4 Distributed store for tensor-based RDF Graph Search

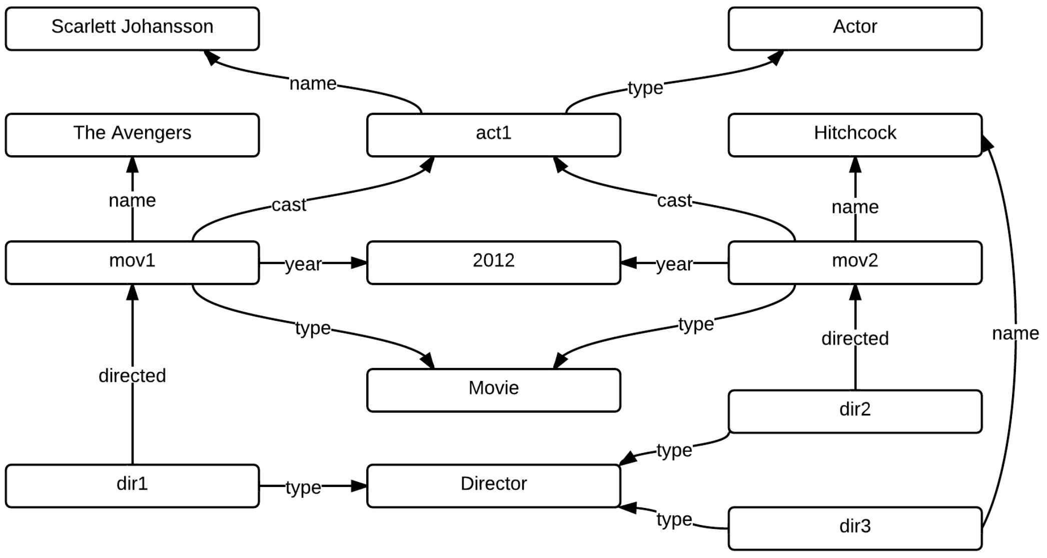
For smaller RDF Databases, a single-machine implementation of the tensor-based proposal could keep all indexes and at least one instance of the sparse matrix in memory for quicker retrieval. However, single-machine storage will invariably become a bottleneck on both performance and reliability of a system. Furthermore, the set of queries proposed access the indexes and the sparse matrix from different perspectives, and enabling uniform performance to all types of queries requires different indexing policies implemented on those data stores, increasing the amount of memory needed. At a certain point, the implementation of an index on a single machine will become unfeasible.

The tensor-based proposal is well suited to be used in combination with innovative Cloud Computing techniques, such as MapReduce over work queues and database sharding over non-relational databases that can be useful to scale out results. The need for increased scale becomes clear when taking into consideration the growth of the Web of Data.

A distributed system should allow the use of an arbitrary number of database nodes on the system as well as indexing policies for the indexes in use and sparse matrices stored. On top of such storage, an API to execute queries should also be provided (so that knowledge of the exact network topology could be hidden from client applications). The main goal of this dissertation is to propose and design such a scalable RDF graph storage system, optimized for keyword search using the tensor-based approach.

The tasks necessary for the process of storing and querying an RDF database fall on three general steps. First is the distributed indexing of nodes, full-paths, and templates as defined by the tensor-based proposal. Second is the assembly of the sharded sparse matrix. Finally, an interfacing layer over the data network must be provided to execute the queries proposed on the tensor-based proposal. To accommodate the data generated on the general steps, a distributed database is needed. The proposed architecture for this database is to use an in-memory key-value datastore, sharded among several nodes.

4.1.Step one: distributed indexing of nodes, full-paths, and templates



id	node
1	mov1
2	The Avengers
3	mov2
4	Hitchcock
5	2012
6	Movie
7	act1
8	dir1
9	dir2
10	Director
11	dir3
12	Actor
13	Scarlett Johansson

id	template
1	*-type-*
2	*-directed-*-year-*
3	*-directed-*-name-*
4	*-directed-*-cast-*-type-*
5	*-directed-*-cast-*-name-*
6	*-directed-*-type-*
7	*-name-*

id	full-path
1	dir1-type-Director
2	dir1-directed-mov1-year-2012
3	dir1-directed-mov1-name-The Avengers
4	dir1-directed-mov1-cast-act1-type-Actor
5	dir1-directed-mov1-cast-act1-name-Scarlett Johansson
6	dir1-directed-mov1-type-Movie
7	dir2-directed-mov2-year-2012
8	dir2-directed-mov2-cast-act1-type-Actor
9	dir2-directed-mov2-cast-act1-name-Scarlett Johansson
10	dir2-directed-mov2-name-Hitchcock
11	dir2-directed-mov2-type-Movie
12	dir2-type-Director
13	dir3-name-Hitchcock
14	dir3-type-Director

Figure 9 - Step one illustration, from RDF to indexes

The raw information of the RDF Graph is extracted from either a SPARQL end point or from a file of tuples. For simplicity, let's assume that the tuples are stored in a file using the N-Triples format, where each line represents one tuple.

Indexing all nodes is a very straightforward task: for each tuple, both the subject and the object are added to a set. Each element of the set can be later taken out and numbered. A simplified initial algorithm to store all indexes could be:

```
def findNodes(tuples)
  nodesSet = new DB::Set();
  for each tuple in tuples do  # iterates over rdf triples
    s, p, o = parseTuple(tuple)  # extracts subject, predicate and object
    nodesSet.add(s)
    nodesSet.add(o)
  end
  nodes = new DB::Array()
  n = 0
  for each node in nodesSet do
    nodes[n] = node  # uses array indexes as node indexes
    n++
  end
  DB.persist(nodes)
end
```

Algorithm 1 - First approach to build nodes index

The next task is the construction of the full-path index. For this, all full-paths of the graph will have to be walked, either using a breadth-first search BFS or a depth-first search DFS. A good starting point can be each source of the graph, and from this source find all paths leading to a sink. The sources and sinks can be found while listing the nodes index with some changes to the algorithm:

```
def findNodesSourcesAndSinks(tuples)
  nodesSet = new DB::Set()
  sources = new DB::Set()
  for each tuple in tuples do  # iterates over rdf triples
    s, p, o = parseInput(tuple)  # extracts subject, predicate and object
    if s not in nodesSet do  # if subject is a new node...
      nodesSet.add(s)
      sources.add(s)  # ... it is a source until proven wrong
    end
    nodesSet.add(o)
    sources.rem(o)  # an object's node is not a source
  end
  nodes = new DB::Array()
  n = 0
  for each node in nodesSet
    nodes[n] = node  # uses array indexes as node indexes
    n++
  end
  DB.persist(nodes)
  DB.persist(sources)
end
```

Algorithm 2 - Second approach to build nodes index

This algorithm can be modified to allow it to be used in a distributed fashion without optimizations - this is a challenging problem on its own and falls outside the scope of this dissertation. The second loop to enumerate the nodes can be incorporated in the first loop that processes the tuples. Assuming that the database architecture will provide a shared data structured for this step, the algorithm can use a hash collection to both check for duplicates and enumerate them in a single operation. As will be seen later, an index to facilitate the retrieval of all predicate-objects originating from a node will speed-up the initial sparse matrix construction, so the algorithm is also modified accordingly:

```
def findNodesSourcesAndSinks(tuples)
  nodes = new DB::TwoWayArray()
  predicateObjects = new DB::Array()
  sources = new DB::Set()
  for each tuple in tuples do  # iterates over rdf triples
    s, p, o = parseTuple(tuple)  # extracts subject, predicate and object
    if s not in nodes do
      nodeIndex = nodes.add(s)
      predicateObjects[nodeIndex] = new DB::Set()
      sources.add(s)  # ... it is a source until proven wrong
    end
    nodes.add(o)
    nodeIndex = nodes.index(s)
    predicateObjects[nodeIndex].add((p, o))  # store node connections
    sources.rem(o)  # an object's node is not a source
  end
end
```

Algorithm 3 - Distributed approach to build nodes index

Finally, a very straightforward way to map the distributed execution of this algorithm is to split the input file list of tuples in *j* jobs on a queue, from where workers can freely pool from (without interdependency):

```
def BRGS.mapTupleParsing(file, j)  # splits the file in j blocks
  k = files.lines.length / j  # how many lines a block has
  for i = 0 to j do
    startLine = i * k  # first block line
    endLine = (i + 1) * k - 1  # last block line
    job = new MR::Job(
      findNodesSourcesAndSinks,
      file.lines[startLine..endLine])  # maps a new job with k lines
    MR.enqueue(job)
  end
end
```

Algorithm 4 - Mapping the distribution of tuple parsing to find nodes

Once all sources are found, all full-paths of the graph can be found by walking the graph with a depth-first search (DFS) that starts from each source. It can be intuitively deduced that two sources will produce two completely different sets of full-paths, since no path starting with one source can reach the other source, by definition. To fit this DFS walk in the MapReduce model, we propose a straightforward mapping for this algorithm that creates a job to do a full depth-first walk for each existing source. A distributed algorithm that implements a DFS to find all full-paths starting at a source can be:

```
def pathDive(path, marked)
  node = path.last  # starts from end of path walked so far
  marked.add(node)  # prevents node from being rewalked
  predicateObjects = DB.loadPredicateObjects(node)  # node connections
  if predicateObjects is empty do  # node has no connections...
    DB.indexFullPath(path)  # ... path ended, store full-path
  else:
    for each (p, o) in predicateObjects do
      path.push(p, o)  # grow path walked so far this way...
      pathDive(path, marked)  # ... and recursively keep walking it
      path.pop(2)  # get back to point before going deeper
    end
  end
  marked.rem(node)
end
```

Algorithm 5 - Recursive DFS to walk all full-paths

The mapping of the sources to distributed execution of the DFS can be simply described as:

```
def mapSourceDFS()
  sources = DB.loadSources()
  for each source in sources do
    path = new DB::Set()  # creates a new empty path to start...
    marked = new DB::Set()  # ... and with no marked nodes yet
    path.push(source)
    job = new MR::Job(pathDive, path, marked)
    MR.enqueue(job)  # maps a job to each source
  end
end
```

Algorithm 6 - Mapping the distribution of sources to DFSs

As noted by the algorithms presented so far, several methods must be available in the implementation of the database network (DB methods) and some from the MapReduce framework (API methods). To facilitate referencing in the rest of the text, we baptize each requirement with a method or structure name, shown in parenthesis (using "::" to designate structures and "." to designate methods). In what follows, we describe the methods that are needed to create data structures, as well as the functionalities that they need to provide. The methods that handle data structure creation have the following requirements:

- Create a database backed collection of unique elements, similar to a Set, with two-way indexing, i.e., assigns an index to new elements and can retrieve elements by their index, or given an element find its index **(DB::TwoWayArray)**;

- Create a database backed array of elements with one way indexing, i.e., can store and retrieve elements given any arbitrary index **(DB::Array)**;
- Create a database backed collections of unique elements without indexing and can randomly access any of them **(DB::Set)**;

The methods that handle the functionality needed to manipulate the data structures described above have the following requirements:

- Load a predicateObjects array given a node **(DB.loadPredicateObjects)**;
- Store the full-path and its template serialized as strings on DB::TwoWayArray collections **(DB.indexFullPath)**;
- Load the graph sources' DB.Set **(DB.loadSources)**;
- Add a unit of work to a job queue **(MR.enqueue)**.

All these need to be present in any implementation of the proposed architecture. This design and the complete relationship to the tools chosen to support our implementation will be described in greater detail in the next chapter.

4.2.Step two: sparse matrix assembly, sharding and storage

During this step, the indexes built on step one will provide the necessary information to assemble and persist the sparse matrix. Each position i, j on the sparse matrix depends only on the data from the indexes. We chose to compute all positions for any given full-path, which already fits a MapReduce parallel computation of this step. An algorithm that populates a line of the sparse matrix is:

```
def storeFullPathPositions(fullPath)
  template = DB.templateFromFullPath(fullPath)
  fpCount = (fullPath.length / 2) + 1  # how many nodes in full-path
  fpPos = 1
  while fpPos < fpCount do
    i = (2 * fpPos) - 1  # odd full-path values, i.e., nodes
    node = fullPath[i]
    DB.storePosition(
        node, fullPath,  # sparse-matrix position to store
        fpPos, fpCount, template)  # position, count and template tuple
    fpPos++
  end
end
```

Algorithm 7 - Store all positions of a full-path

Mapping must make sure that each full-path was de-serialized into a data structured useful for the store algorithm. In this case, the algorithm uses an array, so the distribution could be:

```
def mapStoreFullPathPositions()
  fullPaths = DB.loadFullPaths()
  for each fullPath in fullPaths do
    fullPathArray = new DB::Array(fullPath)
    job = new MR::Job(storeFullPositions, fullPathArray)
    MR.enqueue(job)  # maps a job to each full-path
  end
end
```

Algorithm 8 - Mapping full-paths to store sparse matrix positions

As it is clear from the tensor-based proposed queries, multiple sharding techniques to store the sparse matrix are desired to trade in increased redundancy of data for faster retrieval later. This redundancy must be encapsulated on the DB.storePosition method. Similar to the last step, we summarize the required functionalities for this step and name each as a function:

- Given a node and a full-path, create a tuple of values as described in Section 3.3 - i.e., a tuple with the node position in the full-path, how many nodes exist in this full-path and the index of the full-paths' template - and store in the sparse matrix applying the active sharding policies **(DB.storePosition)**;
- List all stored full-paths **(DB.loadFullPaths)**;
- Find a template given a full-path **(DB.templateFromFullPath)**.

4.3.Step three: distributed graph queries

The tensor-based proposal describes several queries for retrieval and procedures for maintenance. These queries are key to the architecture design and sharding choices made. In this section, we describe the query requirements, while the support for them is described in Section 4.5.

4.3.1.Node Query, Path Query and Final Node Query

A Node Query finds all full-paths containing given nodes that were selected by keyword. As seen on the tensor-based proposal, this translates to extracting the sparse matrix column *j,* where *j* is the given node index. As such, this query presents two requirements for the underlying database architecture. First, select a node index from keywords inputted by the user, with margin for small input errors. Second, the retrieval of a full column of the sparse matrix should be an optimized function given the database sharding. A simplified high-level description of the Node Query could be:

```
def BRGS.nodeQuery(keywords)
    node = SE.selectNode(keywords)  # finds a node by keyword search
    return DB.sparseMatrixColumn(node)  # gets corresponding column
end
```

Algorithm 9 - High-level description of a Node Query

Very similar to the Node Query are the Path and Final Node Queries. The Path Query differs by the fact that selecting a full-path query by keyword is not as user-friendly as selecting a node, so the query assumes that a path was selected by other means. Once selected, the requirement of retrieval is similar, except that the Path Query will require the underlying database to provide a full row retrieval. The Final Node Query is a special case of the Node Query, which the purpose is to find all full-paths where the selected node is at the end of it. Instead of retrieving the whole column of the sparse matrix, only those positions where the tuple indicates that the node position is the same as the full-path nodes count, i.e., where given *{i,j}* = *{o,l,t}*, *o* and *l* are equal. This could generate another requirement for the database, but will be left out of the design scope on this dissertation.

To execute these queries, the functionalities needed are:

- Select one node that best matches the given keywords provided by the user **(SE.selectNode)**;
- Retrieve a full column of the sparse matrix **(DB.sparseMatrixColumn)**;
- Retrieving a full row of the sparse matrix **(DB.sparseMatrixRow)**.

4.3.2.Path Intersection Query

The Path Intersection Query verifies if two given full-paths have intersections. On the sparse matrix, this is translated by retrieving the two corresponding rows of those full-paths and finding if they have intersecting rows. This query could use the previously defined DB.sparseMatrixRow method for quick retrieval of the rows and subsequent intersection, as proposed on the tensor-based approach:

```
def BRGS.pathIntersectionQuery(path1, path2)
  path1Row = DB.sparseMatrixRow(path1)
  path2Row = DB.sparseMatrixRow(path2)
  intersectingNodes = new DB::Set()
  for each node in path1Row do
    if node in path2Row do  # intersects path1 and path2 nodes
      intersectingNodes.add(node)
    end
  end
  return intersectingNodes
end
```

Algorithm 10 - Naïve intersection of the two paths

This intersection can be further optimized on the implementation to take into account the fact that the rows are numbered and have gaps. Those optimizations won't be described here. The functionalities needed for this query have already been listed on previous steps or queries.

4.3.3.Path Intersection Retrieval Query

Another query described on the tensor-based proposal is the Path Intersection Retrieval Query. The purpose of this query is to find all full-paths that intersect a given full-path. The query, as described on the proposal works, uses a previously defined query, the Path Query, to get all nodes of a query. However, previous requirements described the need to find nodes back and forth from their values and indexes. Given that full-paths are string serializations of nodes, the Path Query won't be necessary, and the nodes can be found directly from the full-path de-serialization. With the nodes selected, the query then joins the results of a Node Query on each node of the full-path, representing all full-paths intersecting the given full-path:

```
def BRGS.pathIntersectionRetrievalQuery(path)
  paths = new DB::Set()
  for each node in path do
    intersectingPaths = BRGS.nodeQuery(node)  # full-paths with this node
    paths.add(intersectingPaths)
  end
  return paths
end
```

Algorithm 11 - Path Intersection Retrieval Query

The functionalities needed for this query have already been listed on previous steps or queries.

4.3.4.Path Cutting Query

This final retrieval query from the tensor-based proposal is a two-direction operation to return the part of a given full path that either starts or ends on a given node. Similar to the Final Node Query, this is based on extracting the given full-path corresponding row and returning the nodes which node position values on the tuples are greater or lesser than (depending on the direction chosen for the operation) the selected node position, i.e., given the node i_n and path j_p, the query returns all i where $\{i,j_p\} = \{o_i,l,t\}$, have o_i greater or lesser than o_n in $\{i_n,j_p\} = \{o_i,l,t\}$. The algorithm for this is can be simply designed as:

```
def BRGS.pathCuttingQuery(path, node, direction)
  cells = BRGS.pathQuery(path)  # sparse matrix cells of full-path line
  refPosition = cells[node].position  # position in full-path of node
  selectNodes = new DB::Set()
  for each cell in cells do  # cell has node and full-path position
    if direction == 'gt' do  # from start cutting direction
      if cell.position > refPosition do
        selectedNodes.add(cell.node)
      end
    else  # to end cutting direction
      if cell.position < refPosition do
        selectedNodes.add(cell.node)
      end
    end
  end
  return selectedNodes
end
```

Algorithm 12 - Path Cutting Query

The functionalities needed for this query have already been listed on previous steps or queries.

4.3.5.Node Deletion Procedure and Node Insertion Procedure

The first maintenance procedures are for node removal and inclusion. When removing a node, the full-paths including the node deleted are also deleted. To do so, a Node Query is executed to retrieve all full-paths involving the deleted node and this removes each position from the sparse-matrix and the full-path from the index. Finally, the node is also removed from the index.

The procedure to add a node is similar, but since the node is still not connected to the graph - only the node was added, not an edge connecting it - the procedure will only include a new entry on the nodes index. This means that a new column was created on the sparse matrix, but there are no values for this column yet, so no operations are needed since the sparse matrix doesn't store positions without values. This queries use only queries described before and don't need new functionalities.

4.3.6. Edge Deletion Procedure

It is expected that, coupled with the node deletion procedure, several edge deletions will precede it, removing connections before taking the node out of the graph. This procedure is proposed in two formats.

First is the deletion of all edges by label. Given an edge label *e*, the procedure should look for its index and then select all templates containing that edge. For each template selected, the corresponding full-paths of that template should be deleted as well. In the sparse matrix, this translates to removing all positions. The tensor-based proposal suggests to perform the selection of full-paths directly on the sparse matrix, instead of storing a full-path to template relationship, i.e., finding all positions *{i,j} -> (o,l,t)* where *t* is one of the selected templates, and then deleting the full-path *j* from the full-path index:

```
def BRGS.deleteEdgeByLabel(label)
  templates = SE.selectTemplates(label) # gets template by keyword search
  cells = BRGS.sparseMatrixByTemplates(templates)
  BRGS.clearPositions(cells)
end
def BRGS.clearPositions(cells)  # clears position
  fullPaths = new DB::Set()
  templates = new DB::Set()
  for each cell in cells do
    fullPaths.add(cell['fullPath'])  # cascade full-path removal
    templates.add(cell['template'])  # cascade template removal
    DB.clearPosition(cell['node'], cell['fullPath'])
  end
  for each fullPath in fullPaths do
    DB.deIndex(fullPath)
  end
  for each template in templates do
    DB.deIndex(template)
  end
end
```

Algorithm 13 - Edge Deletion by Label - simplified without MapReduce

The second format for is the deletion of a specific edge between two nodes, i.e., the procedure has an input in the form n_1, *e*, n_2. In this case, only the positions selected by a Node Query on each node n_1 and n_2 that are adjacent in the full-path are deleted, i.e., positions $\{i_1,j_p\} \rightarrow (o_1,l_p,t_p)$ and $\{i_2,j_p\} \rightarrow (o_2,l_p,t_p)$ where $o_2 - o_1 == 1$. Once those positions of the sparse matrix are found, the same procedure follows:

```
def BRGS.deleteEdge(node1, label, node2)
  templates = SE.selectTemplates(label)
  positionsToDelete = new DB::Set()
  positions1 = new DB::Array()
  positions2 = new DB::Array()
  positions1 = BRGS.nodeQuery(node1)
  positions2 = BRGS.nodeQuery(node2)
  for each position1 in positions1 do
    fullPathIdx = position1['fullPath']
    position2 = positions2[fullPathIdx]
    dist = position1['fullPathPos'] = position1['fullPathPos']
    if position1['template'] in templates and dist == 1 do
      positionsToDelete.add(position1)
      positionsToDelete.add(position2)
    end
  end
  BRGS.clearPositions(positionsToDelete)
end
```

Algorithm 14 - Edge Deletion - simplified without MapReduce

To execute these queries, the functionalities needed are:

- Select all templates that include a given edge label **(SE.selectTemplates)**;
- Select all positions of the sparse matrix that belong to full-paths of a given template **(DB.sparseMatrixByTemplates)**;
- Clear a position from the sparse matrix, i.e., remove the tuple from storage **(DB.clearPosition)**;
- Remove a full-path or template from the full-paths or templates indexes **(DB.delndex)**;

4.3.7. Edge Insertion Procedure

The edge insertion procedure is the most complex of all operations, but is a combination of previously discussed maintenance queries. This procedure gets as input the specific edge *e* between two existing nodes in the format n_1*, e,* n_2, with both n_1, and n_2 already in the graph. The procedure will have to generate new full-paths involving *e* based on using Path Cutting Query to find, among the full-paths that contain the node n_1, the parts that end at n_1 and, among the full-paths that contain the node n_2, the parts that start at n_2. All partial paths from the first group are concatenated with all the paths on the second group, generating new lines on the sparse matrix for each combination.

4.3.8.Node Update Procedure and Edge Update Procedure

Those procedures don't access the sparse matrix to execute. They only change values on the indexes generated to search for nodes and templates by name.

4.4.Review of architecture requirements

The analysis of the steps listed several functionalities and data structures with specific requirements to be met by any proposed architecture. These functionalities are referenced here by the method and structure names we used to reference them. These methods can be grouped by what kind of support or specific uses they will need from the architecture.

The first group are the functionalities that require generic shared data structures. These methods need to have some data structure common accessible by all workers that will execute them. The requirements that describe the necessary structures are the structures we named earlier, listed on Table 4.

Structure alias	*Ref.*	*Requirement*
DB::TwoWayArray	4.1	Key-Value store in both directions - k/v and v/k
DB::Array	4.1	Key-Value store in one direction - k/v
DB::Set	4.1	Values store without repetition

Table 4 - Requirements of generic shared data structures

The second group are the functionalities that must store or load specific data structures of the tensor-based model. Even though those procedures are built on top of the generic data structures, they have the additional requirement of being optimized for larger volumes and can be sharded through several generic data structures on different nodes of the database network. The method names we listed before that fall on this group are listed on Table 5.

Method alias	*Ref.*	*Requirement*
DB.loadPredicateObjects	4.1	Load predicateObjects given a subject node
DB.indexFullPath	4.1	Store a full-path
DB.deIndex	4.3.6	Clears a full-path or template from the index
DB.templateFromFullPath	4.2	Retrieve a template given a full-path
DB.loadFullPaths	4.2	Retrieves all full-paths
DB.loadSources	4.1	Retrieve all sources

Table 5 - Methods requiring specific distributed data structures

The third group raises the requirement of a MapReduce framework in place. The algorithms described so far don't detail how the steps of the process are started, i.e., how the map methods are invoked, neither how specific workers will receive their jobs, leaving those details to the implementation. However, it is clear that the framework must be able to handle with failed executions, retries, as well as a large and varying number of worker nodes. The single method present in the algorithms above is the MR.enqueue, but that is expected as other methods from the framework would only surface on the implementation of workers.

The fourth group is the evidence needed for searching nodes and templates by keywords. Even though the node index can be backed up by a generic data structure, as seen on the first and second group, the search for nodes using user-friendly keywords points to the need of a more specialized structure for this type of data, based on a search engine that can be more tolerant and not require exact matches only. Selecting templates by an edge label is one of the requirements for the edge removal procedure. It could also be implemented by keeping the reverse relationship of edges that belongs to a template, however, as it happens with the node searching, a search engine based index for the edges would be required anyway. Storing the whole template text on a search engine index skips the creation of that relationship. The method names we listed before that fall on this group are listed on Table 6.

Method alias	*Reference*	*Requirement*
SE.selectNode	Section 4.3.1	Select a node by keyword search
SE.selectTemplates	Section 4.3.6	Retrieve all templates given an edge label

Table 6 - Requirements for a keyword search engine

The fifth group of methods revolve around the requirement of having a sparse matrix that will have a low density of elements and should save memory by not storing positions that don't have values. Also, the sparse matrix positions will often be retrieved in whole rows or columns at a time, which generates the need to have this kind of request optimized. The method names we listed before that fall on this group are listed on Table 7.

Method alias	*Ref.*	*Requirement*
DB.storePosition	4.2	Store a tuple *(o,l,t)* at the position *(i,l)* of the sparse matrix
DB.clearPosition	4.3.6	Clear a position of the sparse matrix
DB.sparseMatrixColumn	4.3.1	Retrieves all tuples of a column of the sparse matrix
DB.sparseMatrixRow	4.3.1	Retrieves all tuples of a row of the sparse matrix
DB.sparseMatrixByTemplates	4.3.6	Retrieves all tuples matching a template index in the sparse matrix

Table 7 - Methods requiring a sparse matrix with optimized access

4.5.Architectural elements

The architectural elements needed to handle the requirements for the proposed system are:

4.5.1.A Key-value store

As seen before, several non-relational databases with key-value stores are available. To handle with the first, second, and the fifth requirements for two-way indexed collection of unique elements, a basic key-value store could be used. By creating namespaces on the keys names, several indexed collections could be stored on the same store.

key	value
nodes::next_index::	4
nodes:1	Movie
nodes:2	Director
nodes:3	Actor

Table 8 - Example of using namespaces for storing collections

Some key-value NoSQL stores mentioned before have this namespace functionality built-in, calling the second part of the name of the keys of fields. By using this, it won't be necessary to build an auxiliary key-value position to store all used indexes for iteration. However, as seen in the example, a next-index counter is still needed, and the underlying key-value must also provide an atomic get-and-increment operation for this kind of value, which is an operation available on some stores.

Furthermore, the reverse operation is also needed (i.e., find the key from the value). This is not a common functionality found on the NoSQL stores listed earlier. But it can be emulated by duplicating the key-value collections and using an inverted naming convention. In this scheme, values are used on the keys and indexes are the stored values.

key	value
nodes-index:Movie	1
nodes-index:Director	2
nodes-index:Actor	3

Table 9 - Example of using inverted key naming convention

The two-way indexed collection is the most complex type of structure listed, and the other two kinds of collections (Array and Set) can be easily mapped to a DB::TwoWayArray if needed, but most key-value stores available have specific structures for them and won't need this mapping.

So the first element of the architecture is pinned down: a key-value store with an atomic integer read-and-increment for the index building, and preferably a key-value store with structures for sets, arrays and fields.

4.5.2.Use of sharding on the stored collections by key

Since a key-value store is needed, the second and fifth requirements will further dictate that some sharding schemes need to be used. The requirements so far call for at least three forms of optimized access: retrieving all elements of a sparse matrix's row, all elements of a sparse matrix's column, and sequential access of all elements of a collection.

The first two forms of retrieval can also make use of duplicating the involved data and distributing it through the nodes. This way each column and row can be stored in a collection, and the whole collection can either be kept in-memory on a single machine or distributed through several nodes depending on the network environment available. The decision on which strategy has greater performance is one of the parameters for the implementation and testing. The third form of sequential access doesn't need data duplicating, but has to go through the same process of being sharded by key, depending on the network environment.

4.5.3.A MapReduce framework based on job queues

The third requirement calls for a MapReduce framework. Most modern frameworks have evolved from the abstraction of MapReduce to be based on job queues, pooled by a group of worker nodes. This allows for the requirements needed by this proposal, such as handling individual job failures, retries of jobs, adding or removing workers on the fly, etc.

The choice of a job queue based framework has the additional benefit of allowing the control of the workflow described on steps one and two of the indexing and sparse matrix assembly process. In this way, the Map procedure of each step can also be a job, which will in turn generate all the jobs needed for the step, and the Reduce procedure can be a scheduled job that runs periodically and checks if all intermediate jobs have finished, moving the process to the next step. On the third step, queries can be handled in the same fashion.

4.5.4.A search engine

In order to allow for user-friendly keyword search, a strictly key-value store won't be enough. This would require the user to type in exact matches with subject or object values. Furthermore, keyword search should allow for fine-tuning the scheme for finding and selecting entries, applying well established strategies for weighting specific words against another, ignoring words too small, and other language specific details such as pluralisation. This calls for a dedicated search engine with a parallel collection for the nodes. This collection will allow the retrieval of node's indexes given keywords with full or partial matches.

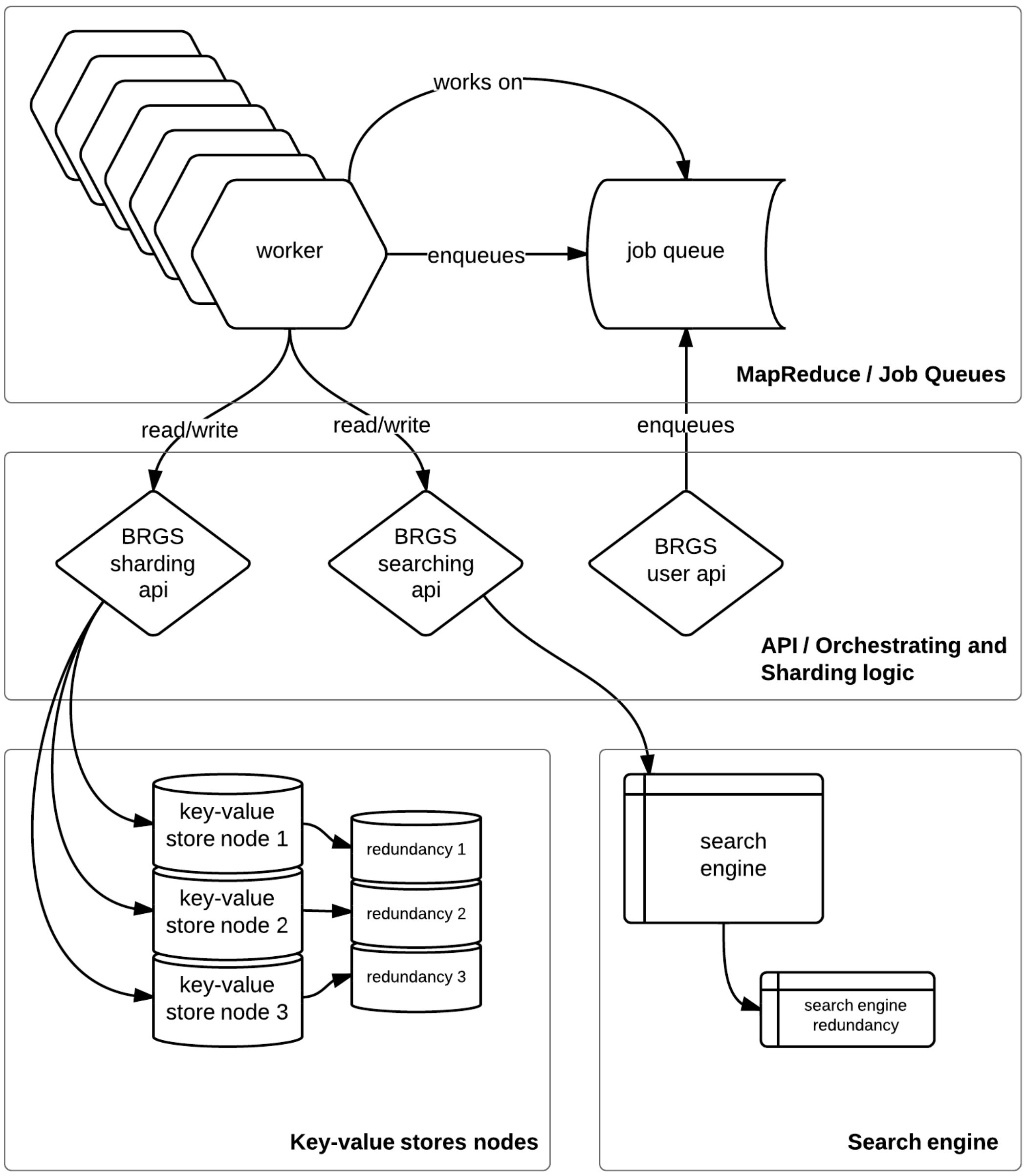


Figure 10 - Architectural elements, orchestrating API and interactions

4.6.Proposed architecture

To cater to the requirements described, we propose using the elements described on the previous section with a central orchestrating element that will implement the proposed algorithms and expose them via an API. The architectural elements interaction is described on Figure 10.

The MapReduce element of the architecture will allow a generic way to harness scalable computing resources. Coupled with Job Queues it will enable the addition or removal of more resources without rewriting or restarting the current network. Also, this abstraction will help the orchestrating element to isolate its API from the sharding logic.

Each of the key-value store nodes should have its own internal redundancy, making it transparent to the orchestrating element. However it will not need to know the distributed structure of the other nodes. This logic will be part of the orchestrating element. This way the key-value stores can have a very simple master-slave setup for redundancy. The search engine element is also isolated as several tools are known to exist that match this functionality.

In this architecture we centralized the communication between the components through the orchestrating element. This will allow the distribution logic of the different elements to work independently, i.e., the MapReduce network may grow or shrink while the Search engine nodes remain the same. This centralization also allows future work to replace individual elements to experiment with other setups.

4.7.Summary

In this chapter, we detailed the tensor-based approach that consists of three independent, sequential steps: indexing, building the matrix, and querying. We discussed each step in detail and provided the requirements for their implementation. For the first step, the construction of the index, we centred our discussion on the shared memory structures that needed to be created, as well as the functionalities that are needed to build a scale up keyword based search mechanism. For the second step, we discussed the algorithms needed to compute and assemble the matrix. The distributed queries, which constitute the third and final step of the process, were discussed and exemplified. A recap of the requirements provides a comprehensible summary to anyone that intends to implement the ideas proposed in this dissertation. We close the chapter by presenting a possible architecture that matches the requirements described. In the next chapter, we detail a possible implementation for the requirements presented in this chapter, used to validate our ideas.