

The Mulgara XA2 Statement Store (Discussion Paper¹)

Andrae Muys
andrae@netymon.com

Netymon Pty Ltd

and

School ITEE
The University of Queensland

The Existing Store

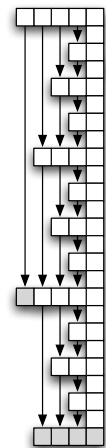
The Mulgara XA1 Statement Store has held up well, scaling comfortably to hundreds of millions of statements before bogging down. The primary limits on its scalability are its resolution, and its spatial locality. As an AVL tree it resolves lookups down to a single block via a binary tree walk. This means that well before our target of $1e11$ triples, significant portions of the index cannot be held in memory and must be loaded from disk. As a binary tree it also has many more layers than something like a B+tree which increases the likely number of seeks required to traverse the tree when it isn't in memory.

The store has two lookup operations critical to mulgara's query performance. Not surprisingly the first is an efficient "indexed find". Mulgara gains most of its query performance from the extensive use of merge-joins, so the other critical operation is slice-iteration. In practical terms this means that the statement-store is a type of Random-Access-List; and both the iterative list-behaviour and the random-access is critical.

Evolution of Design

David Makepeace proposed a skiplist structure for XA2. This structure is exceptionally fast to write, and also extremely fast to merge. When acting as a list, it exhibits good spatial locality, however when indexing the interleaving interferes with the caching of the higher order skip-nodes, and consequently increases the number of seeks required. We can reduce this by collecting the skip-nodes into arrays. These allow the skiplist to be traversed by $\log(N)$ linear scans through a sequential array, providing almost ideal IO complexity. The resulting structure is more commonly known as Indexed Sequential Access Method (ISAM).

David Makepeace also proposed splitting each quad-index into four subindices, one per node in the quad. This can potentially increase the number of seeks required, but compensates for this by first, substantially reducing the amount of duplication in each index; and second, allowing each sequential array to be stored as a list of deltas, which drastically reduces the amount of data that needs to be stored on disk. Just as important is the significant increase in the number of index blocks that can be cached in memory, which will go a long way to alleviating the memory scalability constraints of XA1.

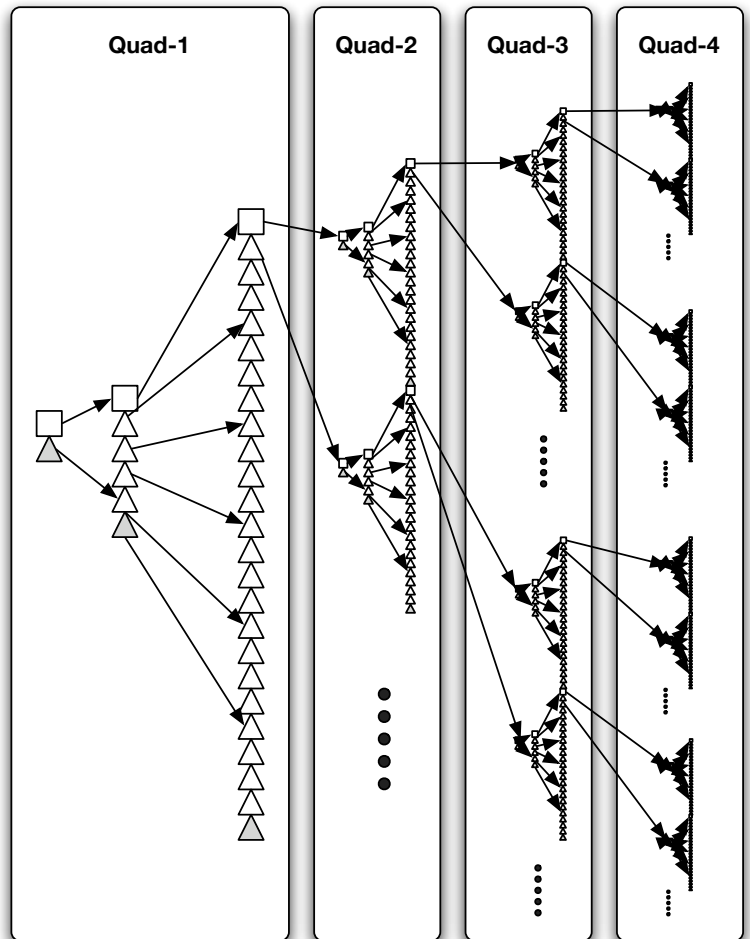


¹ Presented at the Topaz / Public Library of Science, Mulgara Workshop, San Francisco, March 2008

The final optimisation is to increase the size of each block indexed by the tree. Due to disk-caches, OS read-ahead, and various memory cache sizes, it is almost pointless to only request a single disk page at a time. The precise block size may need to be a tunable parameter, however for initial development 32KB should provide a good starting point.

Data Compression

Each entry in a sorted-array contains three fields: the nodes delta from the previous entry; a count of the number of nodes represented by the entry; and a file and/or offset for the children of the node. As the array is sorted, the deltas are likely to be small — a property that will only improve as the number of statements increases. It is also likely that the number of statements under a given tree-node will normally require only 3-4 bytes to represent; indeed, with a maximum of 2^{37} statements in the store, we can never require more than 5 bytes to represent a count. Finally, even uncompressed the offset can only rarely require more than 6 bytes. With 8-bytes



The final proposed structure, including separating each node of the quad into its own tree; storing each subsequent node as a delta from the previous one; and using a larger page size, shown here 4 nodes per page, the proposed structure uses 32KB.

The performance can be further improved by compressing the data being written to disk. Two techniques are being considered, the first uses a header byte to encode the width of the various fields of each quad-node record; the second uses the flag-bit to provide variable width words.

Detailed Disk Layout

Uncompressed each index contains a sorted array of 3 fields. The delta is the difference between the 64-bit node-id represented by this entry, and the previous node-id in the array.

Node-Id Delta	Stmt Count	Offset
---------------	------------	--------

As each block is referenced separately, we assume the previous node-id is 0 when calculating the first node-id in every block. To avoid the

need to traverse an index slice to obtain the number of statements spanned, we also include a statement count field within the index. Naturally this can be excluded in the final index of the 4th quad as in this case the count is always 1. Finally we require an offset to the next level of index; or to the head of the appropriate tree for the next quad.

IO Capacity Calculations

In practice we can expect most deltas fall into one of three classes: Common URI's, including popular and standardised predicates such as RDF, RDFS, Dublin Core, FOAF, etc — as many of these are likely to be included in data loaded early in the lifetime of a store-instance, these are likely to be clustered amongst the low node-ids; Application specific URI's and Literals, which again are likely to be used early and clustered, but possibly clus-

tered apart from the more common standardised predicates; and Document specific URI's, Literals, and all Blank-Nodes, which are unlikely to have been loaded previously (guaranteed in the case of Blank-NNodes) and therefore will be allocated incrementally. This suggests that the delta between any two adjacent node-ids is likely to be $< 64,000$, and will therefore require only 2 bytes.

With a total statement count of 2^{37} a branching factor of over 2^5 will ensure most statement counts at each node are at most 2^{32} . Consequently it is expected that the count field will require between 2-4 bytes. Likewise with a block size of 32KB, and given that offsets reference blocks not individual statements, the offset field should also fit comfortably within 4 bytes.

This means that the expected size of a field, allowing 1-2 bytes for compression overhead, is 9-12 bytes. With a block size of 32KB this means we can expect to store 3K nodes per block. Similarly with a single index level, we can store 9 million nodes in 90MB; with a second index level, 27 billion nodes in 270GB; and 100 billion nodes in 1TB using three index levels. Given 4 nodes per index, and 6 indices per store this suggests that ~ 24 TB will be required for the statement-store.

With three index levels plus the node array per node, and four nodes per statement, an index can guarantee access in 16 seeks. This can be reduced by 2 because the first two levels of index are almost guaranteed to be held in memory, reducing this to 14. Once the appropriate offset has been found for each node, all iteration is via sequential walk through the array, resulting in only inter-track seeks required during iteration.

Modifications for Pair Based Index

Elsewhere Paul Gearon has proposed a pair-based indexing scheme for RDF based on reified statements that shows promise in significantly improving write performance without seriously impacting reads. Using this approach, instead of 4 indices we use a single index with each leaf being 4 ordered lists of statement-ids. Consequently if desired it is possible to use the ISAM tree to implement the index, and to add a payload pointer that points to the lists.

Bibliography

Arge, L., "Efficient External-Memory Data Structures and Applications", PhD Dissertation, University of Aarhus, 1996

Bender, M. et al., "Cache-Oblivious B-Trees", Proceedings of the 41st Annual Symposium on Foundations of Computer Science, 339, 2000

LaMarca, A. et al., "The Influence of Caches on the Performance of Heaps", Journal of Experimental Algorithmics, Vol 1, No 4, 1996

Pugh, W., "Skip Lists: A Probabilistic Alternative to Balanced Trees", Workshop on Algorithms and Data Structures, 437-449, 1989

Ruemmler, C. and Wilkes, J., "An Introduction to disk drive modeling", IEEE Computer, Vol 27, No 3, 17-28, 1994

Vitter, J., "External Memory Algorithms and Data Structures: Dealing with Massive Data", ACM Computing Surveys, Vol 33, No 2, 209-271, 2001