Entity Linking Across Knowledge Graphs

Shramana Thakur
Universität Bonn, Germany
s6shthak@uni-bonn.de

Srinivas Nandagudi Sridharamurthy
Universität Bonn, Germany
s6srnand@uni-bonn.de

Rishi Tripathi
Universität Bonn, Germany
s6ritrip@uni-bonn.de

Supervisors: Dr. Hajeera Jabeen, Gezim Sejdiu, Shimaa Ibrahim

Abstract: This project is done as a part of the Distributed Big Data Analytics Lab during Summer Semester 2019 at the University of Bonn. The aim of this project is to extract pairs of similar entities in heterogeneous and large Knowledge Bases, using Blocking and Similarity Matching approaches, implemented using the Apache Spark framework in Scala.

1. Problem definition

There is an enormous volume of data available as web resources today. However, most of it is complex and unstructured, containing valuable information blocks along with noisy data. A knowledge base (KB) is a system that can be used to store, organize and analyze such data, to extract meaningful inferences from it. This is usually achieved through *'Entity Linking'* (finding out which resources depict the same real-word object), *'Link Prediction'* (deducing new facts about existing resources) and *'Link Clustering'* (grouping similar resources). Entity Linking, in particular, is an important task for unambiguous information retrieval (and finds applications in search engines, master data management, network analysis etc.). Popular KBs like DBPedia, Yago and FreeBase (to name a few), may contain varying information about the same real-world entity, which upon resolution, gives us a more holistic understanding of it.

However, processing these KBs is an expensive task because they are generally large, heterogeneous, noisy and constantly expanding. Naïve approaches for entity linking across two KBs are rendered impractical when dealing with large KBs since they usually require a quadratic number of comparisons. Moreover, processing such large amounts of data in a sequential algorithm on a single machine is hardly feasible. Thus, in the context of Big Data, there is a need to develop efficient (yet effective) solutions to perform parallel and large-scale entity matching.

2. Approach

2.1. Methodology

To perform entity matching, we take two Knowledge Graphs (KG) as input, arranged as RDF[1] triples (Subject, Predicate, Object). The subsequent workflow is divided into the following stages: Preprocessing,

[1] RDF : **R**esource **D**escription **F**ramework is a framework for describing resources on the web

Token-Blocking, Block Filtering, Block Splitting, Candidate Pair Generation, Candidate Pair Filtering and Similarity Matching, as shown in Fig. 1.

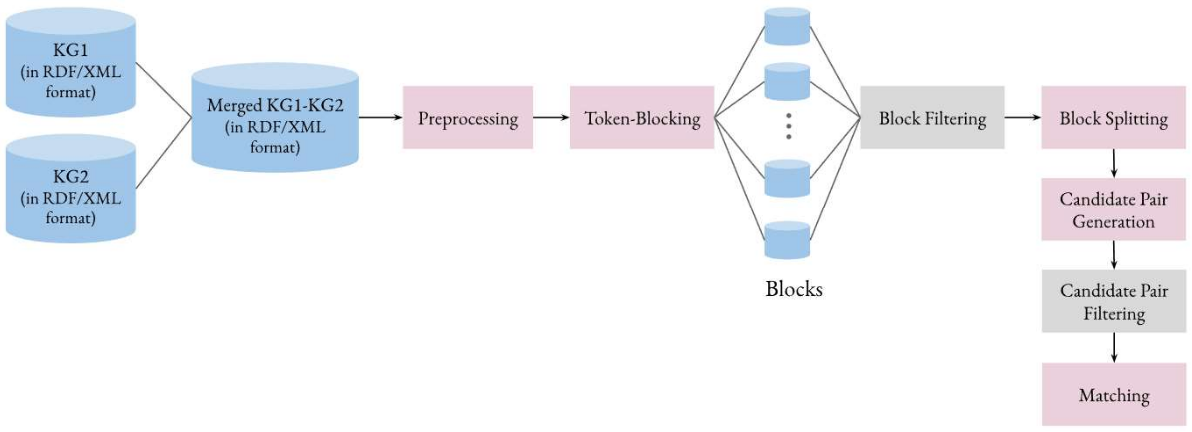


Fig. 1. Workflow

After the input has been preprocessed (discussed in detail in section 3.1), we apply Token-Blocking to reduce the number of candidate entity pairs (which, in the worst case, is of the order of KG_1.size * KG_2.size). Token-Blocking is a technique to cluster similar entities (or subjects) together based on common object literals [1, 2, 3]. This restrains the number of pairwise comparisons to each cluster (or block). Token-Blocking is intuitive (in the sense that two potential entity matches must have common data values). Furthermore, it does not need any domain expertise and can be easily parallelized. We apply two filtering techniques on the blocks obtained to reduce the number of blocks - Block Purging [4] (ensures blocks containing more number of entities beyond a certain threshold are filtered, since, a token shared by multiple entities is evidently less discriminative) and Clean-Clean Entity Resolution constraints [4] (ensures only the blocks containing entities from *both* KGs are retained, since we are only matching entities *across* KGs). Blocks obtained after the filtering procedure are then split if they are 'larger' than the average block 'size'. In our context, we refer to the block size as the number of comparisons that will be performed within each block (number of entities from KG_1 * number of entities from KG_2). This block splitting step tackles skewness to a large extent and ensures equal job distribution amongst the partitions. The Token-Blocking, Block Filtering and Block Splitting procedures are discussed further in sections 3.2 -3.3.

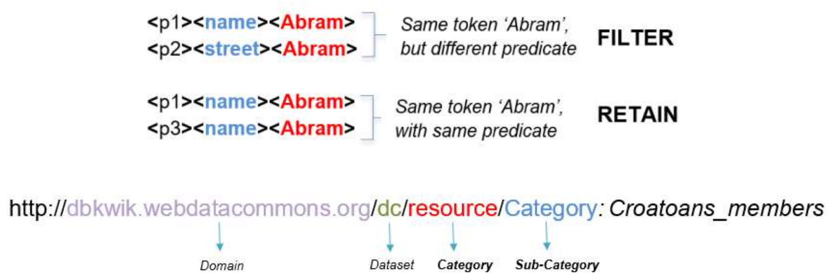
After the Blocking procedures, we generate a list of candidate entity pairs by a pairwise selection of entities within each block, ensuring each pair contains an entity from either KG. We apply a few novel filtering approaches, to reduce the number of candidate pairs by almost half (as found experimentally). Firstly, for each candidate pair, we compare the "predicate" corresponding to their common object literal. This ensures that the entities share the common literal in the same context (Fig. 2(a)). Secondly, we exploit the information within the entity subject URI[2] itself, by extracting its *category* and *sub-category* (Fig. 2(b)). We filter out any candidate pair whose subject URIs do not have the same category and sub-category.

The filtered candidate pairs are then tested for URI (local name) similarity, with the intuition that similar entities would have similar local names [5]. We only use a subpart of the URI as follows:

[2] URI : A **U**niform **R**esource **I**dentifier is a string of characters that unambiguously identifies a particular resource

http://<domain name>/<dataset name>/category/<URI local name used for Similarity Matching>

The extracted local name substrings from either URIs are then converted to bi-grams[3] and the Jaccard Similarity index is used to measure the similarity of these substrings. Finally, candidate pairs having a similarity value beyond a certain threshold, are reported to be predicted entity matches.


<p1><name><Abram>
<p2><street><Abram>
Same token 'Abram', but different predicate
FILTER
<p1><name><Abram>
<p3><name><Abram>
Same token 'Abram', with same predicate
RETAIN
http://dbkwik.webdatacommons.org/dc/resource/Category: Croatoans_members
Domain
Dataset
Category
Sub-Category

a) An example of Predicate-Filtering for entities p1, p2, p3

(b) The 'category' and 'sub-category' of a URI. Used for Category-Filtering

Fig. 2. Candidate Pair Filtering Techniques

Due to limited computing resources, we are only using a certain percentage of the two input KGs. Entity matching, howsoever algorithmically efficient, remains a computationally expensive task (in terms of both memory and time). Thus, we are randomly selecting only a small percentage of the KGs for experimentation (to avoid facing Out-Of-Memory errors). However, this approach does not guarantee we have all the information (triples) pertaining to a certain entity at hand. Hence, in this project, we are limiting all approaches to information extraction from individual triples, rather than, from the set of all triples related to a particular entity. This 'lack of access to full information' further restricts us from using graph-based entity matching techniques that are widely used for entity matching.

2.2. Datasets Used

The datasets were collected from the OAEI-2018 Campaign [6]. Table (1) describes the pairs of KGs used for this project.

Source	Hub	#Entities	#Properties	#Classes	#Triples	#True Entity Matches	#Triples Corresponding to True Entity Matches
Old School RuneScape Wiki	Games	38,563	488	53	598,052	71	4,878
DarkScape Wiki	Games	19,623	686	65	403,132		
Marvel Database	Comics	56,464	99	2	269,106	17	483
DC Database	Comics	128,495	177	5	643,975		
Memory Alpha	TV	63,240	326	0	491,298	29	568
Memory Beta	Books	63,223	413	11	592,263		

Table 1. Datasets Used

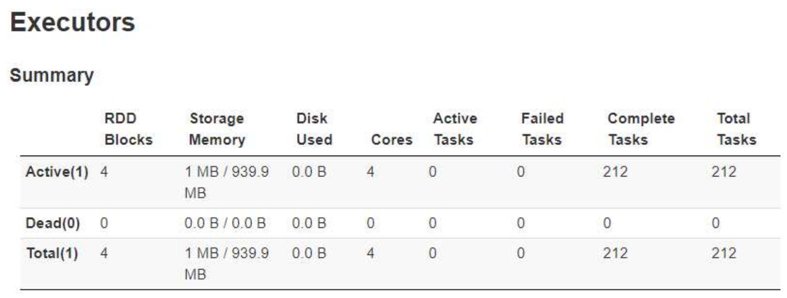
[3] Bi-grams : A pair of consecutive characters in a string. Ex: bigram('word') = {'wo', 'or', 'rd'}

2.3. Parallelization

To make use of all the available cores on a single machine (in our case, 4), the *SparkSession* object is initialized with the following configurations:

spark.executor.cores = 4
spark.executor.memory = 2G
spark.driver.memory = 2G
spark.default.parallelism = 4

Fig. 3(a) shows the Executor tab on the Spark Web UI, indicating the use of all 4 cores. Fig. 3(b) shows the DAG for one stage (with two tasks) within a job. Fig. 3(c) shows the timeline for these two tasks distributed over the 4 cores.

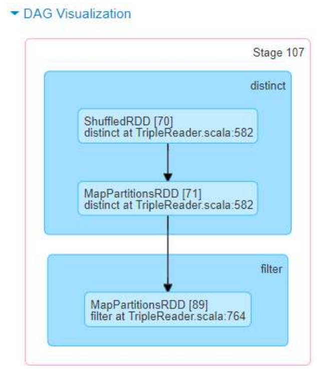


Executors

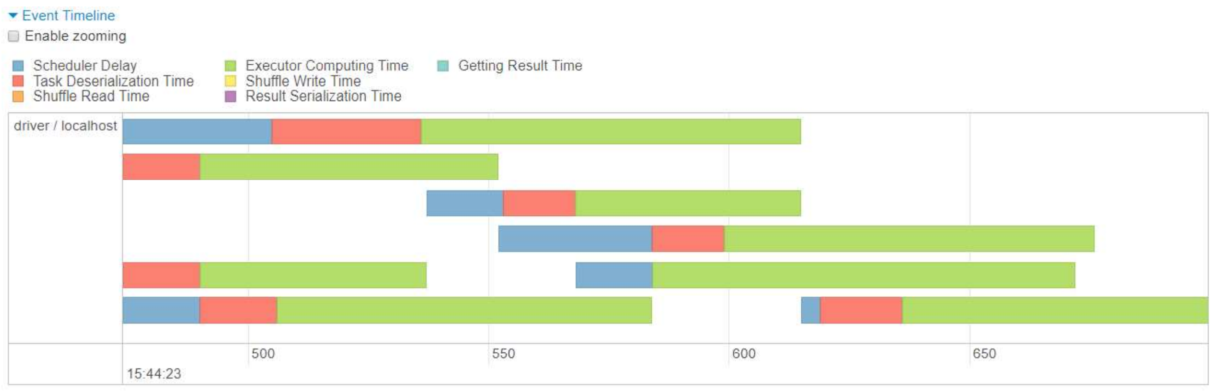
Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks
Active(1)	4	1 MB / 939.9 MB	0.0 B	4	0	0	212	212
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0
Total(1)	4	1 MB / 939.9 MB	0.0 B	4	0	0	212	212

(a) Executors Tab


DAG Visualization
Stage 107
distinct
ShuffledRDD [70]
distinct at TripleReader.scala:582
MapPartitionsRDD [71]
distinct at TripleReader.scala:582
filter
MapPartitionsRDD [89]
filter at TripleReader.scala:764

(b) DAG for a particular stage (with 2 tasks)



(c) Event timeline corresponding to the DAG in (b)

Fig. 3. Spark Web UI

2.4. Frameworks and Libraries

1. **Frameworks:** RDDs, DataFrames (for broadcast joins)
2. **Libraries:**
    - **Triple**: From org.apache.jena.graph package. Used for reading RDF/XML files.

- **StopWordsRemover**: From apache.spark.ml.feature package. Used for removal of stop words.
- **Stanford CoreNLP**: Used for Lemmatization

3. Implementation

3.1. Preprocessing

1. **Input (RDF/XML) files :** KG$_1$, KG$_2$, True matches

2. **GetTrueEntityMatchesTriples( ), GetTrueEntityMatchesNames( ), hasEqualToRelation( ) :** Extracts true entity name (URI) matches as tuples (Entity URI from KG$_1$, Entity URI from KG$_2$) from the 'True matches' file.

3. **GetFractionOfDataset( ) :** To use a partial dataset for experimentation, randomly selects a percentage of triples from KG$_1$ and KG$_2$, and includes all the triples related to the entities that have true matches. Merges them into a single RDD of *Triples*[4] called *MergedTriples*. Additionally, filters out any triple (h,r,t) where the object 't' is a numeric value, or another entity. This step is necessary because we consider only objects with literal values for Token-Blocking.

3.2. Token Blocking

TokenBlocking( ) : The main Token-Blocking procedure performs the following functions:

1. Splits each object literal into individual tokens via the **TokenizeObjLiterals( )** function by performing the following steps:
    a. Converts all literal strings to lowercase
    b. Splits the literals by whitespace/underscore/punctuation into individual tokens
    c. Removes numeric tokens (if parameter keepNumericLiterals is false)
    d. Removes stop words (English)
    e. Removes extra spaces and single characters
    f. Removes duplicate tokens within a single object string
    g. Lemmatizes tokens using the **LemmatizeString( )** function

   The output of this step is a tuple (Entity ID, List of tokens).

2. Expands each (Entity ID, List of tokens) tuple into individual pairs of (Entity ID, Token Name) and groups them by 'Token Name' to obtain (Token Name, List of Entity IDs) tuples. i.e. All the entities (from either KG) having a common token are grouped together. Each of these tuples are referred to as a *Block*[5]. The subsequent procedures (block filtering and splitting) are performed on a RDD[*Block*] structure.

3. *Block filtering*: To reduce the number of blocks, performs the following steps:

[4]**case class Triples** (ID: Long, subject: *String*, predicate: *String*, \`object\`: *String*)
[5]**case class Block** (tokenID: Long, token: *String*, entitiyIDsKG1: *Seq*[Int], entitiyIDsKG2: *Seq*[Int], numBlockComparison : Int)

a. *Block Purging*: Excessively large blocks are removed, as these usually correspond to tokens that occur frequently across both KGs. The condition for block purging adopted here is given by the following pseudo-code:

```
For all blocks b,
    Remove b if:
       Min (No. of KG₁.entities in b, No. of KG₂.entities in b) <
       (Max. possible entity matches between KG₁, KG₂)*fraction
```

The maximum possible number of entity matches is given by the size of the smaller KG while the *fraction* multiplier is determined experimentally.

b. *Clean-Clean Entity Resolution*: To ensure comparisons between entities of both KGs, blocks having entities from only one KG are removed.

3.3 Block Splitting

BlockSplit( ) : Even though the Block Filtering method ensures blocks with a large number of entities are removed, the number of pairwise comparisons within each block can still be very skewed. The number of comparisons for each block can be computed as (No. of KG_1.entities)*(No. of KG_2.entities). For blocks that have an above average number of comparisons, the Block Split procedure splits them into multiple (smaller) blocks such that none of them have an above average number of comparisons. The pseudo-code of the procedure is as follows:

```
avgComparisons = Average no. of comparisons within all the blocks
smallBlocks = set of blocks with comparisons <= avgComparisons
largeBlocks = set of blocks with comparisons > avgComparisons

For all blocks b in largeBlocks,
   Find value p such that:
        ((No. of KG₁.entities in b)*(No. of KG₂.entities in b))/p < avgComparisons

   If (No. of entities from KG₁ > No. of entities from KG₂):
       Split KG₁.entities into p parts and for each part p* create a block with the set of
       entities: (KG₁.entities from p*, KG₂.entities)

   Else
       Split KG₂.entities into p parts and for each part p* create a block with the set of
       entities: (KG₁.entities, KG₂.entities from p*)
```

3.4. Candidate Pair Generation

1. **CandidateGeneration( ) :** Forms candidate pairs of entities as tuples (Entity ID from KG_1, Entity ID from KG_2) from the filtered and split blocks. This step can be performed in parallel with (almost) equal load distribution since Block Splitting ensures there are no disproportionately large blocks.

2. **GenerateCandidateEntityMap( ) :** Performs a broadcast join (based on IDs) between the candidate Entity IDs and the original *MergedTriples* RDD, by converting both to a DataFrame `structure. This is to obtain all subject, object, predicate information of the candidate entities to be used for the subsequent processes.

3.5. Candidate Pair Filtering

FilteringCandidates( ) : Filters candidate entity pairs as described in Section 2.1. Uses two function calls:

1. **FilterPredicates( ) :** Filters out candidate pairs if they have different predicates. (Fig. 2(a))

2. **FilterCategories( ) :** Filters out candidate pairs if they their subject URI has different category/sub-category. (Fig. 2(b))

3.6. Matching

1. **MatchingCandidates( ) :** From the remaining candidate entity pairs, performs a URI (local name) based matching, with the notion that matched entities will have similar names. Uses the **URISimilarity_Jaccard( )** function to compute Jaccard similarity metrics (as described in Section 2.1). The candidate pairs are then sorted in decreasing order of similarity values, and only the pairs with similarity values greater than a particular threshold are reported as matches.

2. **GetPredictedEntityMatchesNames( ) :** Resolves the IDs of the predicted entity matches into their names (subject URIs).

3.7. Evaluation

Evaluate( ) : Calculates the precision, recall and f-score of the predicted matches. Also reports the entity URI pairs that were falsely predicted (false positives) and those that could not be predicted, but were true (false negatives).

3.8. User input Parameters

Table (2) summarizes the different parameters that needs to be specified by the user, and the default values for them.

	User Input Parameter	Function	Description	Default Values
1.	numPartitions	GetFractionOfKG	Number of RDD (re)partitions	4
		CandidateGeneration		4
2.	percentageOfData	GetFractionOfKG	Percentage of triples to be used from the complete KGs	0.01
3.	useOnlyTrueMatches	GetFractionOfKG	Whether to use triples only corresponding to the matched entities or not	FALSE

4.	fraction	TokenBlocking	Fraction of the maximum possible entity matches that will be used as a threshold for block purging	0.02
5.	similarityThreshold	MatchCandidates	Minimum Jaccard similarity score required to declare a candidate pair, a match	0.45
6.	keepNumericLiterals	TokenBlocking TokenizeObjLiterals	Whether or not to include numeric literals during token blocking	TRUE

Table 2. User input Parameters

4. Results

Table (3) summarizes the results for the 3 sets of KGs used, averaged over 5 runs.

Input KGs	Marvel-DC	MemoryAlpha-MemoryBeta	Darkscape-OldSchoolRunescape
Percentage of total #triples used	1%	1%	1%
#Triples in reduced (merged) dataset	~9500	~11,400	~14,800
#Distinct entities in reduced (merged) dataset	~8,600	~10,300	~8,700
#True matches	17	29	71
Precision	0.94	0.864	0.8
Recall	0.94	0.75	0.97
F-measure	0.94	0.8	0.87
#False Positives	1	3.4	17.25
#False Negatives	1	7	2
#Token blocks generated	1,643.2	6,250.4	2,985.75
#Token blocks filtered	633.8	3,889.6	2,064.5
Percentage reduction after filtering	61.40%	37.70%	30.85%
Threshold for Block Purging	0.02	0.02	0.02
#Candidate pairs generated	76,943.20	323,967.80	103,309.75
#Candidate pairs after filtering	17,836.40	119,911	30,022.50
Percentage reduction after filtering	71.80%	62.90%	70.93%
Similarity Threshold for Entity Matching	0.45	0.48	0.45

Table 3. Results (Average of 5 runs)

We can make the following observations from the results:

1. Using *only* the information extracted from individual triples, we obtain a varying amount of precision. However, the recall obtained during each run is quite stable.

2. The similarity threshold needs to be modified experimentally for each pair of KGs.

3. The candidate pair filtering and block filtering processes reduce the number of computations by an amount in the range 30-70% depending on the pairs of KGs used.

Furthermore, we can analyze the cases for false positives and false negatives individually:

False Positives:

a. The number of false positives depend on the random subset of KGs selected at each run, but, remain more or less in the ± 3 range of the average.

b. False positives can be investigated into further by considering entity relationships with other entities. Since we do not have this information available for all entities (due to random selection), we envision it as a task for future scope.

c. Additionally, the ‘gold standard’ used as reference for true entity matches in this project, is not complete [6]. A few predicted matches reported as false positives, may actually be referring to the same real-world entities:

*http://dbkwik.webdatacommons.org/**marvel**/resource/**Category:2008,_May***
*http://dbkwik.webdatacommons.org/**dc**/resource/**Category:2008,_May***

*http://dbkwik.webdatacommons.org/**darkscape**/resource/**Berserker_ring***
*http://dbkwik.webdatacommons.org/**oldschoolrunescape**/resource/**BERSERKER_RING***

False Negatives:

a. The Jaccard Similarity index is sensitive to the length of strings. Hence, similar entities may receive a smaller similarity value than the threshold (and get filtered out), even though they are a candidate pair. For example, the following pair has a similarity value 0.38 while the threshold is 0.45.

*http://dbkwik.webdatacommons.org/**marvel**/resource/**Cyclops***
*http://dbkwik.webdatacommons.org/**dc**/resource/**Cyclopes***

b. Several (true) entity pairs have single triples corresponding to them in their respective KGs with different object literals. Token Blocking, in such cases would not cluster them in the same block. For example,

*http://dbkwik.webdatacommons.org/**memory-alpha**/property/**conductor***
*http://dbkwik.webdatacommons.org/**memory-beta**/property/**director***

5. Conclusion and Future Scope

In this work, we perform entity matching across two KGs using concepts of Token-Blocking and URI (local name) Similarity Matching, using Jaccard Index and obtained positive results. Using Spark framework we were able to optimize the usage of the computing resources available to us.

As future scope, filtered candidate entity pairs could be investigated further using similarities based on their relationships with other entities, through graph-based approaches. While “local name similarity” does give us positive results, it should be used in aggregation with using other heuristics [5], to reduce the number of false positives and negatives.

6. References

[1] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. “Meta-Blocking: Taking Entity Resolution to the Next Level”. *IEEE Transactions on Knowledge and Data Engineering* 26.8 (2013): 1946-1960.

[2] Giovanni Simonini, Sonia Bergamaschi, H. V. Jagadish. "BLAST: a loosely schema-aware meta-blocking approach for entity resolution". *Proceedings of the VLDB Endowment* 9.12 (2016): 1173-1184.

[3] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, Sonia Bergamaschi. “SparkER: Scaling Entity Resolution in Spark”. *EDBT 2019: 22nd International Conference on Extending Database Technology* (2019).

[4] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Peter Fankhauser, (2011) “Efficient Entity Resolution for Large Heterogeneous Information Spaces”. *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM (2011).

[5] Efthymiou, Vasilis, George Papadakis, Kostas Stefanidis, Vassilis Christophides. "Simplifying entity resolution on web data with schema-agnostic, non-iterative matching." In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1296-1299. IEEE (2018).

[6] http://oaei.ontologymatching.org/2018/knowledgegraph/index.html

7. Project Timeline

Week	***Tasks***	***Task Allocation***
1	Reading through research papers Going through Spark/Scala tutorials First Presentation	All
2	Making of rough plan of Implementation Environment Setup	All
3	Preprocessing input KGs, RDF/XML with true results	Srinivas
3+4	Token Blocking, Block Filtering, Block Splitting	Rishi, Shramana
5	Candidate Pair Generation, Candidate Pair Filtering	Rishi, Shramana
5+6	Matching (Trying out different similarity indices)	All
7	Evaluation of the result Implementing suggested modifications	All
8	Report and Final Presentation	All