

The Mulgara XA2 String Pool (Discussion Paper¹)

Andrae Muys
andrae@netymon.com

Netymon Pty Ltd

and

School ITEE
The University of Queensland

Requirements

The Mulgara XA1 String Pool has proved to be the primary bottleneck in attempts to scale the original store. It is therefore imperative that the XA2 design is careful to ensure that its string-pool design will scale to the same level as the rest of the system. The minimum goal of XA2 is to support 10 billion (2^{34}) statements – with a design target of 100 billion (2^{37}). With a potential 4 elements per statement this means that the XA2 string-pool must be capable of indexing 2^{39} “strings”. If we assume the average Literal is 256 bytes long, and the average URI Reference is a third of that. That makes the average statement 512 bytes long, resulting in an expected worst-case gross storage requirement for the string-pool (absent any indexing or compression) of 2^{48} bytes or 256TB.

We don't store duplicates, which allows us to ignore the cost of storing the graph-uri, and greatly reduces the cost of storing predicate; and we don't store blank-nodes, substantially reducing the cost of storing subjects. So if, as we will discuss later, we allow for the fact that many literals can potentially be stored in-place, the expected case is not so bad. So this design is predicated on the understanding that while we need to be capable of handling 248 bytes, we should expect to average only one “string” per statement. I.e. a maximum of 2^{37} strings \times 2^8 bytes or 32TB.

When we refer to a “String Pool” storing “strings”, we have to recognise that this is a lie. What is actually stored are Typed Octet Sequences, or TOS's². It is important to keep this in mind as not all the data stored in the string-pool is in the form of unicode codepoints, and we can and should exploit this. More importantly there are correctness considerations. RDF specifies that the data stored in the string-pool is typed with reference to the XML Schema Datatypes Specification, which means that the data has several properties that must be respected including:

- Multiple equivalent lexical forms, with type-specific canonicalisation functions.
- A type-specific lexical-form \leftrightarrow value function
- Type-specific ordering functions supporting potentially both partial and total orderings
- Type-specific operations such as arithmetic operations for numbers, and concatenation or regular-expression matching for strings.

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

² Hence we should probably consider renaming the String-Pool — current options include “TOS-Store” and “Octet-Store”; “Typed-Octet-Sequence-Store” is probably too much of a mouthful.

The end result is that in addition to the scalability requirements, the String-Pool should also support independent in-memory and on-disk datastructures for independent datatypes.

The String-Pool Interface

The XA1 String-Pool had the following interface:

```
public interface StringPool {  
  
    public SPObjectFactory getSPObjectFactory();  
  
    public void put(long gNode, SPObject spObject) throws StringPoolException;  
  
    public boolean remove(long gNode) throws StringPoolException;  
  
    public long findGNode(SPObject spObject) throws StringPoolException;  
  
    public long findGNode(SPObject spObject, NodePool nodePool) throws StringPoolException;  
  
    public SPObject findSPObject(long gNode) throws StringPoolException;  
  
    public Tuples findGNodes(SPObject lowValue, boolean inclLowValue,  
                            SPObject highValue, boolean inclHighValue) throws StringPoolException;  
  
    public Tuples findGNodes(SPObject.TypeCategory typeCategory, URI typeURI) throws StringPoolException;  
}
```

The semantics provided are:

- `put(Long, SPObject)` corresponds to an insert.
- `remove(Long)` corresponds to a delete.
- `findGNode(SPObject)` and `findSPObject(long)` correspond to lookup.
- `findGNode(SPObject, NodePool)` corresponds to an insert-on-fail.
- `findGNodes(SPObject, boolean, SPObject, boolean)` corresponds to an open/closed bounded/unbounded range query.
- `findGNodes(SPObject, TypeCategory, URI)` is equivalent to: $\text{String-Pool} \cap [\perp_{\text{type}} \dots \top_{\text{type}}]$.

As previously discussed, the maximum possible number of nodes in 100 billion statements is 2^{39} . Clearly we only strictly require 39-bits to represent each node. Given we are using 64-bit nodes we could insert/delete/reinsert all 100 billion statements 2^{25} times before we ran out of node-ids. Which at a rate of one delete/reinsert per second, ~ 1080 years. Recognising this the question has been asked, do we need to support a remove operation? Is there any reason to attempt to reclaim nodeids? The answer appears to be that while we may need to reclaim the disk-space consumed by any large octet sequences that are no longer referenced in the statement-store, this is probably better treated as a background garbage-collection problem; the node-ids themselves should probably not be reused.

In XA1 the SPObject's serve two purposes. The primary is as a translation of the RDF Node's properties to the internal representation of such things as type category, subtype,

and the datatype's comparator for managing its ordering. The secondary is as a tainting mechanism to ensure that all Resources have been properly canonicalised and converted from lexical-form to value prior to insertion/extraction from the pool. Despite the limitations of the SPObjct when trying to represent the full-range of XSD types, both these functions are invaluable and should be provided by any replacement.

The put operation is an unfortunate encapsulation violation and carries the risk that the one-to-one mapping between SPObjct and nodeid may be violated (although this method currently throws an exception in this case), it also interferes with our ability to store data values directly in the nodeid, a desirable optimisation that has been proposed for XA2.

A new requirement in XA2 is the need to be able to merge two string-pools. As discussed in the overview paper, this is a 2n-way merge with type `:: [StringPool] -> StringPool`. As an operation on lists of string-pools, this operation does not belong to a string-pool, but is rather a static function on string-pools.

So modulo handling of datatypes and their respective operations (conversion, ordering, etc), we can reduce the StringPool interface to:

```
public interface StringPool {  
    ... xsd datatype support ...  
  
    public long findGNode(SPObjct spObjct) throws StringPoolException;  
  
    public long findGNode(SPObjct spObjct, NodePool nodePool) throws StringPoolException;  
  
    public SPObjct findSPObjct(long gNode) throws StringPoolException;  
  
    public static StringPool merge(List<StringPool> stringpools) throws StringPoolException;  
}
```

Handling Datatypes

The XML Schema Datatypes standard is a rich tree of built-in datatypes, and provides a framework for user-defined types to be added. It carefully distinguishes between the representation of a type and its values, as well as providing for various properties (facets) of the type to be declared. These include such things as if the type is a partial/total order; if the type is finite or countably infinite; if they type is bounded, and many others. XA1 handles this poorly, and it is still an open question how XA2 will approach it.

One serious constraint imposed by XA1 that would be useful to avoid is the requirement that all datatypes define a total-order, even for those that provide only a partial-order, or have no order relation at all.

With 263 ids covering an absolute maximum of 239 values, it is clear that a large percentage of the id-space will never be used. Exploiting this, if ~7 bits [62-56] are defined to be a type-id that would allow us to provide different string-pool implementations for different types. Doing this would still leave us 56 bits for a node-id³, which would still allow us to re-insert the worst case dataset 128,000 times.

³ The missing bit is the sign-bit, used to differentiate between temporary and permanent node-ids.

One immediate benefit of splitting the datatypes like this is that it will greatly simplify the process of supporting direct-nodeids, where the value is stored directly in the nodeid. For instance: if stored as a `time_t` 56 bits is sufficient to store timestamps from 1401CE to 2539CE with a resolution of μ Sec directly in the nodeid; or if stored as a pair of `time_t`'s any time-interval that falls within 1843CE and 2097CE with a resolution of minutes. As we can safely assume that even if the datatypes assign bits to track their own subtypes, the vast majority of dates, timestamps, and even numbers can be eliminated from the string-pool by simply inlining them. An additional benefit is the ability to choose an encoding that would permit us to perform global-space comparison operations directly on nodeids without the need to globalize them first.

Conversely splitting the datatypes will also allow us to cap the size of strings handled in the standard string-pool by providing a mechanism to cleanly offload oversized strings to a separate blob-store. It should be noted however that datasets such as wikipedia include many nodes that are 50-100KB in size and should therefore be handled gracefully by the default string-pool, at least initially.

XA2 should therefore provide the ability to declare a set of `DatatypeHandlers`, much like Mulgara's current ability to declare a set of `ContentHandlers`, or `ResolverFactories`. The metaroot would have to ensure that it recorded the `typeid->TypeURI` mapping used by a given store; and the issue of what to do if a `DatatypeHandler` is not registered for `typeids` currently stored in a store on a restart. This may be a compelling reason to require a `.class` file be persisted as part of a datatype's store, if only as a read-only backup to permit the interpretation of existing data in the statement-store.

The only datatype operations that need to be supported immediately are interval and comparisons which are required by the `XSDResolver`. It would be nice to be able to support arbitrary datatype-specific operations, and this is perfectly feasible although it would require a complete rewrite of the `XSDResolver` (which is desperately needed anyway). While ensuring that no decision is made that precludes such a feature, this should probably be deferred until after XA2 is completed. The basic design however would entail the `XSDResolver` being configured (presumably via a `?def` graph) with a set of operation predicates, custom-constraints, transformation rules, `TypeURIs`, and operation classnames, sufficient to permit it to configure itself at runtime to identify, collect, and delegate datatype-specific operations to the correct `DatatypeHandler` in the `String-Pool`.

The resulting complete interface for `StringPool` ensues⁴:

```
public interface StringPool {
    // Operations
    public long findGNode(SPObject spObject);

    public long findGNode(SPObject spObject, NodePool nodePool);

    public SPObject findSPObject(long gNode);

    // Merge
    public static StringPool merge(List<StringPool> stringpools);

    // Datatype support
    public void addDatatypeHandler(URI typeURI, DatatypeHandler handler);
```

⁴ This interface is flattened and the signatures simplified to aid discussion of the functions involved.

```
public Tuples executeOperation(URI typeURI, DatatypeOperation operation);  
}
```

The untyped-literal / xsd:String DatatypeHandler

The most critical of the datatype-handlers is the one dedicated to untyped-literals and xsd:Strings. So the requirements are:

- Fast Lookup :: String -> Long
- Fast Lookup :: Long -> String
- Fast Range :: (String*String) -> [Long]
- Fast Merge :: [XSDStringStore] -> XSDStringStore

It is also necessary that if the datastructure chosen does not support a fast insert that we also specify an intermediate datastructure for use during uncommitted transactions that supports:

- Fast Insert :: IntermediateStore -> String -> (Long*IntermediateStore)
- Fast Conversion :: IntermediateStore -> XSDStringStore

The design of the XSDStringStore is by far the least developed component of the XA2 design. The requirement of an XA2 Store Element to support fast merging means that the traditional structures for storing string->value maps (tries and b-trees) will require some modification.

B-Trees exhibit excellent spacial locality, and can be easily optimised for cache sizes, this makes them invaluable in maintaining performance in external memory datastructures. The disadvantage of B-Trees lies in their assumption of a fixed key-length. If a variable length key exceeds the size allocated per index entry, key comparisons can require a page-fault, completely defeating the cache efficiencies of the b-tree.

Tries tend not to be as cache efficient as B-Trees, however they do handle long keys gracefully. Their disadvantage is that, even with various compression techniques, they can be very space inefficient. The crucial advantage offered by Tries is that they don't require key comparisons - iterating over a key automatically walks the tree.

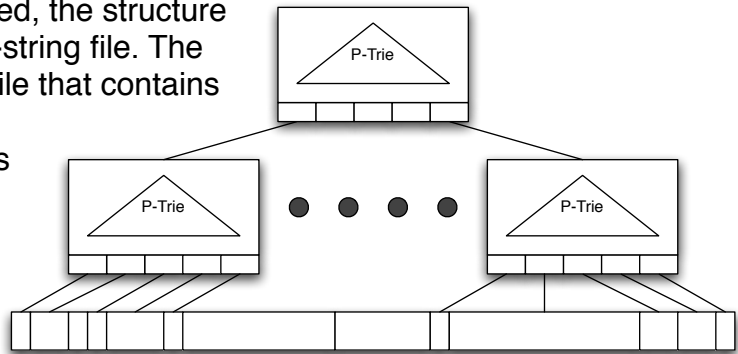
For an in-memory structure the current state of the art appears to be either a HaT-Trie [Askitis 07] or an Adaptive Burst-Trie [Nilsson98], would be an excellent choice — and may well be worth considering as a pre-commit string-pool. However neither decomposes well when adapted to a page-based layout for external memory. On the other hand, a promising candidate is the String B-Tree outlined by Ferragina [Ferragina99]. This is a hybrid B-Tree/Patricia-Trie, that builds a standard b-tree, but uses tries within each node to organise the index. An alternative worth considering would also be the opposite, an adaptive, level-compressed trie that uses b-trees within sparse nodes to organise it's lookup table.

In both cases as we do not need to support insertion it is worth considering if an ISAM-tree would work better than a B-Tree. ISAM-trees fell out of favour due to the amount of re-structuring required to support insert, they do however provide even better cache efficiency than B-Trees.

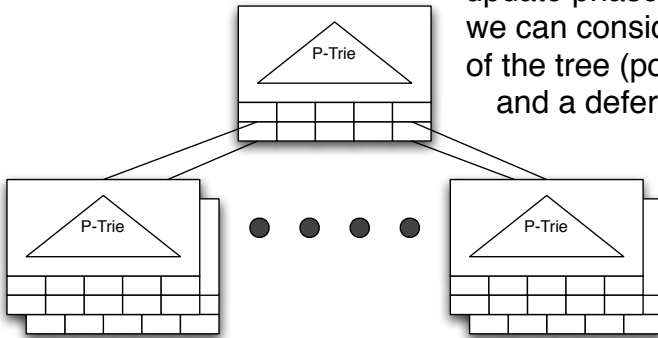
Precisely which approach is preferred is an open question, however both hybrid Tree/Trie and Trie/Tree structures should provide the desired scalability. The ultimate deciding issue will be the efficiency of a merge operation.

The String-Pool Merge Operation

Irrespective of the hybrid structure used, the structure forms an index on an underlying bulk-string file. The leaves of the tree are pointers into a file that contains all the octet-sequences in the string-pool. A merge of these underlying files is clearly $O(n)$, and provided we use retargetable references in the leaves the merge can be pushed into the background. This means that the focus when optimising merge is on the cost of merging indices.



Following the general XA2 paradigm of decomposing the merge operation into a mutating update phase, and an idempotent canonicalisation phase, we can consider a string-pool merge as merging the head of the tree (possibly down to some fixed/tunable depth) and a deferred merge of the subtrees below the head.



We can represent this as each node in the Tree/Trie containing a sequence of child references per entry. Appropriate selection of a reference representation would allow us to share the structure of uncanonicalised sub-trees without copying them.

Summary of Proposed Structure

In summary, the ideal structure for the string-pool is most likely a level compressed adaptive binary multi-trie, that indexes a packed data-file; and uses arrays for very dense nodes, and either b- or isam-trees for very sparse nodes. I recommend against using path-compression to reduce the number of io-operations required to perform a prefix or merge operation. This structure does not need to support dates, timestamps, or numbers as these should be delegated to their own stores, and in most cases stored in-line within the nodeid. Likewise any data over a certain size threshold should be treated as a BLOB and likewise stored external to the pool. Finally, it is worth considering a String-B-Tree — a B-tree that uses Tries internally to manage its nodes — as a possible alternative structure.

Bibliography

Ferragina, P and Grossi, R., "The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications", Journal of the ACM, Vol 46, No. 2, 236-280, 1999

Nilsson, S. and Tikkanen, M., "Implementing a dynamic compressed trie", Proceedings WAE'98, Mehlhorn, K. (Ed), 25-36, 1998

Askitis, N and Sinha, R., "HAT-trie: a cache-conscious trie-based data structure for strings", Proc. 30th Australasian Conference on Computer Science, Vol 62, 97-105, 2007