

Kowari: A Platform for Semantic Web Storage and Analysis

David Wood¹

Tucana Technologies, Inc.
11710 Plaza America Dr., Suite 2000
Reston, VA, USA 20190
+1 703 871 5312

dwood@tucanatech.com

Paul Gearon¹

Tucana Technologies, Pty Ltd
204 Given Tce, Level 1
Paddington, QLD, Australia 4064
+61 7 3876 2188

pag@tucanatech.com

Tom Adams

Tucana Technologies, Inc.
11710 Plaza America Dr., Suite 2000
Reston, VA, USA 20190
+1 703 871 5312

tom@tucanatech.com

ABSTRACT

Large-scale Semantic Web applications require large-scale storage of Resource Description Framework (RDF) information and a means to analyze that information via the Web Ontology Language (OWL) in near real time. The Kowari Metastore was designed as a purpose-built RDF database to fulfill this requirement. Kowari provides a scalable, transaction-safe storage infrastructure for RDF statements and an expressive query language for their analysis, with or without the use of a subset of the RDF Schema and/or OWL languages. OWL Lite plus the full cardinality constraints from OWL Full are currently supported via the interactive Tucana Query Language (iTQL) or the Simple Ontology Framework API (SOFA). Kowari's native quad-store indexing scheme has been shown to scale to hundreds of millions of RDF statements on a single machine. Kowari is an Open Source project sponsored by Tucana Technologies and is licensed under the Mozilla Public License, version 1.1.

Categories and Subject Descriptors

H.3.5 [Information Storage and Retrieval]: Online Information Services – *Web-based Services*.

General Terms

Design, Experimentation.

Keywords

Semantic Web, RDF, OWL, Database, Storage, Analysis.

1. INTRODUCTION

The Resource Description Framework (RDF) [10], the Resource Description Framework Schema (RDFS) [4] and the Web Ontology Language (OWL) [11] are enabling technologies for the Semantic Web and are Recommendations of the World Wide Web Consortium (W3C). The Kowari Metastore provides a storage and analysis platform for RDF, RDFS and OWL information. Kowari has been under active development since 2001 by Tucana Technologies, as an Open Source baseline for Tucana's commercial product, the Tucana Knowledge Server (TKS).

Several systems 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. Since Guha's

first RDF store, rdfDB [8], several other Open Source stores have been developed. These include: Sesame [18], developed by Administrator Nederland b.v. as part of the European IST project On-To-Knowledge; 3store [20], developed by the University of Southampton; and Redland [2], developed by the University of Bristol. The Jena Semantic Web Framework [9], developed by Hewlett Packard's Bristol Laboratory also provides the capability for persistent storage (thus making it an RDF store) by utilizing a persistent backend.

2. ARCHITECTURE

Figure 1 shows Kowari's high-level architecture. The topmost layer provides access APIs, below which is a query engine, backed by a storage layer.

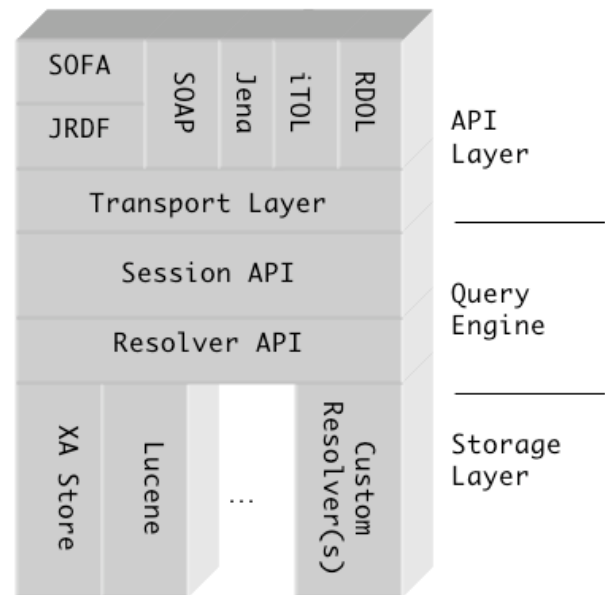


Figure 1. Kowari Architectural Overview.

The upper API layer supports many of the common Semantic Web and industry standard access APIs. The Query Engine is separated from the access API layer by a transport layer, allowing the APIs to be used across a network connection. The Session and Resolver APIs are responsible for taking access requests and

directing them to the storage layer.

Data represented in the query engine that exists in its original state and is said to be in a *globalized* form. Data represented in some portions of the storage layer is represented numerically and is said to be in a *localized* form, that is the numerical values are local to the Kowari instance and cannot be used externally. Globalization is the process of turning these local numeric representations of data into global representations suitable for returning to a user.

2.1 Access APIs

As shown in Figure 1 Kowari provides a number of developer level access APIs. These include Simple Ontology Framework API (SOFA) [1], Java RDF (JRDF) [14], Jena [9], interactive Tucana Query Language (iTQL) [21], RDF Data Query Language (RDQL) [19] and Simple Object Access Protocol (SOAP) [7]. Other APIs including a JavaServer Pages tag library and an RDFS JavaBean are also provided but not shown in the diagram. The JRDF, SOFA, Jena and iTQL APIs can all be used on the client side or in-JVM local to the server.

2.2 Query Engine

The Kowari query engine uses an abstract object model that is bound to a concrete query syntax and other higher level APIs by the Session API. This abstraction allows for easy API integration and the adoption of different query languages. By default, Kowari's query engine parses and executes commands in iTQL.

After a query has been issued, it is parsed into a set of constraints and models. Models can be combined into an object called a model expression using the set operations union, intersection and partition. Individual models are described by URLs, of which the host portion is used to describe the server where the model appears.

Constraints define a subset of a model, and the results may be combined using inner joins and append operations. The full combination of constraints from a query forms an object called a constraint expression. The constraints in the constraint expression will normally be evaluated against the full model expression, however individual constraints may override this and can define a single model to be resolved against. If this override operation is employed then it is stored within the constraint that it applies to.

The full combination of the model expression and the constraint expression forms the basis of a query object, and this is then sent to the first server found in the model expression.

Once a query is received, the first task of the query engine is to estimate the size of the data that each constraint will resolve to, which is determined by resolving each constraint and requesting an upper bound to the size of the result. As results may be very large, using estimates to the upper bound can avoid unnecessary traversal of the data to obtain an exact figure.

Normal constraints are resolved by applying them to each model in the model expression, with the results being combined according to the operations from the model expression. For those constraints that override the model expression, the resolution is performed against a single model.

The constraint results are then used along with the operations defined in the constraint expression to construct an execution plan that will optimize the order in which the individual constraints should be joined together. This initial optimization attempts to minimize both time and space usage, with inner joins on smaller data sets being preferred in order to reduce the size of the working data set.

Once the initial plan is established, the query engine starts to combine the results using the join operations from the query. During this process the sizes of intermediate results may indicate a more efficient execution plan. When this happens the query engine will further optimize the plan, modifying the order of subsequent joins as it proceeds. Once the final join operation has completed the results are returned to the client.

2.3 Resolver API

At the interface between the upper parts of the query engine and the storage layer sits an abstraction layer called the Resolver API. This API gets its name from the resolution of constraints on RDF statements by the storage layer when called by the query engine. An incoming query is broken down to the point where individual constraints are to be resolved against individual RDF models. The Resolver API has been developed to allow these simple operations to be applied against almost any kind of data source. The intention is to allow pluggable graph implementations other than the native Kowari data store. These alternative graph implementations are called Resolvers. There are two types of Resolver: external and internal. Internal resolvers are those whose data source is controlled solely by Kowari (e.g. the XA Statement Store), while external resolvers are those whose data source is not controlled by Kowari (e.g. a relational database).

Generally, RDF models are listed in a special model known as the system model. Models that can be resolved with Internal Resolvers are listed in the system model and are local to a Kowari server. For example, the native XA Statement Store is implemented as an internal Resolver, and all models stored in the XA Statement Store are listed in the system model. Arbitrary model URLs that are external to Kowari are not referred to in the system model and these are handled by the External Resolvers.

An internal Resolver operates on a data store internal to a Kowari server and therefore has its own model listed within the Kowari system model. Usually, internal Resolvers are used alongside data stores with RDF-ready information, which require very little or no conversion. Internally resolved models are the most stable as they are controlled solely by Kowari.

An external Resolver has its model outside of the scope of the system model and outside the control of Kowari. External Resolvers are useful for data stores that are not in an RDF-ready format and require some processing before results can be returned. This is most often used for resolving data in serialized file formats, but can also be useful for connecting to remote systems via wire protocols. A relational database or electronic mail service, for example, may be connected to Kowari as external Resolvers.

There is a danger to using external Resolvers because the data being queried is not controlled by Kowari and there is no guarantee of the model being present. External factors, such as other users, servers, or security protocols may alter or remove files or stores being resolved, thus contaminating or causing errors in the results.

The full API is comprised of three interfaces, known as Resolver, ResolverFactory and Resolution. The ResolverFactory interface has only 3 methods. These handle initialization, cleaning up, and creating instances of the Resolver interface. The Resolver interface has five methods, three of which are used for data modification and transactions, and can be ignored for read-only data stores, such as an RDF/XML file. The Resolution interface is for non-volatile data objects used to return results from the Resolver interface. The simplicity of these interfaces means that

the Resolver API is quite easy to implement, regardless of the type of data being accessed.

When Kowari starts, a configuration file is read that contains a list of Resolvers to use. As each Resolver implementation is loaded, it registers its ability to handle a type of model. For external Resolvers, the type is based on the protocol part of the model's URL. When a constraint is to be resolved against a model on the current server, the query engine looks up the type in the system model, and maps the request to the appropriate Resolver.

2.4 Data Store

Kowari's native data store (the XA Statement Store) is a native Java version 1.4 transactional data store that stores RDF statements persistently on disk without the use of an external database.

The Statement Store stores statements as "quads" consisting of subject, predicate, object and meta nodes. The first three items form a standard RDF statement and the meta node describes which model the statement appears in. Each quad is unique, so a statement that appears in two models will be listed twice, with differing meta node values.

To increase speed and reduce redundancy, the nodes that make up a statement are stored as 64 bit integers, allocated by the Node Pool. These integers are known as node numbers, and correspond to URIs and Literals which are stored in the String Pool. The Statement Store is essentially composed of these two stores that together provide the full requirements of RDF storage.

Kowari stores RDF statements in six different orders. This corresponds to the minimum number of ways the four node types can be arranged such that any collection of one to four nodes can be used to find any statement or group of statements that match. Each of these orderings therefore acts as a compound index, and independently contains all of the statements for the RDF store.

The String Pool is built using an Adelson-Velskii and Landis (AVL) tree to store and order its elements, while the indexes in the Statement Store are based on a hybrid of AVL trees and B-Trees. AVL trees are balanced binary trees, offering fast, simple searching and modification, while B-trees are balanced multi-branch trees which can take more space overhead, with each node taking longer to traverse, but the trees are shallower to search.

The String Pool uses an AVL tree to store indexes into a set of flat files which hold any kind of RDF data. In this context, RDF data includes URI References, String Literals, and recently, all XML Schema data types. The data is ordered in the tree according to the semantics of each type. The pool also holds a reverse index to locate any entry according to its node number.

The Statement Store indexes use an AVL tree to locate a block of statements found in a separate file. Within each block is up to 256 ordered statements. Like a B*-tree, these blocks make the tree much shallower to search, and get split into two blocks when more than the maximum number of statements are inserted. However, it is indexed with an AVL tree, with the speed and simplicity that this structure offers. Thus, the hybrid picks up some of the speed advantages of both approaches.

The statement-based approach of the storage layer treats predicates (properties) as just another data element. This contrasts with existing database formalisms (e.g. [3]), which treat relations as completely different from elements. The statements are indexed using six parallel AVL trees – an index for each required ordering of subject, predicate, object and meta node. 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 [13].

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 can often 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 that 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 model 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.

If the meta nodes are ignored for simplicity, the number of required indexes is reduced to 3. The AVL trees for these indexes 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; and
- 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 re-moved 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 2 illustrates the internal workings of the directed graph implementation in Kowari, showing a portion of an index implemented in an AVL tree (for simplicity the meta node information is not shown). 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 the address of 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.

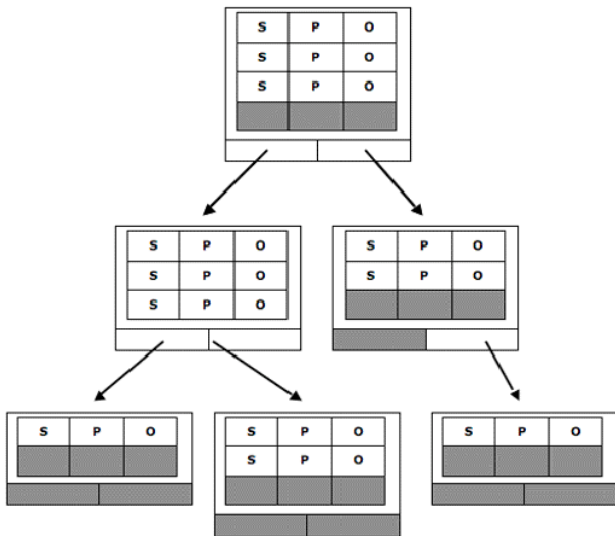


Figure 2. Storing component addressing information in an AVL tree.

3. QUERYING

3.1 Interactive Tucana Query Language

The interactive Tucana Query Language (iTQL) is a declarative language based on SQL and the Squish [12] family of RDF query languages. There is, as yet, no standard RDF query language. Most extant RDF query languages are declarative languages based loosely on SQL. SQL itself cannot be used to directly query graph-like structures, such as Kowari's native data store, because it queries hierarchical (tabular) or linked-tabular data. RDF graphs may not be hierarchical or linked-tabular data structures. SQL-like RDF query languages constrain RDF statements in a WHERE clause, much like SQL constrains table column entries. iTQL does this, then provides a list of variables in a projection which may be selected in the SELECT clause.

iTQL allows one to create RDF models, insert or delete RDF statements into those models and query the results. Loading of bulk data from external files is supported, as is the running of iTQL scripts. iTQL provides support for transactions and the ability to manually control transactions.

Individual RDF models in Kowari, or an entire database, may be backed up while the server is running. Data from these backups may be used to restore an individual model or the entire database.

iTQL also contains other features such as:

- Named RDF models (by URI)
- Views (updatable compositions of models or views)
- Models of varying data types (with type conversion hidden from user)
- Infinitely nested SELECT clauses
- Full aliasing support
- Distributed query support (in TKS only)
- User login, resulting in data view restrictions (in TKS only)

iTQL also contains features to allow for data mining operations including the *trans*, *walk*, *exclude*, *subquery* and aggregation functions. These functions are described in detail below.

3.2 Comparison to RDQL and SPARQL

RDQL is perhaps the most widely implemented RDF query language. Kowari implements a Jena storage backend, which allows (a subset of) RDQL to also be supported in Kowari as an alternative query language to iTQL.

The Simple Protocol and RDF Query Language (SPARQL) [15] is a relatively new query language proposal from the W3C RDF Data Access Working Group (DAWG) which is working to standardize an RDF query language and access protocol. A limited feature set has been defined for SPARQL as of this writing, although this is expected to change as the language development continues.

The major difference between iTQL and these other languages is the greater number of features offered in iTQL. For example, iTQL names models by URIs and refers to those URIs in query FROM clauses. As described above, model expressions may be composed using set operations, and in the case of the Tucana Knowledge Server, these models may reside on separate machines. Users may authenticate to a running server via a login command in order to enable differing views of data, segmented by user or group permissions (available in the Tucana Knowledge Server only). OWL language features are not supported in either RDQL or SPARQL as of yet.

There are also minor syntactic differences between the languages, such as iTQL's delimiting of URIs with angle brackets (\langle , \rangle) or the termination of queries with a semicolon (;). Such differences are easily rectified once SPARQL's syntax becomes stable. Of minor concern is the difference between SPARQL's *not* and iTQL's *exclude* operators, however the Kowari team intends that iTQL will eventually migrate its syntax and operations to become a superset of the evolving standard.

4. ANALYSIS

An important feature of the Semantic Web is the ability to make statements regarding given data. RDF Schema, OWL Lite, OWL DL and OWL Full all provide a framework for testing consistency and making entailment inferences on data. While some inferences

are trivial, many require complex interactions of data that in turn require support in the query language.

Beyond the notions of subject-predicate-object triples and models, iTQL does not directly support concepts from RDF. This means that constructs such as containers, node types, and reification cannot be directly referred to in the language. Similarly, there is no specific support for RDFS and OWL. Instead, iTQL provides a set of fundamental operations appropriate to querying a graph.

iTQL queries are based on constraints that limit individual subject-predicate-object triples, to specified patterns. Using constraints that contain defined RDF predicates, along with a combination of inner join and append operations, many RDF, RDFS and OWL constructs are readily expressible. Any unmet concepts are describable using a set of language features specifically designed to fill those niches.

For predicates that are transitive, such as `rdfs:subClassOf`, the *trans* and *walk* functions have been introduced. The *walk* function will return all statements from a model that follow a path of transitive predicates, starting at a given node. The *trans* function will follow that same path, but will return all statements that can be inferred from that transitive predicate. The *trans* function can also be left without a starting point, allowing it to return all possible inferences of transitive closure on a given predicate in a model.

The *trans* function can be applied to the `rdf:rest` predicate to return the entire list from an RDF container in a single step. Similarly the *trans* and *walk* functions can be used to efficiently infer statements from predicates described by `owl:TransitiveProperty` statements.

The *exclude* function in iTQL will return all data from a model with the exception of the data selected by the excluded constraint. In this way it fills the role of a closed-world *not* operation on a single constraint. This kind of operation is particularly useful for finding statements that violate an explicit condition, as opposed to normal constraints, which find statements that meet a condition.

The final iTQL construct is the subquery. This is a flexible construct that provides the functions of filtering, aggregation and function chaining on projections. The filtering function allows differences to be calculated on complex constraints operations, an important function that the *exclude* function cannot perform. Function chaining allows a new query to be executed for each result which would be returned from the initial query, eliminating the need for an external framework to perform this function. Aggregation is currently limited to counting, and is used to count the occurrence of statements. This can be used to implement Existential quantifiers, Universal quantifiers, and the cardinality restrictions described in all three species of OWL.

In addition to the features of iTQL, Resolvers may produce virtual statements that provide information complementary to the constraints of a query. One such implemented Resolver is the “Node-Type Resolver”, which produces virtual statements stating if nodes are URI References or Literals. This is important for entailment, as statements should never be made about Literals, and inferencing is not always to be extended to blank nodes.

Using these structures as building blocks, it is possible to build efficient and generic iTQL statements that are capable of generating new statements based on entailment rules, and finding statements that violate model consistency. Kowari currently has a library of iTQL statements that can implement entailment rules and consistency checks for all RDFS and OWL Lite statements.

This library also uses subquery aggregation to test consistency for the complete set of OWL cardinality restrictions of OWL Full.

To provide comprehensive inferencing, the full library of rules must be run over a model, along with ontology data supplied as RDFS and OWL statements. The rules are often mutually dependent and may need to trigger each other to be re-executed. For this reason they need to be applied intelligently and efficiently, keeping the entailments up to date when data is modified, and spotting any consistency violations. As of this writing, these rules need to be executed by hand. Automatic updating by rules is in progress.

Using a portion of the iTQL rules library, an RDFS rules engine has been implemented, initially based on Drools. This rules engine is currently being replaced with a native rule engine that uses the “Rete” algorithm [6], with efficiencies gained by using semi-naïve evaluation and some backward chaining. This implementation does not yet handle dynamic changes to data, but it has demonstrated an efficient proof of concept. The rules engine is in the process of being extended to implementing the full OWL Lite suite, as well as the cardinality restrictions of OWL Full.

An alternative implementation of OWL Lite for Kowari is a SOFA API layer [1] that interacts with the JRDF API. SOFA provides a programmable interface for interacting with OWL data and OWL inferences on a data store. It also allows for creating OWL inferences based on data that it discovers. SOFA interacts with the data store at a low level, meaning that many of the efficient iTQL operations mentioned above are not available to it. Instead, SOFA must extract the data from the store in a raw triple form in order to perform inferences internally. This approach is not scalable in the long term, however it does provide an immediate OWL inferencing solution for entailment and consistency. The approach also has the advantage of being able to attach to an existing Kowari framework as a pluggable component.

5. IMPLEMENTATION EXPERIENCE

The current version of Kowari (1.0.x) has been shown to load 50 million RDF statements from an RDF/XML source document in approximately seventeen hours on a 32-bit operating system (Fedora Core 1 on a 2.8 GHz Intel Pentium 4 CPU) using Java version 1.4.2 and explicit I/O. The write performance is roughly linear throughout the first 50 million triples, although a positive logarithmic shape appears shortly thereafter. The same Kowari build loaded approximately 235 million RDF statements in the same time when run on a 64-bit operating system (64-bit RedHat Enterprise Linux on a 1 GHz AMD Operton CPU) using mapped I/O. The 64-bit write operations remain linear through approximately the first 180 million RDF statements, after which time a positive logarithmic shape appears.

Using the same 32-bit system described above, importation of RDF/XML data into Kowari occurs at a rate of roughly 7,500 statements per second during the linear portion of the write phase and reading of results following a query occurs at a rate of roughly 33,000 statements per second. This compares favorably with the storage of RDF data in a MySQL database using a simplistic schema, which resulted in approximately 1,800 write transactions per second and 1,200 read transactions per second using the same data set. It should be noted that many factors could obscure these metrics during use, such as network delays, socket creation and destruction and rendering of results.

Kowari has been tested with 250 simultaneous simulated clients, which queried a single Kowari instance using a random selection from a pool of recorded queries fifteen times in five second intervals. The queries were of varying complexity, from trivial to taxing. These tests showed that Kowari serviced all clients evenly with an average execution time of 150 milliseconds when running on the systems outlined above.

6. FUTURE DIRECTIONS

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 or the use of ski-lists [16].

Kowari's OWL implementation is also an area of focus and will continue to improve. The recent addition of the Drools rules engine [23] and refactoring of the query engine will facilitate the eventual support of both OWL Full and OWL DL. An implementation of the SPARQL syntax and access protocol is planned along with the migration of iTQL syntax operations to become a superset of the evolving standard.

Kowari currently relies on Java's Remote Method Invocation (RMI) protocol for its internal transport layer, although a BEEP protocol has also been developed. The new Resolver API allows for a resolver to implement any transport protocol, although the addition of pluggable protocol handling API would allow for additional protocols to be added more cleanly. Also, the addition of an HTTP-based transport has been much requested and will likely form the basis of the SPARQL protocol.

The additional refactoring necessary to create the above APIs would allow Kowari to be built in different ways. For example, a Kowari server might use an in-memory store and answer SPARQL queries via HTTP or an in-JVM JRDF API backed by the native XA Store.

The Resolver API may also be used to add content handlers that allow the input of MP3 files, JPEG images with EXIF, PDF files with XMP metadata, Word documents, RSS XML feeds, etc, extract metadata from them and convert the metadata to RDF. We currently have implemented content handlers for MP3 files and the mbox email format.

It is hoped that these architectural enhancements will encourage members of the Open Source community to contribute content handlers and Resolvers to Kowari.

7. CONCLUSIONS

Kowari, and its big brother the Tucana Knowledge Server, have been presented as highly scalable RDF data stores which provide substantial analytics capability. Kowari is an important entrant to the field of Semantic Web databases due to the richness of the interactive Tucana Query Language compared to its competitors RDQL and the newly defined SPARQL. Kowari has been successfully fielded in commercial, government and academic settings. Kowari appears to be a reliable base upon which to build Semantic Web applications that require scaling and other "enterprise-ready" features.

8. ACKNOWLEDGMENTS

The authors would like to thank the Tucana engineering team for the work that they've done to make Kowari the great system it is today. We would also like to thank the contributors to the Kowari project made both in source contributions and bug reports.

9. REFERENCES

- [1] Alishevskikh, A., SOFA – Simple Ontology Framework API, <https://sofa.dev.java.net/>
- [2] Beckett, D., Redland RDF Application Framework, <http://librdf.org/>
- [3] Beckett, D. and Grant, J., SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes, http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/, 23 January 2003
- [4] Brickley, D. and Guha, R.V. (eds), RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation, <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>, 10 February 2004
- [5] Elmasri, R. and Navathe, S., Fundamentals of Database Systems, 2nd Ed, Benjamin Cummings Publishing Company, 1994
- [6] Forgy, C., Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence, 1982
- [7] Gudgin, M. et al. (eds), SOAP Version 1.2 Part 1: Messaging Framework, <http://www.w3.org/TR/soap12-part1/>, 24 June 2003
- [8] Guha, R., rdfDB: An RDF Database, <http://www.guha.com/rdfdb/>
- [9] Jena Semantic Web Framework, <http://jena.sourceforge.net/>
- [10] Klyne, G. and Carroll, J. (eds), Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation, <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>, 10 February 2004
- [11] McGuinness, D. and van Harmelen, F. (eds), OWL Web Ontology Language Overview, W3C Recommendation, <http://www.w3.org/TR/2004/REC-owl-features-20040210/>, 10 February 2004
- [12] Miller, L., RDF Squish query language and Java implementation, <http://ilrt.org/discovery/2001/02/squish/>, February 22, 2001
- [13] 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
- [14] Newman, A., JRDF: Java RDF, <http://jrdf.sourceforge.net/>
- [15] Prud'hommeaux, E. and Seaborne, A. (eds), SPARQL Query Language for RDF, W3C Working Draft, <http://www.w3.org/TR/2004/WD-rdf-sparql-query-20041012/>, 12 October 2004
- [16] Pugh, W., Skip Lists: A Probabilistic Alternative to Balanced Trees, Communications of the Association of Computing Machinery, June 1990

- [17] Ramakrishnan, R. and Ullman, J., A Survey of Research of Deductive Database Systems, Journal of Logic Programming, May 1995, <http://citeseer.ist.psu.edu/ramakrishnan93survey.html>
- [18] Sesame, <http://www.openrdf.org/>
- [19] Seaborne, A., RDQL - A Query Language for RDF, W3C Member Submission, <http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/>, 9 January 2004
- [20] Threestore, <http://inanna.ecs.soton.ac.uk/3store/>
- [21] Tucana Technologies, iTQL Commands, <http://kowari.org/271.htm>
- [22] Volz, R., Implementing OWL Lite in rule-based systems and recursion-enabled relational DBs, October 2004, <http://lists.w3.org/Archives/Public/www-webont-wg/2002Oct/att-0033/Paper.pdf>
- [23] The Werken Company, Drools, <http://drools.codehaus.org/>