Shareable Data Structures

MALCOLM CROWE

28 MAY 2019

Data Structures

- What sets you apart from the rest
  - Threading, inter-process comms maybe
  - Code reusability and safety
- Reusability implies a contract for use
  - Should not have hidden surprises
  - E.g. I pass a final/readonly parameter
    - And someone else is able to change it
  - Maybe Java's unmodifiableList(..) ?
    - Relies on caller to do it right
    - Copy-on-write

String in Java and C#

- These are immutable classes
  - Can’t change a string using x[i]=‘A’;
  - Any modification (e.g. +=) creates new
- If you really want a modifiable char[]
  - You need to copy it elementwise
- It’s a wrapper around a modifiable list
  - Better to make everything shareable

We learn about cloning?

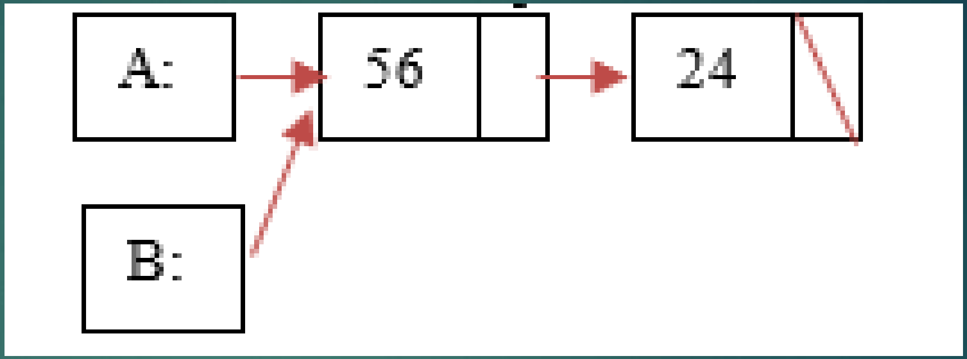
- Often when we pass a structure by value
  - We send an explicit clone for safety
  - But we don’t do this all the time
- So maybe we need to stop using lists!
- Immutable strings are still useful, so
  - Let’s make data structures immutable
- We will still need locking for mutable things
  - We keep it to a minimum to simplify our design

Shareable data structures

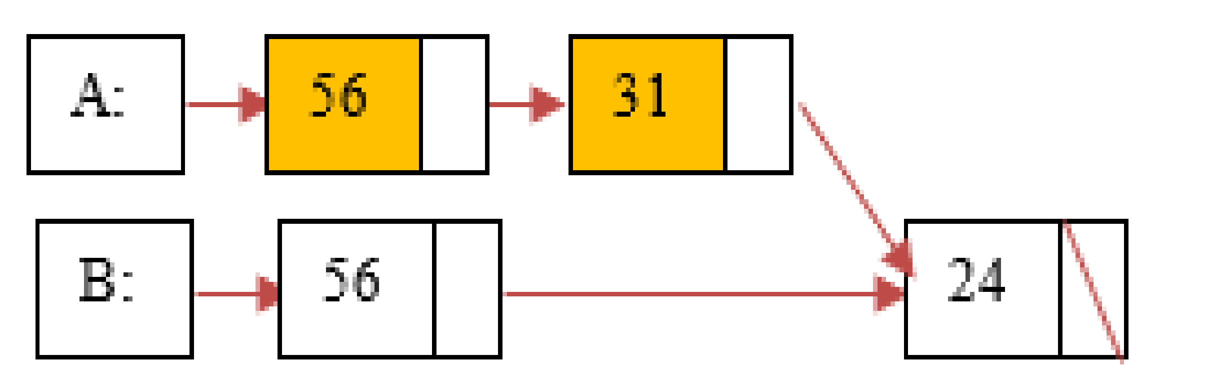
- All levels are immutable
  - Public readonly/final fields
  - All reference fields are shareable too
- No elementwise copying
  - Simply copy reference to the whole thing
- When shareable object is in a mutable
  - We still need to lock the mutable one
  - If we want to make changes to shareable

Shareable linked list

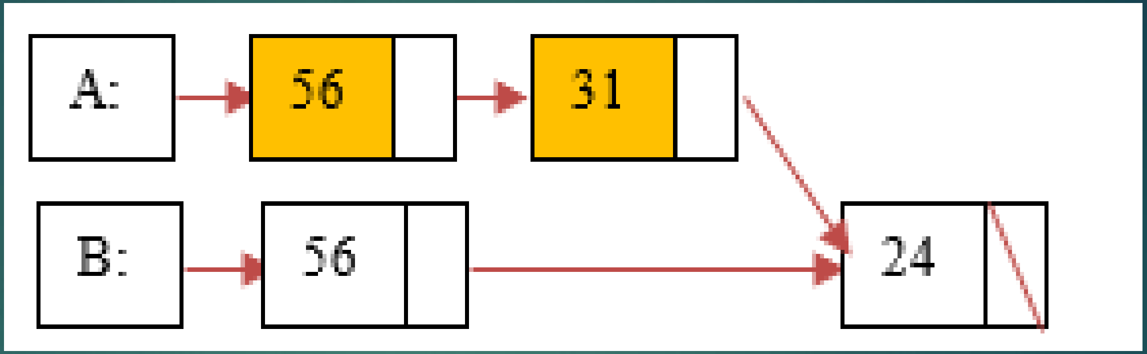
- B=A;



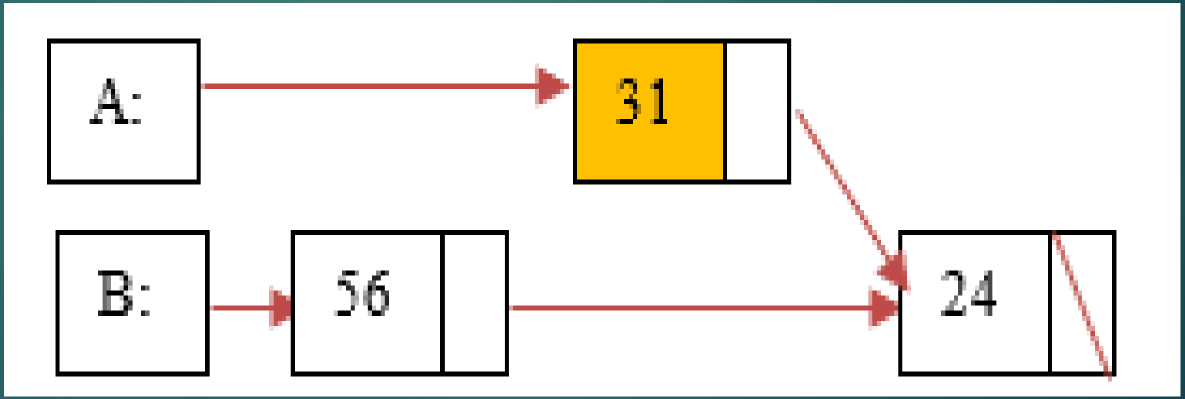
- A=A.InsertAt(31,1)



Shareable linked list



- A=A.RemoveAt(0)



Shareable B-Tree: SDict

- SDict<K,V> leaf nodes contain
  - Key-value pairs c<#<2c (c is e.g. 4)
- Inner nodes contain
  - Key-leaf pairs c<#<2c+1
- The Root node is allowed fewer
  - Even 0 for an empty tree
- The BTree is not kept balanced
  - But worst case is still logarithmic

Operations on SDict<K,V>

- Adding a (k,v) pair

```
T = T.Add(k,v)
```

- Note that we get a new tree!
- In C# we define + so we write

```
T += (k,v)
```

- There is also Remove(k) (or -)
- With T.Lookup(k) we can get v

Advantages and disadv..

- Memory allocator works harder
  - But we don’t need to copy the whole thing
  - We can re-use nodes that remain common
- Linked list uses linear recursion
  - Cost O(N)
  - Same as for arrays
- Much better to use B-Trees
  - Logarithmic behaviour O(logN) instead

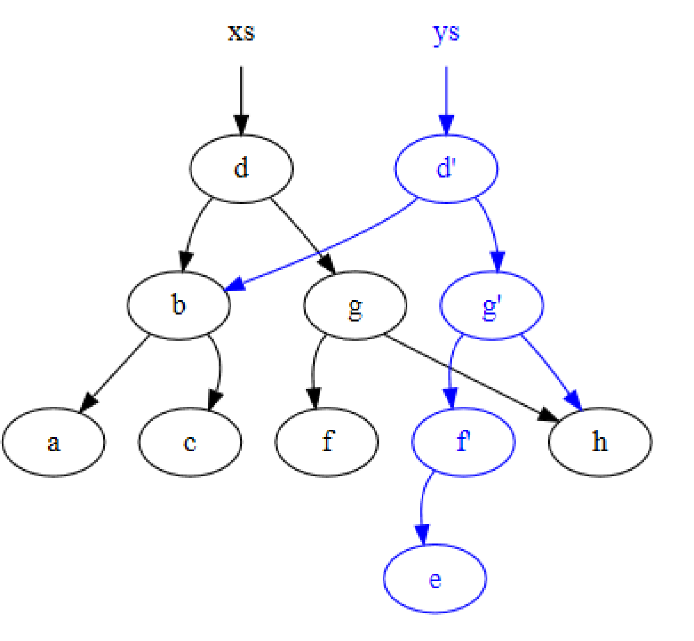
Demo 1: DictDisplay

Memory blocks are shared

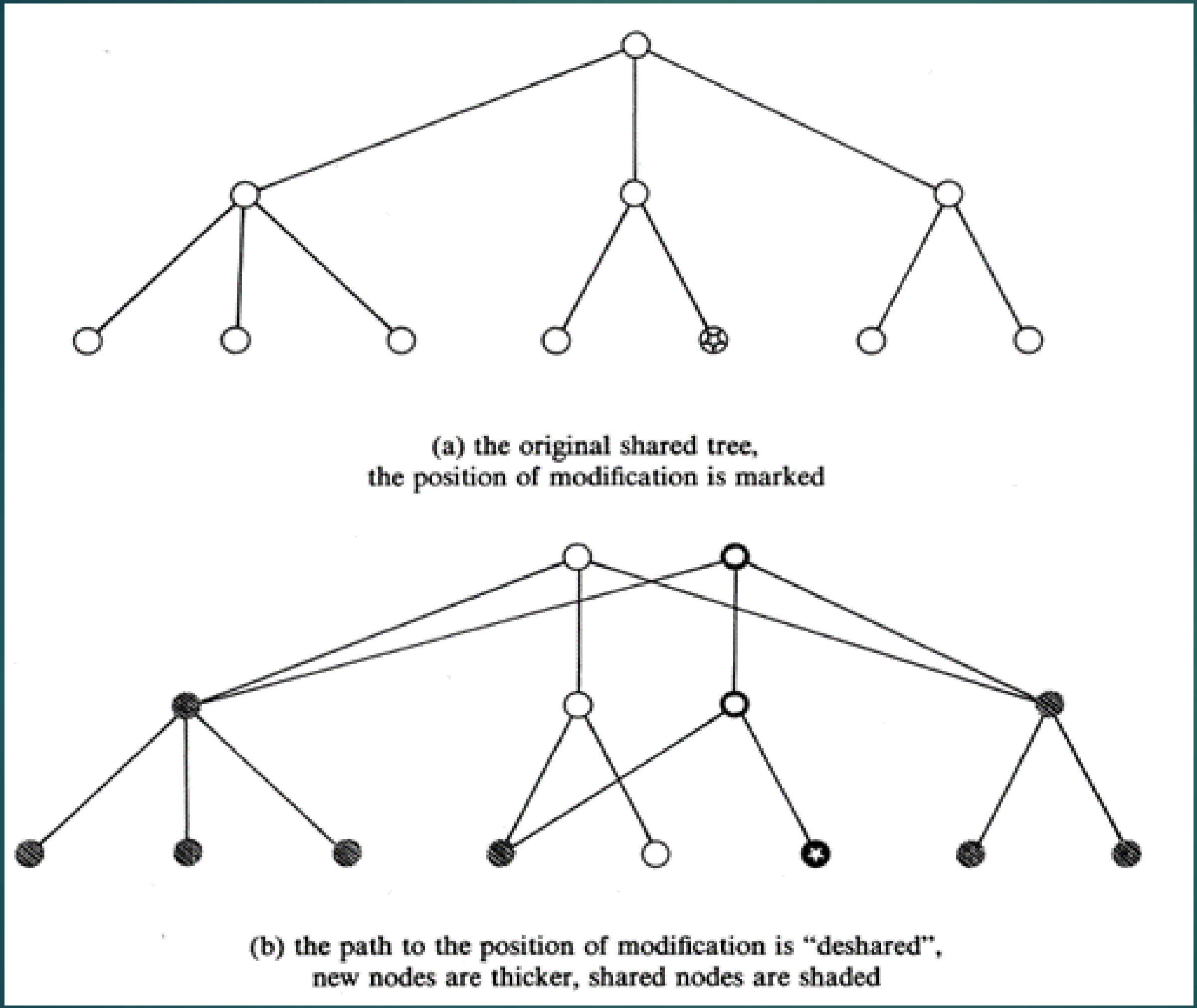
- Thinking about the shareable linked-list
- The versions share nodes after the change
- Similarly for a shareable tree structure
  - For each change the new nodes are a path
  - From the root to the nodes that were changed
- More complex data structures are better
  - Even more efficient since more is shared
- Contrast with mutable structures
  - Where the whole thing needs to be cloned

How new is this?

- “Persistent Data Structures”
  - “Fully Functional” [Okasaki]
  - “Multi-version”, “Concurrent” etc
- These have a similar idea, but a fatal flaw
- A mutable root node
- The Wikipedia article ha
  - But this misses the point



When we add a node



(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

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

When we add a node

- We get at most logN new nodes
  - On a path from root to new leaf
- Remaining nodes all shared
  - Between old and new version
- This proportion only improves
  - As the tree gets larger

Enumerating an SDict

- Bookmarks instead of Iterators

```
for(var b=T.First();
   b!=null;b=b.Next())
```

- Then b.Value is null or a pair (k,v)
- Everything in Strong uses this code

Demo 2: DictDemo

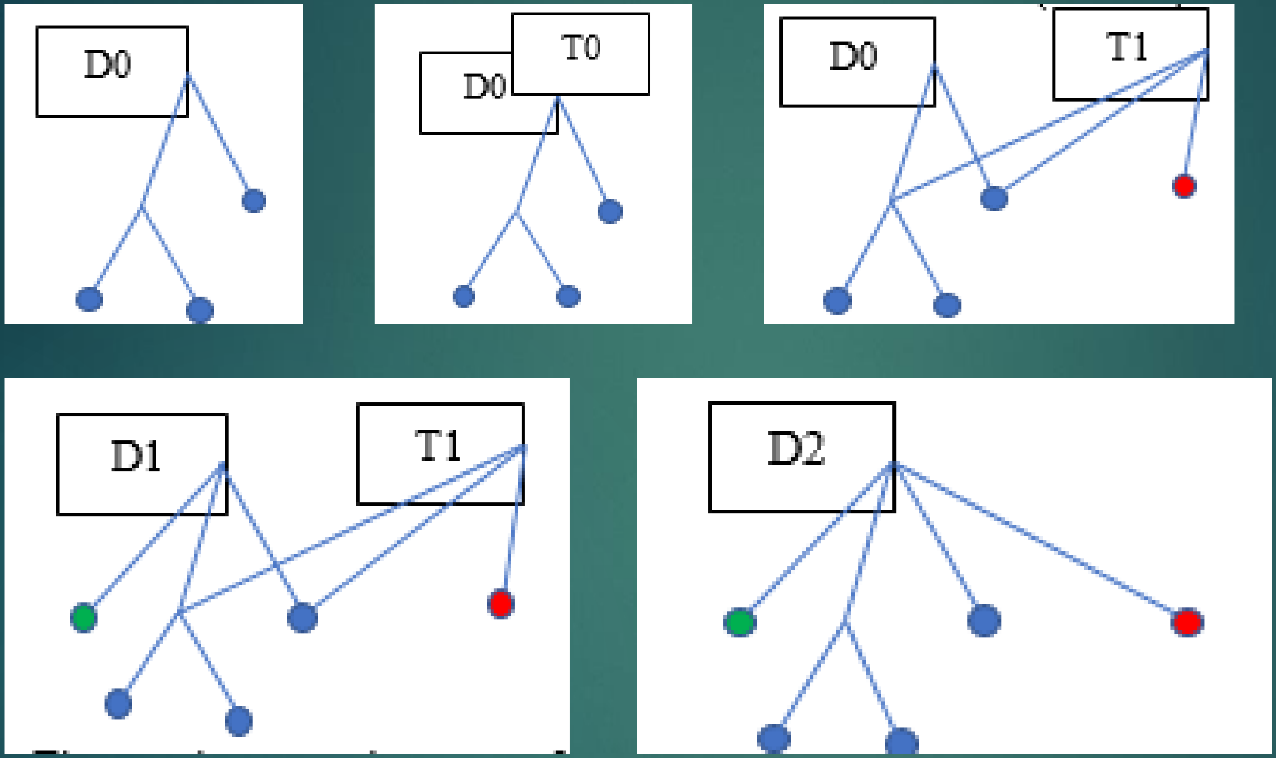
In database technology

- Once we have enough structures
  - We show how a full DBMS can be built
- Taking a snapshot is as easy as B=A above
  - People with copies can consider changes
  - On ROLLBACK they can simply be forgotten
    - For B=A example, simply restore by A=B
  - On COMMIT we need to check for conflicts
    - And the DBMS can accept the changes in master copy
- The list of master copies of databases in use
  - Will be the DBMS' only unsafe data structure!

StrongDBMS has done this

- Built from shareable components
- Designed for data that lasts for ever
  - But with rapid change and concurrency
- On the standard TPC-C test
  - It outperforms all the top databases
  - For a mix of tasks with N clerks

Transaction and B-Tree



Uid is assigned on commit

- When an objects is sent by client
  - It has a client-side uid
  - So each statement from a client
  - Is preceded by a list (uid,name)
- When the server receives this
  - Real uids of Names are looked up
  - Transaction uids for anything new
- Uids assigned on commit

Uid is assigned on commit

- When an objects is sent by client
  - It has a client-side uid
  - So each statement from a client
  - Is preceded by a list (uid,name)
- When the server receives this
  - Real uids of Names are looked up
  - Transaction uids for anything new
- Uids assigned on commit

Commit code for object

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

- Any change constructs a new object

What was the uid before?

- Each transaction has a uid:

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

- But installing this new object
  - Will create a new transaction
  - With a new uid for further objects

Database structure

- Database has a schema and role

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

- Role has names of added objects

A snapshot of SDatabase

- Just copy 4 pointers:

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

Update SDatabase

- Adding a new object gives

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

- Within a transaction there will be an override of this method for a new STransaction

The constructor used here

```
protected SDatabase(SDatabase db,
SDict<long, SDbObject> obs, SRole
ro, long c)
{
    name = db.name;
    objects = obs;
    role = ro;
    curpos = c;
}
```

Install a database object

- We can now use the above
  - As follows

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

Install a new Column

- Tables have their own structure

```
public SDatabase Install(SColumn c, string n, long p)
{
 var obs = objects;
 if (c.uid >= STransaction._uid)
   obs += (c.uid, c);
  var tb = ((STable)obs[c.table]);
  if (role.subs.Contains(c.table) &&
      role.subs[c.table].defs.Contains(n))
    throw new Exception("Table already has column " +
n);
  return New(obs + (c.table,tb+(-1,c,n))+(c.uid,c),
      role+(c.table,-1,c.uid,n)+(c.uid,n), p);
}
```

Transaction

- This is a subclass of SDatabase

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

- By inheritance, has its own objects

Committing a transaction

- We traverse its objects
  - And commit them as above
  - We manage transaction references
    - Using a list of commits in class Writer
- But first we check for conflicts
  - With recent database commits
    - Uids >= inherited curpos

Query processing, RowSet

- RowSet constructed recursively
  - For results of Select
  - And items for Insert
  - Performing grouping, ordering etc
- Traversed with RowBookmark
  - Which contains current context
- Context contains current values
  - Indexed by uid (field, row, group,..)

Demo 3: How many clerks?

Questions?