StrongDBMS

User’s Manual

November 2018

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 serializable 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.

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 Begin, 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.

There is no command-line client as yet. As described later, to connect to a database "Fred", a client program calls new StrongConnect("Fred", 50433). SQL-style syntax is converted to a Serialisable instance using StrongConnect's Prepare method.

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,`		`32-bit integer`
`SNumeric = 3,`	`-567.123`	`64-bit mantissa, 32-bit scale`
`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'`	`64-bit 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 expressions or functions, and no joins or grouping. However, the former can be accommodated by the SSelector abstraction, and the latter by the RowSet abstraction, as in Pyrrho, so these aspects can be added later.

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** '(' Value {',' Value} ')'.

Value: literal .

Delete: **DELETE** Query .

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

Select: **SELECT** [Cols] **FROM** Query .

Query: *table*_id [**WHERE** *col*_id = Value {**AND** *col*_id = 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

To support an initial version of the binary API, we have in namespace Shareable:

```
public enum Protocol
{
    EoF = -1, Get = 1, Begin = 2, Commit = 3, Rollback = 4,
    Table = 5, Alter = 6, Drop = 7, Index = 8, Insert = 9,
    Read = 10, Update = 11, Delete = 12, View = 13
}
public enum Responses
{
    Done = 0, Exception = 1, ETag = 2
}
```

PDUs

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
Get	SQuery	
Begin	(nothing)	
Commit	(nothing)	The transaction
Rollback	(nothing)	
Table	Name,{column name, column type}	The new table
Alter	Name, Child or “”, NewName	The altered object
Drop	Name, Child or “”	The dropped object
Index	Table name, IndexType, References or “”, {column name}	The new index
Read	Int	
Insert	Table name, {col name}opt, {{Serialisable}}	The new Record
Update	Record uid, {col name,Serialisable}	The updated record
Delete	Record uid	The deleted record

System Tables

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

Next steps

Update and Delete are rather primitive so the API will include QueryUpdate and QueryDelete versions before long. 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