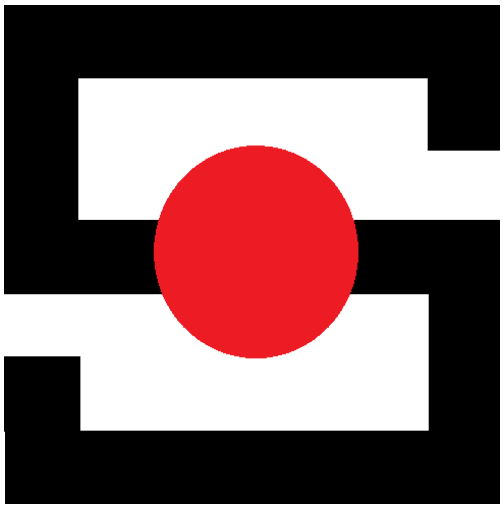
StrongDBMS

User's Manual



Version 0.1 (May 2019)

Introduction

The StrongDBMS is a simple fully-ACID relational DBMS, based on *shareable* data structures. This radical approach involves the use throughout the implementation of classes all of whose fields are read-only. In the server such classes are used for the database itself, its tables and columns and for all transaction objects including row-sets and the bookmarks used when traversing them. The only exceptions to this rule are the DBMS itself, which contains a mutable set of databases, and open streams from the client and from disk files. The only sacrifice to functionality that results from this policy is that only "functional" user-defined methods will be supported: so that local objects can be declared and initialised but can only be modified as targets for FETCH. StrongDBMS will nevertheless support RESTViews and adapter functions as in Pyrrho. SQL parsing is done in the client library.

StrongDBMS is open-source and free to use. The source code is on github.com, available for use by anyone and in any product without fee, provided only that its origin and original authorship is suitably acknowledged. It has Java and C# versions that are fully compatible, and run on Windows and Linux.

From a computer science point of view, Strong builds on the idea in Pyrrho that the database file, as an append-only transaction log, provides physical proof of transaction isolation and durability, and its consistent use of shareable data structures makes atomicity and consistency also provable. Like Pyrrho, StrongDBMS is a First-Committer-Wins database system with persistent row-versioning. It is implemented in Java as well as C#, and runs on both Windows and Linux.

This document provides details of the SQL syntax used, the client library API, the file format, and system tables. 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 currently uses .NET framework 4.7.2, C# version 8.0 (2019), and Java 11.

As of mid-January 2019 (version 0.0), it reached a point where is performs the TPCC benchmark (www.tpc.org/tpcc/) with the standard NewOrder measure at 40/sec on a machine running both client and server (for this application each client takes 3 times as much CPU time as the server and no disk activity, while the server uses more memory and minimal disk activity). An implementation of the benchmark in C# is included with the above distribution. Version 0.1 (March 2019) sees a breaking change to the serializable classes and the API to use uids instead of names for object references and to support column and table constraints. Database files from version 1.0 onwards will be forward and backward compatible between DBMS versions (in addition to being independent of platform[1] and locale).

The use of long integers (uids) instead of names will be used in the introduction of roles in a future version: the role is maintained for the database and associates names with database objects identified by uid. During a transaction the role is supplemented by the aliases of names and queries. As the client does not have access to the server's uids, the SQL parser encodes the names to temporary uids and sends the list separately from the serialised objects. As the query is received by the transaction, references to client-side names are progressively replaced by query and database objects.

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

[1] For example, StrongDBMS and Pyrrho database files use a custom encoding of integers and numerics, up to 128x8 bits, which is independent of the platform. Strings, dates and times follow Unicode and ISO standards.

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[2]. 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. I plan to add more standard RDMS features over the next month or so, in the following order: groups, joins, views/RestView, computational completeness/stored procedures, triggers, window functions, users and roles, and mil-spec access controls.

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 [**-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 **>** .

---

[2] Thus, the file name matches the database name. As it is quite common for database applications to have the same name as the database they use, it is good practice to avoid placing any client binaries in the folder used by the server for data (see the -d flag in the next section below).

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.

It is a design feature of StrongDBMS that the consistency of the database is the responsibility of the server. The StrongDBMS client library does not maintain its own copy of the database schema (as it would always be potentially out of date), and the SQL parser does not have information about tables or columns in the database. There are system tables that enable the client to obtain such information.

The StrongConnect class has the following API for normal use:

Property or Method	Parameters	Comments
void BeginTransaction()		Start an explicit transaction
void Commit()		End explicit transaction with commit
type ExecuteNonQuery(string)	SQL	Return value is Done on success
DocArray ExecuteQuery(string)	SQL	SQL Select returns Json array
void Rollback()		End explicit transaction without commit
StrongConnect(string,int,string)	Host, port, database name	Constructor

Documents contain results in Json format, e.g. if a DocArray is converted to a string it is enclosed with [] and Documents with {}, as in the standard Json specification. Documents have some extra fields whose names beginning with _ for giving uid information for readConstraints.

There is a natural way of programming access to a DocArray, as follows. For C# a DocArray has one field based on the .NET library: List<Document> items. A Document has one field: List<KeyValuePair<string,object>> fields. The Java versions use the ShareableDataStructures library: DocArray has one field SList<Document> items, and Document has one field SList<SSlot<String,Object>> fields. These structures use Lists, because the ordering of items and fields can be significant.

StrongDBMS Data Types

The server avoids operating system and locale dependencies. In due course localisable string collations will be supported. Strings are enclosed in straight single quotes only.

Locales are supported in the client library StrongLink.

The serialisable data types at present are:

Type Byte	Example Literal syntax	Notes
SInteger = 2,		Arbitrary-precision integer
SNumeric = 3,	-567.123	Precision limited to 4 places after decimal point
SString = 4,	'This is a string' ‘Let''s allow embedded quotes'	Unicode, variable-length, no escape characters
SDate = 5,	DATE'2018-12-31' DATE'2018-12-31T22:53:14.785'	ISO 8601
STimeSpan = 6,	TIMESPAN'-3.4:00:34.789' TIMESPAN'3:34.789' TIMESPAN'14'	-3.04:00:34.7890000 0.00:03:34.7890000 14.00:00:00.0000000
SBoolean = 7,	TRUE	
SRow = 8,	(A: 56.7, B:'A string')	

SQL Syntax Reference

For simplicity, identifiers in StrongDBMS are case-sensitive with pattern (A-Za-z_)(A-Za-z0-9_)*, and may not case-insensitively match any reserved word (these are shown in bold in the following syntax rules). The SQL subset is deliberately minimal at this stage. With the implementation of column and table constraints, CREATE INDEX is deprecated: it is not included below and will throw an exception in future versions.

Statement: CreateTable | Insert | Delete | Update | Select | Alter| Drop | TransactionControl .

CreateTable: **CREATE TABLE** id TableDef {[','] TableConstraint } .

TableDef: '(' ColDef {',' ColDef} ')' .

ColDef: id Type { ColumnConstraint }.

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

ColumnConstraint: (**NOTNULL**) | ((**DEFAULT**| **GENERATED** | (**CHECK** id ':')) Value)

| (**PRIMARY KEY**) | (**REFERENCES** id).

TableConstraint: (**PRIMARY KEY**|**UNIQUE**) '(' Cols ')'

| **FOREIGN KEY '**(' 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** ('('Select')'|Values) | UnaryOp Value | **VALUE** .

The VALUE keyword can only be used inside a ColumnConstraint expression, and represents the proposed value for the column at the time the constraint is checked.

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

BinOp = '+' | '-' | '*' | '/' | RelOp | **AND** | **OR** .

RelOp= '=' | '!''=' | '<' | '<''=' | '>' | '>''=' .

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: ColRef [**DESC**] {',' ColRef [**DESC**]} .

ColRef: [id '.'] *col*_id .

Query: TableExp [**WHERE** Value] [**GROUPBY** Cols [**HAVING** Value]].

TableExp: *table*_id [*alias*_id] | '('TableExp|Select ')'| TableExp Join .

Join : (',' | **CROSS JOIN**) TableExp
| **NATURAL** [JoinType] **JOIN** TableExp
| [JoinType] **JOIN** TableExp **ON** ColRef '=' ColRef { **AND** ColRef '=' ColRef }
| [JoinType] **JOIN** TableExp **USING** Cols .
JoinType: [**INNER**| ((**LEFT** | **RIGHT** | **FULL**) [**OUTER**]).

Alter: **ALTER** *table*_id **ADD** (ColDef | TableConstraint)
| **ALTER** *table*_id **DROP** (*col*_id | **KEY** Cols)
| **ALTER** *table*_id **TO** id
| **ALTER** *table*_id **COLUMN** *col*_id **ADD** { ColumnConstraint }
| **ALTER** *table*_id **COLUMN** *col*_id **DROP** (**DEFAULT** | **GENERATED** | **NOTNULL** | *check*_id)
| **ALTER** *table*_id **COLUMN** *col*_id **TO** (id | int | ColDef).
Drop: **DROP** *table*_id .

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

A note on constraints and schema changes: The above syntax for column and table constraints is fairly standard SQL, apart from the new keyword NOTNULL. The combination of keywords NOT NULL is not supported except in a properly constructed CHECK predicate such as CHECK IS NOT NULL.

But the different constraints are not independent. A table can only have one primary key constraint, so it is an error to define more than one. You cannot add a referential constraint to a table unless the data type of the primary key matches the proposed foreign key: this requirement is checked by the server.

Redundancy such as NOTNULL DEFAULT 0 or DEFAULT 0 GENERATED 42 is disallowed. A primary key (or unique) column cannot be null, but specifying NOTNULL for it is tolerated as it can become relevant if the table constraint is later dropped. A foreign key is not required to have a value.

On the server, alter and drop statements are validated against the rest of the schema and current table contents. This happens twice: within the transaction, and again on commit. For example, you cannot drop a key column, or the target of a referential constraint. There is no way to suspend these validation checks, so that changes often need to be done in the correct order or with the help of temporary workarounds. It is good practice to perform related schema changes inside an explicit transaction. The validation on commit will also check for consistency against any concurrent changes to the database that have already been committed (there is no way to prevent these).

The Binary Protocol

The Serialisable.Types enumeration includes bytes used in the protocol and responses. On connection the database name is sent. Then a sequence of PDUs follows.

Each PDU consists of a protocol byte followed by data. Integers in the protocol and on disk are byte-sequences (an unsigned byte n followed by n signed bytes[3]). Simple items in the data such as names are sent as strings (n as a byte sequence, followed by n bytes in UTF8).

Names are encoded as uids. The SDatabase class has a structure called role that allows uids to be translated back into strings. The role also manages a set of global names (for tables, views, stored procedures etc), and for each database object such as table, view, stored query the column names and aliases defined by it. Each PDU may supply a set of client-side (name, uid) pairs called Names., and the prepare phase of query processing computes the corresponding database objects corresponding to non-alias uids. The encoding works as follows (the server encoding is architecture-dependent on the size of a Long):

Purpose	Unique within	Low	High
Role.System		-1	-1
Client side identifiers	Client PDU	-1000000	-2
Client-side aliases	Client PDU	-2000000	-1000001
System tables & columns	Server	-infinity	-2000001
Uncommitted objects and aliases	Transaction	Long.MaxValue>>2+1	Infinity
File positions, aliases*	Database	0	Long.MaxValue>>2

*Alias uids for stored queries and procedure variables will be assigned sequentially from the file position of the stored query or procedure. Empty space will be left in the database file if necessary to support this (e.g. if a very large database defines a procedure with very numerous named variables).

When objects are created the the definer's name is supplied immediately after the new uid. For queries the names for all uids precede the query (see SNames), since the query might refer to these more than once. Where a definer's string follows a new uid this is notated in the following tables by uid+.

The associated PDUs sent by the client are as follows:

Protocol	Response*	ETag
SNames	None	
DescribedGet	Column info, Json	A readConstraint
Get	Json	A readConstraint
SBegin	Done	
SCommit	Committed	The transaction
SRollback	Done	
SCreateTable	Done	The new table
SCreateColumn	Done	
SAlter	Done	The altered object
SDrop	Done	The dropped object
SIndex	Done	The new index
Read	Json	
SInsert	Done	
Insert	Done	The new Record

[3] Thus 0 is coded as [0], 1 as [1 1], -1 as [1 255], -56 as [1 200], 200 as [2 0 200], 400 as [2 1 144] and so on.

SUpdate	Done	The updated record
SDelete	Done	The deleted record

*An exception response is returned if the server reports and exception: it includes the Exception.Message, which for transaction conflicts includes a conflicting Uid. The Done response is followed by two longs: the starting and current positions of the transaction in the log (during an explicit transaction these will be the same), otherwise both are 0. Where the above table shows Done/Committed the response depends on the transaction state: Done indicates that the committed response will follow a successful Commit, and Committed indicates that the transaction was completed as a result of autocommits.

Objects etc are serialised as follows (the parent class shown in italics is serialised first, and ints, longs, strings and sequences in {} are encoded as described above):

Serialisable	1 byte	Types enum
SNames	{uid+}	Names for a following query
SDbObject	*Serialisable*,uid†	
SAlias	*SQuery*, long	Alias name given by uid
SAlter	*SDbObject*, 2 longs, string, byte, int, {string, SFunction}	defpos, [column], name, datatype, sequence, constraints
SArg	*Serialisable*	Like value in C#/Java property setter
SColumn	*SSelector*, string, byte, long, {string, SFunction}	name, datatype,table, constraints
SCreateColumn	*Serialisable*, SColumn	
SCreateTable	*Serialisable*, uid+, {SCreateColumn}, {SIndex}	name,columns, tableConstraints
SDelete	*SDbObject*,2 longs, {long,Serialisable}	table,delpos,keyfields
SDeleteSearch	*Serialisable*,SQuery	
SDrop	*SDbObject*, 2 longs, string	dropobj,[parent],[detail] (for drop of constraint)
SDropIndex	*SDbObject*,long,{long}	table,keycols
SExpression	*SDbObject*,long,SExpression,byte, SExpression	uid,left,operator,right
SFunction	*Serialisable*,byte,Serialisable	function,arg
SGroupQuery	*SQuery*, SQuery, {long} {Serialisable}	source, groupBy, having
SIndex	*SDbObject*, long, bool, {long}, long	table, primary, key columns, references
SInPredicate	*Seialisable,* 2 Serialisable	arg, list
SJoin	*SQuery,* byte, bool, 2 SQuery, {SExpression}, {SRef}	joinType, outer, left, right, ons, uses
SInsert	*Serialisable,* long, {long}, {Serialisable}	table, [cols], values
SOrder	*Serialisable,* long, bool	column, descending
SQuery	*SDbObject*	
SRecord	*SDbObject,* long, {long,Serialisable}	table, {col, value}
SRow	*Serialisable,* {long,Serialisable}	{col,value}
SSearch	SQuery, {Serialisable}	where (list of and-conditions)
SSelector	*SDbObject*	

SSelectStatement	*SQuery*, bool, {long, Serialisable}, SQuery,{SOrder}	distinct, selectList, query, ordering
STable	*SQuery*	Name given by uid
SUpdate	*SRecord*, long, {long,Serialisable}	defpos, {changed key fields}
SUpdateSearch	*SQuery*, {uid,Serialisable}	assignments
SValues	*Serialisable*, {Serialisable}	
SView	*SDbObject,* {long}, SQuery	[cols],view definition

† for client comms only

System Tables

The following system tables are defined at present. The _Log table allows the full transaction log history to be searched. The other tables show objects visible in the current transaction.

_Log	(Uid), Type, Desc, Id†, Affects
_Columns	(Table, Name), Type, Constraints,Uid
_Constraints	(Table,Column,Check), [Expression]
_Indexes	Table, Type, Cols, [References]
_Tables	(Name), Cols, Rows, Indexes,Uid

† id provided where the entry creates a change to the role. See below.

Support for Roles

In a future version of StrongDBMS, it will be possible (as in Pyrrho) to modify the names used for database objects at the role level. To provide for this, and to speed up searching for data objects, StrongDBMS from version 0.1 retrieves all database objects including records by single 64-bit uids instead of strings. When a database object is first defined by name, the name is entered into the current role.

As mentioned above (in section “The StrongDBMS client library”), the client library does not track the identifiers used in the database. It assumes that the client program identifies database objects correctly in queries. The set of identifiers actually used in a particular query are sent immediately before the client PDU, and given special “client-side uids” as described in the section above (“The binary protocol”).

Aliases in the query language have a special range in this uid scheme.

Query Analysis

When a client-side query reaches the server, the table identifiers can be immediately resolved, and the next step (called Prepare) rewrites the query, using server or transaction uids instead of client side uids. This is done without attempting to evaluate expressions.

During traversal of a rowSet, the current bookmark defines a row (an actual row in a base table, or a row defined for the rowSet), and where complex queries draw on subqueries or base tables, the top level bookmark is able to access lower level bookmarks by building these row references into a stack called Context. Thus every row bookmark a readonly context, whose top entry is the row for that bookmark, and links to the next lower context used to construct the row. Each level in the context provides a set of uid-value pairs (for a row this is column uid to column value), and this mechanism can also be used for alias uid to row value, and for computing grouped functions.

Since all of this information is readonly in the row bookmark it must be computed within the rowbookmark constructor. Although the resulting code is challenging for the human reader, it turns out to be very fast in benchmarks.

Current Status

The DBMS is still in the early stages of development, and has reached implementation of joins, constraints and grouped aggregations. There are the beginnings of a test suite, which does not yet test the Alter and drop syntax.

The transaction model will abort a transaction at commit time if its write or read set conflicts with updates made since the start of the transaction (a BEGIN). Outside an explicit transaction, ExecuteQuery (read-only) statements auto-rollback, and ExecuteNonQuery will auto-commit. Most syntax errors are detected by the client-side parser and have no effect on the server, but errors detected at the server (including non-existent columns or tables) will terminate the current transaction immediately.

Next steps

As mentioned above some of the syntax is still being implemented. The next step will be in the direction of “big live data”, including RESTViews. After that it is planned to support stored procedures, structured types, and triggers.

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

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.