4store: The Design and Implementation of a Clustered RDF Store

Steve Harris, Nick Lamb, and Nigel Shadbolt

Garlik Ltd.
{steve.harris, nick.lamb, nigel.shadbolt}@garlik.com

Abstract. This paper describes the design and implementation of the 4store RDF storage and SPARQL query system with respect to its cluster and query processing design. 4store was originally designed to meet the data needs of Garlik, a UK-based semantic web company. This paper describes the design and performance characteristics of 4store, as well as discussing some of the trade-offs and design decisions. These arose both from immediate business requirements and a desire to engineer a scalable system capable of reuse in a range of experimental contexts where we were looking to explore new business opportunities.

1 Introduction

The need for 4store originated from a fundamental business requirement in Garlik. 4store was designed primarily to provide the backend storage for Garlik's DataPatrol[1], a consumer-facing personal information protection product. This product and its variants now have established user bases of many tens of thousands of individuals.

4store was implemented on a low-cost networked cluster with many tens of servers supporting a 24x7 operation. In addition Garlik has built semantically informed search and harvesting software and used industrial strength language engineering technologies across many millions of people-centric Web pages. Methods have been developed for extracting information from structured and semi structured databases. This information is organised against a lightweight people-centric ontology which, when imported comprises many billions of RDF triples.

Since its initial development 4store has been replaced within Garlik by a new clustered store with even greater scalability and efficiency. The 4store source code has been made available under the GNU General Public Licence version 3 [1]. The ANSI C99 source code and documentation can be found at `http://4store.org/`.

[1] `http://www.garlik.com/`. DataPatrol is an online application that checks databases and internet data sources for indications that personal information has "leaked" into the public domain.

1.1 Original Requirements

Due to the expected data volume that would be stored Garlik decided to aim for the storage of 10^9 quads in a cluster of nine machines, each machine having two processor cores, 4GB of RAM, and two SATA disks – A typical configuration for a commodity server at the time.

Average response time to SPARQL [2] queries was required to be in the low milliseconds range – for the typical queries that would be required to provide the service.

One of the features of the application was that a wide range of heterogeneous data would need to be incorporated into the system. Moreover, new data sets were likely to become available whose form and structure we could not anticipate. Since the exact nature of the data could not be known it was decided that the storage system should not be specialised to any particular RDF schema.

Data updates were intended to be applied in bulk. The RDF store refreshing a weeks worth of data at a time. Time allowed for this import was around eight hours. Due to the volume of updates taking place it was felt that transactions would be required to ensure the integrity of results.

1.2 Current Requirements

As the application grew, the requirements evolved. For the last version of 4store that was used to support DataPatrol, the requirement was to hold 15×10^9 triples in a cluster of nine machines with 8GB of RAM each.

The volume of updates averaged 4×10^9 triples per week, updated in a twelve hour period. However, it was found that explicit transactions were not necessary, so this requirement was dropped.

Garlik had also developed other applications backed by 4store during this period, including QDOS[2] and the QDOS FOAF Index[3], which brought their own requirements. The requirement that had the most impact on the design was for live updates, completing in a predictable time, proportional to the size of the RDF file imported. This required moving from an earlier quad index structure to the one described in section 5.2. It was important that we be able to adapt the design of the store to these new requirements where we were experimenting with new kinds of semantically enabled service.

2 Related Work

There are a number of other RDF storage systems which use cluster based storage, or share some design principles with 4store. These include:

[2] `http://qdos.com/`, an online impact measuring application.

[3] `http://foaf.qdos.com/`, an index of around ten million FOAF files, stored in an instance of 4store.

3store Although 3store [3] is not a cluster-based RDF engine many of the design principles originated in 3store. In particular the method of mapping RDF Resources is inherited more or less directly from 3store.

Bigdata Bigdata [4] is another clustered RDF store. It has very high import performance, but little information about its design is available at this time.

Jena Clustered TDB Jena's Clustered TDB backend [5] has similar design goals to 4store. The paper describes an early prototype, rather than a production environment, but the segmentation and storage are substantially different to those in 4store.

Virtuoso Cluster Edition The Clustered Edition of Virtuoso [6] uses yet another clustering model, based on the Map-Reduce algorithm and bitmap quad indexes.

YARS2 YARS2 [7] uses a very different approach to 4store to achieve heavy utilisation of the cluster, following a more conventional clustering model to spread the load across multiple nodes. It is known to scale to 9×10^9 triples.

3 Architecture

At the time of the initial design it was uneconomic to purchase computers with sufficient main memory to hold an adequate proportion of an RDF index for the projected data size. It was estimated that a reasonable average memory footprint for a quad was in the region of 100 bytes, implying that 93GB of RAM would be required to hold the complete index. As a result it was decided to pursue a clustered storage methodology.

For reasons of cost efficiency it was decided to base the cluster on commodity 64-bit, multicore x86 hardware, running the Linux operating system. At the time this was felt to offer the best price/performance ratio, and offers access to a large number of skilled administrators and systems programmers. The communications were to be provided by Gigabit Ethernet network interface controllers and switches. The choice of this hardware platform suggested the "Shared Nothing" architecture [8] as the most practical design.

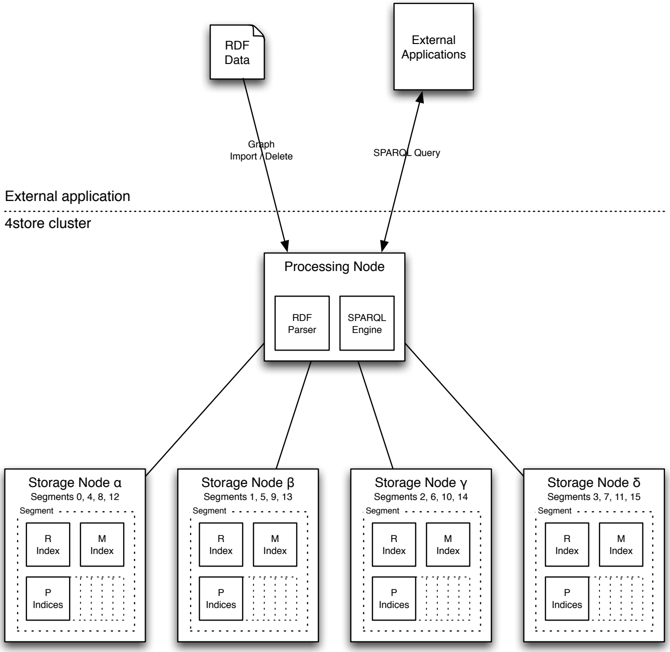
3.1 Cluster Topology

The data is divided among a number of segments (non-overlapping slices of data), with one or more segments on every storage node, as shown in figure 1. These nodes are divided into *Processing* and *Storage* nodes.

It is also possible to run 4store on a single node, running the Processing front-end and one or more Storage back-ends on a single machine. 4store draws little advantage from the proximity however, and the overhead of TCP communications between the Processing and Storage components is still incurred.

3.2 Segmentation

The segmentation model used in 4store is extremely simple. A RID integer (see section 5.1) is calculated for the subject of any given triple. The segment number


RDF Data
External Applications
Graph Import / Delete
SPARQL Query
External application
4store cluster
Processing Node
RDF Parser
SPARQL Engine
Storage Node α
Segments 0, 4, 8, 12
Storage Node β
Segments 1, 5, 9, 13
Storage Node γ
Segments 2, 6, 10, 14
Storage Node δ
Segments 3, 7, 11, 15
Segment
R Index
M Index
P Indices

Dataset	c_v	$\frac{F_r}{F_q}$
BSBM 25MT	2.83×10^{-3}	14.99
FOAF	1.70×10^{-2}	9.09
TIGER/Line	1.59×10^{-3}	5.27

Table 1. Characteristics of various datasets

Drawbacks Synthetic datasets, and potentially real-world ones could skew the distribution of subjects in such as way as to increase the value of c_v, this would have a deleterious effect on the performance and efficiency of the cluster. For example, a large number of triples of the following form would increase c_v substantially:

```
country:US :citizen _:us1 .
country:US :citizen _:us2 .
...
```

Given a triple pattern where the subject is not known, the segmentation algorithm used cannot determine in which segment the matching quad or quads will be found. Because of this it is necessary for the querying process to contact every node in the cluster in order to find matches for this pattern. In practice this limits the application of this algorithm to relatively small clusters. However, in such small clusters it potentially offers an advantage in that the query optimiser is frequently given the choice between a broad shallow query across many nodes, or a narrow deep one against a single node.

When there are two or more potential query operations with similar specificity, but some have constant or known subject values, and some have constant or known object values then the query engine can make the choice between querying all nodes specifying one ore more objects in the *bind* (see section 6.1), or it can specify the subjects and target on the required segments.

The trade-off is that broad bindings consume more resources overall, but complete in a shorter wall-clock time, the IO load being spread across many nodes in the cluster.

3.3 Segment Distribution and Replication

Given a set of nodes N, $\{\alpha, \beta, ...\}$ and a set of segments S, $\{0, 1, ...\}$ the nodes are assigned non-negative integer identifiers, starting from zero. The segments are the divided amongst the nodes, such that the set of segments assigned to node n with r replicas A^{nr} is as below.

$$A^{nr} = \bigcup_{m=0}^{r} R^{nm}$$

$$R^{nm} = \begin{cases} \{s \in S : \ s \bmod |N| = n\} & \text{for } m = 0 \\ \{s \in S : \left(\left(s + \left\lfloor \frac{s}{|N|} \right\rfloor\right) \bmod (|N| - m) + m\right) \bmod |N| = n\} & \text{for } m > 0 \end{cases}$$

For a cluster of 8 nodes, consisting of 32 segments with 2 way replication, the allocations would be as seen in table 2. The aim of this replication method is to ensure that should up to r nodes fail, the increased load is distributed evenly across the remaining nodes.

Node	R^{n0}	R^{n1}	R^{n2}
α	{0, 8, 16, 24}	{7, 14, 21, 28}	{6, 13, 20, 27}
β	{1, 9, 17, 25}	{0, 15, 22, 29}	{7, 14, 21, 28}
γ	{2, 10, 18, 26}	{1, 8, 23, 30}	{0, 15, 22, 29}
δ	{3, 11, 19, 27}	{2, 9, 16, 31}	{1, 8, 23, 30}
ϵ	{4, 12, 20, 28}	{3, 10, 17, 24}	{2, 9, 16, 31}
ζ	{5, 13, 21, 29}	{4, 11, 18, 25}	{3, 10, 17, 24}
η	{6, 14, 22, 30}	{5, 12, 19, 26}	{4, 11, 18, 25}
θ	{7, 15, 23, 31}	{6, 13, 20, 27}	{5, 12, 19, 26}

Table 2. Segment distribution across an eight node cluster with two way replication

4 Inter-Node Communications

Processing nodes communicate with storage nodes via TCP/IP. There is no direct communication between storage nodes. Having discovered the addresses of the storage nodes (see section 4.1) at startup, a processing node asks each node which segments are stored there, and makes one connection per segment on that node. At this point the storage nodes may also optionally (configured at setup time) require an authentication step using a shared secret password to provide some degree of assurance that the processing node is authorised to access the data. The connection is not encrypted because of the likely impact on performance.

Connections between processing nodes and storage nodes are used to send variable sized messages using a type-length-value scheme, each message is either a request or a reply. Communication is always initiated by the processing node sending a message with a request. For most types of request the storage node replies with a message of its own, a few types do not require any reply. In place of the expected reply a storage node can send an error reply, including human readable text if there is a fatal error performing the requested action.

Only one message is sent at a time on any particular connection, but since there is a separate connection for each segment, a request can be sent to all the segments and then all the replies aggregated, meaning the total time to fulfill the request over the whole cluster is limited by the slowest response, rather than the sum of time taken to fulfill the request for each individual segment.

In order to provide replication, requests which write new data to a segment are sent to all replicas of the segment, while to improve performance requests which only read data (e.g. the requests used for the *bind* functions described in section 6.1) make requests to a single replica, and try to choose a replica on a node with least outstanding requests. If a storage node fails while in use, attempts to write data will report errors until it is repaired, but attempts to read data will continue to work normally (but potentially with reduced performance) if at least one replica of each segment is still accessible.

4.1 Discovery

From the outset we wanted processing nodes to be able to discover the storage nodes containing the relevant data without any specific configuration. The processing node needs to identify the complete address (IP address and port number) of a listening TCP socket on each storage node. A "well known port" was not desired, as this would impose a limit of only one instance of the storage node software pre node. DNS Service Discovery [10] seemed well suited to this purpose and, since the storage nodes are on the local network, we used Multicast DNS [11] to enable this without needing a DNS server to be installed or specifically configured for this purpose.

Each storage node advertises a service with the 4store DNS service type (_4store._tcp). To distinguish multiple datasets stored on the same physical nodes, or on different nodes connected to the same network, each dataset has a unique name, which is included in a DNS TXT record, and the total number of segments is also included in the advertisement.

The processing nodes solicit advertisements for the 4store service type and then listen for advertisements. Received advertisements are checked to see that the name matches the desired dataset, and if so the processing node attempts to connect to the advertised address. If the connection fails, other addresses are tried. If after a reasonable time the processing node has not been able to identify and connect to all the storage nodes (or for a processing node which performs only queries, enough nodes to access all the distinct segments) it gives up and reports an error.

5 RDF Representation

5.1 Resources

RIDs RIDs (Resource IDentifiers) are used as a symbol encoding [12] for resource values. RIDs are 64-bit integers which represent URIs, Literals and Blank Nodes using a disjoint value space. The one or two most significant bits of the RID value determine whether the RID encodes a URI, Literals or Blank Node:

MSB1	MSB2	Encodes
0		Literal
1	0	Blank Node
1	1	URI

In the case of URIs and Literals the remainder of the RID is made up of the least significant bits of a UMAC-64 [13] hash of the UTF-8 encoded lexical value of the resource. The cryptographic features of a strongly universal hash are of little relevance, however the collision resistance is desirable. In the case of literals with either a language tag or a datatype, an attribute RID is calculated from the language tag or datatype, stored as the *attr* and additionally exclusive-or'd with UMAC hash of the lexical value. For URIs, Blank Nodes and Plain Literals the value of *attr* is zero.

Blank Nodes are encoded differently. An integer representing the highest blank node identifier is maintained on the storage node(s) holding segment zero. When an importing process wishes to allocate some Blank Node RIDs it requests a block of IDs, represented as (min, max) from segment zero. These IDs are bitwise permuted in such a way as to keep both the MSBs and LSBs of the resulting ID varying frequently, whilst ensuring that distinct input IDs in the range $[0, 2^{62}]$ produce unique output IDs. The variability at both ends of the identifier ensures an even distribution of Blank Nodes across both segments, and in the trie used to store quads (see section 5.2).

Once a collision is detected on any given segment, all future translations from lexical forms to RIDs on that segment must be confirmed by the resource index in the appropriate segment, in order to prevent incorrect results from being returned.

As the RDF Literals and URIs are stored in separate value spaces the point at which collisions are likely depends on the number of unique literals and URIs in a given dataset. The probability of a collision occurring for n values in a space of d possible hash values is given by

$$1 - \prod_{k=1}^{n-1} \left(1 - \frac{k}{d}\right)$$

So, if the number of literals is f_l and the number of URIs is f_u then the overall probability of a collision in a particular segment, where there are s segments is

$$1 - \left(\prod_{k=1}^{\frac{f_l}{s}-1} \left(1 - \frac{k}{2^{63}s}\right) \prod_{k=1}^{\frac{f_u}{s}-1} \left(1 - \frac{k}{2^{62}s}\right) \right)$$

Assuming that the number of resources in each segment is approximately equal, which is likely given the strong universality of the UMAC function [13].

The number of values that can be hashed before we expect to encounter a collision is given by $\sqrt{N}$ for a hash of N values. If we make the assumption[6] that $f_l \approx f_u$, and define f_r as the sum of f_l and f_u then the approximate number of resources that can be imported before we expect to encounter a collision is given by $2\sqrt{2^{62}}$.

[6] Although this situation is not especially likely the statistics for the breakdown into URIs and Literals are not available at this time. Nevertheless, this approximation should still provide a reasonable order-of-magnitude estimate

To know the expected number of quads that can be imported before encountering a collision (e_q) it is necessary to know the ratio of unique quads (f_q) to f_r.

$$e_q = 2^{32} \frac{f_q}{f_r}$$

Some values for $\frac{f_q}{f_r}$ are given in table 1. From this we can estimate that typically 3.9×10^{10} quads of FOAF data could be imported before collision handling would be required. Consequently, it is a worthwhile optimisation to delay handling of collisions until it becomes necessary.

Lexical Value Storage RDF Resources, (URIs, Literals and Blank Nodes) are represented as a 3-tuple of $(rid, attr, lexical\ value)$, and stored in a bucketed, power-of-two sized hash table, shown as the “R Index” in figure 1. The *rid* and *attr* are both RIDs, and the *lexical value* is a text string, encoded in one of a number of ways.

5.2 Quad Storage

In 4store, RDF triples are represented as quads of $(model, subject, predicate, object)$, where a model is somewhat analogous to a SPARQL Graph. The chief differences between a SPARQL Graph and a model are in the handling of empty graphs and the behaviour of the default graph. In 4store, triples assigned to the default graph are placed in a particular model, which is used in query execution against the default graph – when the SPARQL default graph behaviour is enabled.

Although the quads are queried using a flat pattern structure (see section 6.1), the internal structure more closely resembles property tables [12].

Each quad in a particular segment is stored in three indexes. The first two will be described here and the third will be described in section 5.3. The indexes described here are shown as “P Indices” in figure 1.

The P Indices consist of a set of radix tries [14], two for each predicate, using a 4-bit radix. Ordinarily radix tries are at a disadvantage compared to balanced trees as their worst case lookup performance is $\mathcal{O}(k)$, where k is the key length, compared to $\mathcal{O}(\log n)$ for a B-Tree [15]. However the keys in this case have already been mapped to 64-bit integers, so are of finite, short length. Additionally, the integers are already evenly distributed across the space due to the combination of hashing and Blank Node identifier distribution, making the worst case lookup conditions uncommon.

The key for the per-predicate radix tries is the subject or object of the quads to be indexed. The graph and subject/object are stored in a list of rows, pointed to by leaf entry in the radix trie.

Queries with known subjects or objects, but unknown predicates are relatively expensive to execute, as all tries must be consulted to determine if matching subjects or objects appear with that predicate.

5.3 Model Storage

Graphs are indexed using a hash table that points to a list of rows of triples. This index is shown as the "M Index" in figure 1. The function of the graph index is twofold. Primarily it allows queries of the following form to be executed efficiently:

```
SELECT * WHERE { GRAPH <some-graph> { ?s ?p ?o } }
```

A side effect of this is that it allows graph-level deletes to be performed more efficiently, clearing out the Graph index entry and removing matching quads from the appropriate radix trees, this can be performed on each segment independently as all the pertinent information is held locally to the segment.

6 Query Operations

Often the focus on performance of clustered systems is on delegating work to cluster members in order to distribute the work. In large clusters with many parallel task to complete this is a significant efficiency gain, but on the relatively small clusters that 4store is designed to run on this is not always the case.

Running a test on a cluster of five machines, connected by gigabit ethernet on an isolated network, the mean time over one thousand requests for one node to issue an empty request and receive an empty response from one cluster member is 175μs, given an established TCP/IP connection.

For comparison, a join on two 2,000 row binding tables, with one common variable can be completed on the same hardware in 520μs. Consequently, given a pure response-time consideration it's only advantageous to push the join down to cluster members if the tables can be split, sent, joined and returned by the remote cluster members in 520μs. Given the situation that CPUs manufacturers are offering increasing numbers of cores, there is a potential to perform multiple joins simultaneously without incurring network IO overhead. As many such parallel operation can be performed locally, this optimisation looks increasingly unattractive for small operations.

There are however still situations in which performing the joins on cluster members has an overall time advantage. For instance, when the data for both sides of the join is already present on one member, one such situation will be discussed in section 8.2.

Equally a cluster where a very large volume of queries is being performed, saturating the CPU cores on the front-end machines, would benefit from pushing more joins down into the cluster. Another situation would be when the joins are typically performed across very large tables. A join between two binding tables has a complexity around $\mathcal{O}(n \log n)$, so breaking this down into m $\mathcal{O}(\frac{n}{m} \log \frac{n}{m})$ operations is a win in net processor time for sufficiently large n, regardless of the parallelism.

4store has two fundamental distributed query operations, *bind* and *resolve*, these are used to perform the underlying operations for all SPARQL expression evaluation and are described below.

6.1 The Bind Functions

The *bind* functions are used by the query engine when it wishes to produce a binding set for some SPARQL graph pattern. One function takes four multisets of RIDs, and the other a set of quads of RIDs that describe the match to be performed (see below). These arguments are presented to the network distribution algorithm. The network distribution algorithm decides what segment or segments should be consulted to return the complete set of bindings and divides the sets into one set for each segment that is to be consulted.

Once the results have been obtained from the segments of interest the network distributor performs a multiset union on the results and returns this to the query engine.

The primary form of the *bind* function takes four multisets of RIDs, M, S, P and O and a set of flags indicating which columns should be projected. These multisets are matched against the quads held by a particular segment (Q_s), containing a set of quads of RIDs such that quads of the form (m, s, p, o) are selected where:

$$\{(m, s, p, o) \in Q_s : m \in M \vee M = \emptyset, s \in S \vee S = \emptyset, p \in P \vee P = \emptyset, o \in O \vee O = \emptyset\}$$

The resulting multiset of quads is then projected to produce a multiset of n-tuples where n is the cardinality of the projection set.

There is also a second *bind* function, called the *reverse-bind*, for historical reasons. This *reverse-bind* function takes a set of quads R, and returns a multiset of quads quads such that:

$$\{(m, s, p, o) \in Q_s : (m', s', p', o') \in R,\ m = m' \vee m' = \omega, s = s' \vee s' = \omega,$$
$$p = p' \vee p' = \omega, o = o' \vee o' = \omega\}$$

This multiset is then projected as per the main *bind* form. ω in this expression is the “null” value, some nominated RID value which cannot appear in real data.

6.2 The Resolve Function

The *resolve* function is used to map RIDs to attribute RIDs and the lexical value of the input RID.

It takes a set of RIDs and returns a set of tuples of the form (*rid, attr, lexical value*), for a given segment. As in the *bind* case there is a network distributor that is responsible for sending *resolve* requests to the appropriate segments and the result is the union of the returned values from each segment.

7 Query Execution

The 4store query engine is largely based on Relational Algebra. This is historically due to the fact that 4store’s query engine predates the finished SPARQL algebra. Moreover, there is a large body of literature around optimisation that can be applied to relational algebra. Implementations that started after the publication of the SPARQL specification are more likely to use the SPARQL algebra. However many of the observations that follow are likely to be relevant to SPARQL algebra implementations.

4store uses the Rasqal SPARQL parser [16]. Rasqal produces a parse tree representing the underlying structure of the SPARQL expression. 4store walks this tree looking for occurrences of variables, which it records as metadata, and labels blocks of binding patterns with IDs. For example, consider the following query, where the block IDs are show in parentheses:

```
SELECT * WHERE {
 ?x a <Foo> .           (0)
 ?x <has> ?y .          (0)
 OPTIONAL {
  ?y <factor> ?z .      (1)
 }
 { ?y <value> ?v .      (2)
   ?v <label> ?l .      (2)
 } UNION {
   ?y <label> ?l . }    (3)
}
```

Additionally it records which blocks are joined to which other blocks, and by what operation. So, in this case we have:

child	parent	operation
1	0	⟕
2	0	$\bowtie$
3	2	$\cup$

The query executor descends the expression tree, internally joining the expressions to produce a table for each block. Although the projection is performed internally by the *bind* function, in the expressions below it will be shown separately, for clarity:

$$\begin{aligned} b_0 \leftarrow\ & \pi_x \rho_{x/subject}(\text{bind}(\emptyset, \emptyset, \{\texttt{rdf:type}\}, \{\texttt{<Foo>}\})) \bowtie \\ & \pi_{x,y} \rho_{x/subject} \rho_{y/object}(\text{bind}(\emptyset, x, \{\texttt{<has>}\}, \emptyset)) \end{aligned}$$

$$b_1 \leftarrow \pi_{y,z} \rho_{y/subject} \rho_{z/object}(\text{bind}(\emptyset, b_0.y, \{\texttt{<factor>}\}, \emptyset))$$

$$\begin{aligned} b_2 \leftarrow\ & \pi_{y,v} \rho_{y/subject} \rho_{v/object}(\text{bind}(\emptyset, b_0.y, \{\texttt{<value>}\}, \emptyset)) \bowtie \\ & \pi_{v,l} \rho_{v/subject} \rho_{l/object}(\text{bind}(\emptyset, v, \{\texttt{<label>}\}, \emptyset)) \end{aligned}$$

$$b_3 \leftarrow \pi_{y,l} \rho_{y/subject} \rho_{l/object}(\text{bind}(b_0.y, \emptyset, \{\texttt{<label>}\}, \emptyset))$$

The next phase collapses all the UNION expressions. Relational algebra has no equivalent to SPARQL's UNION, but we will use the $\cup$ symbol to represent SPARQL's UNION. UNION blocks are collapsed bottom-up (from highest block ID to lowest), first any FILTER expressions are evaluated, and non-satisfying rows are removed, then the binding tables for co-UNIONs (any blocks related by the $\cup$ operation in the operations table) are concatenated:

$$b_2 \leftarrow b_2 \cup b_3$$

Next the joins across the remaining blocks are performed:

$$b_0 \leftarrow b_0 \bowtie b_2 ⟕ b_1$$

Finally, any remaining FILTERs, ORDER BY, and DISTINCT are applied. FILTERs are left to as late as possible to avoid having to *resolve* more RIDs than required. The presence of LIMIT without ORDER BY, internal complexity limiting and other factors may indicate that not all lexical values for bindings are required. Calls to the *resolve* operation are relatively expensive, as they are more likely to require random access IO in the storage nodes, and can transfer large volumes of data.

It has been our experience that when dealing with queries over large volumes of data using the SPARQL protocol it is often necessary to use the LIMIT keyword, or enable some form of effort limiting, or soft limit to reduce the volume of answers that will be returned. Few HTTP client libraries expose sufficient support for flow control to indicate to the SPARQL server that enough answers have been obtained, or else that the query has been running for too long.

Where possible FILTER expressions are evaluated as results are streamed to the client, with blocks of RIDs from b_0 being resolved at once, based on an heuristic estimation of how many rows of values will be required to satisfy the query.

8 Notable Optimisations

8.1 Join Ordering Optimisation

The primary source of optimisation is the conventional ordering on the joins internal to a block join. We attempt to predict which *bind* will be the most specific, perform that one first, then successively apply the same specificity estimate, given the values from the binding table at that point.

Earlier versions of 4store had access to comprehensive quad histogram data. The current version only has access to predicate frequency information in order to perform this heuristic evaluation. This is due to the structure of the radix trie indexes, an earlier index form providing a highly efficient way to obtain occurrence histograms as a side-effect of its design.

8.2 Common Subject Optimisation

Where two or more binds of the form

$$\mathrm{bind}(M^1, S, \{p_1\}, \{o_1\}), \mathrm{bind}(M^2, S, \{p_2\}, \{o_2\}), \ldots \mathrm{bind}(M^b, S, \{p_b\}, \{o_b\})$$

are encountered, where $|M^n| \leq 1$ this pair of binds can be transformed to a single *reverse-bind*:

$$\text{reverse-bind}\left(\bigcup_{n=1}^{b}\{(m, s, p_n, o_n) : s \in S, m \in M^n\}\right)$$

Where M^n is treated as $\{\omega\}$ if $|M^n| = 0$.

This has a twofold advantage. Firstly it reduces the number of network operations, and secondly (and more importantly) it breaks the join operation across the storage nodes, due to the way quads are segmented.

In the example of the following query:

```
SELECT ?x WHERE {
  ?x <givenName> "John" ;
     <familyName> "Smith" .
}
```

Any pair of triples matching this pattern will fall into the same segment, as they must share a subject RID, so when the storage node performs the join it will only have to consider one segment at a time, eliminating unnecessary bindings for ?x before they reach the front-end, and thus reducing the search space.

8.3 Cardinality Reduction

If the REDUCED or DISTINCT keywords are used then the cardinality of *bind* functions need not be preserved. Because of this, the presence of one of these keywords is passed down to the storage node, and it takes any time-efficient measures that are available to reduce the cardinality of the result set. For example, by removing adjacent identical rows, and also by use of index structures. Given a *bind* of the form $\pi_{predicate}(\mathrm{bind}(\emptyset, \emptyset, x, \emptyset))$ it is sufficient to to consult the list of predicate indices, to return the RID values for the matching predicates present in the segment. If DISTINCT is specified then the front-end still needs to perform the DISTINCT operation, but typically the size of the result set returned will be greatly reduced.

Similar optimisations are available for bindings to all subjects, objects, models and resources.

8.4 Unreferenced Variables

It is often necessary to place variables in graph patterns where the value of the variable is not required. Consider a query to find all the people that have some employer:

```
SELECT DISTINCT ?x WHERE {
  ?x a <Person> ;
     <has> ?employer .
}
```

By inspection it is possible to ascertain that bindings for ?employer are not required, we simply have to ensure that such a binding exists. Given this, the bind call for the second triple pattern above can be reduced to:

$$\pi_x \rho_{x/subject}(\text{bind}(\emptyset, x, \{\texttt{<has>}\}, \emptyset))$$

Without the DISTINCT or REDUCED keywords the engine is still required to preserve the cardinality of ?x, but in either case we can avoid holding a column of bindings in the binding table.

9 Future Work

9.1 Updates

Currently in 4store, as of the writing of this article the only update operations that are supported in the front-end are deleting an entire model, and adding triples to a model. The nascent SPARQL/Update specification will require fine-grained updates.

9.2 Full Text Indexing

Currently 4store has no index that can efficiently address full text searches. This is supported in SPARQL via the *regex* function. More sophisticated full-text searching is commonly offered as a non-standard extension. This is an area for future work.

10 Conclusion

In this paper we have described in detail the architectural design principals and methods of implementation for key aspects of our clustered RDF store. We discussed the merits and demerits of a number of fundamental features of the store such as its segmentation model. We have detailed a number of optimization strategies and we have also reviewed the performance characteristics and trade-offs that informed our design.

We believe that there is much to be gained by sharing effective design patterns, best practice and hard won insights amongst our emerging community.

References

1. Free Software Foundation: GNU General Public License. http://www.gnu.org/licenses/gpl.txt (June 2007)
2. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/ (2005)
3. Harris, S., Shadbolt, N.: SPARQL query processing with conventional relational database systems. In Dean, M., Guo, Y., Jun, W., Kaschek, R., Krishnaswamy, S., Pan, Z., Sheng, Q.Z., eds.: WISE Workshops. Volume 3807 of Lecture Notes in Computer Science., Springer (2005) 235–244 http://eprints.ecs.soton.ac.uk/11126/1/harris-ssws05.pdf.
4. Personick, M.: Bigdata: Approaching web scale for the semantic web. http://www.bigdata.com/whitepapers/bigdata_whitepaper_07-08-2009.pdf (2009)
5. Owens, A., Seaborne, A., Gibbins, N., mc schraefel: Clustered TDB: A clustered triple store for Jena. In: WWW2009. (November 2008) http://eprints.ecs.soton.ac.uk/16974/.
6. Erling, O., Mikhailov, I.: Towards web scale RDF. In: Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledge. (October 2008) http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticles/VOSArticleWebScaleRDF.pdf.
7. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A federated repository for querying graph structured data from the web. In Aberer, K., Choi, K.S., Noy, N.F., Allemang, D., Lee, K.I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P., eds.: ISWC/ASWC. Volume 4825 of Lecture Notes in Computer Science., Springer (2007) 211–224 http://www.deri.ie/fileadmin/documents/DERI-TR-2007-04-20.pdf.
8. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9**(1) (1986) 4–9 http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf.
9. Bizer, C., Schultz, A.: Berlin SPARQL benchmark (BSBM) specification - v2.0. http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/
10. Cheshire, S.: DNS service discovery (DNS-SD). http://www.dns-sd.org/ (2009)
11. Cheshire, S.: Multicast DNS. http://www.multicastdns.org/ (2006)
12. Wilkinson, K.: Jena property table implementation. Technical report, Hewlett-Packard Labs (October 2006) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.530.
13. Krovetz, T.: UMAC: Message authentication code using universal hashing. IETF **RFC 4418** (March 2006) http://tools.ietf.org/html/rfc4418.
14. Morrison, D.R.: PATRICIA - practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM **15(4)** (October 1968) 514–534 http://portal.acm.org/citation.cfm?doid=321479.321481.
15. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes. Technical report, Boeing Scientific Research Laboratories (July 1970) http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/Bayer-McCreight.pdf.
16. Beckett, D.: Rasqal RDF Query Library. http://librdf.org/rasqal/ (2005)