StrongDBMS: Built from Immutable Components

Malcolm Crowe, Santiago Matalonga
University of the West of Scotland
Paisley, UK
email:{malcolm.crowe;santiago.matalonga}@uws.ac.uk

Martti Laiho
DBTechNet
www.dbtechnet.org
email:martti.laiho@gmail.com

***Abstract*—StrongDBMS is a new relational Database Management System (DBMS). Atomicity, Consistency, Isolation and Durability (ACID) properties are guaranteed through the use of an explicit transaction log and immutable software components. The shareable data structures used allow instant snapshots and provide thread-safety even for iterators, and minimize the need for locking mechanisms without compromising consistency. StrongDBMS has been implemented in C# and Java, and both versions are inter-operable on Windows and Linux. Benchmarking measures are included in this paper. StrongDBMS is open-source and free to use. This paper presents the design rationale for StrongDBMS and benchmarks its current version. Benchmarking results using the Transaction Processing Council's TPC/C benchmark show performance comparable with standard commercial products.**

***Keywords* – Optimistic; relational; thread-safety; transactions.**

I. Introduction

StrongDBMS has as its design goal to build a simple fully-ACID relational DBMS, based on an append-only transaction log file, and *shareable* data structures. The transaction log file gives guarantees of transaction isolation and durability, and shareable data structures as described below provide guarantees of atomicity and consistency.

The rest of section 1 gives some background to the work, Section 2 introduces shareable data structures, Section 3 describes the resulting database architecture, and section 4 discusses some benchmarking data on the resulting DBMS.

A. Background

Most modern DBMSs, including StrongDBMS and Pyrrho [1], employ Multi-Version Concurrency Control (MVCC), in which each transaction effectively works with a private copy of the database. For higher levels of isolation, such as Snapshot Isolation (SI) and Serializable SI (SSI), a transaction which reads a data item x sees a private copy of x with the value that it had when the transaction began, and a transaction which writes x does so on a private copy of x which is only made available globally (i.e., to other transactions) upon a successful commit operator of the writer.

However, in most systems, this ideal strategy is made more complex by the sharing of index structures between concurrent transactions, so that many DBMS use the First Updater Wins strategy (FUW) so that the first transaction to announce an update locks the index until it commits [2]. With StrongDBMS and Pyrrho each transaction uses its own indexes and access data structures, and so these DBMS are able to implement First Committer Wins (FCW), in which transactions proceed without interfering with each other until commit time. Upon commit, if there have not been other commits on objects which T has written or read, it is allowed to commit. Otherwise, T must be aborted. With this approach, integrity constraints against commits made since the start of T cannot be made until T begins to commit.

Both StrongDBMS and Pyrrho use immutable objects with maximal sharing for indexes and access structures. The objects are immutable in the sense that any modification of the value of a data object results in a new object; pointers cannot be updated, and values are never overwritten. The objects admit maximal sharing in that when an object is modified (or a new object is created), that part which is the same as the previous object is re-used; only the part which is different uses new storage.

StrongDBMS extends the use of immutable objects to all serializable objects and all objects used in query processing including row sets, and so these desirable properties can be guaranteed throughout the transaction implementation. The main goal of this paper is to show how such immutable objects with sharing may be used in the implementation of a DBMS.

B. Relationship to previous work

In Pyrrho and StrongDBMS each database is stored on disk as a single append-only transaction log file. The data format is independent of machine architecture, word size, byte order, or locale. Entries in the log from each transaction are appended as a group for atomicity and serialization, as explained below. Database objects have a unique identity given by their definition point in the log. They retain this identity when updated or modified even though the modification details are recorded later in the log.

In this way, both are optimistic-execution DBMS with persistent row-versioning, as discussed in [3]. Unlike Pyrrho, StrongDBMS is implemented in Java as well as C#, taking advantage of its novel aspects of the features of each programming language, and both implementations can run on Windows and Linux.

StrongDBMS' internal data structures (in the Shareable namespace) are serializable and used both in the server and the client, with a binary Application Programming Interface (API) and a Structured Query Language(SQL) parser in the client library. There are some system tables that provide relational access to the internal mechanisms of the DBMS.

The DBMS is still under development and many standard SQL features will be added later. It currently supports integrity constraints, aggregation, grouping, and joins. Roles, views and support for "big live data" [4] will be added during 2019, followed by executable modules and triggers.

The data types supported are arbitrary-precision integers and numeric, Unicode string, date and timespan, and row. Identifiers are case-sensitive. The set of system tables is currently limited to the log and the list of base tables.

StrongDBMS uses a client-server architecture with a client API based on serializable objects rather than SQL. Parsing of SQL is performed in the client library (StrongLink). The server, StrongDB, opens a Transmission Control Protocol (TCP) port on 50433. There is a command-line utility StrongCmd. The implementation uses .NET framework 4.7.2, C# version 8.0 (2019), and Java 11. It is open-source and free to use. The source code is on github.com [5], together with an introduction to the serializable classes of StrongDBMS.

StrongDBMS is available for use by anyone and in any product without fee, provided only that its origin and original authorship is suitably acknowledged.

II. Shareable data structures

The unique interest of StrongDBMS is the use of shareable (or immutable) data structures. Such structures are particularly appropriate for DBMS, since they are inherently thread-safe, provide instant snapshots, and do not require locking. This section briefly introduces this concept.

A. *On value semantics and thread-safety*

The study of Data Structures is an essential early stage in any Computing program [6] and needs to be revisited later on when the student has mastered threading [7]. Students quickly learn that the standard string data type in modern languages, such as Java and C#, is immutable, but are often not told why.

As an unsafe example, consider arrays of characters in Java. Suppose A, declared as char[] A, contains the characters NOW. If we assign this array to a similar array B, then both A and B share the same data. After an update to A, say A[2] = 'T' , they *both* contain NOT . This may be what the programmer intended, but from the viewpoint of this study, this behavior is seen as unsafe. There is nothing wrong with the original assignment of A to B or with sharing the array elements. But A[2] = 'T' represents a problem (and Java wisely disallows such an operation for Strings). So, for a shareable list structure we support A=A.InsertAt('T',2) and A=A.RemoveAt(1), and both these operations create new lists without changing the contents of B. The implementation, of course, will be as a linked list.

This is not to criticize Java, which has built on its String structure and championed interfaces such as Cloneable. Linked Lists are not the best structures for databases either, but shareable data structures with logarithmic behavior can transform the performance of databases.

When a shareable structure such as a linked list or tree is updated, new nodes are required from the start of the structure to the updated position. The rest of the structure is unchanged and does not need to be copied.

Before leaving the notion of thread-safety, consider the behavior of data structures passed as parameters. Java (as a requirement) and C# (by default) pass parameters "by value", a comfortable phrase that obscures a major source of difficulty. There is nothing to stop the called procedure from modifying a structure passed in. Such modifications are often useful but can be a difficult source of error.

The solution to both problem areas is to use, as far as possible, data structures that contain no mutable fields. In C# and Java, immutable fields can be declared **public readonly** or **public final**; they receive their values in constructors and these cannot be changed. It is a huge advantage that whole indexes and even whole databases in memory can then be copied by a single machine instruction, and database rollback or Prolog Unbind consists simply of forgetting the new pointer. With such data structures, there is no need for locking, because the values inside can never change. Managing locking in complex software has been a problem for many decades [9]and it is a great relief to reduce this burden.

The most commonly used shareable data structure in StrongDBMS is a key-value dictionary called SDict<K,V> , which is used to build shareable searchable arrays (e.g., SDict<int,bool>) and multi-level indexes that use such dictionaries at each level. In C#, it is possible to define operators so that we can use d += (k,v) for adding an entry to dictionary d, which is safer than having to remember to write d= d.Add(k,v) . (Java's dictionaries are not safe, and many programmers used to them will accidentally write d.Add(k,v) and lose the updated dictionary.)

Instead of using linear linked lists or arrays, for scalability it is strongly recommended to use structures with logarithmic behavior such as B-Trees. Figure 1 shows the picture of an update for a B-tree, reproduced from [8], and shows that only a few nodes need to be created on an update.

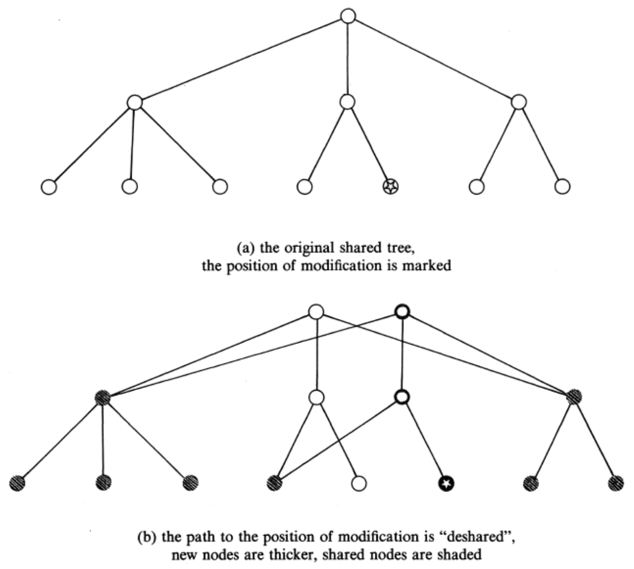
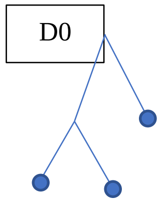

(a) the original shared tree,
the position of modification is marked
(b) the path to the position of modification is "deshared",
new nodes are thicker, shared nodes are shaded

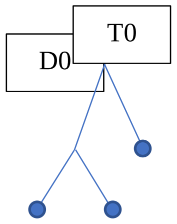
Figure 1: Updating a B-Tree (from[8])

Figure2 shows how this approach affects the scenario of databases and transactions described in the Introduction.

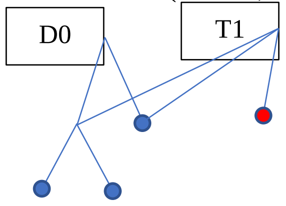
Let us start with a database D initially synchronized with the data abase file. There will be data structures including a list of objects and a list of names, but we show just one in the illustration.



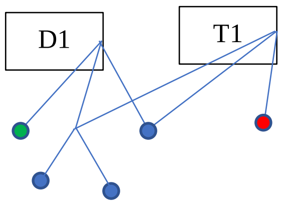
A transaction T starts by using the same data structures as the database.



When the transaction makes a change, it creates new root nodes as required, but continues to share the rest of the data structures (indexes, lists of objects etc.).



If another transaction commits in the meantime, the database will be replaced by a new one sharing the same data structures apart from the new root node.



These changes do not affect the transaction's data structures.

When our transaction commits, (it can only do so after checking for no conflicts), its new information is added into the database's data structures, creating new root nodes as required.

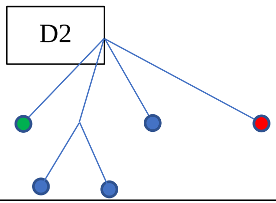


Figure 2: Shareable data structures and transaction behaviour

Care is needed when shareable data structures are used inside mutable structures. If such an unsafe object A contains an immutable dictionary d then we need to remember to write **lock**(A) d += (k,v); in Java we use a synchronized block **synchronized**(A) { d = d.add(k,v); } . The only place StrongDBMS finds it necessary to use such locking for the global list of databases, and for file and stream objects.

The memory allocator must work harder with shareable data structures, but in complex software, this happens anyway, and if arrays are used a lot of time is spent in copying.

B. *Bookmarks instead of Enumerators*

Both C# and Java always use Enumerators or Iterators in the standard libraries [9][10]. For an enumerator E, one moves to the next item in a collection using E.MoveNext(). This is obviously unsafe even for an immutable collection, as E might have been passed in as a parameter or copied somewhere else. In this work, we exclusively use Bookmarks for our shareable collections.

For any shareable collection as defined here, there is a method called First() that returns a bookmark to the first entry of the collection (First() returns null if the collection is empty). And given any Bookmark B, we get a bookmark to the next entry if any by B = B.Next().

Neither language provides us with a useful syntax for iteration using bookmarks, but it is easy to get used to writing

for (**var** b=C.First(); b!=**null**; b=b.Next())

Bookmarks iterate through the list as it stood when the First() bookmark was created. It is very convenient to be allowed to modify the list as it is being traversed. (The standard libraries do not allow mutable List structures to be modified during iteration.)

III. Architecture of the DBMS

The design goals mentioned at the start of section I almost dictate some important features of the DBMS architecture. Each element of the binary API should be serializable, for transport from the client to the server and for serialization to the transaction log as persistent database objects. Some of the serializable objects represent SQL constructs, translated from SQL into this form in the client library. Each database object has a readonly *uid* field consisting of its immutable file position on disk.

A. *Permanent uids for database objects*

A file position will not be known until it is committed (serialized) to disk, and so because of the readonly nature of the uid, the commit will be done inside a constructor. In this section we will use C# for the code illustrations (the Java code is similar and can be reviewed in [4]):

```
protected SDbObject(SDbObject s, AStream f) :base(s.type)
{
    uid = f.Length;
    f.uids = f.uids + (s.uid, uid);
    f.WriteByte((byte)s.type);
}
```

In this fragment, we can see the uid being set as the current file length before we write the first byte (the type) of the SDbObject subclass. We also see a dictionary called uids maintained by the file stream structure f, which associates the previous unique identifier with the new permanent uid.

The next step in the design is to decide what the previous uid was. This is assigned at the time the SDbObject is created for addition to the transaction. The transaction maintains a private sequence of uids for its SDbObjects so when the SdbObject is created for the transaction we have:

```
protected SDbObject(Types t,STransaction tr) :base(t)
{
   uid = tr.uid+1;
}
```

Objects such as table or column references arriving from the client may have uids assigned by the client-side parser if the names alone would be ambiguous. Because of separation of concerns between client and server, the client does not interpret these (for example, it will not know what columns are defined for a table). This is an important point since any schema information held in the client will, in general, be out of date.

B. *Database*

The database knows what its schema objects are, indexed by their permanent uids, and has a name catalogue for top-level objects such as base tables.

The first constructor for a database (cold start) gives the name and initializes the other information:

```
SDatabase(string fname)
{
   name = fname;
   objects = SDict<long, SDbObject>.Empty;
   names = SDict<string, SDbObject>.Empty;
   curpos = 0;
}
```

If there is a database file on disk it is loaded into memory, deserializing its contents from the file and installing them in the database structure. We see some examples of this process below.

Creating a copy of the database is just:

```
protected SDatabase(SDatabase db)
{
   name = db.name;
   objects = db.objects;
   names = db.names;
   curpos = db.curpos;
}
```

We see that copying (taking a snapshot of) a database costs almost nothing (just four pointers).

The database structure is immutable so that any update requires the construction of a new instance. The Database structure has a constructor that updates its dictionaries of schema objects in a new instance:

```
protected virtual SDatabase New(SDict<long,SDbObject> o,
      SDict<string,SDbObject> ns, long c)
{
    return new SDatabase(this, o, ns, c);
}
```

The current database (this) is made available to the constructor so that other data pointers (in this case, just the database name) that have not been changed can be copied into the new object. Here is the constructor:

```
protected SDatabase(SDatabase db, SDict<long,
SDbObject> obs, SDict<string,SDbObject> nms, long c)
{
   name = db.name;
   objects = obs;
   names = nms;
   curpos = c;
}
```

C. *Installing database objects*

The method in the database class for installing a table, e.g., when loading the database on startup, is very simple – of course it returns a new database object using the New method given above:

```
public SDatabase Install(STable t, long c)
{
  return New(objects+(t.uid, t),names+(t.name, t), c);
}
```

Tables maintain their own readonly lists of columns, rows and indexes, so installing a column creates a new version of the table object as well as a new database:

```
public SDatabase Install(SColumn c, long p)
{
  var obs = objects;
  if (c.uid >= STransaction._uid)
      obs += (c.uid, c);
  var tb = ((STable)obs[c.table])+c;
  return New(obs+(c.table,tb), names+(tb.name,tb), p);
}
```

It is important that very little data copying is required to make a new table object: it contains merely a small set of references to the roots of tree structures, some of which will have been updated.

The database does not directly include columns in its list of objects (but transactions do as described below). There is a global static mutable collection of file streams with exclusive access to the databases currently open on the server, and a database looks up the appropriate file and locks it when it needs to access the disk.

Records, updates and deletes do not need to be in these memory structures as they can be retrieved from the disk file when required (However, if a lot of clients use the same database, the saving in memory is at the cost of increased contention on the file stream. An object cache would also be worth considering).

D. *Transaction*

We make STransaction a subclass of SDatabase so that it inherits the immutable information from the database on creation, including the current file position of the database (c in the above New method) at the start of the transaction. The code for starting a transaction is just a constructor:

```
public STransaction(SDatabase d,bool auto) :base(d)
{
   autoCommit = auto;
   rollback = d._Rollback;
   uid = _uid;
   readConstraints = SDict<long, bool>.Empty;
}
```

The code called for the base constructor is just the code for copying a Database, shown earlier.

Objects proposed for addition in the transaction (including records, updates and deletes) are added to the transaction's objects using its private sequence of uids, and this sequence is traversed on commit. (Recall that transactions cannot see other transactions so their sequences of uids are separate.)

In order to support long transactions, we remember that schema objects and records defined in a transaction should be usable in the transaction, so the transaction needs to install them in the dictionaries and indexes it has inherited from the database. It uses the same install code as the database, but with its own version of the New method that creates a transaction object instead of a database object.

When the transaction commits, the transaction's objects are installed in the database, and the transaction object can be forgotten.

E. *Detection of transaction conflicts*

The Commit method for the transaction object needs to consider whether conflicting changes may have occurred in the database before a commit can be agreed. As mentioned above, the transaction already has the file position at the start of the transaction. The transaction now looks at the current state of the file: if objects have been added by other transactions it compares with the changes to be committed. If there are conflicts or anything read by the transaction has been modified, the commit cannot proceed, and a transaction conflict exception will be raised. If all is well, the database file is locked, and the process is repeated for any commits that may have happened before the lock. If there are still no conflicts, as the file is already locked the transaction's objects can be committed to the database using the mechanism described at the start of subsection A above. These installation steps result in a new database object, which is then installed in the server's static mutable list of databases:

```
public static void Install(SDatabase db)
{
   lock(files) databases = databases+(db.name, db);
}
```

The database file is then unlocked.

F. *Query processing and RowSets*

StrongDBMS follows the SQL standard closely except that it allows case-sensitive identifiers and a small set of primitive data types. Full details are in [5], but some example SQL statements may help:

```
create table Voc (Id integer, Word string, Notes string)
insert Voc values (1,'a','Indefinite article')
select from Voc where Word>'Z'
```

As mentioned above, parsing of SQL queries is done on the client, so that the client sends the server a Serialisable object such as an SQuery or an SInsertStatement. As the server receives these, there is a *Lookup* method to identify the columns and tables referred to by name, and construct versions of the received object where the object references are to the correct schema objects.

The next step is that a RowSet is constructed for the results of the query or the data for the insert or other command. In the presence of subqueries or grouping, etc., this process may be recursive, so that the query's RowSet method supplies a stack called Context in which the current values of selectors can be found. The RowSet contains a copy of the transaction that has the readConstraints list populated during this recursion.

RowSets are traversed using a special subclass of Bookmark (RowBookmark), which holds a row object for the current row of the traversal, and a base table record if this is a row of a base table. In general, the selectors in the query can be expression objects, so that for returning results to the client, the selector expressions use the same Lookup method to compute the results, using new Context extended by the current RowBookmark.

The RowSet method recursively constructs row sets for traversing tables in joins and subqueries, for applying an ordering or a search condition and for evaluating aggregations. For join processing the row sets participating in the join are first ordered using the columns specified in the join-condition or implied by a natural join.

The final traversal for the client serializes the results using Json format. Non-query client requests that use RowSets include insert, update and delete statements for base tables, and these use the records referred to in the RowBookmark.

G. *Transaction Programming Paradigm*

As described above, a transaction in StrongDBMS operates on the database as if it were private since the start of the transaction. This isolation provides true conflict serializability [11], which is strictly stronger than that required by the ISO SQL standard [12]. The private transaction context allows a straight-forward programming for the transaction logic without concern on lock timeouts or concurrency conflicts before the COMMIT. Further details on the isolation model are given in Chapter 1 section "Concurrency Control" in [3].

IV. Benchmarking StrongDBMS

A. *Parameter tuning*

As suggested above, StrongDBMS adopts a standard data format for the database file that is independent of locale or machine architecture. Within the server it is obvious that standard **int** and **long** data types will be used for integers where possible and a multibyte alternative for big integers.

It is less obvious how to fine-tune the size of B-Tree "buckets". B-Trees have a fixed bucket size N, and then allow nodes other than the root to have between N and 2N (or 2N+1) child nodes. Experimentally it can be established that performance is independent of the value of N over the range 6 to 32. Currently StrongDBMS uses N=8. With smaller values of N there are more nodes and deeper trees, while if N is larger the cost of copying bucket contents becomes more significant.

B. *Performance Benchmarks*

Relational DBMS traditionally use the Transaction Processing Council's TPC/C benchmark [13] which models a 1980s-style Online Transaction Processing application. An interesting measure is provided by the New Order transaction, which models a clerk filling in a warehouse order for a customer. Each warehouse serves ten districts, each with 3000 customers, and 100000 products. An order may have up to 20 lines, each a given quantity of a specific product identified by its code. As the clerk enters the fields on the form the database supplies details: the customer's name and address, the customer's discount, orders to date, etc., and for each line of the order supplies the product description and price, updates the current stock level, and computes the total cost per line and per order. On completion of the order the transaction is committed. Each order takes dozens of server round-trips.

On a personal computer, an implementation following the details prescribed by TPC typically will execute about 20 New Order transactions per second. The initial state of the database on Strong occupies 100MB. In Table 1, this initial database file has already been constructed (it was excessively slow to recreate in the Java on Linux configuration), and timings were taken for the initialization of the system (cold start) and for 2000 New Order transactions. The client and hardware were the same for all four tests (Intel i5 processor, 16 GB of memory).

TABLE I. TPC/C 2000 NEW ORDER TRANSACTIONS

Server Implementation	**Operating System**	**Cold start**	**2000 New Orders**
C# 8.0	Windows 10	16 sec	48 sec
C#	Debian 9 (mono)	10 sec	79 sec
Java 11 (32 bit)	Windows 10	7 sec	51 sec
Java 11 (32 bit)	Debian 9 (mono)	6 sec	242 sec

V. CONCLUSIONS

This paper has introduced StrongDBMS, a new database management system based on the ideas of append-only transactions log-file and shareable data structures. Together they provide the capabilities of transaction isolation, durability, atomicity and consistency. This paper presented the design rationale and trade-offs for the StrongDBMS approach. StrongDBMS has been co-developed in Java and C#, and will continue to be supported in both programming languages. As mentioned above, StrongDBMS is still under development. The current state has enabled us to envision and discuss the benefits and limitations of the approach. The design of StrongDBMS is intended to support multithreading, so that the server handles each transaction in a different thread, and threads sharing a database use the database file for synchronization when a commit is requested. Our next steps will include tests for verifying performance with multithreading.

We have presented how the current implementation of StrongDBMS performs in the Transaction Processing Council's TPC/C benchmark. In time for the conference we will provide comparable performance figures for commercial database products.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge the inspiration and encouragement and contributions of members of the DBTech community and from Stephen Hegner.

REFERENCES

[1] M. K. Crowe, "Transactions in the Pyrrho Database Engine. in Databases and Applications" [ed.] M H Hamza. Innsbruck, Austria : ACTA Press, 2005.. pp. 71-76.

[2] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. E. Shasha, "Making snapshot isolation serializable", ACM Transactions on Database Systems, 30:2 (2005) pp. 429-528.

[3] M. Laiho, M. Kurki, M. Crowe, F. Laux, D. Dervos, and K. Hirvonen, Introduction to Transaction Programming: DBTechNet.org, [Online] 2019. dbtechnet.org/papers/IntroToTransactionProgramming.pdf.

[4] M. Crowe, C. Begg, F. Laux, and M. Laiho, "Data Validation for Big Live Data. Barcelona" : DBKDA 2017, The Ninth International Conference on Advances in Databases, Knowledge and Data Applications, 2017, pp.30-36.

[5] M. Crowe, "Shareable Data Structures". GitHub. [retrieved 31 March 2019] https://github.com/MalcolmCrowe/ShareableDataStructures.

[6] M. Ardis, D. Budgen, G. W. Hislop, J. Offutt, M. Sebern, and W. Visser, SE 2014: "Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering", Long Beach, CA : 2015, Computer, Vol. 48 (11), pp. 106-109.

[7] B. P. Shults, "Teaching Data Structures: thread safety and components". Boston, MA : IEEE, 2002. 32nd Annual Frontiers in Education.

[8] T. Krijnen, and G. L. T. Meertens, "Making B-Trees work for B". Amsterdam : Stichting Mathematisch Centrum, 1982, Technical Report IW 219/83.

[9] Microsoft. System.Collections.Immutable. 2015 www.nuget.org [retrieved 31 March 2019].

[10] Oracle. Java Collections Framework.

[11] H. Berenson, J. Gray, J. Melton, E. J. O'Neill, P. E. O'Neill, "A Critique of ANSI SQL Isolation Levels". San Jose, CA : ACM, 1995. Proceedings of the 1995 ACM SIGMOD International Conference. pp. 1-10

[12] J. Melton, Information Technology - Database Languages - SQL.. : ISO/IEC, 2016. 9075.

[13] Transaction Processing Council. [retrieved 31 March 2019]. http://www.tpc.org.