# An Indexing Scheme for a Scalable RDF Triple Store

David Makepeace[1], David Wood[1], Paul Gearon[1], Tate Jones[1] and Tom Adams[1]

[1] Tucana Technologies, Inc.,
11710 Plaza America Drive, Reston, VA, 20147, USA
{davidm, dwood, pag, tjones, tom}@tucanatech.com
http://www.tucanatech.com/

A database of Resource Description Framework (RDF) statements suggests indexing schemes different from those commonly used in relational databases in order to achieve scalability and robustness required during write transactions. This paper suggests an indexing scheme for RDF databases that uses AVL trees in place of B-tree-like indexes.

## 1 Introduction

The Resource Description Framework (RDF) is a World Wide Web Consortium (W3C) Recommendation for the representation of metadata. At its most fundamental level, RDF consists of statements of short fixed sentences, comprising a subject, a predicate and an object. This short sentence of three terms is collectively known as an RDF statement or triple, which when combined, together form a directed graph with subjects and objects comprising the nodes and predicates forming the arcs [1, 2].

## 2 RDF Data Storage

Several databases currently exist for the storage of RDF data. These databases – often referred to as RDF stores – exist as both open source projects and commercial product offerings. The first RDF store was Guha’s rdfDB [4], an intellectual successor to this is the Kowari project [5]. Kowari is an open source, massively scalable, transaction-safe, purpose-built database for the storage and retrieval of metadata.

Native RDF stores exist due to difficulties storing large quantities of RDF data in relational databases. Storage of metadata in a relational database typically requires a design with a few (one to six) very narrow, deep tables. Each table might consist of two to five columns, most of which are indexed more than once for performance. A basic query would require all tables to be joined with additional table aliases for nested joins. Large numbers of individual queries are required to perform more complex queries, resulting in slower response times.

Updating metadata statements in a relational database is expensive since all columns are indexed. This would require data pages and indexed columns to be locked. Updates in a relational database are slow when multiple data locks attempt to obtain the same page lock (a "hot spot"). These types of locks also affect reading performance and maintenance.

These problems are avoided in Kowari, which allows inline writes into its indexes, does not have the concept of data pages and makes use of phase trees to allow for long-lived read transactions during writing [3].

## 3 AVL Trees

Kowari is composed of a number of components, including a query layer at the top of the stack, and an RDF statement storage layer at the bottom. The statement-based approach of the storage layer treats predicates (properties) as just another data element. This contrasts with existing database formalisms (e.g. [6]), which treat relations as completely different from elements. The statements are indexed using three parallel Adelson-Velskii and Landis (AVL) trees – an index for each subject, predicate and object. AVL trees are balanced binary search trees where the height of the two subtrees of any node differs by at most one, the children of any node are themselves AVL trees. This structure has the useful property that searching, insertion, and deletion times are O(log n), where n is the number of nodes in the tree. O(log n) times are guaranteed by the fact that the trees are balanced [7, 8].

Each index in Kowari is an AVL tree that provides addressing information into a large random-accessed file. Using this structure allows the AVL trees to remain relatively small and are often able to fit into physical memory. In practice, the speed of searching, insertion and deletion operations are linear when the depth of the AVL trees is relatively small. As trees may become unbalanced during write operations, they must be rebalanced, often by rotation of a node's children. This is a simple operation on the addressing information tree which does not affect the underlying data. In the general case, rebalancing times are also O(log n). Nodes are split into two when full.

AVL trees are particularly useful for RDF stores whose query approach systematically constrains one or more elements of a triple (or quad if grouping information is added to the "triple"). Triples may be indexed in order and rotated to create parallel indexes. Any given constraint on one or more triple elements may thus be satisfied as a simple range in one of the indexes. For example, a query asking for RDF statements whose subject is a given string may be satisfied by performing a search for triples in a subject-first index where the subject returned is the number corresponding to that string literal that can be used as a key in a hash table lookup.

The AVL trees each have a different key ordering, defined as follows:

```
(subject, predicate, object),
(predicate, object, subject) and
(object, subject, predicate).
```

Each node in the AVL tree holds the following information:

- A set of triples sorted according to the key order for this tree;
- The number of triples in the set for this node;
- A copy of the first triple in the sorted set;
- A copy of the last triple in the sorted set;
- The ID of the left subtree node;
- The ID of the right subtree node;
- the height of the subtree rooted at this node.

All triples in the left subtree compare less than the first triple in the sorted set and all triples in the right subtree compare greater than the last triple in the sorted set. Space for a fixed maximum number of triples is reserved for each node. A triple is added to a tree by inserting it into the sorted set of an existing node. If the only appropriate node is full then a new node will be allocated and added to the tree. A triple is removed from the tree by identifying the node that contains it and removing it from the sorted set. If the sorted set becomes empty then the node is removed from the tree.

A design goal was to keep index addressing information in memory for as long as possible. AVL tree nodes are split between two files such that the sorted set of triples for a node are stored as a block in one file while the remaining fields are stored as a record in the other file. This ensures that the traversal of an AVL tree does not result in sorted sets of triples being unnecessarily read into memory. This also allows for different file I/O mechanisms to be used for the two files. This is similar to approaches using B-trees in many relational databases.

B-trees generally do not fill their nodes completely, with each node having some number of entries between N/2 and N, where N is the order of the tree. This results in an index being (on average) about 25% larger than it needs to be. This pushes B-tree indexes out of a range that can be memory-mapped or cached, much sooner. This is a major concern when indexes can reach several gigabytes in size. AVL trees also tend to be faster to traverse than B-trees when the entire tree fits in memory.

Figure 1 illustrates the internal workings of the directed graph implementation in Kowari, showing a portion of an index implemented in a AVL tree. The data (stored as a series of triples) is sorted by the first component of the triple. The first component of each triple in the figure represents a subject. The data is stored only in the indices and is not stored separately elsewhere. Storing the data multiple times increases the storage requirements for the data set but allows for very rapid responses to queries since each query component can use the most appropriate index.

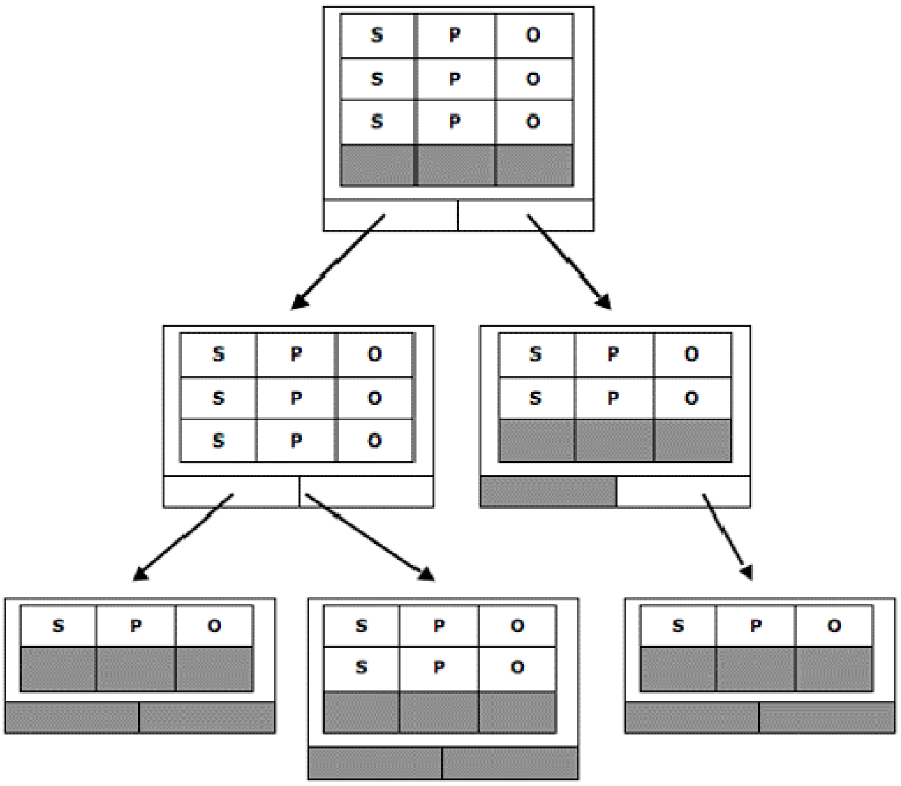

**Fig. 1.** Storing RDF statements in an AVL tree, in subject-predicate-object form

## 4 Conclusions and Further Work

Our work has shown that AVL trees are sufficient for storing hundreds of millions of RDF statements and allowing access to them in near real time. With current approaches, scaling above this level may force the AVL trees out of physical memory, moving searching, insertion and deletion operations out of the linear behavior zone and into a logarithmic one. Future scaling work above this level focuses on more complex indexing structures, such as keeping hashtables of sorted triples.

## References

1. Iannella, R., Guide to the Resource Description Framework, The New Review of Information Networking, Vol 4, (1998)
2. RDF/XML Syntax Specification (Revised), World Wide Web Consortium, http://www.w3.org/TR/rdf-syntax-grammar/ (2004)
3. Wood, D., Jones, T. and Hyland, B., A New Type of Data Management, Tucana Technologies, Inc. white paper, http://www.tucanatech.com/downloads/new_type_of_data_management.pdf, 2004-06-21
4. Guha, R., rdfDB: An RDF Database, http://www.guha.com/rdfdb/
5. Kowari, http://www.kowari.org/, 2004-06-21
6. Elmasri, R. and Navathe, S., Fundamentals of Database Systems, 2nd Ed, Benjamin Cummings Publishing Company (1994)
7. National Institute of Standards and Technology (NIST), Dictionary of Algorithms and Data Structures, Definition of AVL Tree, http://www.nist.gov/dads/HTML/avltree.html (1998)
8. Morris, J.,: Programming Languages and Data Structures course notes, http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/AVL.html (1998)