

# Overview of the Mulgara XA2 Disk Structure (Discussion Paper<sup>1</sup>)

Andrae Muys  
andrae@netymon.com

Netymon Pty Ltd

and

School ITEE  
The University of Queensland

## Persistent Datastructures

One of the most fundamental features of Mulgara is its use of snapshot isolation to provide efficient read-only transactions. With XA2, and a move to support concurrent writers, it remains important to retain this feature. Snapshot isolation is a form of multiversion concurrency which is closely related to the idea of persistent, or “purely functional” datastructures used in primary memory. In these structures all update operations return a new instance of the data-structure, without affecting the arguments to the operation. In programming theory this is referred to as “referential transparency”, in database theory “isolation”. Consequently it is not unreasonable to consider reconceive the problem of designing an on-disk format to support snapshot isolation, as designing persistent datastructures for external memory.

The seminal work on persistent datastructures is Chris Okasaki’s book, “Purely Functional Data Structures”[Okasaki98]. Okasaki introduces several techniques for designing persistent datastructures, the ones relevant to XA2 are structure-sharing, lazy-evaluation, lazy-rebuilding, numerical-representations, datastructural-bootstrapping.

- *Structure Sharing* is involved where the updated instance contains references to/into the original. This includes simple reuse, such as implementing a linked-list append by returning a cons-cell that points to the head of the original list; and more complex reuse such as the copy-on-write techniques used by the current XA1 store.
- *Lazy Evaluation* involves returning an instance paired with a suspended operation that can be executed to perform the update on a read. When applied to external memory, where we are without the ability to thunk an operation, this technique amounts to journaling.
- With *Lazy Rebuilding* rather than suspending the update operation we defer invariant restoration until some triggering event. The most obvious example is to defer rebalancing a tree after a delete until a sufficient number of elements have been removed that the imbalance starts affecting lookup complexity. In databases this routinely takes the form of background vacuum tasks.
- *Numerical Representations* exploit a resemblance between number representation systems and datastructures. One common datastructure that can be derived, analysed, and

---

<sup>1</sup> Presented at the Topaz / Public Library of Science, Mulgara Workshop, San Francisco, March 2008

modified using this technique is the Binomial Heap, where each tree of rank- $n$  resembles a binary digit  $2^n$ .

The basic building block in XA2 for both the Statement-Store and the String-Pool is a redundant positional numerical representation of a random access list using lazy rebuilding to defer carries. The following discussion will assume a binary system, although other bases are possible and may be desirable.

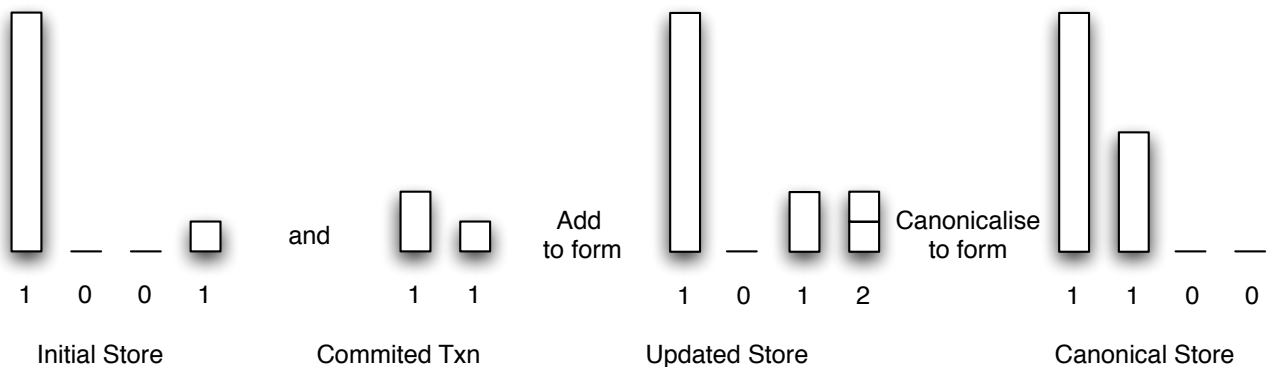
### Numerical Representation of the XA2 Store

A redundant binary number is one in which each digit can take on values other than 0 or 1. For instance the number  $302_2$  would expand to  $3 \times 2^2 + 0 \times 2^1 + 2 \times 2^0$ , ie.  $14_{10}$  or  $1110_2$ . This representation is redundant as clearly there are multiple ways of representing each number. Unsurprisingly we define the canonical form of a number to be the same as its non-redundant representation. To us the relevant features of this form are that we can add two redundant numbers in  $O(\min(D_1, D_2))$  time where  $D_1$  and  $D_2$  are the number of digits in each term; and that canonicalisation can be performed in  $O(\log(D) \times \log(d))$  time where  $d$  is the maximum (worst-case) or average (expected-case) digit in the number, and is idempotent in the number represented.

In mapping this onto a datastructure we represent a number as a list of digits  $[d_0, d_1, \dots, d_n]$  with weights  $w=2^i$ . We then proceed to represent each digit as a list of elements each with  $w_i$  nodes  $d_0=[^0e_0, ^0e_1, \dots, ^0e_n]$ ,  $d_1=[^1e_0, ^1e_1, \dots, ^1e_n]$ , etc. As the number of digits is worst-case  $\log(N)$ , provided none of the digits are excessively large the cost of merging two such datastructures is only  $O(\log N) \times O(\text{merge-of-two-elements})$ .

The reason this is useful to XA2 is that this means that we can keep the updates of any given transaction completely separate from the committed store for a cost of  $\log(N_{\text{update}})$  provided we keep the store in canonical form. This greatly reduces the complexity of maintaining a multi-phasic tree structure and eliminates almost all the data-hazards that would otherwise require locking the tree. An related benefit is that if the updates of each uncommitted transaction are kept separate, rollback becomes a simple matter of deleting the now invalid structures. The corollary of this is that as only committed data is ever added, the need to maintain free-lists is greatly reduced — in fact if we select an appropriate design to represent the elements, they can be eliminated completely.

In practice what this looks like is



It should also be noted that the 'add' step simply updates the collection of elements that form the store and can therefore be performed in a single constant time operation, as the elements themselves do not need to be copied — an instance of structure-sharing. The

obvious disadvantage is that as the elements are independent, a lookup operation must treat each element independently, reducing read performance by a factor equal to the number of elements. As noted the canonicalisation operation is idempotent, so it can be deferred — an instance of lazy rebuilding. Moreover we can exploit lazy evaluation by keeping the lowest digit of an uncommitted transaction in memory, allowing us handle multiple small updates without touching disk until the transaction commits.

This datastructure places an unusual demand on the structure chosen to represent each element as we can ignore insertion cost, but must instead focus on reducing the cost of a merge. In one respect this is also a disadvantage as most datastructure research has focused on balancing read performance with insert/delete performance, and has largely ignored merges. On the other hand it is expected that the elimination of tree/page locking, the drastic simplification of the larger store, and the ability to change the implementation of an element without affecting the global datastructure will compensate for this.

### Some Definitions

A sequence of read and update operations is called a *History*. When these operations are contained within separate concurrent transactions the history is defined in terms of a *partial order*, with the intuitive ordering of each operation within a transaction, and between the operations of transactions that commenced after an earlier transaction successfully committed. Any history that respects the above ordering and which contains the entire history of any contained transaction is called a *complete history*. The result of removing from a history, any operations belonging to transactions that did not or are not committed is called the *committed projection* of the history[Bernstein87] — naturally all committed histories are also complete. The result of removing from a history all read operations is called the *update projection* of the history. There are additional issues of consistency and serializability, however these are addressed in the companion concurrency control paper. For the purposes of this paper we can assume that all committing transactions are consistent and serializable, allowing us to focus on what happens to them before and after concurrency control validation.

### Representing Transactions

Every Mulgara store starts empty, and its state is fully defined by the operations that have been performed on it. More formally, what we have been referring to as the committed store is therefore a topological sort of the committed projection of the mulgara instance's history. Given that we are assuming all transactions are serialized on commit, in practice we maintain a temporal sort of the update projections of the committed transactions. The transactions themselves consist of a history potentially containing interleaved reads and updates, however as the store is an update projection of the history, we are able to exploit the following identities (where  $S_1$  and  $S_2$  are sets of statements).

1.  $H = \{ i(S_1), i(S_2) \} = \{ i(S_1 \cup S_2) \}$
2.  $H = \{ d(S_1), d(S_2) \} = \{ d(S_1 \cup S_2) \}$
3.  $H = \{ d(S_1), i(S_2) \} = \{ i(S_2), d(S_1 - S_2) \}$
4.  $H = \{ i(S_1), d(S_2) \} = \{ d(S_2), i(S_1 - S_2) \}$

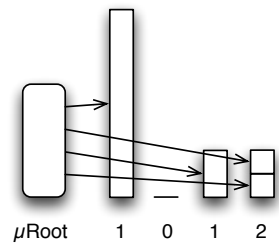
These allow us to reduce every updating transaction to either a single insert or delete operation of the serial execution of a single insert and a single delete operation. However we

must be careful to ensure that incomplete transactions maintain sufficient intermediate statement-sets to provide read-isolation for any interleaved queries.

Transactions are therefore best maintained as disjoint insert/delete sets. If we are careful to keep them disjoint — by performing a remove operation on one whenever we perform an add operation on the other — we can amortize the cost of the difference operations over the individual operations avoiding an excessive performance hit on the commit operation. We should also seriously consider using an in-memory multiversion datastructure to buffer update operations and reduce the impact of multiple small updates. If we assume we require 512 bytes per statement<sup>2</sup>, we can buffer 32K Statements in ~16MB.

### Representing The Store

As discussed above the store is represented by a set of sets of elements. As each element is the result of a partially canonicalisation of a separate transaction merge, we should expect each element will need to be its own file. If we assume that each file self-identifies its position, which given this is equivalent to tracking its size is reasonable, we can store the flattened set of all elements that constitute the store. Given we need to identify this set, missing from the earlier diagram is the coordinating metaroot-file which is needed to keep track of the files included in a given phase. It should be noted here



that if we are going to support JTA recovery, the  $\mu$ Root will also have to record the files belonging to any prepared transactions still pending commit. On disk the  $\mu$ Root need only maintain the files that form the current committed projection, and any prepared transactions and consequently need only be a single-version structure — although with the redundancy required to provide durability over a mid-write system failure. Naturally to maintain snapshot isolation the in-memory version of the  $\mu$ Root must be multi-version, but this would not need to be persisted to disk unless we wished to support maintaining read-only (or even read-write) transactions across a system restart.

### Representing Elements

We have previously been discussing elements in terms of single files, but this is not necessary provided the element is self describing. For instance a statement-store element, once it reaches a certain size threshold, is almost certainly going to be split into 6 or 7 separate files, one for each index, and a string-pool element will most likely consist of at least 2 files, as it needs to maintain a bidirectional map. The only requirement is that they support an efficient  $2^n$ -way merge operation,  $\text{merge} :: [\text{Element}] \rightarrow \text{Element}$ . This then, along with a need to provide efficient query operations, becomes the primary requirement of an element. One thing that might be interesting to consider, is should we require a serialised class file adequate to interpreting an element be included, at least by reference, within an element? This may have implications regarding backup, archival, and migration of large scale RDF data to new structures/platforms.

### Bibliography

Bernstein, P. et al., "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987

Okasaki, C., "Purely Functional Data Structures", Cambridge University Press, 1998.

---

<sup>2</sup> assuming the average object is 256 bytes, the average amortized distinct predicate and subject are 32 bytes each and including 192 bytes to store 4 8-byte longs in 6 indicies.