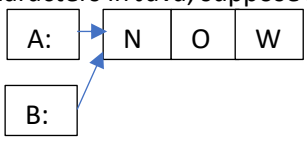
Shareable Data Structures

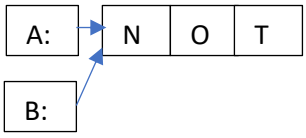
Malcolm Crowe, University of the West of Scotland

1 Introduction

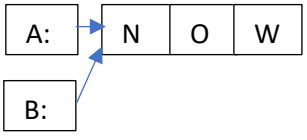
The study of Data Structures is an essential early stage in any Computing programme, and needs to be revisited later on when the student has mastered threading. Early exercises on threading use example programs containing unsafe data structures such as arrays to illustrate the need for locking. Students quickly learn that the standard string data type in modern languages such as Java and C# is immutable, but are often not told why: even if they are taught that this is to enable sharing of strings, they may fail to appreciate that other data structures can also benefit from being made safe, immutable and shareable. Telling them that in these languages "strings are values" only serves to obscure the real issue.

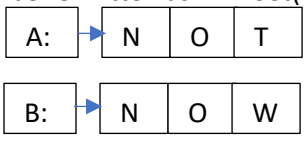
Using arrays A and B of characters in Java, suppose A contains the characters NOW. After the assignment B=A we have: 

(In languages such as Java, C#, or Python, the assignment of a whole data structure just involves updating a pointer.)

After A[2]='T' we have: 

, so that the value of B has been changed too. This may be what the programmer intended.

With a safe array, or a string, the assignment B=A would still give 

. The assignment A[2]='T' might need to be rewritten as A=A.Set(2,'T'), but would give a new safe array or string, so that we would end with 

. This behaviour is called value semantics. With safe arrays or strings the assignment operation for the whole structure still involves updating a single pointer. At the very least the programmer needs to be clear which behaviour is intended.

Value Semantics

Strings as values were mentioned above. In fact, the notion of value semantics is key to our approach in this booklet. Whatever type S you are working with, S should have value semantics if possible. This means that if I place a value x of type S into a variable v, any subsequent changes to x will not be visible from v. Thus, every assignment of a value of any shareable type S assigns a snapshot of that value, as at the time of the assignment. This is just what happens when we assign an integer or a string.

But importantly, as we are told when we begin programming, this is not the case with structured data. When we use such everyday structures as lists, arrays, stacks, trees, or enumerators/iterators, we have to be careful to copy the data elementwise ("cloning") when a snapshot is what we want. Sharing these data structures is a nuisance in a multi-threaded program (such as one with a graphic user interface) as they need to be locked by any method that modifies them. We even have to be careful when a parameter is passed into a method "by value". If the parameter is an array in C# or

Java, there is nothing to stop the method from modifying it, so it is “the programmer’s responsibility” to ensure that nothing unexpected happens as a result. These habits make the task of programming unnecessarily complex.

In this little book I want to present a useful set of data structures that all have value semantics but still cover the same needs as the standard collection types listed above. To emphasise that these are all shareable in the above sense the type names will begin with S. We will also use generics a lot for strong typing: so we will have classes such as SList<T>, SDict<K,V> etc. As we will see they all have the property that their fields are all *readonly* or *final*.

But, as with strings, that does not make them less useful. If you manipulate a string in C# or Java, you get a new string. In just the same way, the method to change the nth entry in an SList will return a new SList. This ensures that the previous value remains accessible as long as anyone has a copy of it. And as we shall see, because any substructure will also be shareable, much of the substructure will be the same as in the previous value, so that this strategy ends up reducing memory allocation operations.

There is a place for unsafe data structures such as arrays. They can be used as local variables and where needed for using standard library methods. For example, array sorting algorithms can be safely done using unsafe arrays provided the arrays are copied beforehand. There is also nothing wrong with using an unsafe data structure as a parameter for a constructor of a shareable type.

In this booklet we will study the advantages and disadvantages of using dense arrays compared with more complex shareable indexed structures. Unless otherwise signalled, everything in this booklet applies equally to Java and C#, and nearly always also to other imperative programming languages. For the code examples though we need to choose a language, and C# is used here. In most cases the only change needed to convert the code from C# to Java is to replace the keyword **readonly** by **final**. Where the Java version contains interesting differences this is explained in the text. The associated software repository on github.com (github.com/MalcolmCrowe/ShareableDataStructures) has versions of these classes for C#, Java and Python.

Threads and Locking

With a shared unsafe data structure, locking is needed to manage concurrent access. For example, in the case of the arrays A and B above, if another thread changes B, there is in principle a race between the two threads to see which update succeeds. This is not a problem for an array of characters as the simple assignment shown is thread-safe. But, without locking, a more complex change might lead to inconsistency. In elementary courses it is a standard exercise to have two threads with a shared file such as standard output (or input!) to demonstrate that locking is needed to distentangle the operations.

If there are many structures to lock, deadlock can be a problem. Using safe structures will not avoid locking altogether, but can greatly reduce it, as we will show in this booklet. We will still need locking somewhere, as our application will surely have some variable data. *Any method that modifies the fields of an object containing such data should still lock the object*, as otherwise some important changes will get lost sooner or later. Obviously this does not apply if all of the fields are readonly as they can only be modified in the constructor, and nobody else has access to the object at this time. (C# and Java both prevent some uses of **this** inside a constructor.)

The repository contains a number of Test programs that compare the performance of these data structures with standard, non-shareable data structures. The occurrence of garbage collection affects the timings for both sorts of data structure, and as may be expected, the shareable structures make the memory allocator work harder. Otherwise, the timings are broadly similar. In considering the results of these tests, bear in mind that the reasons for using shareable data structures are (a)

thread-safety (b) quick snapshots (c) use of bookmarks instead of iterators. This last point will be discussed later.

Application to Database technology

By the time we get to the end of this booklet we will have enough shareable data structures to implement something major like a DBMS. Rather obviously, a DBMS server will have a List of Database objects that are in use, and we can ensure that these are shareable (SDatabase, say). The list of databases itself can be mutable: it contains the current committed state of each database, and is modifiable only by the transaction manager (the main thread of the DBMS). Someone (a client service thread) wanting to work with one of these simply copies the SDatabase they want (in all of these languages, this copy is just of the 64-bit pointer to the structure). This will be a snapshot of the current database state and they can start to modify this as they please (it is private to them). If they want their changes to be made permanent, they need to keep a note of the changes they want.

If the client service thread aborts or rolls back, their copy of the database can be simply forgotten, since none else knows about it. If they request for their copy to be committed, the DBMS will examine their list of changes against the current state of the database in its list – this will no longer match the snapshot the client was given. If the proposed changes do not conflict with changes that other threads have made, they can be applied to the master copy, and the SDatabase value in the master list of databases will be modified to be this new value. As the assignment of a single SDatabase pointer is an atomic operation nothing needs to be locked, and the commit process need not interfere with the operation of any other threads.

However, if we also want durability, the changes need to be written to disk (say in a transaction log) as part of the commit process. Since the disk file will be busy, providing copies of records being worked on by other threads, we need to lock it during commit and while seeking a record. This is the only locking that we need to perform for this case where commit is just for a single database. This solves the entire database-deadlock problem at a stroke.

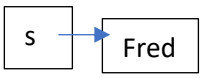
First Steps: a safe Linked List

We begin with an introduction to the terms used in the above discussion, by means of the simplest structures encountered by students: List and Linked List. As normally defined, both are unsafe, even though documentation tells us that their methods are thread-safe.

Instances or values are stored in one or more memory locations, e.g.

Where a value occupies more than one location (as here) it is helpful to show the origin or starting position. A variable is something that refers to a value: a variable also needs a memory location to hold it: what it holds is either null or a reference to the current value.

String s = "Fred";



If I now have another variable, String t and assign t= s, both of these will refer to the same place in memory. There are no methods that allow the value Fred to change. But either t or s can be made to refer to a different string, or be null (not point to anything). We will illustrate a reference with an arrow, and a null pointer with a diagonal.

> Java and C# go to quite a lot of trouble to pretend that the same sort of picture is accurate for simple values such as int. But it is not really the case. If I have int x=17; and set int y=x; there are two different memory locations, both with the value 17. But as with strings there are no methods that can change the value 17. If I now say x = x+1, it will place the value 18 into x. This has exactly the same effect as if x now pointed to a location containing 18. Even though Java also has a reference class Integer that really does point to an integer value, the behaviour will appear to be the same.

Example 1: Java always (and C# by default) passes method arguments by value. We can add a new entry to a List **a** either by passing it as a value parameter **x** to a function that calls **x**.Add(**n**), or by simply calling **a**.Add(**n**) . In both cases **a** will have been modified. *If someone else has a copy of the list **a**, it will be modified as well.* If we want to avoid this behaviour, we need to make an elementwise copy of the List (this used to be called a clone).

We will return to the List structure later. We turn first to consider the linked list.

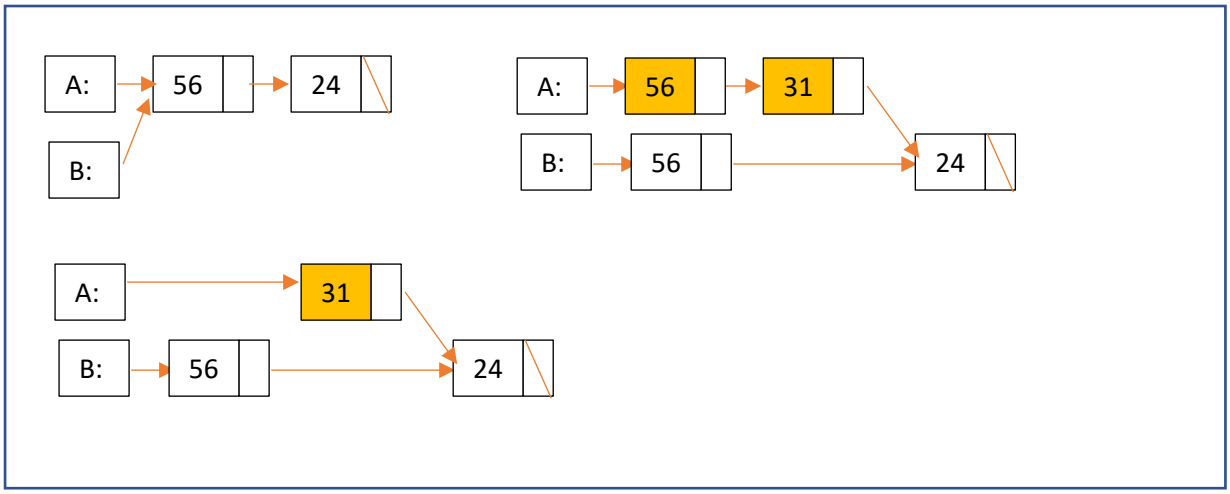
Example 2: Consider a simple Linked List data structure:

```
class ListOfInt {int element; ListOfInt next;}
```

If I have an instance **a** of this class (so **a** is not null), it is the head of a list of integers, and contains at least one. But if I let anyone have a copy of this list, there is nothing to stop them changing the value of the first element, or any of the links.

Example 3: Consider a *shareable* Link data structure:

```
public class SListOfInt {
       public readonly int element;
       public readonly SListOfInt next;
       public SListOfInt(int e, SListOfInt n) { element = e; next = n; }
       // more methods may get added here
}
```



Readonly is an access modifier that ensures that a value can be assigned in the constructor, but then cannot be changed. It is well implemented in C# and Java.

Now I can share an instance of this class in the sure knowledge that no-one can change it. Moreover, no method of this class can change the list represented by an instance. However, if I have a variable of this class, such as SListOfInt **a** , I can always change **a** to refer to a different instance. I can even make a refer to a longer list by a statement such as **a** = new SListOfInt(22,**a**); . But importantly, anyone with a copy of my old list will see no changes.

Now in either of these linked list structures we need to allocate a piece of memory for each element of the list. This does require some extra resources provided by our programming library and/or operating system: a memory allocator and a garbage collector. If we have a very large linked list we might be tempted to “save resources” by using an array of int instead. But notice that when we added a new entry at the head of a long list, we did not need to copy the whole thing. We would

probably have had to do so if it was an array (especially if someone else wanted to keep the original version).

A major advantage with the shareable list is that we never need to check for a cycle. The constructor always creates a new head and no next-pointer can be altered to point to it.

Example 4: A function to test if a given int is in the list:

```
public bool Contains(int x)
{
    return x == element || (next?.Contains(x) ?? false);
}
```

This definition is recursive, and in some systems its execution will cause the stack to grow temporarily in the worst case by N stack frames, where N is the length of the list. But it is a very special sort of recursion, called tail recursion, and many programming language implementations will automatically replace it by an equaivalent loop. If ours does not, we could program the loop ourselves:

(Example 4a)

```
public bool Contains(int x)
{
    for (var a = this; a != null; a = a.next)
        if (x == element)
            return true;
    return false;
}
```

We will leave this alterative mechanism to further examples of tail recursion as an exercise for the reader.

Example 5: Let us add a method to the SListOfInt class, to remove the **n**th element (n>=0).

```
public SListOfInt RemoveAt(int n) {
       if (n==0)
              return next;
       return new SListOfInt(element,next.RemoveAt(n-1));
}
```

Now, anyone who has a copy of the list will see no change, but I have a new list. Importantly, we did not make a copy of the whole thing first, and if the list is long we can see that most of the links are common to both lists. Again we could replace the tail recursion with a loop if we wanted.

Programmers familiar with the usual implementations of List<T> need to remember that is x is an SList<T>, then

```
x.RemoveAt(1);
```

will do nothing (except possibly throw an exception if x has only one element). It is important to remember to write

```
x = x.RemoveAt(1);
```

if you now expect x to be the shortened list. Reminder: if this x is a field in another data structure **a**, you must lock the data structure before doing this, e.g.

```
lock(a) { a.x = a.x.RemoveAt(1); }
```

Example 6: Another method could insert a new element at position n in the list (n>=0):

```
public SListOfInt InsertAt(int x, int n) {
```

```
            if (n==0)
                  return new SListOfInt(x,this);
            return new SListOfInt(element,next.InsertAt(x,n-1));
      }
```

In both of these examples, there is a point (**n**) in the list where we have made changes: we had to allocate **n** or **n**+1 new pieces of memory, for the path from the head of the list to where we resume the old list. The statements that we use when n==0 (at this stage the recursion has reached the head of the list) simply reuse the pieces of memory that made up the previous value.

Note that this method does not allow us to create the first entry in a list. For that we need the constructor. Since an empty list would be null, for a method that works even for an empty list you will want either (a) a static method (and as we will see that will cause problems in Java for generic classes) or (b) use a two stage mechanism where SlistOfInt is an object containg a readonly pointer to the start of the list. We will come back to these options later.

Example 7: We need at least one method that accesses the list. In C# we can implement a subscript method:

```
public int this[int n]
{
    get { return (n == 0) ? element : next[n - 1]; }
}
```

And something similar in Java (would need to be called something like getAt(int i) . Note some C# 2017 shortcut tricks here:

Example 8: Here is another tail-recursive property to get the length of the list:

```
public int Length
{
    get { return 1 + (next?.Length ?? 0); }
}
```

The Length property uses a shortcut mechanism for C#. The ?. means if the left-hand side is not null, evaluate the right hand side, otherwise return null. The ?? means if the left hand side is null, use the right hand side. So the method is equivalent to

```
public int Length
{
    get { return (next!=null)? next.Length +1 : 1; }
}
```

In Java this is a method getLength() as Java does not have properties.

Some people like facilities such as Length and subscripts: they can make a complicated data structure look encouragingly like an array, which may be more familiar. But it might lead a tired programmer to write a very innefficient version of the Contains function:

```
public bool Contains(int x)
{
    for (var i = 0; i < Length; i++)
        if (this[i]== x)
            return true;
    return false;
}
```

In fact, we are so used to loops of this sort that we should pause and make some criticisms of the foreach (Iterator) concept found in many programming languages. Typically the iterator programming

paradigm implements foreach loops by having a method on the data structure (called GetEnumerator() or begin()) return an iterator; and once we have an iterator there should be methods such as hasNext() or end() to say if we have reched the end of the list, ++ or next() or MoveNext() to advance the iterator, and * or .Current to give the object at the current position. The details vary from one language to another (the above examples are variously from C++, C# and Java), but in no published case is there any way of implementing a shareable iterator of any sort, as all of them assume the same iterator object is kept during the iteration.

On the other hand, the pattern we have for traversing our list with a loop (Example 4a) looks great. Instead of an Iterator, we will use Bookmarks. We will have a function First() giving a Bookmark for the first element (if any) of a structure, and if we have a Bookmark we can have a function Next() that gives the a Bookmark for the next item in the structure (if any). Either of these can be null indicating that the list is empty or has no more entries.

We could formalise this by the following interfaces:

```
public interface Shareable
{
    Bookmark First();
}
public interface Bookmark
{
    Bookmark Next();
    Shareable Value();
    int Position();

}
```

Then the pattern for traversing a list (whenever we need to) becomes

```
        for (var b = list?.First(); b != null; b = b.Next())
            DoSomethingWith(b.Value());
```

We will do something very like this using generic definitions starting with the next section.

Summary: The SListOfInt class.

```
public class SListOfInt
{
    public readonly int element;
    public readonly SListOfInt next;
    public SListOfInt(int e, SListOfInt n) { element = e; next = n; }
    public bool Contains(int x)
    {
        return x == element || (next?.Contains(x) ?? false);
    }
    public int this[int n]
    {
        get { return (n == 0) ? element : next[n - 1]; }
    }
    public int Length
    {
        get { return 1 + (next?.Length ?? 0); }
    }
    public SListOfInt InsertAt(int x, int n)
    {
        if (n == 0)
            return new SListOfInt(x, this);
        return new SListOfInt(element, next.InsertAt(x, n - 1));
    }
    public SListOfInt RemoveAt(int n)
```

```
        {
            if (n == 0)
                return next;
            return new SListOfInt(element, next.RemoveAt(n - 1));
        }
    }
```

Classes or Structs?

Despite everything found in books if a struct has mutable fields then it will never have value semantics. In this booklet the distinction between classes and structs is not important. But for best results use classes for shareable types: assignment of a value (creation of a snapshot) is a single machine instruction.

A small advantage of using structs for your own shareable types in C# is that you can declare them as readonly struct, and this restriction will also apply to any structure inheriting from your shareable type. But there is currently no way to impose this as a constraint on a generic type parameter, so for the reason given above, you may prefer to stick to shareable class types as suggested here.

Using the code in this booklet

I don't like having to include a huge library in my executable software when I only want one or two classes. All of the class definitions in the remaining sections of this booklet will be made available on shareabledata.org in separate class files. The C# files will conform to Windows line termination, and the Java files will conform to Linux line termination (this policy will be kept under review).

All of the code uses the latest versions of C# and Java at the time of writing.

Importantly, the classes are made freely available for you to use. I would love to receive suggestions for improving them: email me at Malcolm.crowe@uws.ac.uk .

2 List and Array

In this chapter we give a generally-useful shareable SList<T> class. Instances of this class will be shareable lists whose elements have type T: T should be a shareable type. Unfortunately, there isn't currently a way of enforcing this.

Shareable<T>

As we are going to use generics, let us provide some abstract base classes:

```
public abstract class Shareable<T>
{
    public readonly int Length;
    protected Shareable(int c = 0) { Length = c; }
    abstract Bookmark<T> First();
    public T[] ToArray()
    {
        var r = new T[_count];
        for (var b = First(); b != null; b = b.Next())
            r[b.Position] = b.Value;
        return r;
    }
}
public abstract class Bookmark<T>
{
    public readonly int Position; // >=0
    protected Bookmark(int p) { Position = p; }
    public abstract Bookmark<T> Next();
    public abstract T Value { get; }
```

```
    }
```

For a motivation for these declarations, see the discussion above. We have made these into abstract classes so that we automatically have the ToArray() method. And the _count field ensures among other things that even an empty Shareable is not actually null (this helps with Java).

On the other hand we don't want millions of empty nodes so we will often make a special Empty Shareable<T> node and protect the constructor that makes an empty node so that everything shares the one Empty node. Empty is more useful than it would appear: it will be a single memory pointer, se we won't be calling a constructor to make it each time, and testing to see if this is Empty is really fast.

We can now see that ToArray will have the same code no matter what T is, so it might as well go in the base class.

The only new bits are the properties Length and Position. Note that these are not recursive (and so they are cheaper). The idea with Position is that First() should give a bookmark with position 0, and the Next() should have a position 1 higher. With more complex infrastructure we could make this setting automatic, but it would complicate things unduly. As with checking that all fields are declared public-readonly we will simply ask implementors to get things right.

SList<T>

```
namespace Shareable
{
    public class SList<T> : Shareable<T>
    {
        /// <summary>
        /// An empty list is Empty.
        /// SLists are never null, so don't test for null. Use Length>0.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        public readonly T element;
        public readonly SList<T> next;
        public static readonly SList<T> Empty = new SList<T>();
        protected SList() { element = default(T); next = null; }
        internal SList(T e, SList<T> n) : base(n.Length+1)
        {
            element = e;
            next = n;
        }
        public static SList<T> New(params T[] els)
        {
            var r = Empty;
            for (var i = els.Length - 1; i >= 0; i--)
                r = new SList<T>(els[i], r);
            return r;
        }
        public SList<T> InsertAt(T x, int n)
        {
            if (Length==0 || n == 0)
                return new SList<T>(x, this);
            return new SList<T>(element, next.InsertAt(x, n - 1));
        }
        public SList<T> RemoveAt(int n)
        {
            if (Length==0)
                return Empty;
            if (n == 0)
                return next;
            return new SList<T>(element, next.RemoveAt(n - 1));
```

```
        }
        public SList<T> UpdateAt(T x,int n)
        {
            if (Length==0)
                return Empty;
            if (n == 0)
                return new SList<T>(x, next);
            return new SList<T>(element, next.UpdateAt(x, n - 1));
        }
        public Bookmark<T> First()
        {
            return (Length==0)?null:new SListBookmark<T>(this);
        }
    }
    public class SListBookmark<T> : Bookmark<T>
    {
        internal readonly SList<T> _s;
        internal SListBookmark(SList<T> s, int p = 0) :base(p) { _s = s; }
        public override Bookmark<T> Next()
        {
            return (_s.Length <= 1) ? null
                     : new SListBookmark<T>(_s.next, Position+ 1);
        }
        public override T Value => _s.element;

    }

}
```

This is not much to add about the three methods InsertAt, RemoveAt and UpdateAt, which are very similar to the two methods of the last chapter.

The New method is not available in Java. It provides a way of creating an SList<T> from a (possibly empty) array T[] .The params keyword means that a client program can write SList<T>.New(a,b,c) if a, b, and c are of type T, as well as SList<T>.New(t) where t is an array of type T. Without the keyword params you would need to write SList<T>.New(new T[]{a,b,c}) . The New method is declared static because we don't need to have an Slist<T> already, and it can't be a ordinary constructor unless we are sure the parameter T[] is non-empty.

In Java it turns out to be easier to use null for an empty SList, and use the constructor to create a list with one element.

Finally, the ToArray() method is similar to the corresponding method in the usual List<T> class in returning an ordinary array of T. When compiling the Java version we need to suppress warnings as Java does not like generic array creation and we need an unchecked type conversion instead.

SCList<T>

This is a version of Slist for the case where elements of the list can be compared. There is not much to say about it. We override methods to ensure that things have the right types.

```
    /// <summary>
    /// An empty list is Empty.
    /// SCLists are never null, so don't test for null. Use Length>0.    ///
    /// </summary>
    /// <typeparam name="K"></typeparam>
    public class SCList<K> : SList<K>, IComparable where K : IComparable
    {
        public new static readonly SCList<K> Empty = new SCList<K>();
        SCList() { }
        public SCList(K el, SCList<K> nx) : base(el, nx) { }
        public new static SCList<K> New(params K[] els)
```

```
        {
            var r = Empty;
            for (var i = els.Length - 1; i >= 0; i--)
                r = new SCList<K>(els[i], r);
            return r;
        }
        public int CompareTo(object obj)
        {
            if (obj == null)
                return 1;
            var them = obj as SList<K> ?? throw new Exception("Type mismatch");
            SList<K> me = this;
            for (; me.Length > 0 && them.Length > 0; me = me.next, them = them.next)
            {
                var c = me.element.CompareTo(them.element);
                if (c != 0)
                    return c;
            }
            return (me.Length > 0) ? 1 : (them.Length > 0) ? -1 : 0;
        }
        public override Bookmark<K> First()
        {
            return (Length == 0) ? null : new SCListBookmark<K>(this);
        }
    }
    public class SCListBookmark<K> : SListBookmark<K> where K : IComparable
    {
        internal new readonly SCList<K> _s;
        internal SCListBookmark(SCList<K> s, int p = 0) : base(s, p) { _s = s; }
        public override Bookmark<K> Next()
        {
            return (_s.Length <= 1) ? null :
              new SCListBookmark<K>((SCList<K>)_s.next, Position + 1);
        }

    }
```

SArray<T>

For completeness we include a shareable array class SArray<T>, but it is not very useful in practice.

```
    public class SArray<T> : Shareable<T>
    {
        public readonly T[] elements;
        public SArray(params T[] els)
        {
            elements = new T[els.Length];
            for (var i = 0; i <els.Length; i++)
                elements[i] = els[i];
        }
        public SArray<T> InsertAt(int n,params T[] els)
        {
            var x = new T[elements.Length + els.Length];
            for (int i = 0; i < n; i++)
                x[i] = elements[i];
            for (int i = 0; i < els.Length; i++)
                x[i + n] = els[i];
            for (int i = n; i < elements.Length; i++)
                x[i + els.Length] = elements[i];
            return new SArray<T>(x);
        }
        public SArray<T> RemoveAt(int n)
        {
            var x = new T[elements.Length - 1];
```

```
            for (int i = 0; i < n; i++)
                x[i] = elements[i];
            for (int i = n + 1; i < elements.Length; i++)
                x[i - 1] = elements[i];
            return new SArray<T>(x);
        }
        public SArray<T> UpdateAt(T x, int n)
        {
            var a = new T[elements.Length];
            for (int i = 0; i < n; i++)
               a[i] = elements[i];
            a[n] = x;
            for (int i = n+1; i < elements.Length; i++)
                a[i] = elements[i];
            return new SArray<T>(a);
        }
        Public override Bookmark<T> First()
        {
            return (Length==0)? null : new SArrayBookmark<T>(this,0);
        }
    }
    public class SArrayBookmark<T> : Bookmark<T>
    {
        internal readonly SArray<T> _a;
        internal SArrayBookmark(SArray<T> a,int p) : base(p)
        {
            _a = a; Position = p;
        }
        public Bookmark<T> Next()
        {
            return (Position+1 >= _a.elements.Length) ? null
                     : new SArrayBookmark<T>(_a, Position+1);
        }

        public T Value()
        {
            return _a.elements[Position];
        }
    }
```

There is a lot of copying going on here with loops, and wasteful reallocation of memory. If the length of the array is likely to be large and/or there are a lot of changes, it will be generally faster and less memory-intensive to use SList<T> instead.

Unusally this bookmark retains a memory of the snapshot it is working on, and the current position in the array. This enables the bookmark to continue to traverse the snapshot it has been given even if the oiriginal source of the array has modified it. Moreover, anyone we give this Bookmark to will be able to continue using the snapshot it was created for.

An objection might be that any shareable copy of the SArray<T> is immutable, and we don't need a separate Bookmark class to traverse it. This is perfectly true: but we provide a Bookmark class (conforming to our standard pattern) for the same reason that ordinary arrays in Java have iterators. It means that we can get used to traversing any Shareable structure the same way. For example, in the ToArray() implementation.

3 Binary Search Trees

Recall that a search tree can be used to sort an array of items. The code for an unbalanced search tree is very simple, but it will perform very badly for sorting if the data is already in order. The code for the corresponding bookmark is more tricky and will be explained below.

```
    /// <summary>
```

```
/// Implementation of an UNBALANCED binary search tree
/// </summary>
/// <typeparam name="T"></typeparam>
public class SSearchTree<T> :Shareable<T>
    where T : System.IComparable
{
    public readonly T node;
    public readonly SSearchTree<T> left, right;
    public static readonly SSearchTree<T> Empty = new SSearchTree<T>();
    SSearchTree() { node = default(T); left = null; right = null; }
    internal SSearchTree(T n,SSearchTree<T> lf,SSearchTree<T> rg)
        : base(1+lf.Length+rg.Length)
    {
        node = n;
        left = lf;
        right = rg;
    }
    public static SSearchTree<T> New(params T[] els)
    {
        var r = Empty;
        foreach (var t in els)
            r = r.Add(t);
        return r;
    }
    public SSearchTree<T>Add(T n)
    {
        if (this == Empty)
            return new SSearchTree<T>(n, Empty, Empty);
        var c = n.CompareTo(node);
        if (c <= 0)
            return new SSearchTree<T>(node, left.Add(n),right);
        else
            return new SSearchTree<T>(node, left, right.Add(n));
    }
    public bool Contains(T n)
    {
        if (this == Empty)
            return false;
        var c = n.CompareTo(node);
        return (c == 0) ? true : (c < 0) ? left.Contains(n) : right.Contains(n);
    }
    public override Bookmark<T> First()
    {
        return (this==Empty)?null:new SSearchTreeBookmark<T>(this, true);
    }
}
public class SSearchTreeBookmark<T> : Bookmark<T> where T : System.IComparable
{
    internal readonly SSearchTree<T> _s;
    internal readonly SList<SSearchTree<T>> _stk;
    internal SSearchTreeBookmark(SSearchTree<T> s, bool doLeft,
        SList<SSearchTree<T>> stk= null,
        int p = 0) : base(p)
    {
        if (stk==null) stk = SList<SSearchTree<T>>.Empty;
        for (;doLeft && s.left != SSearchTree<T>.Empty; s = s.left)
            stk = stk.InsertAt(s, 0);
        _s = s; _stk = stk;
    }
    public override T Value => _s.node;
    public override Bookmark<T> Next()
    {
```

```
            return (_s.right != SSearchTree<T>.Empty)?
                new SSearchTreeBookmark<T> (_s.right, true, _stk, Position + 1)
                : (_stk == SList<SSearchTree<T>>.Empty) ? null
                : new SSearchTreeBookmark<T> (_stk.First().Value, false,
                    _stk.RemoveAt(0), Position + 1);
        }
    }
```

The Bookmark class does the traversal. In books you we see a traversal written as

```
void Traverse(Tree n, ref int i)
{
      If (n==null) return;
      Traverse(n.left,ref i);
      DealWith(n.node,i++);
      Traverse(n.right,ref i);
}
```

This means that the First() node visited will involve moving down the left branches as far as possible, building a stack of nodes to be visited later. Next() involves going right from current node if possible, otherwise taking the top off the stack and using it without first going down its left branches.

The New method isn't allowed in Java.

For now we observe that in the worst case Add-ing a node to a tree of size N might create N new nodes. This means that the worst case for building a tree of size N will involve N(N-1)+1 steps. This is not good.

On the other hand the code is very safe, as cycles cannot occur.

In the next chapter we will present a B-tree implementation for SDict<K,V>, and we will find that SDict<T,bool> gives a more efficient sorting algorithm than SSearchTree<T>.

4 B-Trees

The next example is a shareable implementation of B-trees called SDict. B-Trees are not binary trees: we avoid calling them BTree in the code to avoid confusion. These manage sorted lists of key-value pairs using an n-ary tree (so we have two type parameters SDict<K,V>). The design is quite standard and described in many textbooks, but is rather complex to code: it usually includes a self-balancing part of the algorithm called spilling. We avoid spilling in our implementation: this is because without spilling the only new nodes (for add or remove) are on the path from the root to the key's location; whereas with spilling all of the nodes on the same level as the key's location might get replaced. Without spilling we continue to avoid the worst-case scenario mentioned above, but not all paths from root to leaves have the same length.

In a binary search tree described above each node has a value and two child nodes: we can think of such a node as (k,T1,T2) where T1 is a subtree of values less than k and T2 is a subtree of values greater than k (T1 and/or T2 may be empty).

With the B-tree, nodes can have a number of children up to a fixed maximum S, which is usually 2^n for leaf nodes and 2^n+1 for inner nodes, for some n. At lower values of S the memory allocator works harder, and at higher values there is less sharing of nodes between old and new trees, in experiments these aspects tend to balance out so that values of S in the range 8 to 33 are satisfactory. The number of comparisons needed to locate an entry is O(log N) in any case because binary search is used within nodes O(log n) and the depth of the B-tree will be O(log (N/n)).

Leaf nodes contain key-value pairs (k,v) say. Inner nodes contain an ordered group of pairs (k,T) and a subtree G; where each T is a non-empty subtree (either an inner node or a leaf node) whose

greatest key value is k, and G is a subtree containing larger than the last k. All nodes apart from the root have at least S/2 children; the root can have fewer (if the tree is empty it will have none).

If the tree has fewer than S keys, there will be just one node in the tree (the root). At this point the addition of another key will cause the node to split, and the two resulting nodes will becoime children of a new root node. When nodes are removed, the reverse process happens, and nodes with fewer than S/2 children will be coalesced, reducing the depth of the tree.

For consistency with the previous chapter, let us consider an SDict<K,V> to be a Shareable list of pairs (K,V), so we define an auxiliary class called SSlot<K,V> (whose fields key and val are public readonly of course). This makes SDict<K,V> a subclass of Shareable<SSlot<K,V>>, and if we apply our bookmark machinery we will have a bookmark of type SBookmark<SSlot<K,V>>, so for convenience we create a subclass SDictBookmark<K,V>

It is clear from this discussion that there should be two classes of nodes, for inner and leaf nodes, and it makes sense to have a common base class, which we call SBucket<K,V>. Again all of these classes will have public readonly fields.

In C# it also helps to have an common interface, which we call IBucket: it helps the SDictBookmark to access the current position in the bucket as well as the number of slots in a bucket.

SDict<K,V>

From the above discussion we have quite a group of new classes just for implementation of SDict<K,V>, so we will not give full listings of everything in this document. SDict<K,V> will have an API rather like the classic Hashtable<K,V> or Dictionary<K,V> classes in standard libraries, and its implementation will have methods that call the methods of the Shareable base class (and SSlot).

SDict<K,V> itself is clear enough:

```
public class SDict<K,V> : Shareable<SSlot<K,V>> where K: IComparable
{
    /// <summary>
    /// Size is a system configuration parameter:
    /// the maximum number of entries in a Bucket.
    /// It should be an even number, preferably a power of two.
    /// It is const, since experimentally there is little impact on performance
    /// using values in the range 8 to 32.
    /// </summary>
    public const int Size = 8;
    public virtual int Count { get { return root?.total ?? 0; }}
    public readonly SBucket<K, V> root;
    protected SDict(SBucket<K,V> r) { root = r; }
    internal SDict(K k, V v) : this(new SLeaf<K, V>(new SSlot<K, V>(k, v))) { }
    /// <summary>
    /// Avoid unnecessary constructor calls by using this constant empty tree
    /// </summary>
    public readonly static SDict<K, V> Empty = new SDict<K, V>(null);
    /// <summary>
    /// Add a new entry or update an existing one in the tree
    /// </summary>
    /// <param name="k">the key</param>
    /// <param name="v">the value to add</param>
    /// <returns>the modified tree</returns>
    public virtual SDict<K,V> Add(K k,V v)
    {
        return (root == null || root.total == 0)? new SDict<K, V>(k, v) :
            (root.Contains(k))? new SDict<K,V>(root.Update(k,v)) :
            (root.count == Size)? new SDict<K, V>(root.Split()).Add(k, v) :
            new SDict<K, V>(root.Add(k, v));
    }
```

```
        /// <summary>
        /// Remove an entry from the tree (Note: we won't have duplicate keys)
        /// </summary>
        /// <param name="k"></param>
        /// <returns></returns>
        public virtual SDict<K,V> Remove(K k)
        {
            return (root==null || root.Lookup(k)==null) ? this :
                (root.total == 1) ? Empty :
                new SDict<K, V>(root.Remove(k));
        }
        public bool Contains(K k)
        {
            return (root == null) ? false : root.Contains(k);
        }
        public V Lookup(K k)
        {
            return (root==null)?default(V):root.Lookup(k);
        }
        /// <summary>
        /// Start a traversal of the tree
        /// </summary>
        /// <returns>A bookmark for the first entry or null</returns>
        public override Bookmark<SSlot<K, V>> First()
        {
            return (SBookmark<K, V>.Next(null, this) is SBookmark<K,V> b)?
                new SDictBookmark<K, V>(b):null;
        }
    }
```

We see in the above Add and Remove methods some indications of the tree restructuring dicsussion above. The SDictBookmark class is also simple enough:

```
    public class SDictBookmark<K, V> : Bookmark<SSlot<K, V>> where K : IComparable
    {
        public readonly SBookmark<K, V> _bmk;
        public override SSlot<K, V> Value =>
            ((SLeaf<K,V>)_bmk._bucket).slots[_bmk._bpos];
        internal SDictBookmark(SBookmark<K,V> bmk) :base(bmk.position())
            { _bmk = bmk; }
        public K key => Value.key;
        public V val => Value.val;
        public override Bookmark<SSlot<K, V>> Next()
        {
            return (_bmk.Next() is SBookmark<K,V> b)? new SDictBookmark<K,V>(b):null;
        }
    }
```

We can see an advantage in defining this class in that we have shortcut properties key and val that access the corresponding fields in the Value SSlot. More importantly note that the standard First() and Next() methods return instances of the SDictBookmark class instead of Bookmark<SSlot<K,V>>.

For further details , see the code files.

5 Multi-level indexes

The next sample is a shareable implementation of multi-level indexes, allowing duplicate and null keys, here called SMTree<K,int>, since for simplicity we assume the records being indexed are in a list somewhere (e.g. on disk), and retrieved using an integer address. The multi-level key consists of several columns (an SCList), and the implementation uses a B-Tree for each one with the final level leading to an integer value identifying the record. If a level allows duplicate key values, there will be an extra SDict<pos,true> that can be traversed to find all of the rows with matching keys (pos gives

the disk location of the record). From the outside, the MTree is a SDict<SList,int>, where the key type is the multi-column key. In principle, the lookup for a given key (k0,k1,..,kn) proceeds as T1 = M.Lookup(k0), T2=T1.Lookup(k1), and so on until pos=Tn.Lookup(kn) where the desired row is then SA[pos] , however, the partially-ordered stage if any complicates this picture, as do null key values (e.g. if we allow a short key where later fields of a key might be null).

SMTree

From this description we can already see that the implementation will require a number of auxiliary classes and enumerations. We will already need an SList<TreeInfo> to describe the multicolum key class. Then a single level of the implementation can be done be an internal class SITree<K1,V1>, say. But its K1 and V1 types can't be just K and int, since the intermediate objects T1, T2, .. in the above discussion can be trees of at least two kinds (partial and compound) and the last value will be an integer. So we invent a Variant class to handle this variety and an enumeration Variants for the case that arise.

To implement the recursive process described here, it is convenient to generalise the SMTree to use the Vaiant class we have just introduced. A shortened version of the SMTree class can be represented as follows:

```
public class SMTree : SDict<SCList<Variant>, Variant>
{
    public class SITree ...
    public readonly SITree _impl;
    public readonly SList<TreeInfo> _info;
    public readonly int _count;
    public override int Count => _count;
    SMTree(SList<TreeInfo> ti, SITree impl,int c) :base(null)
    {
        _info = ti;
        _impl = impl;
        _count = c;
        if (ti.Length>1 && ti.element.onDuplicate != TreeBehaviour.Disallow)
            throw new Exception("Duplicates are allowed only on last TreeInfo");
    }
    public SMTree(SList<TreeInfo> ti) : this(ti, (SITree)null, 0)  { }
    public SMTree(SList<TreeInfo> ti,SList<Variant> k,int v) :this(ti) ...
    public override Bookmark<SSlot<SCList<Variant>,Variant>> First() ...
    public MTreeBookmark PositionAt(SCList<Variant> k) ...
    public SMTree Add(int v,params Variant[] k)
    {
        var r = Add(SCList<Variant>.New(k), v,out TreeBehaviour tb);
        return (tb == TreeBehaviour.Allow) ? r :
          throw new Exception(tb.ToString());
    }
    public SMTree Add(SCList<Variant> k,int v, out TreeBehaviour tb) ...
    public override SDict<SCList<Variant>, Variant>
          Add(SCList<Variant> k, Variant v) ...
}
```

The MTreeBookmark class implements the recursion decsribed in the text, with an outer Bookmark for the current level, and an MTreeBookmark for the inner recursion if any.

```
public class MTreeBookmark : Bookmark<SSlot<SCList<Variant>, Variant>>
{
    readonly SDictBookmark<Variant,Variant> _outer;
    internal readonly SList<TreeInfo> _info;
    internal readonly MTreeBookmark _inner;
    readonly Bookmark<SSlot<int, bool>> _pmk;
    internal readonly bool _changed;
    internal SCList<Variant> _filter;
```

```
    MTreeBookmark(SDictBookmark<Variant, Variant> outer, SList<TreeInfo> info,
        bool changed, MTreeBookmark inner,Bookmark<SSlot<int, bool>> pmk,
        int pos, SCList<Variant> key=null) :base(pos)
    {
        _outer = outer; _info = info; _changed = changed;
        _inner = inner; _pmk = pmk; _filter = key;
    }
    public SCList<Variant> key()
    {
        if (_outer == null)
            return null;
        return new SCList<Variant>(_outer.key, _inner?.key());
    }
    public int value()
    {
        return (_inner!=null)?_inner.value() : (_pmk!=null)?_pmk.Value.key :
            (_outer.val!=null)?(int)_outer.val.ob : 0;
    }
    public override Bookmark<SSlot<SCList<Variant>, Variant>> Next() ...
}
```

Full details can be found in the code files.

6 Files and Streams

It is natural to ask how files and streams can be handled with a shareable infrastructure. To start the discussion we observe that a sequential file generally may have multiple readers but at most one writer. We will wish to avoid any kind of file that allows arbitrary changes to file contents, so we will limit our discussion to the “append storage” model where the only changes to a disk file involve the addition of new data at the end, and “Direct access” to the file contents is allowed for the readers. We can easily implement shareable access classes to disk files that embody these ideas, such as SBuffer, which is a shareable contiguous block of data read from the file, with a readonly integer giving the current position within it.

Streams can be viewed as a special kind of SLIst: an input stream is like a queue of objects to be processed (so that taking off the head object gives a new input stream), and an output stream is like a list where new objects are added at the end. Standard input, standard output, and web services are the most common uses of streams. It is quite common for these streams to be passed between threads, and making them into shareable classes is not a large step.

There are choices to be made as to the level of abstraction: does the file/stream consist of a sequence of characters, or strings, or records? This means that the shareable class for files and streams should also have have a generic type parameter for the objects in the file or stream. Generally, these types will be serialisable Strangely, the seralisation classes in .NET do not support shareability, so we provide our own Seralisable framework in this section.

Since the implementation of Serialisable will be based on low-level framework classes for input/output and network access, the implementation of i/o will need to use locking to ensure the shareable behaviour we require. We will create a subclass of FileStream called AStream with the methods we need for the implementation.

Types

The enumeration Types gives a list of 18 types starting with Serialisable, which will be enough for us to implement a simple DBMS with Tables, Columns, Views and Transactions.

```
    public enum Types
    {
        Serialisable = 0,
        STimestamp = 1,
```

```
        SInteger = 2,
        SNumeric = 3,
        SString = 4,
        SDate = 5,
        STimeSpan = 6,
        SBoolean = 7,
        SRow = 8,
        STable = 9,
        SColumn = 10,
        SRecord = 11,
        SUpdate = 12,
        SDelete = 13,
        SAlter = 14,
        SView = 15,
        STransaction = 16,
        SPartial = 17,
        SCompound = 18
    }
```

Serialisable

Serialisable has no public or internal constructors: its role is as a base class for the Serialisable types. All the other types listed in the Types enumeration have the following basic operations:

- A public constructor to create a new instance of the Serialisable class (not associated with a file).
- A static Get method to deserialise an instance of the type from a stream. There will be a protected constructor with parameter StreamBase to create this new instance. There will be two sorts of Stream: AStream which is a database file, and AsyncStreams which are for communication with the database client.
- A virtual Put method to serialise an instance to a stream. Other serializable classes will call the base.Put first and then serialise their extra fields to the stream.

```
    public class Serialisable
    {
        public readonly Strong type;
        protected Serialisable(Types t)
        {
            type = t;
        }
        public Serialisable(Types t, AStream f)
        {
            type = t;
        }
        public static Serialisable Get(StreamBase f)
        {
            return new Serialisable(Types.Serialisable);
        }
        public virtual void Put(StreamBase f)
        {
            f.WriteByte((byte)type);
        }
...
    }
```

StreamBase

This is a standard subclass of FileStream with methods such as GetInt, PutInt for implementing the serialisable types. It also has a Buffer class for buffering the input and output. It is not a shareable data structure.

AStream

AStream is mutable and not shareable, and is a subclass of StreamBase. It has methods for obtaining database objects and data from the database file (GetOne, GetAll, etc), Commit for appending new objects and data, and and a constructor that obtains exclusive access (FileShare.None) to the underlying FileStream object. All calls to its Get and Put methods are protected within the Commit and Get/GetOne methods (Commit will be protected in Transaction, see next chapter).

```
public class AStream : StreamBase
 {
     public readonly string filename;
     internal FileStream file;
     long position = 0, length = 0;
     Buffer rbuf, wbuf;
     public AStream(string fn)
     {
         filename = fn;
         file = new FileStream(fn,FileMode.Open,FileAccess.ReadWrite,
                FileShare.None);
         length = file.Seek(0, SeekOrigin.End);
         file.Seek(0, SeekOrigin.Begin);
     }
     Serialisable _Get(long pos)
     {
         rbuf = new Buffer(this, position);
         Types tp = (Types)ReadByte();
         Serialisable s = null;
         switch (tp)
         {
             case Types.Serialisable: s = Serialisable.Get(this); break;
                  ...
         }
         return s;
     }
     public Serialisable GetOne()
     {
         lock (file)
         {
             if (position == file.Length)
                 return null;
             var r = _Get(position);
             position = rbuf.start + rbuf.pos;
             return r;
         }
     }
     public Serialisable Get(long pos)
     {
         lock (file)
         {
             return _Get(pos);
         }
     }
 ...
 }
```

7 Databases and Transactions

A relational database begins with a list of tables, and a file for persisting their state to disk. If the disk file is the transaction log, then it provides a guarantee that transactions are serialisable, so we follow this design. If our database is shareable, then we automatically get snapshot isolation. We will build

on these two simple properties to implement a strong DBMS based on shareable objects wherever possible.

As a preparation for this development, the list of Seralisable types given in the last section contains useful shareable objects such as STable, SColumn etc. But before it is snesible to give the full details, we need to sketch out the database and transaction structure.

SDatabase

Asd the name implies, SDatabase is shareable. It has a list of database objects accessible by name (such as STables) , and the same list of STables accessible by uid. Uids are long integers consisting of file positions in the transaction log.

The class has a static lock for manipulating the list of AStream files that the DBMS has exclusive access to.

```
public class SDatabase
{
    public readonly string name;
    public readonly SDict<long, STable> tables;
    public readonly SDict<string, SDbObject> objects;
    public readonly long curpos;
    static object files = new object(); // a lock
    protected static SDict<string,AStream> dbfiles = SDict<string,AStream>.Empty;
    protected static SDict<string, SDatabase> databases =
           SDict<string,SDatabase>.Empty;
    public static SDatabase Open(string fname)
    {
        if (dbfiles.Contains(fname))
            return databases.Lookup(fname);
        var db = new SDatabase(fname);
        lock (files)
        {
            dbfiles = dbfiles.Add(fname, new AStream(fname));
            databases = databases.Add(fname, db);
        }
        return db.Load();
    }
   ...

   }
```

STransaction

A database transaction is about new or updated objects and data, and committing the transaction involves writing these to the transaction log. This results in an updated Database. With a shareable Database structure this means a new Database instance, but of course as discussed above the new instance will contain only a few new or changed allocated memory locations.

A Transaction object will hold a list of Serialisable objects to be committed to the AStream. The Commit method is simple if there is no concurrency. If there are concurrent transactions there will be more code in this method to check for conflicts with the Serialisables recorded since the start of the transaction.

```
public class STransaction : SDatabase
{
    // uids above this number are for uncommitted objects
    public static readonly long _uid = 0x80000000;
    public readonly long uid;
    public readonly SDict<int,Serialisable> steps;
```

```
        public Transaction(SDatabase d)
        {
            uid = _uid;
            steps = SDict<int,Serialisable>.Empty;
        }
        /// <summary>
        /// This routine is public only for testing the transaction mechanism
        /// on non-database objects.
        /// Constructors of Database objects will internally call this method.
        /// </summary>
        public STransaction(STransaction tr,Serialisable s) :base(tr)
        {
            steps = tr.steps.Add(tr.steps.Count,s);
            uid =  tr.uid+1;
        }
...
        /// <summary>
        /// We will single-quote transaction-local uids
        /// </summary>
        /// <returns>a more readable version of the uid</returns>
        internal static string Uid(long uid)
        {
            if (uid > _uid)
                return "'" + (uid - _uid);
            return "" + uid;
        }
...
         public SDatabase Commit()
         {
...
         }
    }
```

We discuss the Commit() method below. The basic idea is clear: when we Commit a transaction we get a modified SDatabase, obtained by serialising the Serialsables to the (possibly updated) database file. We only do this once we are sure that the database has received no updates that conflict with the steps of the transaction. For details see the Conflicts section below.

SDbObject

For database objects such as STable, we will want to record a unique id based on the actual position in the transaction log, so the Get and Commit methods will capture the appropriate file positions in AStream – this is why the Commit method needs to create a new instance of the Serialisable. The uid will initially belong to the Transaction. Once committed the uid will become the position in the AStream file.

```
    public abstract class SDbObject : Serialisable
    {
        public readonly long uid;
        /// <summary>
        /// For a new database object we add it to the transaction steps
        /// and set the transaction-based uid
        /// </summary>
        /// <param name="t"></param>
        /// <param name="tr"></param>
        protected SDbObject(Types t,Transaction tr) :base(t)
        {
            uid = tr.Add(this);
        }
        /// <summary>
        /// A modified database obejct will keep its uid
```

```
        /// </summary>
        /// <param name="s"></param>
        protected SDbObject(SDbObject s) : base(s.type)
        {
            uid = s.uid;
        }
        /// <summary>
        /// A database object got from the file will have
        /// its uid given by the position it is read from
        /// </summary>
        /// <param name="t"></param>
        /// <param name="f"></param>
        protected SDbObject(Types t,StreamBase f) : base(t)
        {
            uid = f.Position;
        }
        /// <summary>
        /// During commit, database objects are appended to the
        /// file, and we will have a (new) modified database object
        /// with its file position as the uid.
        /// </summary>
        /// <param name="s"></param>
        /// <param name="f"></param>
        protected SDbObject(SDbObject s,AStream f) :base(s.type)
        {
            uid = f.Length;
            f.WriteByte((byte)s.type);
        }
        internal bool Committed => uid < Transaction._uid;
        internal string Uid()
        {
            return Transaction.Uid(uid);
        }
    }
```

STable

There are methods for adding and dropping columns and rows.

The code for Commit is interesting. A statement for defining a table will create an STable and several SColumns. The commit for STable will therefore have to update the table reference in the columns that remain to be committed. Although most SQL statements are committed and they are executed, it is possible for batch together number of SQL statements in a single explict transaction, so the table may also have a set of rows whose table references also need to be updated.

In addition to the methods shown here, there will be methods for detecting conflicts with other serialisables. For example, an STable Serialisable defines a new table by name, so a conflict should be detected if another transaction created a table with the same name since the start of our transaction. We return to this point at this end of this section.

```
    public class STable : SDbObject
    {
        public readonly string name;
        public readonly SDict<long,SColumn> cols;
        public readonly SDict<long, long> rows; // defpos->uid of latest update
        public STable(Transaction tr,string n) :base(Strong.STable)
        {
            if (tr.objects.Contains(n))
                throw new Exception("Table n already exists");
            name = n;
            cols = SDict<long,SColumn>.Empty;
```

```
        rows = SDict<long, long>.Empty;
    }
    public STable Add(SColumn c)
    {
        return new STable(this,cols.Add(c.uid,c));
    }
    public STable Update(SColumn c)
    {
        return new STable(this, cols.Add(c.uid, c));
    }
    public STable Add(SRecord rec)
    {
        return new STable(this,rows.Add(rec.Defpos, rec.uid));
    }
    public STable Remove(long n)
    {
        if (cols.Contains(n))
            return new STable(this, cols.Remove(n));
        else
            return new STable(this, rows.Remove(n));
    }
    STable(STable t,SDict<long,SColumn> c) :base(t)
    {
        name = t.name;
        cols = c;
        rows = t.rows;
    }
    STable(STable t,SDict<long, long> r) : base(t)
    {
        name = t.name;
        cols = t.cols;
        rows = r;
    }
    STable(SDatabase d,StreamBase f):base(Types.STable,f)
    {
        name = f.GetString();
        cols = SDict<long,SColumn>.Empty;
        rows = SDict<long, SRecord>.Empty;
    }
    STable(SDatabase tr,STable t,AStream f) :base(t,f)
    {
        name = t.name;
        // if we already have columns, they need to be updated
        var nc = SDict<long,SColumn>.Empty;
        for (var b = t.cols.First(); b != null; b = b.Next())
            nc = nc.Add(b.Value.key,new SColumn(b.Value.val, uid));
        cols = nc;
        rows = t.rows;
    }
    /// <summary>
    /// Database objects should only be committed once.
    /// So we only commit a table when it is first mentioned.
    /// Its new columns get committed as they are defined.
    /// </summary>
    /// <param name="f"></param>
    /// <returns></returns>
    public override Serialisable Commit(Transaction tr,AStream f)
    {
        if (Committed) // nothing to do!
            return this;
        var r = new STable(this, f);
        base.Commit(tr,f);
```

```
            f.PutString(name);
            return r;
        }
        public static STable Get(SDatabase d,AStream f)
        {
            return new STable(d,f);
        }
        public override string ToString()
        {
            return "Table "+name+"["+Uid()+"]";
        }

    }
```

SColumn

```
    public class SColumn : SDbObject
    {
        public readonly string name;
        public readonly Types dataType;
        public readonly long table;
        public SColumn(Transaction tr,string n, Types t, long tbl) :
base(Types.SColumn,tr)
        {
            name = n; dataType = t; table = tbl;
        }
        internal SColumn(SColumn c,long t) :base (c)
        {
            name = c.name; dataType = c.dataType;
            table = t;
        }
        SColumn(SDatabase d,StreamBase f) :base(Types.SColumn,f)
        {
            name = f.GetString();
            dataType = (Types)f.ReadByte();
            table = f.GetLong();
        }
        public SColumn(SDatabase tr,SColumn c,AStream f):base (c,f)
        {
            name = c.name;
            dataType = c.dataType;
            table = tr.Fix(c.table);
            f.PutString(name);
            f.WriteByte((byte)dataType);
            f.PutLong(table);
        }
        public static SColumn Get(SDatabase d,StreamBase f)
        {
            return new SColumn(d,f);
        }
       ...
        public override string ToString()
        {
            return "Column " + name + " [" + Uid() + "]: " + dataType.ToString();
        }
    }
```

SRecord

There is an interesting point here: we will allow renaming of columns and tables, but it will be expensive to replace records if column names change, since records committed both before and after the change need to refer to the updated columns. So on disk we record the column uid instead of the column name for fields.

This makes sense for SRecord, because we will not be keeping all SRecords in memory: they will be refetched from disk when we want them.

```
    public class SRecord : SDbObject
    {
        public readonly SDict<string, Serialisable> fields;
        public readonly long table;
        public SRecord(Transaction tr,long t,SDict<string,Serialisable> f)
:base(Types.SRecord,tr)
        {
            fields = f;
            table = t;
        }
        public virtual long Defpos => uid;
        protected SRecord(SRecord r,long tb) :base(r)
        {
            fields = r.fields;
            table = tb;
        }
        public SRecord(SDatabase tr,SRecord r,AStream f) : base(r,f)
        {
            table = tr.Fix(r.table);
            fields = r.fields;
            f.PutLong(table);
            var tb = tr.tables.Lookup(table);
            f.PutInt(fields.Count);
            for (var b=fields.First();b!=null;b=b.Next())
            {
                var k = b.Value.key;
                long p = 0;
                for (var c = tb.cols.First(); c != null; c = c.Next())
                    if (c.Value.val.name == k)
                        p = c.Value.key;
                f.PutLong(p);
                b.Value.val.Commit(tr,f);
            }
        }
        protected SRecord(SDatabase d,StreamBase f) : base(Types.SRecord,f)
        {
            table = f.GetLong();
            var n = f.GetInt();
            var tb = d.tables.Lookup(table);
            var a = SDict<string, Serialisable>.Empty;
            for(var i = 0;i< n;i++)
            {
                var k = tb.cols.Lookup(f.GetLong());
                a = a.Add(k.name, f.GetOne(d));
            }
            fields = a;
        }
        public static SRecord Get(SDatabase d,StreamBase f)
        {
            return new SRecord(d,f);
        }
        protected void Append(StringBuilder sb)
        {
            sb.Append(" for "); sb.Append(Uid());
            var cm = "(";
            for (var b = fields.First(); b != null; b = b.Next())
            {
                sb.Append(cm); cm = ",";
                sb.Append(b.Value.key); sb.Append("=");
```

```
                sb.Append(b.Value.val.ToString());
            }
            sb.Append(")");
        }
...
        public override string ToString()
        {
            var sb = new StringBuilder("Record ");
            sb.Append(Uid());
            Append(sb);
            return sb.ToString();
        }
    }
```

SUpdate

The SUpdate record preserves the Defpos of the updated SRecord. For efficiency all of the current fields are recorded in the SUpdate.

```
public class SUpdate : SRecord
    {
        public readonly long defpos;
        public SUpdate(Transaction tr,SRecord r) : base(tr,r.table,r.fields)
        {
            defpos = r.Defpos;
        }
        public override long Defpos => defpos;
        SUpdate(SUpdate u,long tbl,long dp) :base(u,tbl)
        {
            defpos = u.defpos;
        }
        public SUpdate(SDatabase tr,SUpdate r, AStream f) : base(tr,r,f)
        {
            defpos = tr.Fix(defpos);
            f.PutLong(defpos);
        }
        SUpdate(SDatabase d,StreamBase f) : base(d,f)
        {
            defpos = f.GetLong();
        }
        public new static SRecord Get(SDatabase d,StreamBase f)
        {
            return new SUpdate(d,f);
        }
...
        public override string ToString()
        {
            var sb = new StringBuilder("Update ");
            sb.Append(Uid());
            sb.Append(" of "); sb.Append(Transaction.Uid(defpos));
            Append(sb);
            return sb.ToString();
        }
    }
```

SDelete

```
public class SDelete : SDbObject
    {
        public readonly long table;
        public readonly long delpos;
        public SDelete(Transaction tr, long t, long p) : base(Types.SDelete,tr)
        {
            table = t;
```

```
            delpos = p;
        }
        internal SDelete(SDelete u, long tbl, long del) : base(u)
        {
            table = tbl;
            delpos = del;
        }
        public SDelete(SDatabase tr, SDelete r, AStream f) : base(r,f)
        {
            table = tr.Fix(r.table);
            delpos = tr.Fix(r.delpos);
            f.PutLong(table);
            f.PutLong(delpos);
        }
        SDelete(SDatabase d, StreamBase f) : base(Types.SDelete,f)
        {
            table = f.GetLong();
            delpos = f.GetLong();
        }
        public static SDelete Get(SDatabase d, StreamBase f)
        {
            return new SDelete(d, f);
        }
...
        public override string ToString()
        {
            var sb = new StringBuilder("Delete ");
            sb.Append(Uid());
            sb.Append(" of "); sb.Append(Transaction.Uid(delpos));
            sb.Append("["); sb.Append(Transaction.Uid(table)); sb.Append("]");
            return sb.ToString();
        }
    }
```

Conflicts

This is not another class, just the additions to the above code to detect conflicts. Note that the definitions of Conflicts should be such that a.Conflicts(b) iff b.Conflicts(a).

In AStream:

```
        /// <summary>
        /// Called from Transaction.Commit(): file is already locked
        /// </summary>
        /// <param name="tr"></param>
        /// <param name="pos"></param>
        /// <returns></returns>
        public Serialisable[] GetAll(SDatabase d, long pos, long max)
        {
            var r = new List<Serialisable>();
            position = pos;
            rbuf = new Buffer(this, pos);
            while (position<file.Length)
            {
                r.Add(_Get(d));
                position = rbuf.start + rbuf.pos;
            }
            return r.ToArray();
        }

        public SDatabase Commit(SDatabase db,SDict<int,Serialisable> steps)
        {
            wbuf = new Buffer(this);
```

```
            uids = SDict<long, long>.Empty;
            for (var b=steps.First();b!=null; b=b.Next())
            {
                switch (b.Value.val.type)
                {
                    case Types.STable:
                        {
                            var st = (STable)b.Value.val;
                            var nt = new STable(db, st, this);
                            db = new SDatabase(db,nt,Length);
                            break;
                        }
```

...

```
              }
```

In Transaction.Commit:

```
        public SDatabase Commit()
        {
            AStream dbfile = dbfiles.Lookup(name);
            SDatabase db = databases.Lookup(name);
            long pos = 0;
            var since = dbfile.GetAll(this, curpos, db.curpos);
            for (var i = 0; i < since.Length; i++)
                for (var b = steps.First(); b != null; b = b.Next())
                    if (since[i].Conflicts(b.Value.val))
                        throw new Exception("Transaction Conflict on " + b.Value);
            lock (dbfile.file)
            {
                db = databases.Lookup(name);
                since = dbfile.GetAll(this, pos,dbfile.Length);
                for (var i = 0; i < since.Length; i++)
                    for (var b = steps.First(); b != null; b = b.Next())
                        if (since[i].Conflicts(b.Value.val))
                            throw new Exception("Transaction Conflict on " + b.Value);
                db = dbfile.Commit(db,steps);
            }
            Install(db);
            return db;
        }
```

In Serialisable:

```
        public virtual bool Conflicts(Serialisable that)
        {
            return false;
        }
```

In STable:

```
        public override bool Conflicts(Serialisable that)
        {
            switch (that.type)
            {
                case Types.STable:
                    return ((STable)that).name.CompareTo(name) == 0;
            }
            return false;
        }
```

In SColumn:

```
public override bool Conflicts(Serialisable that)
{
    switch (that.type)
    {
        case Types.SColumn:
            {
                var c = (SColumn)that;
                return c.table == table && c.name.CompareTo(name) == 0);
            }
    }
    return false;
}
```

In SRecord:

```
public override bool Conflicts(Serialisable that)
{
    switch(that.type)
    {
        case Types.SDelete:
            return ((SDelete)that).delpos == Defpos;
    }
    return false;
}
```

In SUpdate:

```
public override bool Conflicts(Serialisable that)
{
    switch (that.type)
    {
        case Types.SUpdate:
            return ((SUpdate)that).Defpos == Defpos;
    }
    return base.Conflicts(that);
}
```

In SDelete:

```
public override bool Conflicts(Serialisable that)
{
    switch(that.type)
    {
        case Types.SUpdate:
            return ((SUpdate)that).Defpos == delpos;
        case Types.SRecord:
            return ((SRecord)that).Defpos == delpos;
    }
    return false;
}
```

7 Parsing Input

Our DBMS will need to process a subset of SQL, so let's implement some shareable parsing infastructure.