Transaction Manager

The TransactionManager is the central unit of the SpeegQLe™ DBMS. It is the initial DBMS process which must be started on a server, and spawns the other necessary processes in sequence. It first spawns the FileManager, BufferManager (getting a buffer pool size from the DBMS configuration stored in metadata), and LogManager, in that order, then runs its private recover() function to ensure the database remains consistent in case of a crash. Once recover() has completed, the TransactionManager starts the LockManager (not needed during recovery because TransactionManager is a singleton) and QueryProcessor.

Once the DBMS has started, the TransactionManager takes on a server role and listens for connections from clients (user interfaces) on a TCP port specified in the DMBS configuration. It exposes three functions to these clients.

```
Enum connectionStatus { CONNECTED, FAILED, DISCONNECTED }

connectionStatus ← connect(authentication_data)

bytestream ← query(string)

connectionStatus ← disconnect()
```

The connect() command creates a new connection to the SpeegQLe™ DBMS. The TransactionManager uses the authentication_data (defined in DBMS configuration) given to create a new sessionId, which maps the connection socket to a specific user with some permission set (both held in the DBMS metadata). If the authentication_data is invalid, or the connection fails for other reasons, connect() returns connectionStatus.FAILED. If the connection is successful, connect() returns connectionStatus.CONNECTED, and a new thread is started to handle the connected client and its associated socket; the TransactionManager continues to listen for new connections. disconnect() works similarly, ending any existing connection on the given socket after sending connectionStatus.DISCONNECTED. If no connection exists, disconnect() is ignored.

The query() function receives a query (in string format) from a client. If no transaction for that client currently exists, the TransactionManager creates one and logs it with messageType.START; if a transaction does exist, the new query is added to it (nested transactions are not supported). The TransactionManager then passes the query string, along with the current transactionId, to the QueryProcessor using the evaluateQuery() function. Queries within a single connection are processed sequentially; new queries are queued until the running query completes. If the server receives a call to query() outside the context of a connected client, it discards that query.

QueryProcessor.evaluateQuery() returns a Smarterator™ of query results. If the Smarterator’s getStatus() returns START, or if it returns NORMAL and there are other result Smarterators already in the results set, the new results Smarterator is added to the current transaction’s results set and the TransactionManager simply holds the current transaction open, waiting for further queries.

If the Smarterator’s getStatus() returns NORMAL and the transaction’s results set contains no other transactions, or if getCommitFlag() returns COMMIT, the TransactionManager then runs its internal commit() function, which iterates through each Smarterator in the transaction’s results set, adding the Smarterator’s output to a bytestream which is pushed to the client. Each Smarterator internally performs write-ahead-logging; acquires quasi, shared, and exclusive locks; and pins pages in memory as necessary. Once the final Smarterator completes, commit() logs messageType.COMMIT with the associated transactionId and then calls release() on each Smarterator (in query order), which releases all locks held by that Smarterator and unpins all pages that Smarterator has pinned through the BufferManager. Finally, commit() returns a success status, which query() then returns as the last set of bytes in the bytestream.
If any Smarterators in the results set cause a failure, or if any Smarterator in a transaction has `getStatus()` of ROLLBACK, `commit()` rolls back the transaction using the recovery API `rollback()` function (which undoes all writes and sends a `messageType.ABORT` entry to the log (with the `transactionId`). `commit()` then calls `release()` on each Smarterator in the results set, and returns a failure status, which is written to the `query()` bytestream (and sent to the client).