A Distributed In-Memory Redis-Backed Triple Store Database

Drew Shafer
me@drewshafer.com
May 9, 2013

Abstract
This paper discusses the architecture, data partitioning scheme, and query execution strategies of a Redis-backed SPARQL Triple-Store endpoint. Redis is an advanced Key-Value store with properties that potentially make it very suitable as a backend storage engine for a triple-store database. As with other in-memory data stores, Redis offers simple deployment and high single-node performance. Features of Redis that set it apart include a rich set of primitive datatypes (String, List, Hash, Set, Sorted Set) with associated specialized commands, optional on-disk persistence with adjustable robustness, and an embedded Lua interpreter that can be used to implement arbitrarily complex behavior.

The implementation described in this paper uses the Redis Set datatype to store and index triple data, and has a pseudo-MapReduce architecture. Incoming SPARQL queries are transformed into a physical execution plan that uses Lua scripting to execute a single map phase across all nodes in parallel. The map results are merged and joined by a single reduce node implemented in Java. Performance measurements are presented and discussed for the BSBM 10M and 100M datasets.

Introduction
Triple Store databases are critical as the backing technology of the Semantic Web, and accordingly the description and evaluation of various triple store implementations has provided a rich vein of fodder for recent publication [1][2][3][4][5][6][7][8][9][10]. Due to the large size of interesting triple-store datasets and benchmarks, most implementations are understandably built on spinning-disk-backed storage and optimize for the physical limitations of the medium. Meanwhile, computer hardware continues to become more affordable, and main memory, while still approximately two orders of magnitude more expensive than magnetic disk storage[11], is growing ever larger and has reached the point where software engineers can access large datasets in their entirety without the need to page to disk.

This paper describes a completely in-memory triple-store base on Redis[12], a proven and popular advanced key-value store.

Background on Redis
Redis (REmote DIctionary Server[13]) is an in-memory datastore that is widely deployed as a key-value store for caching-oriented tasks, similar to other systems like memcached[14]. Redis performs ably in this specialized role, but is not-so-secretly capable of much more. While other in-memory key-value databases exclusively store strings, Redis has the ability to store and manipulate higher-order data structures at each key. In addition to strings, Redis supports Hash

Tables, Lists, Sets, and Sorted Sets as primitive objects, and provides efficient implementations of common operations on these structures. As such, Redis is sometimes referred to as a “Data Structure Server”, and is useful as a performant and robust base from which to experiment with software systems that require data persistence and manipulation. Of special importance to this project, Redis includes an in-process Lua scripting host, which allows arbitrary manipulation of datasets to be executed on the Redis server before returning results to the requester.

System Architecture

The triple store runs on a commodity cluster with a single master and 0 or more nodes. Each node hosts a combination of Alias Databases (mapping URIs to shorter character sequences for efficient storage) and Triple Databases (storing the familiar Subject-Predicate-Object data that gives Triple databases their name). Redis is single-threaded, so multiple Redis servers, or “Shards”, are started on each node to efficiently utilize CPU resources. The typical advice when deploying Redis is to run one copy of the server for each physical core available. However, due to the design of this datastore, Alias DBs and Triple DBs are not accessed simultaneously during queries. Therefore, each node is typically configured to host twice as many Redis shards as cores available, split evenly between Alias and Triple databases, to get maximum performance during each phase of query execution. Additionally, the Master node can be configured with both Alias and Triple DBs if there is available memory that is not needed by the Reducer.

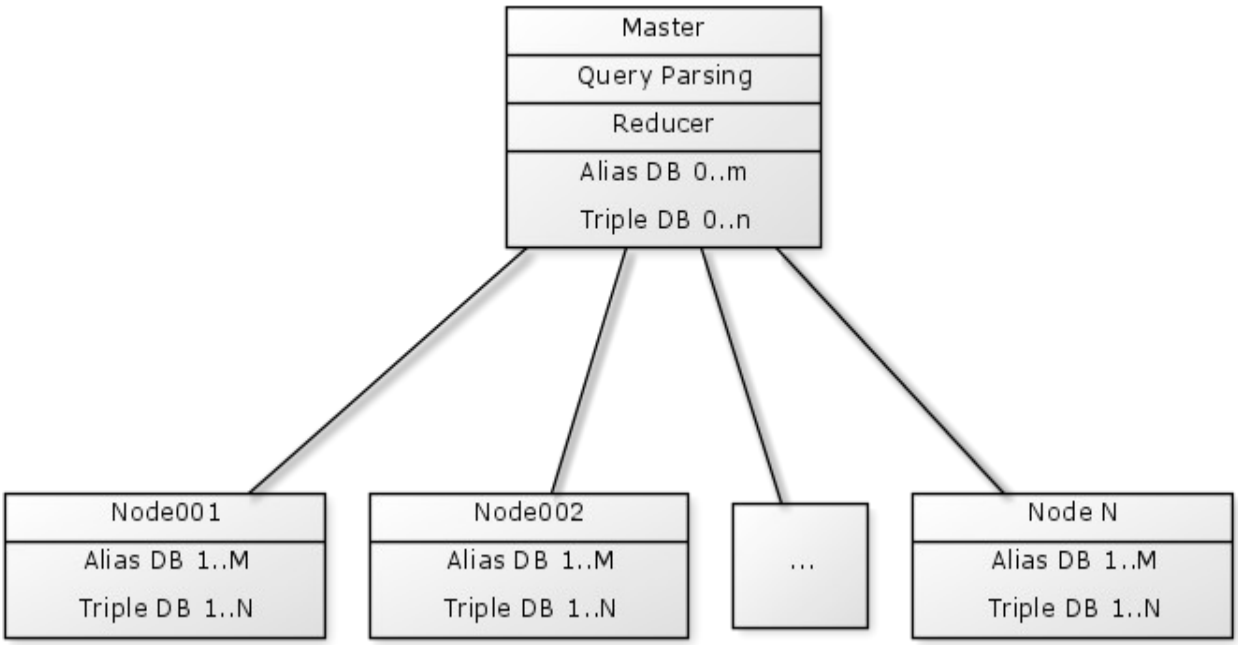


Figure 1. High-level cluster architecture. Typically both M and N are chosen to be equal to the number of cores available on each node.

In an effort to keep development and deployment simple, no software components are executed on the Nodes apart from the Redis servers.

Because the working dataset is stored entirely in memory, the Disk I/O performance of the cluster (traditionally a bottleneck in distributed databases) is not a primary consideration in the

architecture of the system. Rather, system design must strike a balance between proportionally high-memory or proportionally high-CPU nodes. Choosing nodes with relatively high memory will allow the cluster to process larger datasets on fewer nodes, but each query will take longer to execute. Conversely, nodes with proportionally high CPU will execute queries more quickly, but more nodes will be required to contain an equivalent dataset.

URI Aliasing

The most common standard used for exchanging and importing RDF Triple data is the Named Triple (*.nt) file format. Named Triple files are are extremely simple to parse, but this simplicity comes at the expense of being almost comically inefficient in the number of bytes required to store each triple. Since available memory is the limiting factor in the size of the dataset that this server can query, each URI is assigned a much shorter alias as it is imported, and it is this alias that is actually stored and indexed in the Triple Shards. The mapping of Aliases->URIs is held in a separate set of Alias shards hosted on the same nodes as Triple Shards. Each URI is assigned to a particular Alias shard according to a hash function, and then sent to that shard for the actual integer alias to be assigned. In the global context, the alias value is prefixed by the shard index, so looking up a URI by its aliased value is a simple matter of splitting off the prefix and sending the remainder of the alias to the shard index encoded by the prefix. For example, if the URI “bsbm-inst:ProductType3” hashed to the 13th Alias Shard, and it was the 53,419th URI to be aliased on that shard, the corresponding global alias would be “12:53418” (Both the Shard Prefix and the Alias value are 0-indexed in this implementation). Aliases could be made shorter still by using a more compact encoding for the counters, but simple ASCII-Integer encoding was used here because it can be simply and efficiently implemented in Lua.

Triple Partitioning

The primary goal of the triple distribution algorithm for the datastore was to maximize the potential for joins that can be completely evaluated on the individual nodes, before any data is returned to the master/reducer. SPARQL queries are generally assumed to be highly reliant on Subject-Subject joins, with Subject-Object joins being almost as important[15]. Other work analyzing real-world datasets and query mixes have shown this assumption to be valid[16]. Knowing this, the triples are partitioned into Redis Shards by hashing the Subject URI, guaranteeing that all Subject-Subject joins can be completed on the individual nodes. This specific category of subject-subject “Star Joins” (so called because each solution takes the form of several edges emanating from a common vertex when visualized as a graph) are very efficiently executed by the Redis-SPARQL triple store.

SPARQL Query	Graph Representation of the BGP

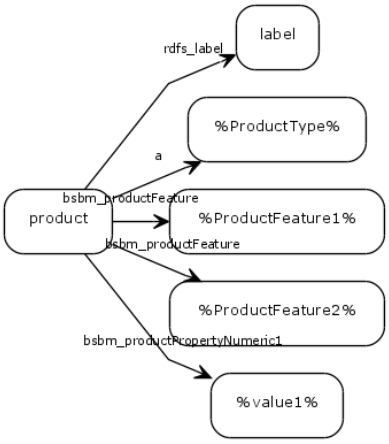
SELECT DISTINCT ?product ?label
WHERE {
?product rdfs:label ?label .
?product a %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productFeature %ProductFeature2% .
?product bsbm:productPropertyNumeric1 ?value1 .
FILTER (?value1 > %x%)
}
ORDER BY ?label
LIMIT 10
	

Table 1. The BSBM Explore Query #1 is an example of the Subject-Subject Star Joins that are common in SPARQL NOTE: In this and other query examples, the URI Namespace-prefix section is omitted for brevity.

As Subject-Object joins are also important, for Triples where the Object is a URI (instead of a RDF Literal) the system additionally hashes the Object and inserts a copy of the triple into the corresponding database shard. This allows a single Object-Subject join between exactly two triple patterns to be completely evaluated on the distributed nodes as well.

SPARQL Query	Graph Representation of the BGP

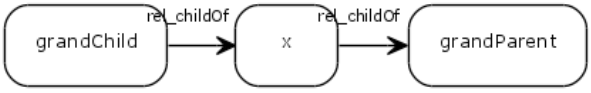
SELECT DISTINCT ?grandchild ?grandparent
WHERE {
?grandchild rel:childOf ?x .
?x rel:childOf ?grandparent
}
	

Table 2. A Single Object->Subject Join query and corresponding graph representation.

Literal Representation

RDF Literals consist of a String value, plus an optional DataType URI and optional Language tag. Unlike URI values, RDF Literals are commonly referenced in filter and sort expressions. In order for these expressions to be evaluated on the nodes where possible, then, the actual literal value must be available to the node - not just a global alias. To accomplish this, the entire RDF literal is encoded in a JSON string and sent to the Triple shard as a part of inserting the corresponding triple value. In the individual Triple DB, the Literal value is assigned an alias that is used only on that node.

RDF Literal Value	"2000-06-28"^^<http://www.w3.org/2001/XMLSchema#date>
JSON Representation	{“v”:”2000-06-28”,”t”:”xsi:date”}
Node Alias	#381782

Table 3. A RDF Literal as represented in a) Named Triple format for import into the triple store, b) JSON for exchange between the Master and the Triple DB on a Node, c) the Literal Alias used internally by the node.

Triple Storage and Indexing

After having global URI aliases and local Literal aliases (if any) assigned, each triple is encoded as a JSON array for storage in the Triple DB Shard on a node.

Subject Alias	Predicate Alias	Object Alias	JSON String Encoding
15:29741	0:7721	#401988	[“15:29741”,”0:7721”,”#401988”]

Table 4. Representation of a Triple value within a node

Once the JSON triple encoding is calculated, the string is indexed by its Subject, Predicate, and Object URIs to allow for efficient retrieval during queries. This indexing leverages the Set datatype provided by Redis, and set keys are simply constructed as “S:[URI Alias]”, “P:[URI Alias]”, or “O:[URI Alias]”, for Subject, Predicate, and Object URIs, respectively.

When querying RDF Triples, there are six discrete query patterns that can be constructed (S, P, O, SP, PO, SO). While this limited number of potential indexing problems might suggest that it is reasonable to maintain six separate indexes on the triple data in order to efficiently service any query that might be encountered[17], this strategy comes at a high cost of storage overhead, particularly because the compound index keys created by the SP, PO, and SO combinations create many times more index entries than are created when indexing by S, P, and O individually. When operating under in-memory constraints as this system must, this extra storage overhead becomes quite onerous, reducing the number of triples that a given memory size can accommodate by an order of magnitude.

Because the system indexes triples individually by adding them to sets S, P, and O, compound queries by any two elements can be expressed as the Set Intersection of the two corresponding indices. Fortunately, Redis comes with a built-in, highly optimized command to return the intersection of two or more sets called SINTER that executes in O(N*M) time, where N is the length of the shortest set and M is the number of sets being intersected. Since triple queries can result in the intersection of at most 2 sets, this simplifies to O(N).

The JSON encoding for triples is a relatively heavyweight representation, and indeed the overall memory usage of the system can be further optimized by creating Node-local integer aliases of each triple, and inserting this alias into each of the [S,P,O] sets instead of the triple itself. Converting unique triples to integers for storage is not a novel concept, and has been described

as a part of the 4Store database[18], and probably elsewhere as well. Preliminary experiments along these lines showed that approximately a 30% reduction in memory footprint is possible, at a cost of 50% increased query execution time due to the additional lookups required. While the faster, less memory-efficient JSON representation was chosen for this project, the option of creating Node-Local aliases should be considered for memory-constrained systems.

Query Analysis and Physical Plan Optimization

For each logical component of a SPARQL query, the Query Analyzer/Optimizer module in the Master node constructs a physical query plan from a combination of Redis built-in commands, Lua scripts that can be executed in parallel on the individual nodes, and sequential routines (written in Java) which must be executed on the Master node. Amdahl's Law[19] shows that the lower bound of execution time for a program containing both parallel and sequential elements is the execution time of the sequential elements. Applying this maxim, the overriding rubric of the Query Analyzer/Optimizer is to move as much computation as possible into the Nodes and off of the Master. If possible, computation should be done using only Redis' builtin commands, which are *fast*. Next, computation is implemented in Lua script running within the Redis process, which is not fast (relatively), but can be executed in parallel. Only when it is impossible to complete a computation on the remote nodes is the computation assigned to the Java reducer stage. Of course, some query components (notably Order-By) are implemented by a combination of both distributed computation on the nodes and single-node computation by the Java reducer.

A detailed exploration of all aspects of the Analyzer/Optimizer is beyond the scope of this paper. A sample before-and-after for one BSBM query is presented here as an example of the high-level output of the Query Analyzer/Optimizer. One aspect to note is that the five non-optional triples in the BGP Pattern from the SPARQL query have been split into two separate BGP patterns that get evaluated during the distributed Redis-Lua computation (BGP_74 and BGP_73), which are then joined in java during the Reduce phase. This is necessary because the strategy used to shard triples across the cluster cannot assure that the join can be completed on a single shard.

The following page shows an example of the result of the executing the query optimizer/analyzer on a BSBM query.

Example SPARQL Query and Resulting Physical Execution Plan

SPARQL Query:

```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1 ?rating2 ?rating3 ?rating4
WHERE {
        ?review bsbm:reviewFor <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer257/Product12578> .
        ?review dc:title ?title .
        ?review rev:text ?text .
        FILTER langMatches( lang(?text), "EN" )
        ?review bsbm:reviewDate ?reviewDate .
        ?review rev:reviewer ?reviewer .
        ?reviewer foaf:name ?reviewerName .
        OPTIONAL { ?review bsbm:rating1 ?rating1 . }
        OPTIONAL { ?review bsbm:rating2 ?rating2 . }
        OPTIONAL { ?review bsbm:rating3 ?rating3 . }
        OPTIONAL { ?review bsbm:rating4 ?rating4 . }
}
ORDER BY DESC(?reviewDate)
LIMIT 20
```

SPARQL-Redis Physical Execution Plan:

```
ReducePhaseSlice {
 start : 0,
 length: 20,
 parent: ReducePhaseProject {
  projectVars: {'?title','?text','?reviewDate','?reviewer','?reviewerName','?rating1','?rating2','?rating3','?rating4'}
  parent : ReducePhaseOrder{
   expr  : ?reviewDate,
   parent: ReducePhaseJoin {
     left : BGP_74 {
      pattern : { {'?review','7:36568','?reviewer'},{'?reviewer','9:36567','?reviewerName'} } }
     right: MapPhaseLeftJoin {
      left : MapPhaseLeftJoin {
       left : MapPhaseLeftJoin {
        left : MapPhaseLeftJoin {
         left : BGP_73 {
          filter: {(vars['?text']['l'] == (string.lower('EN')))},
          pattern : {
                    { {'?review','2:35896','14:981'},
                      {'?review','16:36689','?title'},
                      {'?review','16:36694','?text'},
                      {'?review','31:36338','?reviewDate'},
                      {'?review','7:36568','?reviewer'} } }

         right: BGP_78 {
          pattern : { {'?review','28:36243','?rating4'} } }
        }
        right: BGP_77 {
         pattern : { {'?review','31:36340','?rating3'} } }
       }
       right: BGP_76 {
        pattern : { {'?review','7:36569','?rating2'} } }
      }
      right: BGP_75 {
       pattern : { {'?review','19:36328','?rating1'} } }
    }
   }
  }}
```

Description of redis-sparql-server

The primary component of this system is a single Java executable called redis-sparql-server.jar that runs on the Master node. This executable has three separate modes for deploying the cluster, importing triple data, and executing queries. Each mode requires a common configuration file that defines the cluster layout. This configuration file has a simple, terse JSON format that defines how many of which type of Redis shard to deploy to each slave.

A sample configuration file for a 4-node cluster is shown:

```
{
   "redisCmd": "/sparql/redis-server",
   "aliasBasePort": 49000,
   "tripleBasePort": 50000,
   "nodes":{
      "master":[2,2],
      "node001":[4,4],
      "node002":[4,4],
      "node003":[4,4],
   },
}
```

The first entry (redisCmd) specifies that a redis shard can be launched by executing "/sparql/redis-server" on any node. For each hostname defined the "nodes" list, a 2-element array is given that dictates the number of Alias shards (first entry) and Triple shards (second entry) that reside on that node. Instead of specifying the listening port for each Alias and Triple shard, by convention Alias shards start at aliasBasePort and Triple shards start at tripleBasePort on each node, and increment as more shards are added to the node. These port values default to 50720 for aliasBasePort and 50820 for tripleBasePort, and do not need to be explicitly specified in the configuration file unless those defaults cause conflicts, or if a single node will need to hold more than 100 Alias shards (in which case the port numbers would start to conflict with the Triple shards on the same node).

Mode 1: Cluster Configuration and Setup

In a cluster of 16 quad-core nodes, this system runs 120 separate Redis instances - clearly too many for the user to launch manually. When given the --start-redis option on the command-line, sparql-redis-server reads the cluster configuration JSON and uses SSH to start the specified redis-server processes listening on the correct ports on each node. This mode is dependent on the cluster being configured for password-free SSH logins from the master to any slave node.

Mode 2: Triple Data Import

When launched with the --populate <FileName.nt> option, redis-sparql-server connects to the Redis shard databases, then reads triples from the specified file and inserts them into the database. In order to achieve reasonable performance during data import, the system aggregates input into blocks of hundreds or thousands of triples as it reads them from disk. Once a block is ready complete, it is dispatched to a background thread to be inserted into the Alias and Triple database shards. Each block goes through an insertion pipeline similar to that shown in Figure 2.

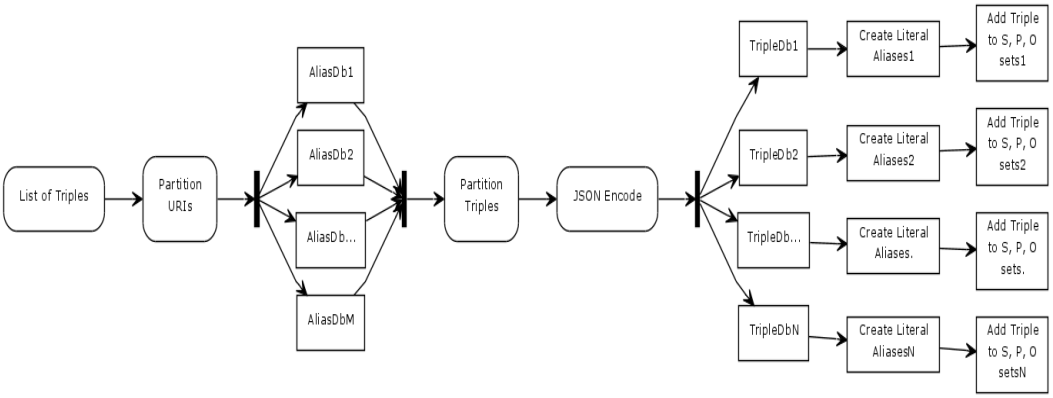


Figure 2. Triple Batch Import Execution Pipeline

Typically data is split into several files, and this multi-threaded import process runs multiple nodes simultaneously, with each node importing a subset of the full dataset (this is the only mode where redis-sparql-server executes on any of the slave nodes). Because of the multi-node and multi-threaded inserts, the actual order that triples are aliased and inserted is not deterministic. This means that the same URI or RDF Literal will likely be assigned a different Alias each time a dataset is imported. Data integrity is guaranteed, however, because each insert operation to a Redis backend shard executes atomically, so the system is always assured that there will be either 0 or 1 Alias mappings for a given URI.

Although data is split for insertion by several nodes, it should be noted that this spit is a “dumb” operation - it has no knowledge of the hash scheme used to partition URIs and Triples among the databases, so each batch of triples will likely require network communications with all Alias and Triple shards in the cluster. Due to this, the scaling potential from adding more simultaneous nodes to the insert process is limited, because as more inserting nodes are added, more contention for atomic inserts to the Redis backends can be expected.

Mode 3: SPARQL HTTP Endpoint

With the --listen [--port <n>] command-line option(s), redis-sparql-server starts a HTTP SPARQL Endpoint on the specified port with an embedded Jetty[24] web server and waits for incoming connections. When an incoming SPARQL query is received, it is parsed into a logical representation by the Apache JENA/ARQ library[25] and run through the query execution pipeline shown in Figure 3.

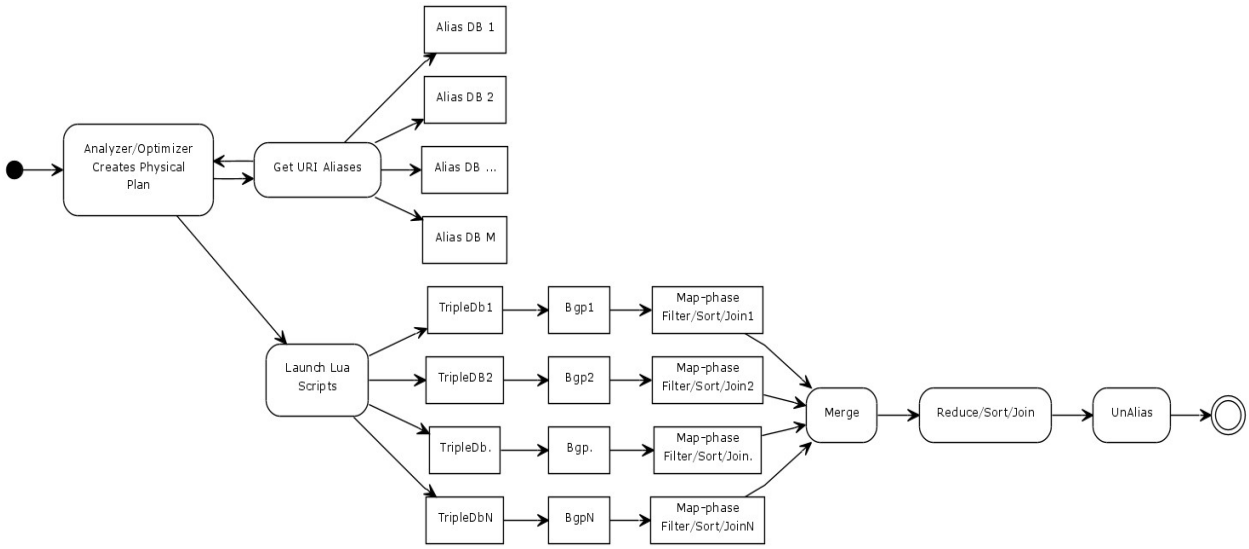


Figure 3. Query Evaluation Pipeline

Once the Query Evaluation Pipeline is complete, the JENA library is again used to translate the result into RDF-XML format for return to the requester.

As a final note, the different modes of redis-sparql-server are not mutually exclusive. A single invocation from the command-line can specify all three modes, and the system will start the Redis instances, load them with data, and finally start the SPARQL endpoint once data import is complete.

Performance Evaluation

Methodology: Clusters of various sizes and machine specifications were provisioned on Amazon's EC2 cloud infrastructure using MIT's excellent Starcluster tool[20], and the Berlin SPARQL Benchmark (BSBM)[26] "Explore" query mix was benchmarked for 10M and 100M triple datasets (Not all cluster configurations were able to fit the 100M dataset). For each configuration, Load times and selected Average Query Execution Times were measured.

The primary EC2 Instance Type used was m2.2xlarge (4 cores, 34 GB RAM)[21]. Originally plans were made to get measurements on a cluster of m1.large machines, but this was abandoned for two reasons. First, the m1.large machines have relatively little RAM (7.5 GB), which limited the cluster to small datasets. The second and much more compelling reason, however, was one of cost to run the cluster. In order to minimize EC2 expenses, experiments were primarily run using spot instance bidding[22]. This strategy exposed a very curious aspect of the Amazon EC2 market: some instance types are severely overvalued on the spot market, and others are reliably undervalued. The m1.large instance type falls into the overvalued category - while "On Demand" instances are charged at $0.24/hr, the spot instance price is routinely over $5.00/hr for multiple hours per day. Meanwhile, the much more powerful m2.2xlarge instance type fluctuates between $0.07 - $0.10 on the spot instance market, despite having a "regular" on-demand price of $0.82/hr. Because of these irregularities in the Spot Instance market, it was much more economical to purchase time on the more powerful m2.2xlarge instances.

Query Execution Time

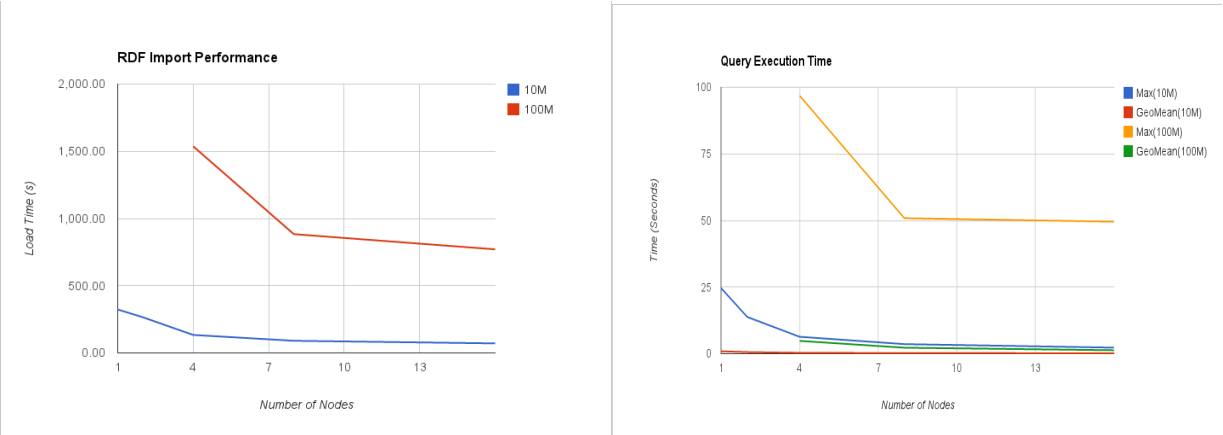


Figure 4. Elapsed Import Time and Query Execution Time for various cluster sizes with 10M and 100M datasets

Full performance results, including logs of individual query measurements, are available at the project Github page[23].

Of the 11 different queries in the BSBM Explore usecase, 7 of them (Queries 1, 2, 3, 4, 9, 11, 12) are executed very quickly by redis-sparql-server. Of the remaining low-performing queries (Queries 5, 7, 8, 10), Query 5 stands out as the only query that appears to hit a scaling performance limit for cluster sizes larger than 8 nodes:

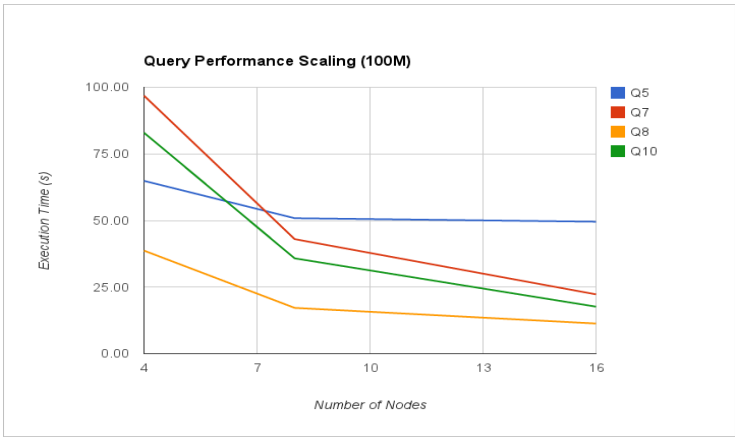


Figure 5. Query Performance Scaling for Queries 5, 7, 8, and 10. Note that Query 5 is unique in failing to scale from 8 to 16 nodes.

The BSBM specification describes Query 5 as a search that aims to find products with similar features to one a customer has already identified:

```
SELECT DISTINCT ?product ?productLabel
WHERE {
        ?product rdfs:label ?productLabel .
FILTER (%ProductXYZ% != ?product)
        %ProductXYZ% bsbm:productFeature ?prodFeature .
```

```
        ?product bsbm:productFeature ?prodFeature .
        %ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
        ?product bsbm:productPropertyNumeric1 ?simProperty1 .
        FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1 > (?origProperty1 – 120))
        %ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
        ?product bsbm:productPropertyNumeric2 ?simProperty2 .
        FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2 > (?origProperty2 – 170))
}
ORDER BY ?productLabel
LIMIT 5
```

When running the query, %ProductXYZ% is randomly chosen to be different product URIs present in the dataset.

Examining the the logs produced by redis-sparql-server for single executions of Query 5 in 8-node and 16-node configurations, it may be possible to get a better picture of execution performance:

8 Nodes	16 Nodes
Query parsing/optimizing: 0.004 seconds Lua script execution: 46.971 seconds Merge: 0.03 seconds ReducePhaseJoin: 7.079 seconds ReducePhaseFilter: 0.183 seconds ReducePhaseFilter: 0.03 seconds	Query parsing/optimizing: 0.004 seconds Lua script execution: 40.082 seconds Merge: 0.035 seconds ReducePhaseJoin: 5.23 seconds ReducePhaseFilter: 0.223 seconds ReducePhaseFilter: 0.037 seconds

Table 6. Selected log traces of Query 5 execution on 8-node and 16-node clusters when %ProductXYZ% = bsbm-inst:dataFromProducer5044/Product255158. Note that the execution profile is wildly different than the previous table regarding proportion of time spent in the merge phase.

The first sample query log shows the majority of processing time spent during Lua Script Execution on the various nodes. This should bode well for scaling performance as more nodes are added, because the each node will have less data so the Lua scripts should execute faster. However, other samples of Query 5 execution show significantly different execution profiles:

8 Nodes	16 Nodes
Query parsing/optimizing: 0.004 seconds Lua script execution: 40.149 seconds Merge: 0.03 seconds ReducePhaseJoin: 20.556 seconds ReducePhaseFilter: 0.043 seconds ReducePhaseFilter: 0.006 seconds	Query parsing/optimizing: 0.004 seconds Lua script execution: 49.365 seconds Merge: 0.035 seconds ReducePhaseJoin: 4.218 seconds ReducePhaseFilter: 0.057 seconds ReducePhaseFilter: 0.008 seconds

Table 6. Selected log traces of Query 5 execution on 8-node and 16-node clusters when %ProductXYZ% = bsbm-inst:dataFromProducer5017/Product253855. Note that unlike the previous example, the execution profiles of the two samples are dissimilar.

These wildly different execution profiles for the same basic query structure indicate that certain query configurations partition the triples among the nodes in a way that is particularly bad for query #5 when it is executed with certain parameters..

Memory Utilization

For each 1M triples of the BSBM dataset, the cluster requires approximately 430 MB RAM for the Triple DB and 50 MB RAM of Alias DB space. As a rough gauge, this represents approximately 2x the size of the equivalent Named-Triple format file on disk. In addition to the space needed to store triples, additional memory must be left available on the nodes for the Lua processing scripts - this requirement was not rigorously quantified in time for this report, but is estimated to require about 10-15% additional memory above that needed to store the triples.

The single reducer process was observed to require about 3 GB RAM for the 10M dataset, and up to 20 GB RAM for the 100M dataset. This is obviously not a trivial figure, and could be a limiting factor to scaling this strategy to dataset sizes of over 1Bln triples on commodity hardware (though as a counterpoint to concerns about lack of available RAM, Amazon offers the cr1.8xlarge EC2 instance type with 244GB RAM at an on-demand price of $3.50/hr, which happens to be their most economical option when evaluated on the basis of $/GB RAM).

Based on experience with the 100M dataset, it is estimated that querying the 1000M BSBM dataset would require a cluster of between 4 and 6 cr1.8xlarge EC2 nodes. Due to the memory required by the master node, the 1000M dataset probably would not fit on any number of m2.2xlarge nodes, though one potential avenue would be to have a single cr1.8xlarge master coupled with a large number (~40) of m2.2xlarge slave nodes.

Discussion

This project achieved its design goal of building a distributed triple store where as much of the distributed processing as possible is pushed down into Lua scripts executing on unmodified Redis servers. While this strategy may simplify deployment of the cluster, this exercise has shown that there are significant downsides as well:

- The **Lua scripts account for the largest portion of query execution** time for slow queries, even at cluster sizes of 8 and 16 nodes. This is suggestive of a disparity in processing efficiency for routines implemented in Lua vs. those implemented in Java.

This is not to say that Lua is a slow language for the tasks it is designed for, but is instead an indicator that...

- Lua is **not well suited for large data processing**. Lua is primarily designed to be simple to implement into C/C++ programs as an extension language[27], and lacks features and optimizations that would help it efficiently operate on large collections of data.
- **The Redis Lua implementation is purposefully restricted** in order to guarantee deterministic execution of scripts. Among other things, this means that no timekeeping functions are available, making it difficult to profile Lua script execution within the Redis server.
- **Redis does not include many common Lua libraries**, and it is not possible to load libraries dynamically, meaning that including new Lua libraries would require forking the Redis server code. Most significantly, the Lua Regular Expression library is not included in Redis, which precludes the possibility of executing SPARQL Regex filters in Lua.

None of these points are meant to be taken as a criticism of Lua in general. The language is a valuable tool to have available for the class of problems it was designed to solve - namely, Lua's strength is as a tool for expressing connections between existing software components in a way that creates new, useful features that that the original developer might not have thought of, or that might be deemed too specialized to implement as a core feature of the program that hosts the Lua script.

With this lesson in mind, a follow-on Triple Store endpoint based on Redis should make more judicious use of Lua scripting within Redis. Some uses of Lua in this system were successful, most notably the scripts for inserting URIs into the Alias database and returning aliases. Before moving the Alias logic into Lua, the system had to insert URIs into the Alias database one-by-one, incurring multiple network round-trips for each URI. By implementing more logic in Lua, the system today sends hundreds or thousands of URIs to the Alias database at once, and receives all of the new aliases back in a single network round-trip.

In final analysis, the primary benefit of using Redis as the backing store for a Triple Store comes from the strategy of indexing triples by inserting them into Redis Sets, then using the SINTER command to efficiently retrieve all triples that match any two constraints of Subject/Predicate/Object without needing to create secondary join indices on the possible combinations.

Future Work

The findings from this project suggest several avenues for further work. The primary focus will be to reduce or eliminate the use of Lua in performance-critical paths during query evaluation, replacing it with Java code running on the nodes. Apart from this goal, there are several open questions following the implementation of this system:

What precisely is happening on BSBM Query #5? Is this a pathological case for one query/dataset, or (seemingly more likely) is it indicative of a need for a more sophisticated data partitioning strategy?

How does the system perform in the context of other common Triple-Store benchmarks, such as DBPedia?

Can the system be adapted to run on the upcoming release of redis-cluster? If using redis-cluster is not feasible, how can the system be adjusted to handle failed nodes?

Can data be more efficiently partitioned such that multiple nodes can participate in data import with little or no contention for the backend databases?

What would be the performance increase (if any) vs. Memory Overhead tradeoff from using additional compound indices instead of leveraging the Redis SINTER command to match triple patterns?

Source Code Repository

The source code, test scripts, and execution logs for this project are all available at https://github.com/dshafer/sparql-redis-server

Bibliography

1. Rohloff, Kurt, et al. "An evaluation of triple-store technologies for large data stores." *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*. Springer Berlin Heidelberg, 2007.
2. Cheng, Long, et al. "Runtime Characterization of Triple Stores." *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*. IEEE, 2012.
3. Haque, Albert, and Lynette Perkins. "Distributed RDF Triple Store Using HBase and Hive." (2012).
4. Cheng, Long, et al. "Runtime Characterisation of Triple Stores: An Initial In-vestigation." http://issc.exordo.com/data/paper_attachments/31/final_draft/Runtime_Characterisation_of_Triple_Stores_An_Initial_Investigation.pdf
5. Bail, Samantha, et al. "FishMark: A Linked Data Application Benchmark." *Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+ HPCSW 2012)*.
6. Oliver, Ian, Jukka Honkola, and Ora Lassila. "APPARATUS, METHOD AND COMPUTER PROGRAM PRODUCT FOR PROCESSING RESOURCE DESCRIPTION FRAMEWORK STATEMENTS." U.S. Patent No. 20,120,303,678. 29 Nov. 2012.
7. Fletcher, George HL, et al. "SAINT-DB: A structural indexing based triple store." http://www.win.tue.nl/~yluo/files/poster_ictopen2012.pdf
8. Voigt, Martin, Annett Mitschick, and Jonas Schulz. "Yet Another Triple Store Benchmark? Practical Experiences with Real-World Data." *Semantic Digital Archives* (2012): 85.
9. Punnoose, Roshan, Adina Crainiceanu, and David Rapp. "Rya: a scalable RDF triple store for the clouds." *Proceedings of the 1st International Workshop on Cloud Intelligence*. ACM, 2012.
10. Xiufeng, Liu, Christian Thomsen, and Torben Bach Pedersen. "3XL: An Efficient DBMS-based Triple-store." 284-288. http://vbn.aau.dk/ws/files/66882593/3xldemo.pdf
11. At the time of writing, 8 GB of consumer-class DDR3 memory costs approximately $60, roughly the same price as a 750 GB hard drive.
12. http://redis.io
13. http://redis.io/topics/faq
14. http://memcached.org/
15. Huang, Jiewen, Daniel J. Abadi, and Kun Ren. "Scalable SPARQL querying of large RDF graphs." *Proceedings of the VLDB Endowment* 4.11 (2011): 1123-1134.

16. Gallego, Mario Arias, et al. "An empirical study of real-world SPARQL queries."*1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India*. 2011.
17. Weiss, Cathrin, Panagiotis Karras, and Abraham Bernstein. "Hexastore: sextuple indexing for semantic web data management." *Proceedings of the VLDB Endowment* 1.1 (2008): 1008-1019.
18. Harris, Steve, Nick Lamb, and Nigel Shadbolt. "4store: The design and implementation of a clustered RDF store." *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. 2009.
19. Amdahl, Gene M. "Validity of the single processor approach to achieving large scale computing capabilities." *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967.
20. http://star.mit.edu/cluster/.
21. http://www.ec2instances.info/. I recommend this website when weighing the specifications and per-hour cost of various EC2 instance types, as Amazon's own documentation seems to be purposefully engineered to make such comparisons nearly impossible.
22. http://aws.amazon.com/ec2/spot-instances/
23. https://github.com/dshafer/sparql-redis-server
24. http://www.eclipse.org/jetty/
25. https://jena.apache.org/
26. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
27. http://www.lua.org/about.html