Efficient Linked-List RDF Indexing in Parliament

Dave Kolas, Ian Emmons, and Mike Dean

BBN Technologies, Arlington, VA 22209, USA
{dkolas,iemmons,mdean}@bbn.com

Abstract. As the number and scale of Semantic Web applications in use increases, so does the need to efficiently store and retrieve RDF data. Current published schemes for RDF data management either fail to embrace the schema flexibility inherent in RDF or make restrictive assumptions about application usage models. This paper describes a storage and indexing scheme based on linked lists and memory-mapped files, and presents theoretical and empirical analysis of its strengths and weaknesses versus other techniques. This scheme is currently used in Parliament (formerly DAML DB), a triple store with rule support that has recently been released as open source.

1 Introduction

As the number and scale of Semantic Web applications in use increases, so does the need to efficiently store and retrieve RDF [1] data. A wide variety of RDF and OWL [2] applications are currently being developed, and each application's scenario may demand prioritization of one performance metric or another. Current published schemes for RDF data management either fail to embrace the schema flexibility inherent in RDF or make restrictive assumptions about application usage models.

Despite the fact that RDF's graph-based data model is inherently different than relational data models, many published schemes for RDF data storage involve reductions to a traditional RDBMS [3–7]. This results in the deficiencies of RDBMS's (inflexible schemas, inability to efficiently query variable predicates) being propagated to RDF storage; arguably, avoiding these deficiencies is one of the major reasons for adopting an RDF data model. Other published approaches eschew the mapping to an RDBMS, but suffer either inadequate load or query performance for many applications. In this paper, we argue that the storage approach in Parliament provides excellent load and query performance with low space consumption and avoids the pitfalls of many other specialized RDF storage systems.

Parliament [8] (formerly DAML-DB [9]) is a triple store developed by BBN that has been in use since 2001. During that time, Parliament has been used for a number of applications from basic research to production. We have found that it offers an excellent tradeoff between load and query performance, and compares favorably to commercial RDF data management systems [10].

Recently, BBN has decided to release Parliament as an open source project. Parliament provides the underlying storage mechanism, while using Jena [11] or Sesame [12] as an external API. This paper explains in detail the underlying index structure of Parliament, and compares it to other published approaches. Our hope is that open-sourced Parliament will provide a fast storage alternative for RDF applications, create a platform upon which storage mechanism and query optimizer research can be built, and generally advance the state of the art in RDF data management.

The remainder of this paper is structured as follows. Section 2 addresses related work. Section 3 describes the index structure within Parliament. Section 4 explains how the operations on the structure are performed. Section 5 provides both worst case and average case analysis of the indexing mechanism, and Section 6 provides a small empirical comparison to supplement [10].

2 Related Work

The related work on RDF data management systems falls into two major categories: solutions that involve a mapping to a relational database, and those that do not.

2.1 RDBMS Based Approaches

A large proportion of the previously published approaches involve a mapping of the RDF data model into some form of relational storage. These include triples-table approaches, property tables, and vertical partitioning. There is a strong temptation to use relational systems to store RDF data since such a great amount of research has been done on making relational systems efficient. Moreover, existing RDBMS systems are extremely scalable and robust. Unfortunately, each of the proposed ways of doing this mapping has deficiencies.

The triples-table approach has been employed in 3store [3], and is perhaps the most straightforward mapping of RDF into a relational database system. Each triple given by (s, p, o) is added to one large table of triples with a column for the subject, predicate, and object respectively. Indexes are then added for each of the columns. While this approach is straightforward to implement, it is not particularly efficient, as noted in later work [4, 5, 13, 14, 10]. The primary problem is that queries with multiple triple patterns result in self-joins on this one large table, and are inefficient.

Property tables were introduced later, and allowed multiple triple patterns referencing the same subject to be retrieved without an expensive join. This approach has been used in Jena 2 [4]. A similar approach is used in [6]. In this approach, each database table includes a column for a subject and several fixed properties. The intent is that these properties often appear together on the same subject. While this approach does eliminate many of the expensive self-joins in a triples table, it still has deficiencies leading to limited scalability. Queries with triple patterns that span multiple property tables are still expensive. Depending

on the level of correlation between the properties chosen for a particular property table, the table may be very sparse and thus be less space-efficient than other approaches. Also, it may be complex to determine which sets of properties are best joined within the same property table. Multi-valued properties are problematic in this approach as well. Furthermore, queries with unbound variables in the property position are very inefficient and may require dynamic table creation. In a data model without a fixed schema, it is common to ask for all present properties for a particular subject. In the property table approach, this type of query requires scanning all tables. With property tables, adding new properties also requires adding new tables, a consideration for applications dealing with arbitrary RDF content. It is the flexibility in schema that differentiates RDF from relational approaches, and thus this approach limits the benefit of using RDF.

The vertical partitioning approach suggested in [5] may be viewed as a specialization of the property table approach, where each property table supports exactly one property. This approach has several advantages over the general property table approach. It better supports multi-valued properties, which are common in Semantic Web data, and does not sacrifice the space taken by NULL's in a sparsely populated property table. It also does not require the property-clustering algorithms for the general property tables. However, like the property table approach, it fails to efficiently handle queries with variables in the property position.

2.2 Other Indexing Approaches

The other primary approaches to RDF data storage eliminate the need for a standard RDBMS and focus instead on indexing specific to the RDF data model. This set of approaches tends to better address the query models of the semantic web, but each suffers its own set of weaknesses.

The RDF store YARS [15] uses six B+ tree indices to store RDF quads of a subject, predicate, object, and a "context". In each B+ tree, the key is a concatenation of the subject, predicate, object, and context, each dictionary encoded. This allows fast lookup of all possible triple access patterns. Unlike the RDBMS approaches discussed above, this method does not place any particular preference on the subject, predicate, or object, meaning that queries with variable predicates are no different than those with variable subjects or objects. This structure sacrifices space for query performance, repeating each dictionary encoded triple six times. The design also favors query performance to insertion speed, a tradeoff not necessarily appropriate for all Semantic Web applications. Our approach is more efficient both in insertion time and space usage, as will be demonstrated. Other commercial applications use this method as well [16]. Kowari [14] is designed similarly, but uses a hybrid of AVL and B trees instead of B+ trees for indexing.

The commercial quad store Virtuoso [7] adds a graph g element to a triple, and conceptually stores the quads in a triples table expanded by one column. While technically rooted in a RDBMS, it closely follows the model of YARS [15],

but with fewer indices. The quads are stored in two covering indices, g, s, p, o and o, g, p, s, where the IRI's are dictionary encoded. Several further optimizations are added, including bitmap indexing and inlining of short literal values. Thus this approach, like YARS, avoids the pitfalls of other RDBMS based work, including efficient variable-predicate queries. The pattern of fewer indices tips the balance slightly towards insertion performance from query performance, but still favors query performance.

Hexastore [13], one of the most recently published approaches, takes a similar approach to YARS. While it also uses the dictionary encoding of resources, it uses a series of sorted pointer lists instead of B+ trees of concatenated keys. Again, this better supports the usage pattern of Semantic Web applications and does not force them into a RDBMS query model. Hexastore not only provides efficient single triple pattern lookups as in YARS, but also allows fast merge-joins for any pair of two triple patterns. Again, however, it suffers a five-fold increase in space for storing statements over a dictionary encoded triples table, and favors query performance over insertion times. It is our experience that applications often do require efficient statement insertion, and thus our approach seeks to balance query performance and insertion time. Since this approach was published most recently and compares favorably to previous approaches, we will focus our empirical comparison evaluation on Hexastore.

Other commercial triple stores such as OWLIM [17] have been empirically shown to perform well, but their indexing structure is proprietary and thus no theoretical comparison can be made.

3 Index Structure

This section explains the three parts of the storage structure of Parliament: the resource table, the statement table, and the resource dictionary. This description is simplified for the sake of clarity; it does not discuss using quads instead of triples, optimizations for blank nodes, the rule engine, or some small implementation details. Parliament can be compiled in either 32 or 64 bit modes, and the width of the fields described varies accordingly.

3.1 Resource Table

The Resource Table is a single file of fixed-length records, each of which represents a single resource or literal. The records are sequentially numbered, and this number serves as the ID of the corresponding resource. This allows direct access to a record given its ID via simple array indexing. Each record has eight components:

- Three statement ID fields representing the first statements that contain this resource as a subject, predicate, and object, respectively
- Three count fields containing the number of statements using this resource as a subject, predicate, and object, respectively

– An offset into the string representations file described below, used to retrieve the string representation of the resource
– Bit-field flags encoding various attributes of the resource

The first subject, first predicate, and first object statement identifiers provide pointers into the statement table, which is described below. The subject, predicate, and object counts benefit the find operations and query optimization, discussed in Section 4. For the remainder of the paper, these counts will be referred to as $count(resource, pos)$ for the count of a resource in the given position. The usage of the offset into the string representations file will be explained below.

3.2 Statement Table

The Statement Table is the most important part of Parliament's storage approach. It is similar to the resource table in that it is a single file of fixed-length records, each of which represents a single statement. The records are sequentially numbered, and this number serves as the ID of the corresponding statement. Each record has seven components:

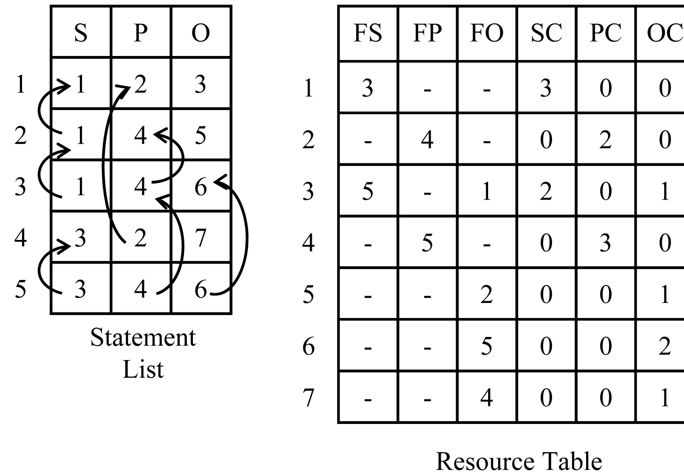
– Three resource ID fields representing the subject, predicate, and object of the statement, respectively
– Three statement ID fields representing the next statements that use the same resource as a subject, predicate, and object, respectively
– Bit-field flags encoding various attributes of the statement

The three resource ID fields allow a statement ID to be translated into the triple of resources that represent that statement. The three next statement pointers allow fast traversal of the statements that share either a subject, predicate, or object, while still storing each statement only once.

Figure 1 shows an example knowledge base consisting of five triples. Each triple row in the statement list table shows its resource identifier as a number and the pointers to the next statements as arrows. Omitted arrows indicate pointers to a special statement identifier, the null statement identifier, which indicates the end of the linked list.

3.3 Resource Dictionary

Like many other triple stores [13, 5, 6, 12], Parliament uses a dictionary encoding for its resources. This dictionary provides a one-to-one, bidirectional mapping between a resource and its resource ID. The first component of this dictionary is the mapping from a resource to its associated identifier. This portion of the dictionary uses Berkeley DB [18] to implement a B-tree whose keys are the resources' string representations and whose values are the corresponding resource ID's. This means that inserts and lookups require logarithmic time.



Statement List

	S	P	O
1	1	2	3
2	1	4	5
3	1	4	6
4	3	2	7
5	3	4	6

Resource Table

	FS	FP	FO	SC	PC	OC
1	3	-	-	3	0	0
2	-	4	-	0	2	0
3	5	-	1	2	0	1
4	-	5	-	0	3	0
5	-	-	2	0	0	1
6	-	-	5	0	0	2
7	-	-	4	0	0	1

Fig. 1. Example Statement List and Resource Table

The second half of the dictionary is the reverse lookup from a resource ID to a string representation. This is implemented in a memory-mapped file containing sequential, variable-length, and null-terminated string representations of resources. A resource ID is translated into the corresponding string representation by using the resource ID to index into the resource table, retrieving the string representation offset, and using this to index into the string representations file to retrieve the associated string. Thus, looking up a string representation from a resource identifier is a constant time operation.

The current approach stores each string representation twice. Future implementations may eliminate this redundancy.

3.4 Memory-Mapped Files

Three of the four files that comprise a Parliament triple store (the resource table, the statement table, and the string representations file) are stored and accessed via a common operating system facility called "memory mapping". This is independent of the index structure of the store, but is worth mentioning because it confers a significant performance advantage. Most modern operating systems use memory mapping to implement their own demand-paged virtual memory subsystem, and so this mechanism for accessing files tends to be highly optimized and keeps frequently accessed pages in memory.

4 Triple Store Operations

The three fundamental operations that a triple store can perform are query, insertion (assertion), and deletion (retraction). These are discussed below.

4.1 Query

Parliament performs a lookup of a single triple pattern according to the following algorithm:

1. If any of the triple pattern elements are bound, Parliament uses the B-tree to translate the string representations of these resources into resource ID's.
2. If any bound elements are not found in the B-tree, then the query result is the empty set, and the query algorithm terminates.
3. If none of the elements are bound, then the query result is the entire statement list. Parliament enumerates this by iterating across all of the records in the statement table and retrieving the string representations of the elements.
4. If exactly one element is bound, then Parliament looks in the resource table for the resource table the ID of the first statement using that resource in the position the resource appears in the given triple pattern.
5. If two or three elements are bound, then Parliament looks in the resource table for those resource ID's to retrieve $count(resource, pos)$ for each. Parliament selects the resource whose count is smallest, and retrieves from the resource table the ID of the first statement using that resource in the position the resource appears in the given triple pattern.
6. Starting with that statement ID, Parliament traverses the linked list of statement records corresponding to the position of the minimal count resource.
7. If the triple pattern contains exactly one bound resource, then this list of statements is exactly the answer to the query, and again Parliament retrieves the string representations of the elements as it enumerates the list to form the query result.
8. If two or three elements are bound, then as Parliament enumerates the linked list of statements, it checks whether the resources in the positions of the non-minimal count resources are the same as the bindings in the given triple pattern. Whenever a match is found, Parliament retrieves the string representations of the elements and adds that triple to the query result.

Whenever Parliament is enumerating statements, it skips over statements whose "deleted" flag has been set. See Section 4.3 below for details.

Parliament is designed as an embedded triple store and does not include a SPARQL or other query language processor. Such queries are supported by accessing Parliament as a storage model from higher-level frameworks such as Jena or Sesame. Single find operations (as discussed above) are combined together by the higher-level framework, with Parliament-specific extensions for optimization. In particular, when using Parliament with Jena's query processors [11], we have used several different algorithms for query planning and execution, which will be detailed in subsequent publications. The basis of these optimizations is the ability to quickly access the counts of the resources in the given positions.

4.2 Insertion

To insert a triple (s, p, o), Parliament executes the following algorithm:

1. Parliament uses the B-tree to translate the string representations of the three resources into resource ID's.
2. If all three elements are found in the B-tree, then Parliament performs a query for the triple pattern (s, p, o). Note that this is necessarily a fully bound query pattern. If the triple is found, then no insertion is required, and the algorithm terminates.
3. If any elements are not found in the B-tree, then Parliament creates new resources for each of them as follows:
   (a) Parliament appends the string representation of the resource to the end of the string representations file. If the file is not large enough to contain the string, then the file is enlarged first. The offset of the beginning of the string is noted for use in the next step.
   (b) Parliament appends a new record to the end of the resource table. If the file is not large enough to contain the new record, then the file is enlarged first. The number of the record is saved as the new resource ID for use in the steps below, and the offset from the string representations file is written to the appropriate field in this record. The record's counts are initialized to zero, and the first statement ID's are set to null.
   (c) Parliament inserts a new entry into the B-tree. The entry contains the resource's string representation as its key and the new resource ID as its value.
4. Parliament now has three valid resource ID's representing the triple, and knows that the triple is not present in the statement table.
5. Parliament appends a new record to the end of the statement table. If the file is not large enough to contain the new record, then the file is enlarged first. The number of the record is saved as the new statement ID for use in the steps below, and the three resource ID's obtained above are written to the appropriate fields in this record. The record's next statement ID's are all set to null.
6. For each of the three resources, Parliament inserts the new statement record at the head of that resource's linked list for the corresponding triple position as follows:
   (a) The resource record's first statement ID for the resource's position is written into the corresponding next statement ID field in the new statement record. Note that if this resource was newly inserted for this statement, then this step will write a null into the next statement ID field.
   (b) The ID of the new statement is written into the resource record's first statement ID for the resource's position.

4.3 Deletion

The index structure of Parliament's statement table is not conducive to the efficient removal of a statement record from the three linked lists of which it is a member. These linked lists are singly linked, and so there is no way to remove a record except to traverse all three lists from the beginning.

Due to these difficulties, Parliament "deletes" statements by marking them with a flag in the bit field portion of the statement record. Thus, the algorithm consists of a find (as in the case of an insertion, this is a fully bound query pattern) followed by setting the flag on the found record. In the future, we may utilize doubly linked lists so that the space occupied by deleted statements can be reclaimed efficiently. However, in our work to date deletion has been infrequent enough that this has been deemed a lower priority enhancement.

5 Theoretical Analysis

As is readily apparent, the presented approach suffers some unfortunate worst case performance, but the average case performance is quite good. This is consistent with empirical results presented in [10] and this paper. We will address both find operations on a single triple pattern and triple insertions.

5.1 Worst Case Analysis

The worst-case performance for a single triple pattern lookup is dependent on how many of the elements in the pattern (s, p, o) are bound. If zero elements are bound, the triple pattern results in a total scan of the statement list, resulting in $O(n)$. Since all triples are the expected result, this is the best possible worst case performance. If one element is bound, the chain for that particular element will be traversed with time $O(count(bound, pos))$. Again exactly the triples that answer the pattern are traversed.

Things change slightly for the cases where two or three of the (s, p, o) elements are bound. If two elements are bound, the shorter of the two lists will be traversed. This triple pattern can be returned in

$$O(min(count(bound_1, pos_1), count(bound_2, pos_2)))$$

However, this could be $O(n)$ if all triples use the two bound elements. If all three elements are bound, the shortest of the three lists will be traversed. This shortest list will be longest when the set of statements is exactly the three-way cross product of the set of resources. In this case, if the number of resources is m, then the number of statements is m^3 and every list is of length m^2. Thus the list length is $n^{2/3}$, and a find operation for three bound elements is $O(n^{2/3})$.

Since an insertion first requires a find on the triple to be inserted, it incurs the worst-case cost of a find with three bound elements, $O(n^{2/3})$. It also incurs the cost of inserting any nodes in the triple that were not previously known into the dictionary, but this logarithmic time $O(\log m)$ is overshadowed by the worst case find time. After that, adding the triple to the head of the lists is done in $O(1)$ constant time. Thus the worst-case of the insertion operation is $O(n^{2/3})$.

Here we note that this worst-case performance is indeed worse than other previously published approaches, which are logarithmic. However, the scenarios that produce these worst-case results are quite rare in practice, as will be shown in the following section.

5.2 Average Case Analysis

While the worst-case performance is worse than other approaches, analyzing the relevant qualities of several example data sets leads us to believe that the average case performance is actually quite good.

The most relevant feature of a data set with respect to its performance within this indexing scheme is the length of the various statement lists for a particular subject, predicate, or object. For instance, the worst-case time of the insert operation and the find operation with three bound elements is $O(n^{2/3})$, but this is associated with the case that the set of triples is the cross-product of the subjects, predicates, and objects, which is a highly unlikely real world situation. Since these bounds are derived from the shortest statement list, analysis of the average list lengths in a data set is a key measure to how this scheme will perform in the real world.

Table 1. List Length Means (Standard Deviations)

Data Set	Size	Subject	Predicate	Non-Lit Object	Lit Object
Webscope	83M	3.96 (9.77)	87,900 (722,575)	3.43 (2,170)	4.33 (659)
Falcon	33M	4.22 (13)	983 (31,773)	2.56 (328)	2.31 (217)
Swoogle	175M	5.65 (36)	4,464 (188,023)	3.27 (1,793)	3.38 (569)
Watson	60M	5.58 (56)	3,040 (98,288)	2.87 (918)	2.91 (407)
SWSE-1	30M	5.25 (15)	25,404 (289,000)	2.46 (1,138)	2.29 (187)
SWSE-2	61M	5.37 (15)	83,773 (739,736)	2.89 (1,741)	2.87 (300)
DBpedia	110M	15 (39)	300,855 (3,560,666)	3.84 (148)	1.17 (22)
Geonames	70M	10.4 (1.66)	4,096,150 (3,167,048)	2.81 (1,623)	1.67 (15)
SwetoDBLP	15M	5.63 (3.82)	103,009 (325,380)	2.93 (629)	2.36 (168)
Wordnet	2M	4.18 (2.04)	47,387 (100,907)	2.53 (295)	2.39 (271)
Freebase	63M	4.45 (15)	12,329 (316,363)	2.79 (1,286)	1.83 (116)
US Census	446M	5.39 (9.18)	265,005 (1,921,537)	5.29 (15,916)	227 (115,616)

Table 1 shows the mean and standard deviations of the subject, predicate, and object list lengths for several large data sets [19]. There are a few nice properties of this data that are worth noting. First, the average number of statements using a particular subject is quite small in all data sets. The average number of statements using a particular object is generally even smaller, though with a much higher standard deviation. Finally, only the predicate list length generally seems to scale with the size of the data set.

These observations have important implications with respect to the average time of find and insert operations. For find operations, we now know several things to be generally true:

- Either the object or subject list will likely be used for find operations when all three elements are bound. Thus these operations will often touch fewer than 10 triples.

- Due to the previous, insert operations should generally be quite fast.
- The predicate list is only likely to be used for find operations when only the predicate is bound, and thus only when all statements with the given predicate must be touched to be returned anyway.
- Find operations with two bound elements, which have the most troubling theoretical worst-case performance, necessarily include either a bound subject or bound object. As a result, these too should generally be quite fast.

These conclusions collectively suggest real world performance that is much more impressive than the worst-case analysis would imply, and this is shown empirically in the following section.

6 Empirical Analysis

Since Hexastore [13] is the most recently published work in this area, and its indexing structure out performed several of the other approaches, we have focused our empirical evaluation on Parliament as compared to Hexastore. At the time of our evaluation, only the prototype Python version of Hexastore was available for comparison. Future work will compare against the newly released version. This limitation resulted in the relatively small size of this empirical evaluation; we could not go beyond the size of main memory without the comparison becoming unfair to Hexastore. Parliament was tested with 850 million triples in [10].

Evaluation was performed on a MacBook Pro laptop with a 2.6 GHz dual core CPU, 4 GB of RAM, and a 7200 RPM SATA HD, running Mac OS X 10.5.7. This platform was most convenient for execution of both systems. While Hexastore's evaluation focused on only query performance, we feel it is important to include insertion performance and memory utilization as well, as there are many Semantic Web applications for which these factors are significant. We have focused on the Lehigh University Benchmark [20], as it was used in the Hexastore evaluation and contains insertion time metrics as well. We have evaluated LUBM queries 1, 2, 3, 4, and 9. Since the version of Hexastore used does not perform inference, we were forced to modify queries 4 and 9 such that none was required.

The insertion performance graph is shown in Figure 2. The throughput of Parliament stays fairly stable at approximately 35k statements per second. This throughput is 3 to 7 times larger than that of Hexastore, which starts at approximately 9k statements per second, and declines to less than 5k statements per second as the total number of triples increases. Parliament's throughput results include both persisting the data to disk (the Python version of Hexastore is entirely memory-based) and forward chaining for its RDFS inference capabilities.

Figures 3, 4, 5, 6, and 7 show the relative query performance of Parliament and Hexastore on LUBM queries 1, 2, 3, 4, and 9 respectively.

Queries 1, 3, and 4 produce results where both systems appear to be following the same growth pattern, though Hexastore performs slightly better on queries 1 and 3 and Parliament performs better on query 4. Parliament also demonstrates more variability in the query execution times, which is likely a result of the dependency on the operating system's memory mapping functionality.

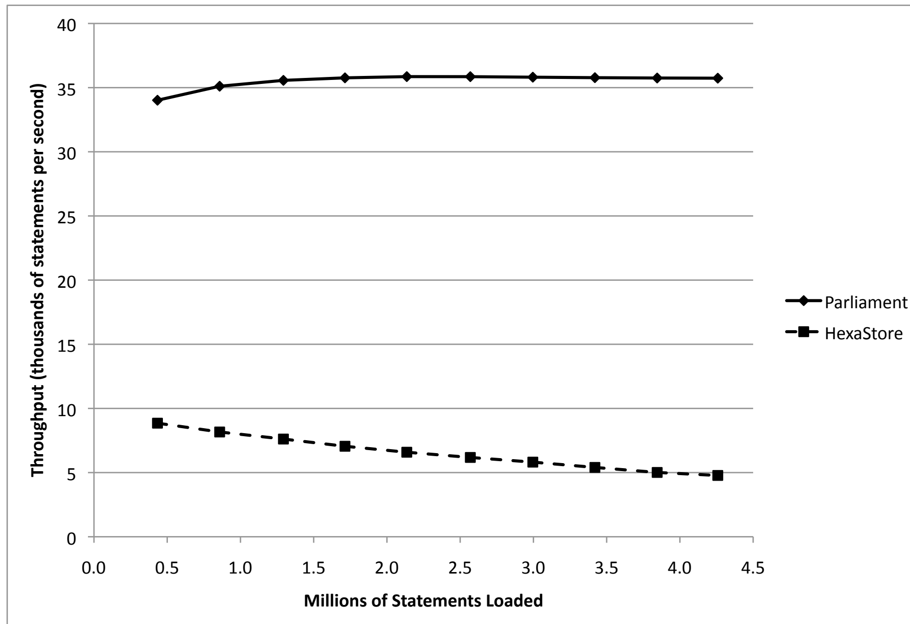


Fig. 2. Insertion Performance

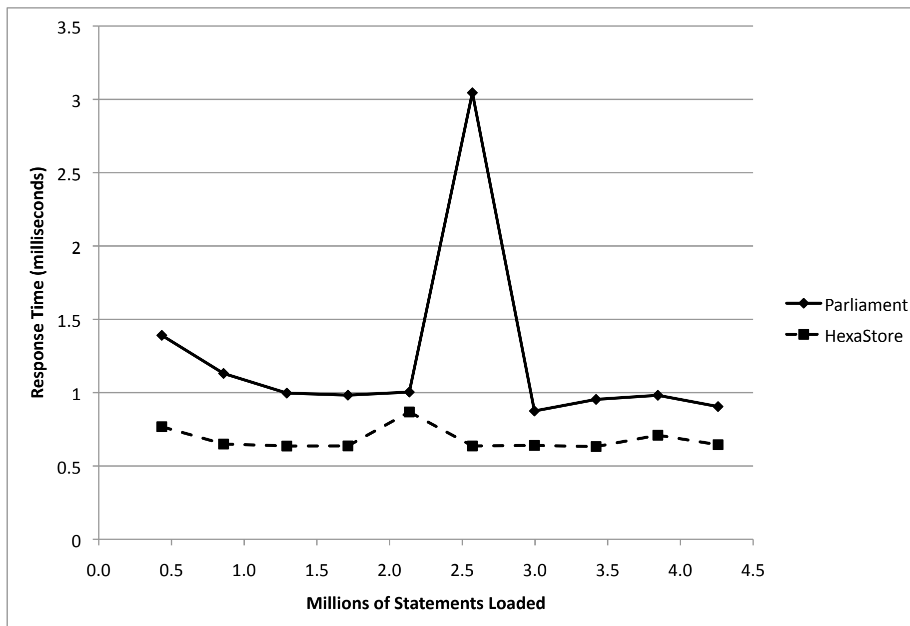


Fig. 3. LUBM Query 1 Response Time

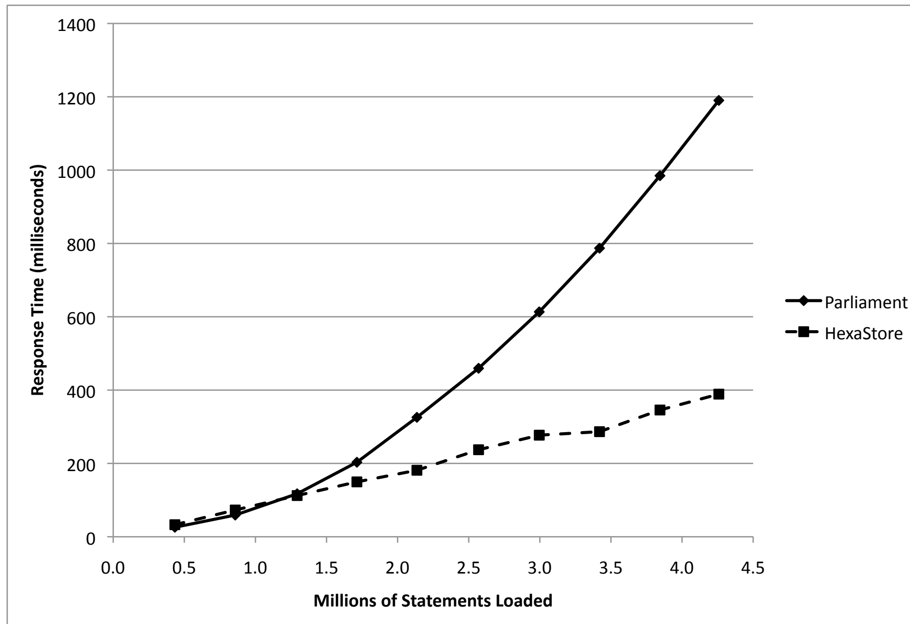


Fig. 4. LUBM Query 2 Response Time

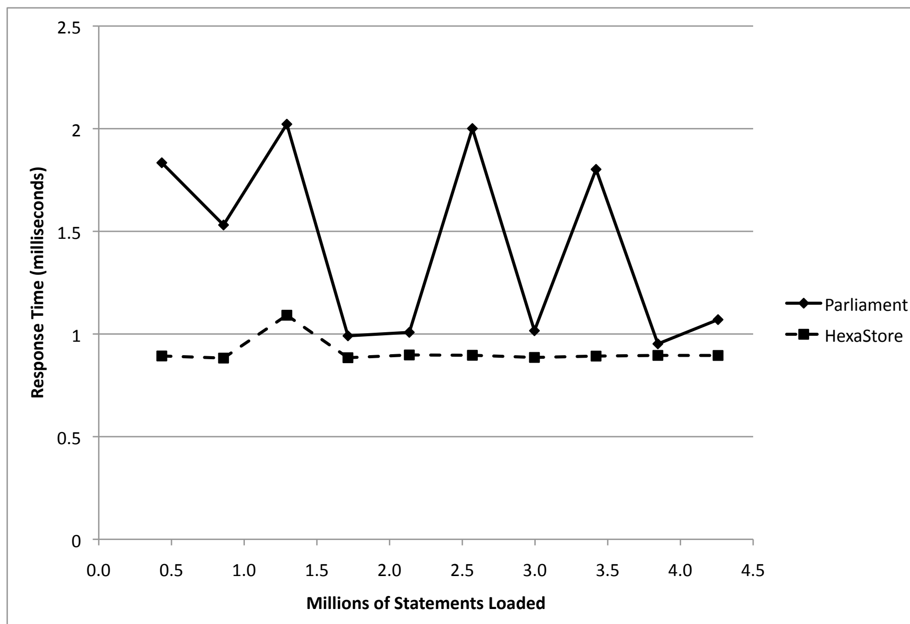


Fig. 5. LUBM Query 3 Response Time

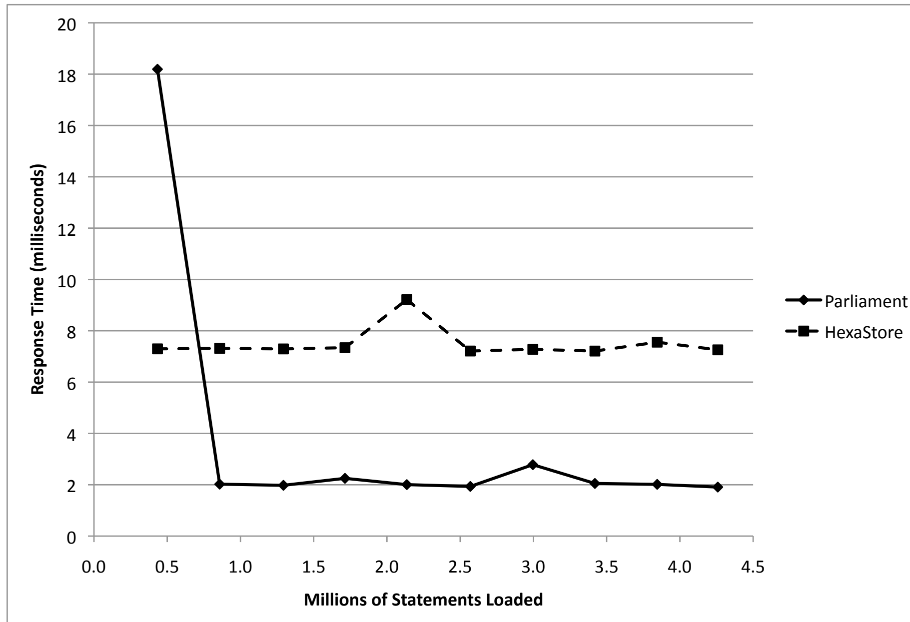


Fig. 6. LUBM Query 4 (modified) Response Time

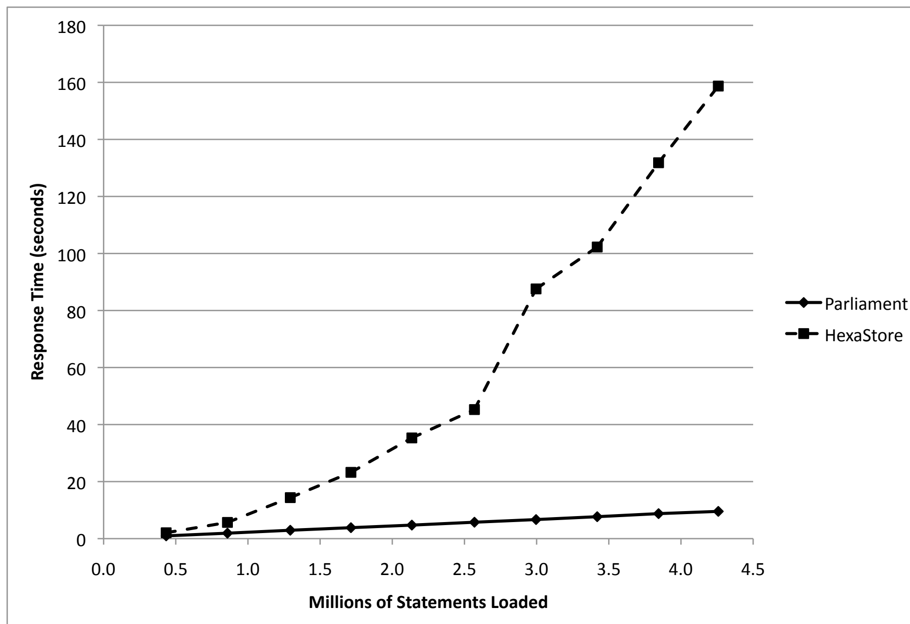


Fig. 7. LUBM Query 9 (modified) Response Time

Queries 2 and 9 show Parliament and Hexastore following different growth curves, with Parliament performing better in query 9 and Hexastore performing better in query 2. This is more likely the result of differing query plans within the two systems than a strength or deficiency of the storage structure, but without insight into the query planner of Hexastore we cannot verify this claim.

Finally, Table 2 shows an estimate of memory used by Hexastore and Parliament with all 4.3M statements loaded. These numbers are as reported by Mac OS X, but as is often the case with virtual memory management, the memory metrics are only useful as course estimates. However, they show what was expected; Parliament's storage scheme requires significantly less storage space.

Table 2. Space Utilization for 4.3M Triples (in GB)

	Hexastore	Parliament
Real Memory	2.02	0.50
Virtual Memory	2.59	1.38
Disk Space	N/A	0.36

Overall, we conclude that Parliament maintains very comparable query performance to Hexastore, while significantly outperforming Hexastore with respect to insertion throughput and required space.

7 Conclusions

In this paper, we have shown the storage and indexing scheme based on linked lists and memory mapping used in Parliament. This scheme is designed to balance insertion performance, query performance, and space usage. We found that while the worst-case performance does not compare favorably with other approaches, average case analysis indicates good performance. Experiments demonstrate that Parliament maintains excellent query performance while significantly increasing insertion throughput and decreasing space requirements compared to Hexastore. Future work will include experiments focusing on different query optimization strategies for Parliament, explanations and analysis of Parliament's internal rule engine, and further optimizations to the storage structure.

References

1. Klyne, G., Carroll, J., eds.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (February 2004) http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.
2. Dean, M., Schrieber, G., eds.: OWL Web Ontology Language Reference. W3C Recommendation (February 2004) http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

3. Harris, S., Shadbolt, N.: Sparql query processing with conventional relational database systems. In: Lecture Notes in Computer Science. Springer (2005) 235–244
4. Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D., Database, J.: Efficient rdf storage and retrieval in jena2. In: EXPLOITING HYPERLINKS 349. (2003) 35–43
5. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 411–422
6. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient sql-based rdf querying scheme. In: VLDB '05: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment (2005) 1216–1227
7. Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In Auer, S., Bizer, C., Müller, C., Zhdanova, A.V., eds.: The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW), September 26-28, 2007, Leipzig, Germany. Volume 113 of LNI., GI (2007) 59–68
8. BBN Technologies: Parliament http://parliament.semwebcentral.org/.
9. Dean, M., Neves, P.: DAML DB http://www.daml.org/2001/09/damldb/.
10. Rohloff, K., Dean, M., Emmons, I., Ryder, D., Sumner, J.: An evaluation of triple-store technologies for large data stores. In: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, Vilamoura, Portugal, Springer (2007) 1105–1114 LNCS 4806.
11. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, New York, NY, USA, ACM (2004) 74–83
12. Broekstra, J., Kampman, A., Harmelen, F.V.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: Lecture notes in computer science. Volume 2342., Springer (2002) 54–68
13. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. **1**(1) (2008) 1008–1019
14. Wood, D., Gearon, P., Adams, T.: Kowari: A platform for semantic web storage and analysis. In: XTech2005: XML, the Web and beyond, Amsterdam (2005)
15. Harth, A., Decker, S.: Optimized index structures for querying rdf from the web. Web Congress, Latin American **0** (2005) 71–80
16. Franz, Inc.: AllegroGraph http://www.franz.com/products/allegrograph/.
17. Kiryakov, A., Ognyanov, D., Manov, D.: Owlim — a pragmatic semantic repository for owl. In: Lecture Notes in Computer Science. Volume 3807/2005. Springer (2005) 182–192
18. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley db. In: ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (1999) 43–43
19. Dean, M.: Toward a science of knowledge base performance analysis. In: Invited Talk, 4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008), Karlsruhe, Germany (October 2008) slide 20 http://asio.bbn.com/2008/10/iswc2008/mdean-ssws-2008-10-27.ppt.
20. Guo, Y., Qasem, A., Pan, Z., Heflin, J.: A requirements driven framework for benchmarking semantic web knowledge base systems. IEEE Transactions on Knowledge and Data Engineering **19**(2) (2007) 297–309