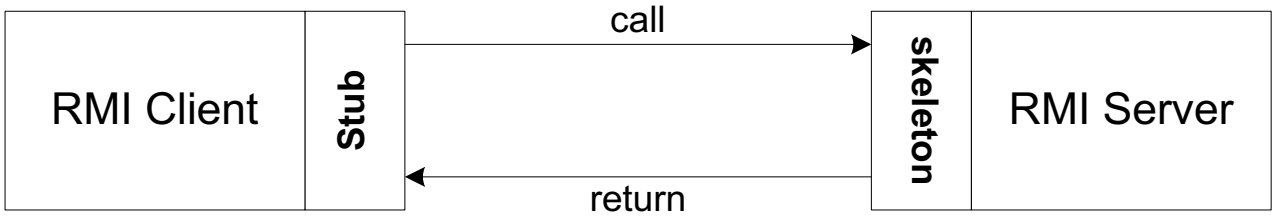
Outline

- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Java RMI

- Java RMI allows methods of an object at server VM to be invoked remotely at a client VM
  - We call this object **a *remote object***
- How?

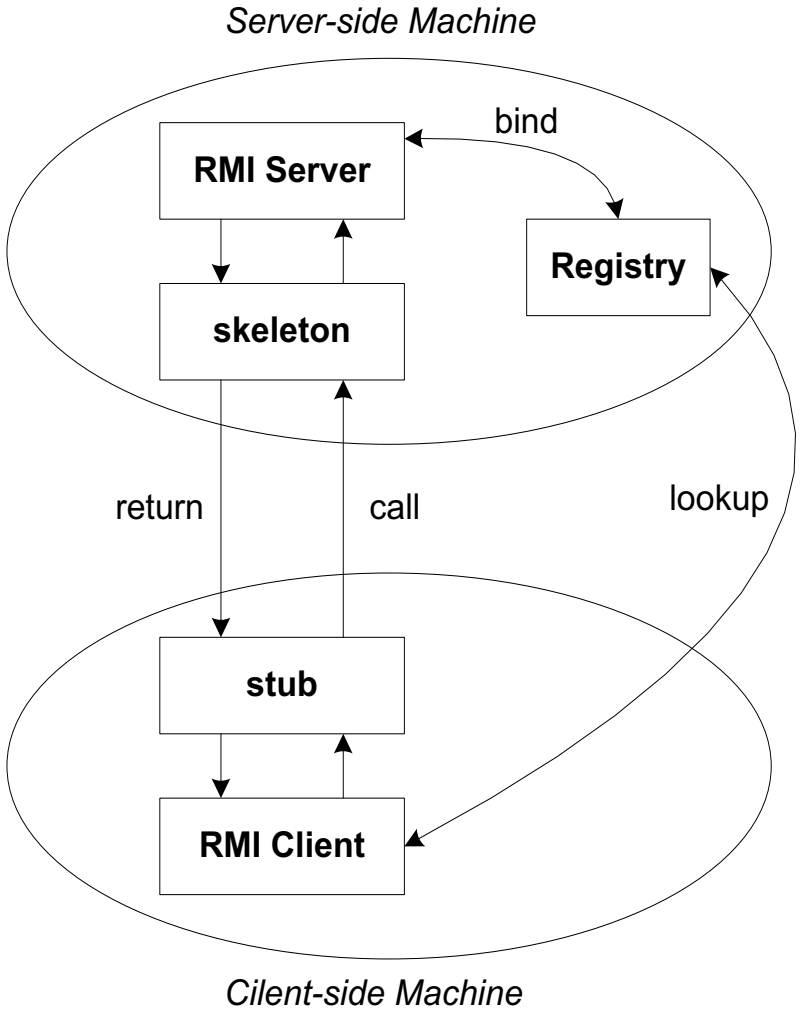
The Stub and Skeleton



1. The ***skeleton*** (run by a server thread) binds the interface of the remote object
2. A client thread looks up and obtain a ***stub*** of the skeleton
3. When a client thread invokes a method, it is blocked and the call is first forwarded to the stub
4. The stub marshals the parameters and sends the call to the skeleton through the network
5. The skeleton receives the call, unmarshals the parameters, allocates from pool a worker thread that runs the remote object's method on behalf of the client
6. When the method returns, the worker thread returns the result to skeleton and returns to pool
7. The skeleton marshals the results and send it to stub
8. The stub unmarshals the results and continues the client thread

RMI registry

- The server must first bind the remote obj's interface to the registry with a name
  - The interface must extend the `java.rml.Remote` interface
- The client lookup the name in the registry to obtain a stub



Things to Note

- A client thread and a worker thread is synchronized
- The same remote object is run by multiple worker threads (each per client)
  - ***Remote objects bound to registry must be thread-safe***
- If the return of a remote method is another remote object, the stub of that object is created automatically and sent back to the client
  - That object can be either thread-local or thread-safe, depending on whether it is created or reused during each method call
- A remote object will ***not*** be garbage collected if there's a client holding its stub
  - Destroy stub (e.g., closing connection) at client side ASAP

Outline

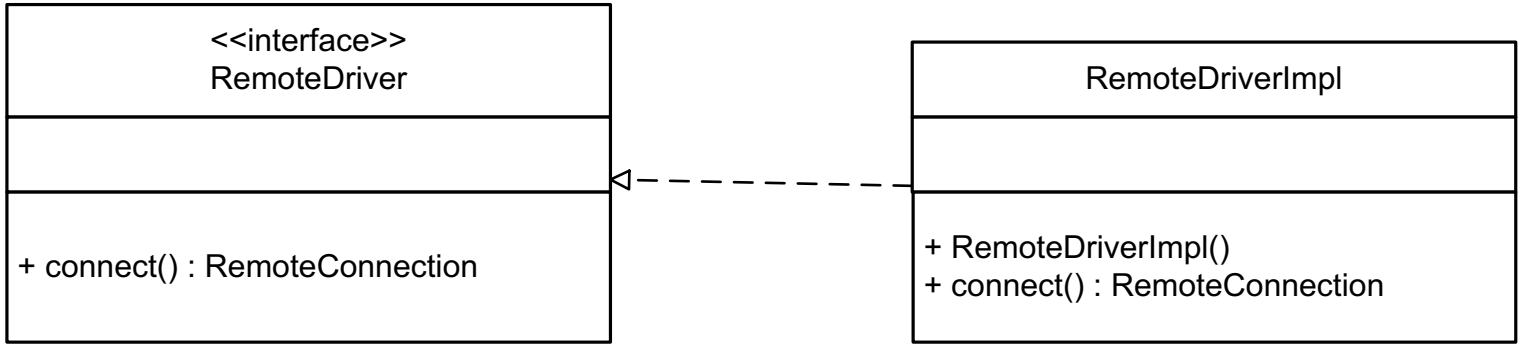
- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Server-Side JDBC Impl.

- `RemoteXxx` classes that mirror their corresponding ***JDBC interfaces*** at client-side
  - Implement the most essential JDBC methods only
- Interfaces: `RemoteDriver`, `RemoteConnection`, `RemoteStatement`, `RemoteResultSet` and `RemoteMetaData`
  - To be bound to registry
  - Extend `java.rml.Remote`
  - Throw `RemoteException` instead of `SQLException`

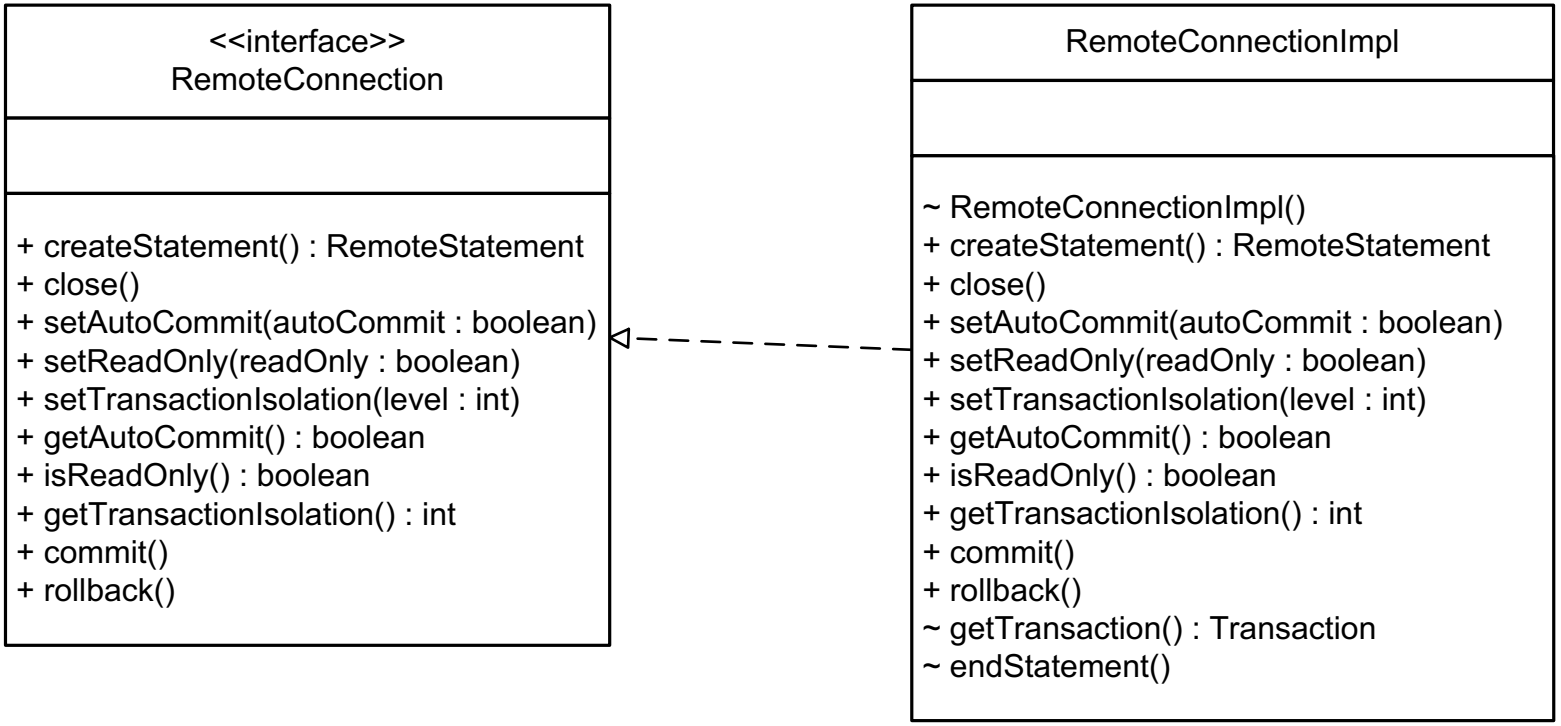
`RemoteDriver`

- Corresponds to the JDBC `Driver` interface



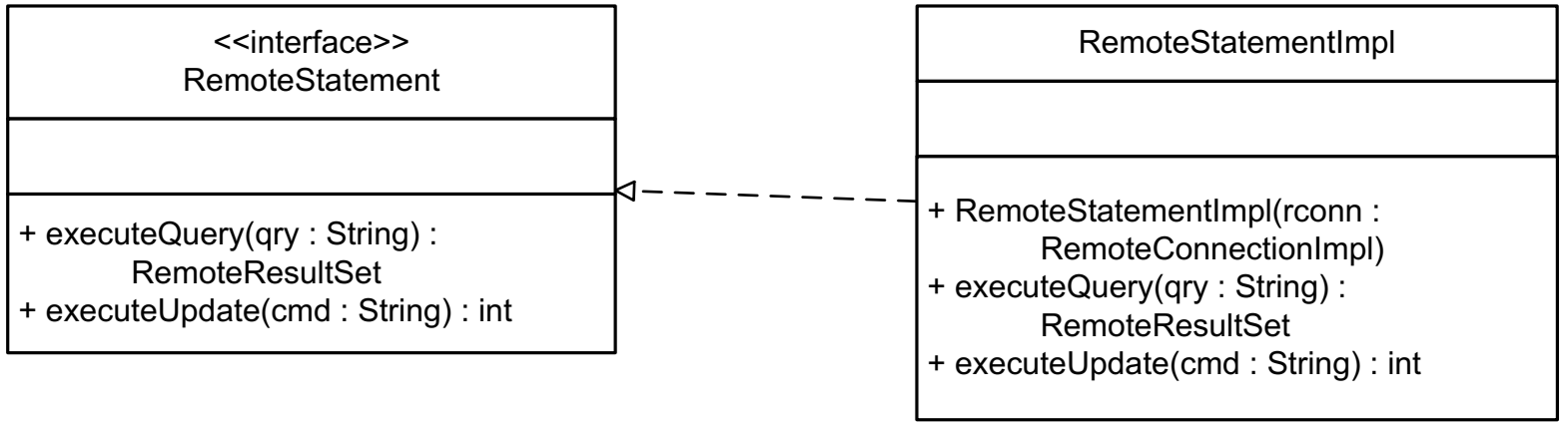
RemoteConnection

- Corresponds to JDBC `Connection` interface



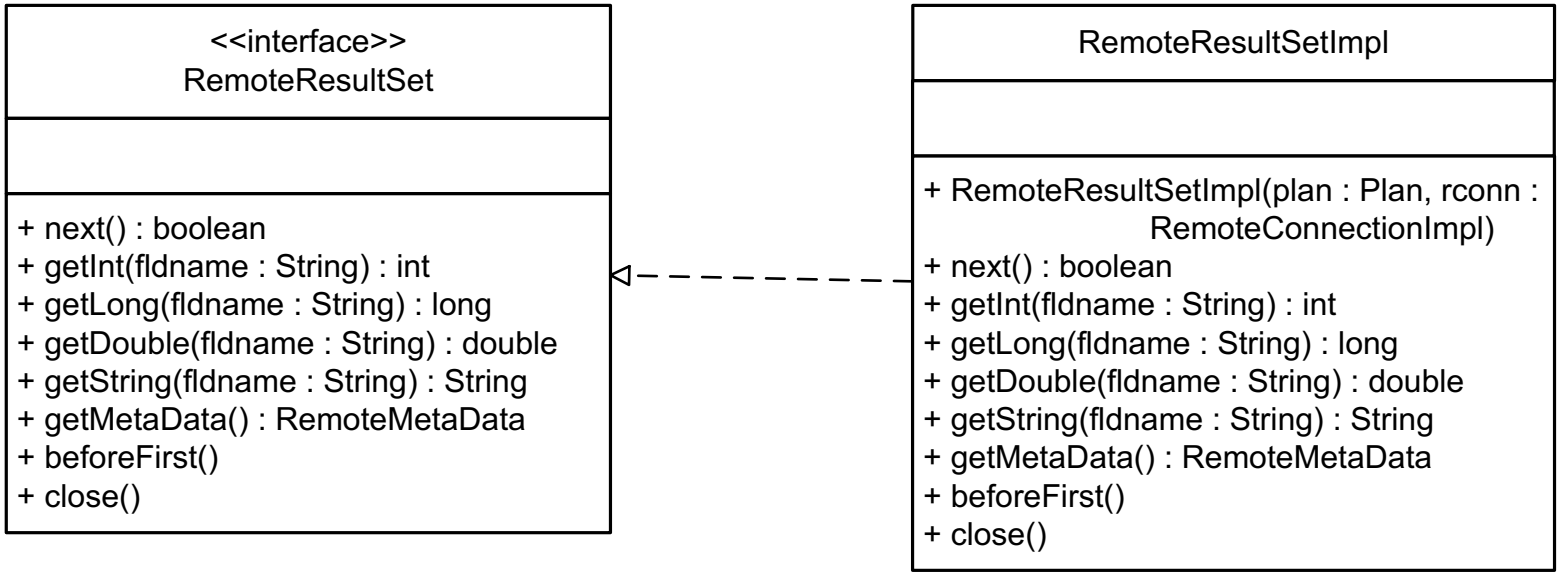
RemoteStatement

- Corresponds to JDBC `Statement` interface



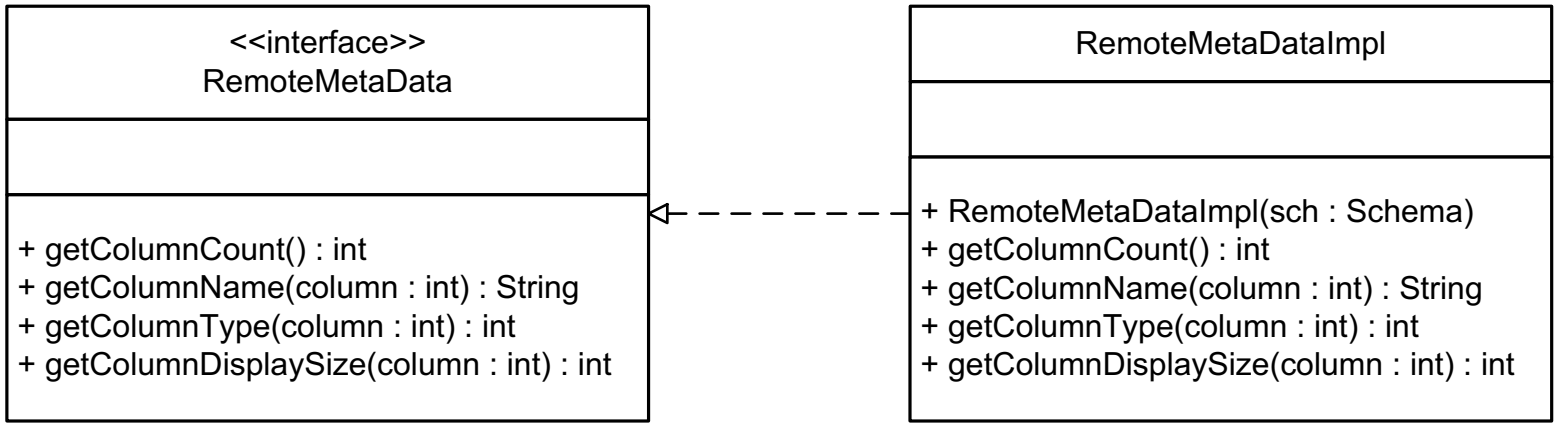
RemoteResultSet

- Corresponds to JDBC `ResultSet` interface



RemoteMetaData

- Corresponds to JDBC `ResultSetMetaData` interface



Registering Remote Objects

- Only the `RemoteDriver` need to be bound to registry
  - Stubs of others can be obtained by method returns
- Done by `JdbcStartUp`:

```
/* create a registry specific for the server on
        the default port 1099 */
Registry reg = LocateRegistry.createRegistry(1099);

// post the server entry in it
RemoteDriver d = new RemoteDriverImpl();

/* create a stub for the remote implementation object d,
        save it in the RMI registry */
reg.rebind("vanilladb-jdbc", d);
```

Obtaining Stubs

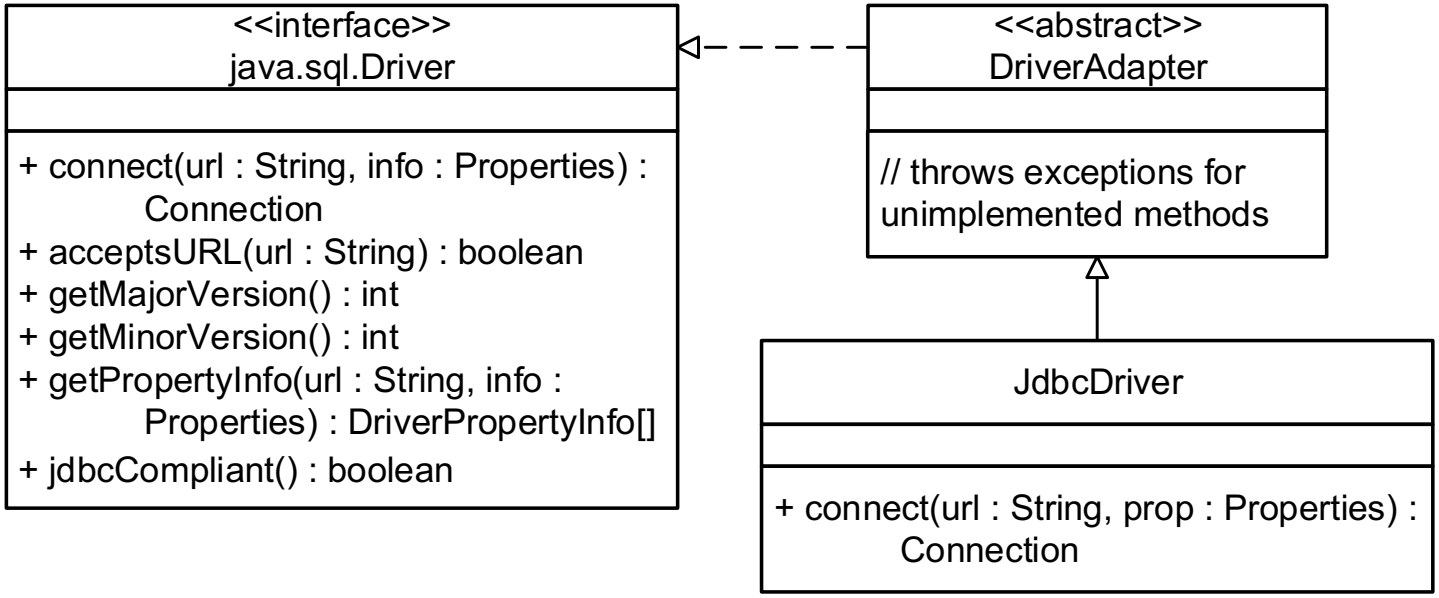
- To obtain the stubs at client-side:

```
// url = "jdbc:vanilladb://xxx.xxx.xxx.xxx:1099"
String host = url.replace("jdbc:vanilladb://", "");
Registry reg = LocateRegistry.getRegistry(host);
RemoteDriver rdvr = (RemoteDriver)
                          reg.lookup("vanilladb-jdbc");
// creates connection
RemoteConnection rconn = rdvr.connect();
// creates statement
RemoteStatement rstmt = rconn.createStatement();
```

- Directly through registry or indirectly through method returns

JDBC Client-Side Impl.

- Implement `java.sql` interfaces using the client-side ***wrappers of stubs***
  - E.g., `JdbcDriver` wraps the stub of `RemoteDriver`



DriverAdapter and JdbcDriver

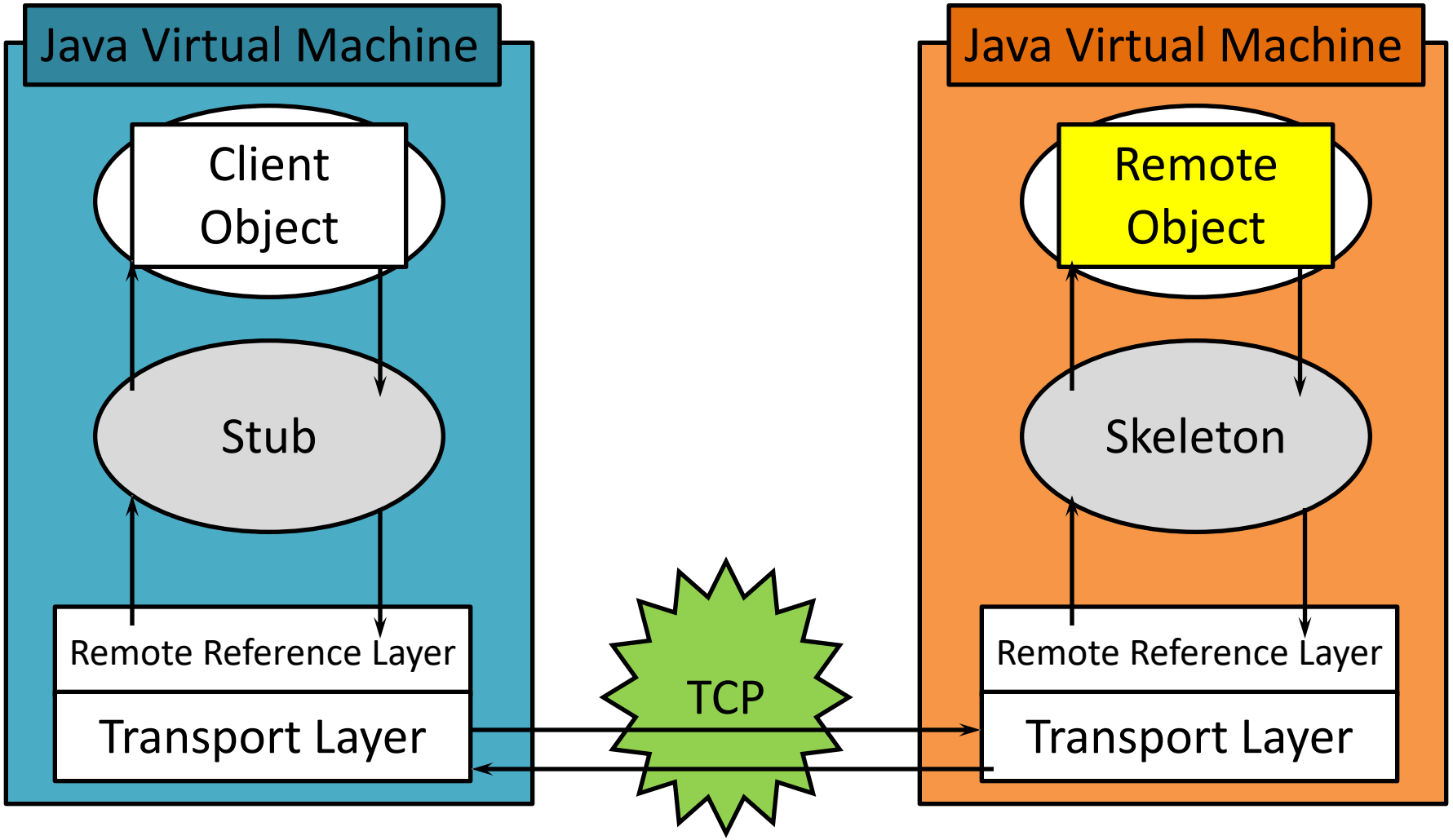
- DriverAdapter
  - Dummy impl. of the Driver interface (by throwing exceptions)
- JdbcDriver:

```
public class JdbcDriver extends DriverAdapter {

    public Connection connect(String url, Properties prop) throws SQLException
    {
        try {
            // assumes no port specified
            String host = url.replace("jdbc:vanilladb://", "");
            Registry reg = LocateRegistry.getRegistry(host);
            RemoteDriver rdvr = (RemoteDriver) reg.lookup("vanilladb-jdbc");
            RemoteConnection rconn = rdvr.connect();

            return new JdbcConnection(rconn);
        } catch (Exception e) {
            throw new SQLException(e);
        }
    }
}
```

Outline

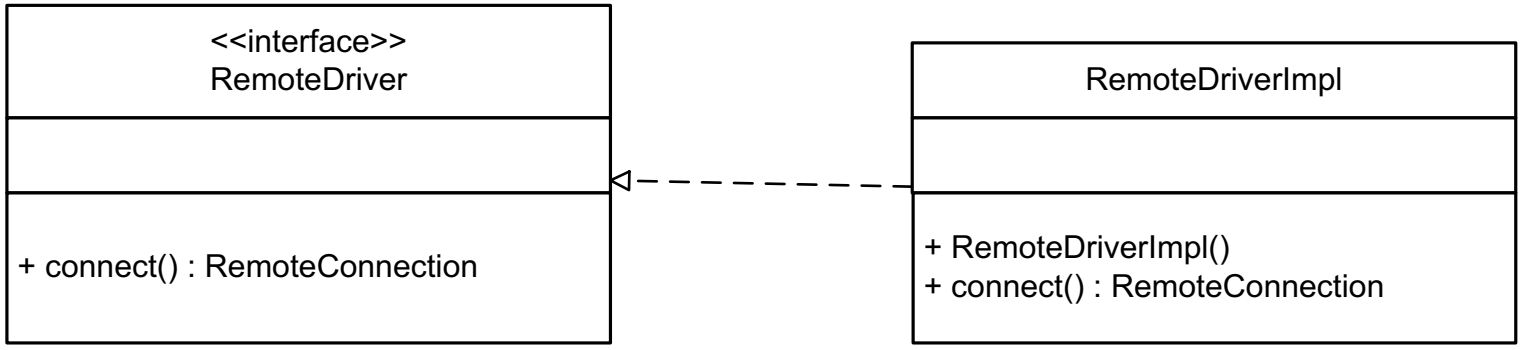
- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Remote Class Implementation in RMI Layers



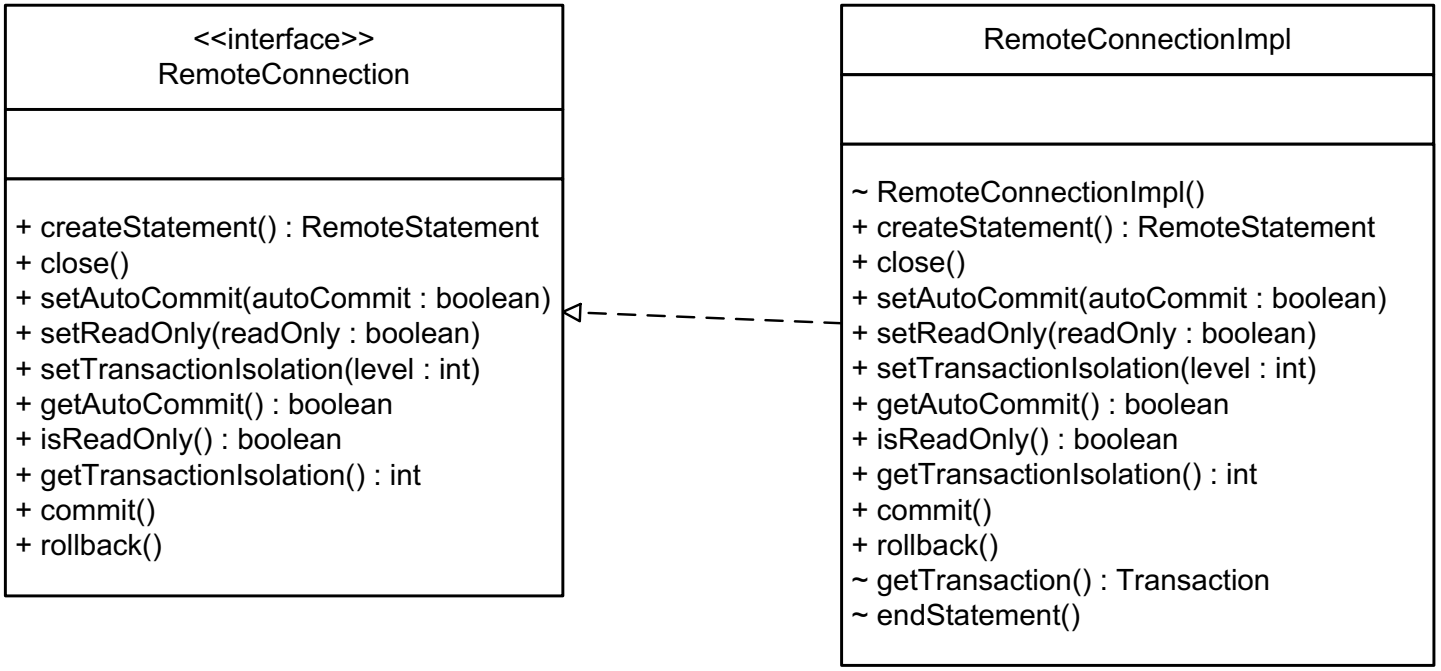
RemoteDriverImpl

- `RemoteDriverImpl` is the entry point into the server
- Each time its connect method is called (via the stub), it creates a new `RemoteConnectionImpl` on the server
  - RMI creates the corresponding stub and returns back it to the client
- ***Run by multiple threads, must be thread-safe***



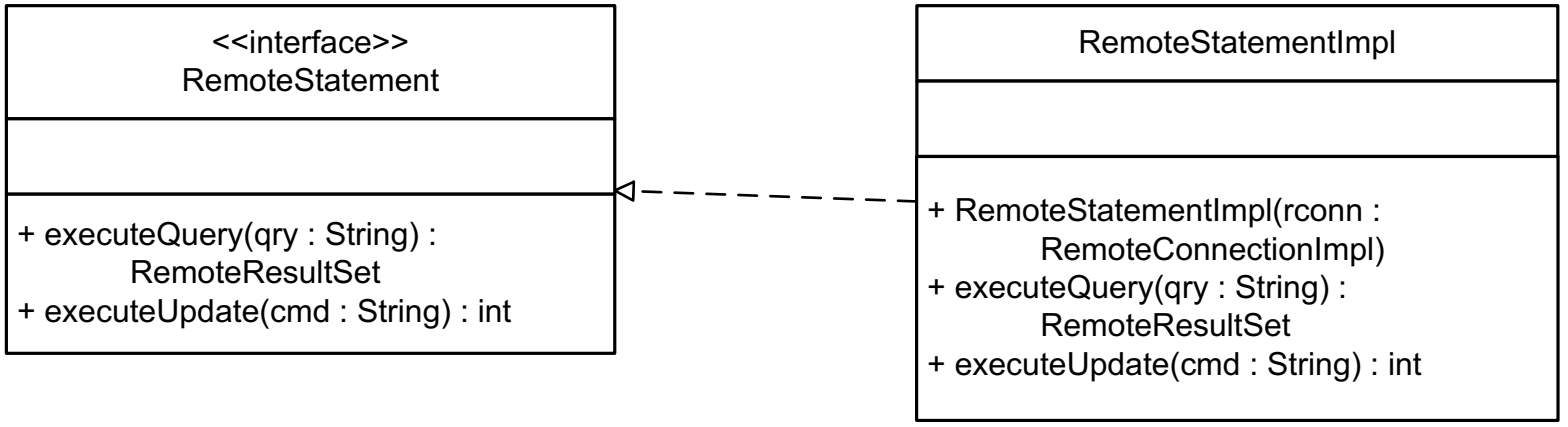
`RemoteConnectionImpl`

- Manages client connections on the server
  - Associated with a tx
  - `commit()` commits the current tx and starts a new one immediately
- Thread local



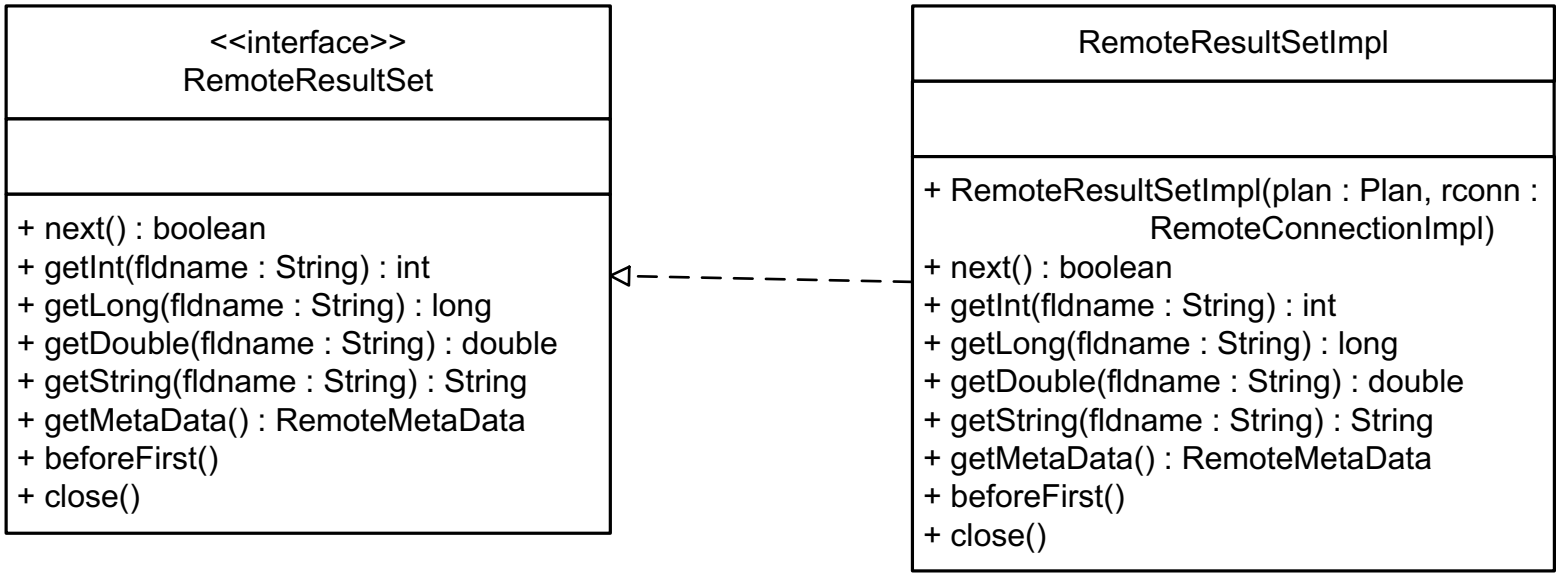
`RemoteStatementImpl`

- Executes SQL statements
  - Creates a planner that finds the best plan tree
- If the connection is set to be ***auto commit***, the `executeUpdate()` method will call `connection.commit()` in the end
- Thread local



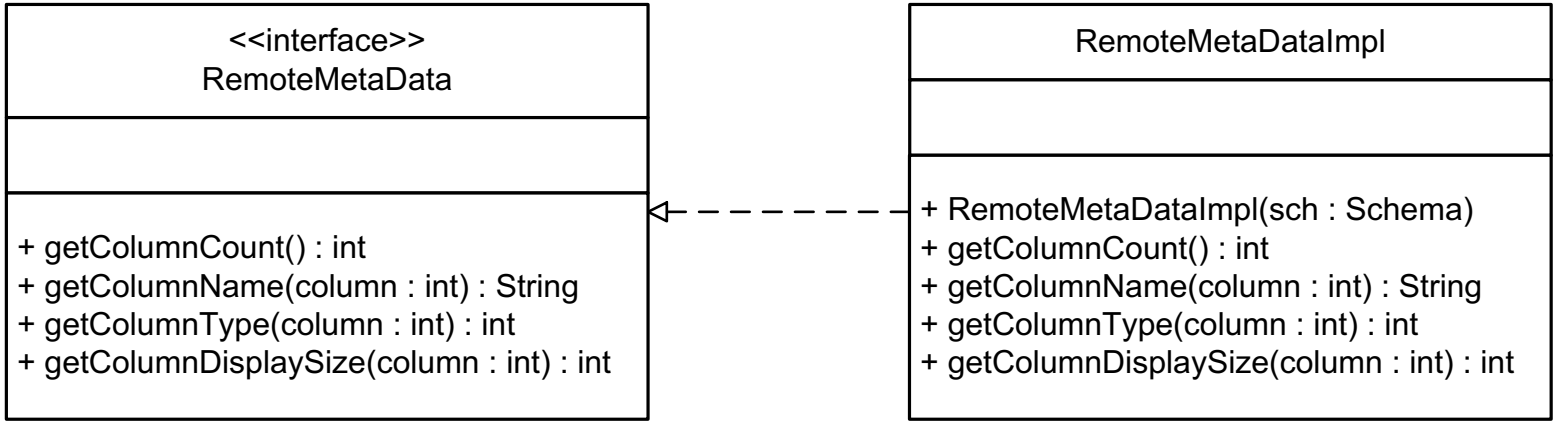
RemoteResultSetImpl

- Provides methods for iterating the output records
  - The scan opened from the best plan tree
- Tx spans through the iteration
  - Avoid doing heavy jobs during the iteration
- Thread local



RemoteMetaDataImpl

- Provides the schema information about the query results
  - Contains the `Schema` object of the output table
- Thread local



Outline

- Processes, threads, and resource management
  - Processes and threads
  - Supporting concurrent clients
  - Embedded clients
  - Remote clients
- Implementing JDBC
  - RMI
  - Remote Interfaces and client-side wrappers
  - Remote Implementations
  - StartUp

Starting Up

- `StartUp` provides `main()` that runs VanillaCore as a JDBC server
  - Calls `VanillaDB.init()`
    - Sharing global resources through static variables
  - Binds `RemoteDriver` to RMI registry
    - One thread per connection

Threading in Engines

- Generally,
- Classes in the query engine are thread-local
- Classes in the storage engine are thread-safe

Assignment Reading

- The following packages in VanillaCore
  - `org.vanilladb.core.server`
  - `org.vanilladb.core.remote.jdbc`

References

- Java Threads and Concurrency
- Java RMI