StrongDBMS

User's Manual

January 2019

Introduction

The StrongDBMS is a simple fully-ACID relational DBMS, based on shareable data structures. It is open-source and free to use, and the code is available for use by anyone and in any product without fee, provided only that its origin and original authorship is suitably acknowledged. Implementations in C# and Java are currently under development.

The internal data structures (in the Shareable namespace) are serialisable and used both in the server and the client, with a binary API. There is an SQL parser in the client library. There are some system tables that provide relational access to the internal mechanisms of the DBMS.

This document provides details of the SQL syntax used, the client library API, and the file format.

The source code is available on github.com/MalcolmCrowe/ShareableDataStructures , together with an introduction to the serializable classes of Strong DBMS. The server is called StrongDBMS and it opens a TCP port on 50433. It uses .NET framework 4.7.2 and C# version 8.0. The Java version will use Java 11.

The transaction protocol is optimistic and fully isolated so that a transaction cannot see concurrent changes. At present read-only transactions don't conflict with anything. By default each call to StrongConnect starts a transaction that auto-commits on success. Explicit transactions are started with BeginTransaction, and end with Commit or Rollback. When the server handles an exception it automatically rolls back any current transaction.

There is one append-only transaction file per database, with no extension. The results of queries are returned in Json format and a _uid if present is a permanent 64-bit defining position in the database file. Strong uses a simplified version of SQL and a limited set of data types, but there is much that will be familiar. Despite its name, Strong is not as strongly typed as Pyrrho. Scale and precision cannot be specified: the maximum number of digits after a decimal point is 4, and integers are limited to about 2040 bits.

There is a command-line client (StrongCmd), documented briefly below. For the API, as described later, to connect to a database "Fred", a client program calls new StrongConnect(host,"Fred", 50433). Then there are versions of ExecuteQuery(sql) and ExecuteNonQuery(sql) in addition to the binary API.

The StrongDBMS server

This is the executable StrongDB.exe defined in the Shareable solution. When started without command-line arguments, it establishes a StrongDBMS TCP service on port 50433 and the local host 127.0.0.1, and the database folder is the current folder.

The command-line syntax is

```
    StrongDB [-p:port] [-h:host] [-d:path]
```

Where the arguments set the port number (default 50433), the host address (default "127.0.0.1") and the database folder path (default .). Any other arguments lead to a usage advisory output.

On startup, the server echoes the command arguments followed by "Enter to start up". The enter key confirms the start. The server obviously needs to be left running in order to undertake work on behalf of its clients.

The StrongCmd client

This is the executable StrongCmd.exe defined in the StrongCmd solution. When started without command-line arguments, it attempts to communicate with a StrongDBMS server on port 50433 on the local machine, using a database called temp. Any named database is created when required.

The command-line syntax is

StrongCmd [**-c:***culture*] [**-h:***host*] [**-p**:*port*] [**-e:***cmd*] [**-f:***file*] *database*

Remember that Windows uses case-sensitive file names: if a name matches apart from case, Windows will use it, and this can lead to conflicts. There is no file-extension.

The normal command prompt is **SQL>** . If an explicit transaction is in progress, the prompt changes to **SQL-T>** . There is no semicolon statement terminator, and the command normally is considered to end with the newline character input. However, multiline commands can be enclosed in **[ ]** , in which case the prompt for the next line of input is **>** .

The syntax for command line input is given in the syntax reference section below, which includes the three transaction control statements BEGIN, ROLLBACK and COMMIT. Strong normally operates in auto-commit mode, whereby each statement is executed in a new transaction which is automatically committed unless an exception is reported. If an exception occurs (including syntax errors), the transaction is aborted, and the database is restored to its state before the transaction began.

The BEGIN statement switches off the auto-commit mode for the duration of the transaction. In such an explicit transaction, nothing is written to disk (or visible to other clients) until the COMMIT statement is executed to finish the transaction. The transaction will be terminated without making any changes if an exception occurs or if the ROLLBACK statement is executed.

The StrongLink client library

Shareable.dll and StrongLink.dll are constructed respectively by the Shareable and StrongLink projects, and should be referenced by any application program using StrongDBMS. The API is provided by the StrongConnect class, and is documented below.

StrongDBMS Data Types

The serialisable data types at present are:

Type Byte	**Example Literal syntax**	**Notes**
`STimestamp = 1,`	`DATE'2018-12-31 23:15:01'`	
`SInteger = 2,`		Arbitrary-precision integer
`SNumeric = 3,`	`-567.123`	Precision limited to 4 places after decimal point
`SString = 4,`	`'This is a string'`	Unicode, variable-length, no escape characters or embedded quotes
`SDate = 5,`	`DATE'2018-12-31'`	
`STimeSpan = 6,`	`TIMESPAN'12345678'`	Integer (ticks)
`SBoolean = 7,`	`TRUE`	
`SRow = 8,`	`(A: 56.7, B:'A string')`	

SQL Syntax Reference

For simplicity, identifiers in StrongDBMS are not case-sensitive, must consist of ANSI alphabetic characters only, and may not match any reserved word (these are shown in bold in the following syntax rules). The SQL subset is deliberately minimal at this stage, with no joins or grouping.

Statement: CreateTable | CreateIndex | Insert | Delete | Update | Select | TransactionControl .

CreateTable: **CREATE TABLE** id '(' ColDef {',' ColDef} ')' .

ColDef: id Type .

Type: **TIMESTAMP** | **INTEGER** | **NUMERIC** | **STRING** | **DATE** | **TIMESPAN** | **BOOLEAN** .

CreateIndex: **CREATE** [**PRIMARY**] **INDEX** id **FOR** *table*_id '(' Cols ')' [**REFERENCES** id] .

Cols: id {',' id} .

Insert: **INSERT** *table*_id ['('Cols')'] (**VALUES** Values)|Select .

Values: '(' Value {',' Value} ')' .

Value: literal | [*table*_id '.'] *col*_id | Value BinOp Value | Func '(' Value ')' | '(' Value ')' | Select |
'(' Value [**AS** id] {','Value [**AS** id]}')'| Values | Value **IS NULL** | Value **IN** Value |
UnaryOp Value .

Func = **COUNT** | **MAX** | **MIN** | **SUM** .

BinOp = '+' | '-' | '*' | '/' | '=' | '!''=' | '<' | '<''=' | '>' | '>''=' | **AND** | **OR** .

UnaryOp = '-' | **NOT** .

Delete: **DELETE** Query .

Update: **UPDATE** Query **SET** *col*_id = Value {',' *col*_id = Value} .

Select: **SELECT** [**DISTINCT**] **[** Value [**AS** id] {',' Value [**AS** id]}] **FROM** Query [**ORDERBY** Order].

Order: [id '.'] *col*_id [**DESC**] {',' [id '.'] *col*_id [**DESC**]} .

Query: *table*_id [*alias*_id] [**WHERE** Value] .

Alter: **ALTER** *table*_id **ADD** id Type
| **ALTER** *table*_id **DROP** *col*_id
| **ALTER** *table*_id [**COLUMN** *col*_id] **TO** id [Type] .
Drop: **DROP** *table_or_index*_id .

TransactionControl: **BEGIN** | **ROLLBACK** | **COMMIT** .

The Binary Protocol

The Serialisable.Types enumeration includes bytes used in the protocol and responses.

Each PDU consists of a protocol byte followed by data. Simple items in the data such as names are sent as strings (length as 32-bit integer, chars in UTF8) or integers (64-bit). Integers are sent highest byte first. { } denotes a sequence of items, preceded by a 32-bit count.

The associated PDUs sent by the client are as follows:

Protocol	Data	ETag
DescribedGet	SQuery	
Get	SQuery	
SBegin	(nothing)	
SCommit	(nothing)	The transaction
SRollback	(nothing)	
SCreateTable	Name,{column name, column type}	The new table
SAlter	Name, Child or “”, NewName	The altered object
SDrop	Name, Child or “”	The dropped object
SCreateIndex	Table name, IndexType, References or “”, {column name}	The new index
Read	Int	
SInsert	SInsertStatement	
Insert	Table name, {col name}opt, {{Serialisable}}	The new Record
SUpdate	Record uid, {col name,Serialisable}	The updated record
SDelete	Record uid	The deleted record

System Tables

There are two system tables at present called “_Log” and “_Tables”. _Log gives a list of all of the SDbObjects in the database as strings together with their defining positions. _Tables gives the current statistics (number of columns, number of rows) for base tables in the database. These only show committed data.

Query Analysis

At the lowest level in the database we have SColumn and STable, SRecord etc. These have names and uids for lookup purposes. At the client we only have uids for disambiguation purposes, and these will not be the same as the DBMS’s uids: in the query language we will basically just have selectors (names only) and expressions built from these.

When the query language reaches the server, we can associate the selectors with actual fully-featured columns. But we won’t be able to evaluate expressions until we have an actual row in a Rowset. So in this phase we recurse into expressions to lookup the selectors.

Current Status

The DBMS is still in the early stages of development. There are the beginnings of a test suite.

Next steps

It would be good to have more system tables, e.g. to obtain ETags for database objects, the steps of multi-step transactions, and the current list of columns of a Table.

There is a clear need for defining users, roles, and permissions. The plan at present is to have the format of records in the database file expand to include transaction times and users as soon as the first role is defined.

At present there is no way of dropping an index except by dropping the table.

But first there needs to be an SQL-style parser in the client library.

Read constraints and Auditing

We now move on to define and manage usage of the database. If no users are defined, the database is public: this may be appropriate for an embedded system. But it will be good practice to define a user (the owner of the database) when the database is created. Today there is considerable interest in access auditing, and a requirement in some jurisdictions for companies to record use of sensitive data.

As an academic exercise at least, let us consider how this could be accomplished. We can add auditing records to the transaction log whenever a user accesses sensitive data, to specific records in a table, or all of them. Even where we do not need to create an audit record, some information of this sort is useful in transaction management (up to now we have not considered conflicts between reading and writing). This means there are already good reasons for considering such transactional read constraints even for data that is not sensitive.

Such considerations lead to the following machinery.

- Implementation of transaction read constraints
- Flagging of sensitive data (at column level)
- Implementation of authentication and authorisation of users and roles
- Recording of these users and roles for committed transactions (who made changes)
- Immediate recording of access information for sensitive data during transactions