Spark-SAKey: Scalable Almost Key discovery in RDF data using Spark

Robin Stenzel[1] and Tasneem Tazeen Rashid[2]

[1] Universität Bonn, Germany
s6rosten@uni-bonn.de , Matriculation number: 3000946
[2] Universität Bonn, Germany
s6tarash@uni-bonn.de, rashid.tasneem@gmail.com , Matriculation number: 3032710

1 Problem definition

With the purpose of availability of data for semantic web mappings, RDF is vastly used. Applications can effectively integrate data by utilizing the links among RDF resources and keys are very efficacious for this objective. Key constraints are denoted as a set of properties which uniquely identifies facts from the resources. Nonetheless, automatically finding keys in the large RDF resources can be obscuring due to the large size of the data. Additionally, large databases often include duplicates and erroneous data which makes it impossible to find pure keys with no exceptions.

The SAKey approach ref [2] provides a way to solve this problem by finding n-almost keys, namely keys with less than n exceptions. It also uses a wide variety of pruning techniques to handle larger amounts of data. In this paper, we will build on that approach and try to parallelize certain functions using Spark GraphX. This should lead to an improved run time on clusters and be more suitable for even larger databases.

2 Approach

In this chapter, we will describe the main ideas of the SAKey approach and describe afterwards where our approach adds new elements or deviates from the original respectively. The main methods used are:

1. creating a data structure called Final Map,
2. finding potential n-non keys by mining a graph for maximal cliques,
3. the n-non key discovery,
4. derivation of (n-1)-almost keys.

First, we introduce some basic definitions. An n-almost key is a property set with less than n exceptions or collisions, in other words, it exists at most n instances that share the same values for this set. A n-non key on the other side, represents a property set with at least n collisions. This similarity is later used to derive (n-1)-almost keys from n-non keys [2].

In order to find a collision for a set of properties, the SAKey [2] approach represents the data in the following map structure, called Final Map. The Final Map stores for each property a set of sets of subjects. Every set corresponds to a group of instances that share one value for a given property . To illustrate this approach we take a look at the movie dataset shown in Figure 1 and the corresponding Final Map shown in Table 1.

Dataset D1:
d1:Film(f1), $d1{:}hasActor(f1,'' B.Pitt'')$, $d1{:}hasActor(f1,'' J.Roberts'')$,
$d1{:}director(f1,'' S.Soderbergh'')$, $d1{:}releaseDate(f1,'' 3/4/01'')$, $d1{:}name(f1,'' Ocean's\ 11'')$,
d1:Film(f2), $d1{:}hasActor(f2,'' G.Clooney'')$, $d1{:}hasActor(f2,'' B.Pitt'')$,
$d1{:}hasActor(f2,'' J.Roberts'')$, $d1{:}director(f2,'' S.Soderbergh'')$, $d1{:}director(f2,'' P.Greengrass'')$,
$d1{:}director(f2,'' R.Howard'')$, $d1{:}releaseDate(f2,'' 2/5/04'')$, $d1{:}name(f2,'' Ocean's\ 12'')$
d1:Film(f3), $d1{:}hasActor(f3,'' G.Clooney'')$, $d1{:}hasActor(f3,'' B.Pitt'')$
$d1{:}director(f3,'' S.Soderbergh'')$, $d1{:}director(f3,'' P.Greengrass'')$, $d1{:}director(f3,'' R.Howard'')$,
$d1{:}releaseDate(f3,'' 30/6/07'')$, $d1{:}name(f3,'' Ocean's\ 13'')$,
d1:Film(f4), $d1{:}hasActor(f4,'' G.Clooney'')$, $d1{:}hasActor(f4,'' N.Krause'')$,
$d1{:}director(f4,'' A.Payne'')$, $d1{:}releaseDate(f4,'' 15/9/11'')$, $d1{:}name(f4,'' The\ descendants'')$,
$d1{:}language(f4,'' english'')$
d1:Film(f5),$d1{:}hasActor(f5,'' F.Potente'')$, $d1{:}director(f5,'' P.Greengrass'')$,
$d1{:}releaseDate(f5,'' 2002'')$, $d1{:}name(f5,'' The\ bourne\ Identity'')$, $d1{:}language(f5,'' english'')$
d1:Film(f6),$d1{:}director(f6,'' R.Howard'')$, $d1{:}releaseDate(f6,'' 2/5/04'')$,
$d1{:}name(f6,'' Ocean's\ twelve'')$

Figure 1: Movie dataset

Property	Output
d1:hasActor	{{f1, f2, f3}, {f2, f3, f4}}
d1:director	{{f1,f2,f3}, {f2, f3, f5}, {2, f3, f6}}
d1:releaseDate	{{f2, f6}}
d1:language	{{f4, f5}}

Table 1: Final Map

If we look at the property hasActor we can see that it has two sets of size 3. Each representing a value that is equal for each movie in the set. It is therefore by itself a 3-non key. The first set is caused by "B. Pitt" who plays in the films f1, f2 and f3 and the second set is based on the movies which feature "G. Clooney". Other actors who only appear in one movie do not have their corresponding set, as one film will never lead to a collision. Removing all singleton sets (Singleton Filtering) is therefore a useful step to reduce the number of values in the map.

The next step of the SAKey [2] approach is to build property pairs which have at least n common subjects/instances. Each pair that fulfills this criteria will be added to a graph as an edge. This graph will then be mined for maximal cliques using a greedy algorithm. The received cliques represents the potential n-non keys (PNKs) and will be given as input to next algorithm, the n-non key discovery.

This algorithm takes the Final Map as a second argument. It then executes a depth first search on it, to test every subset of properties for being a n-non key. However, to reduce the amount of property sets, the algorithm use multiple pruing techniques. First, the algorithm only considers sets which are a subset of a PNK. Any sets not covered by that can not have n collisions by construction of the graph. Second, if all PNKs containing this properties are already in the discovered n-non key set, the algorithm discards this property set (Antimonotonic Pruning). There is also a set called seenIntersections. This ensures that no intersection is checked twice (Intersection Pruning).

Finally, the last algorithm derives the (n-1)-almost keys by using the complement of the found n-non keys.

Now, in order to parallelize the finding of maximal cliques and the discovery of n-non keys, we introduce an additional step in between the first and second algorithm. In this step, we

make use of the Apache Spark GraphX library and build a graph out of the final map and use the built-in function to compute its connected components. Since connected components are disjoint by definition, cliques can only be found within one component. Therefore, it is possible to search for max cliques in each connected component in parallel. The same is true for the n-non key discovery. Since the found cliques in one component do not share nodes with another one, the later discovered n-non keys will also be disjoint. Consequently, the danger of losing performance by considering the same candidates or intersection multiple times is non existent.

However, adding the computation of connected components adds another time consuming process to the algorithm. We hope to justify this increase by enabling parallelization for two steps that were not running in parallel beforehand.

Another approach could be parallelizing the non key discovery by running it on each clique in parallel. That approach would not add new algorithms and therefore not increase the time through new computations. Nevertheless, the cliques are in general not disjoint which could reduce the effect of the used pruning techniques. For example, since the parallel processes do not share the same set of already found non keys, many candidates would be processed multiple times.

3 Dataset

Data sets were taken from [2] from DBPedia. Link to the data set repository: . At the same page, one can find the sakey.jar and how to use it.

The datasets can also be found in our solution under src/main/resources/datasets . Some of the original datasets were missing some ending points, which are corrected there.

4 Implementation

Structure of implementation:

1. Set the value of n to find n-non keys and (n-1)-almost keys.
2. Read the triple file.
3. Generate a map (Final Map) with the properties from the dataset along with their set of sets of subjects.
4. Create a graph from the map (GraphX).
5. Compute connected components.
6. Find maximal cliques for each connected component in parallel (maximal potential n-non keys, PNK).
7. Generate n-non keys from the PNK in parallel for each component.
8. Compute (n-1)-almost keys from all n-non keys and the Final Map.

4.1 Final Map

Input: RDF triple
Output: Pair including the Final Map with elements of form (property/predicate -> set of sets of subjects) and a set of already found keys

First, the input is mapped to ((predicate, object), subject) and aggregated by key. As the result, one receives a set of subjects for each (predicate, object)-pair. In the movie example, that means aggregating all movies where for example Brad Pitt plays as an actor.

Afterwards, the result is mapped as follows: result -> (predicate, (object, subject)) and then again is aggregated by key. This results in a set of set for each property/predicate. For example for the actor property, this would include one set of movies per actor. This step also includes two filtering techniques during the aggregation. First, only sets > 1 will be appended, since singleton sets can not cause any collision and therefore will never create a new n-non key ("Singleton Sets Filtering"). Second, while adding new sets to the working set, they will be checked if they are a subset of an already included set, also the other way is checked, namely, if an included set is a subset of the new one. ("v-exception Sets Filtering").

Finally, all properties with an empty set are filtered out and added to the almost keys. These properties can not cause any collisions and are therefore keys in the normal sense. The rest is returned as the final map. All steps use only core elements of Spark (map, filter, aggregate) and can therefore be fully parallelized. Table 1 shows the out of final map of the dataset.

4.2 GraphX and Connected Components

Input: Final Map
Output: Connected components for the graph consisting of property combinations

This step is added in order to parallelize the SAKey algorithm and is the main difference in our implementation compared to the original. However, before creating the graph, one has to generate edges by testing each possible (property1, distinct node set1, property2, distinct node set2) quadruple for the size of their intersection. That means, if two properties have at least n common nodes in their sets stored in the Final Map, an edge is added containing both properties. The reason behind this is that only then, they could cause enough collisions to be part of a n-non key (still part of SAKey).

Afterwards, we use the function "triple.asGraph" (provided by SANSA) to create a GraphX graph and use the built-in function "getComplements" to compute all connected components. This enables the user to run the steps c and d in parallel on each of them. The function "getComplements" can also be executed in parallel.

4.3 Max Cliques for PNK

Input: Graph
Output: set of maximal potential n-non keys (PNK).

This function computes approximately the maximal cliques for a given graph using a greedy algorithm. Since the algorithm requires deletion of edges, we transformed the GraphX graph into one of the scalax library. The algorithm consists of two node-ordering algorithms. The first one, called "MIN-FILL" uses the min-fill heuristic to order the nodes and to create a chordal graph by adding edges. The min-fill node is the one that needs the least edges to be filled so that its parent set is fully connected. In each step, the algorithm selects the min-fill node, puts it in the next position and connects all its neighbors with each other. Then it deletes the node and continues with the next [1].

Subsequently, the function uses the max cardinality ordering to determine the maximal cliques in the chordal graph [1]. The found cliques represent the potential n-non keys and the function returns only the maximal ones.

4.4 n-Non Key Discovery

Input: Potential n-non keys, Final Map
Output: n-non keys

The following algorithm, traverses the Final Map in a depth first search manner and checks all sets of properties if they have at least n collisions and therefore are n-non keys. To avoid the immense overhead of testing each of the $2\hat{|}$properties| sets, the authors of SAKey use a number of pruning techniques.

4.5 Almost Key Derivation

Input: n-non keys.
Output: (n-1) almost keys

This algorithms derives the (n-1)-almost keys by using the complement of all found n-non keys. Since n-non keys have per definition at least n collisions, the complement includes all property sets with strictly less collisions. As for this algorithm, one needs all n-non keys, it is hard to parallelize parts of it.

5 Evaluation

In this section we examine our implementation considering completeness and time analysis. In order to evaluate our code, we use the movie dataset which is used throughout the SAKey paper to explain its algorithms. For that, we examined the following cases for n = 2, 3 and n>3. Within each case we checked if our algorithm produces not only the expected (n-1)-almost keys but also all intermediate results such as "max cliques" and "n-non keys". In Table 2, we display an extract of such a case for n = 2.

Step	Output
Max cliques	{{language}, {director,release}, {director, actor}}
n-non keys	{{language}, {director, release}, {director, actor}}
(n-1)-almost keys	{{actor, release}, {director, language}, {language, release}, {language, actor}, {name}}

Table 2: Correctness test case

The algorithm correctly identifies the expected results for each such case.

Afterwards, we tried compare our results to the underlying SAKey paper for the same datasets. Unfortunately, the authors of the paper use some pruning techniques not mentioned in the paper. Although, the numbers of found keys only differ from each other in a small fraction ($< 10\%$) due to pruning or mistakes on our side, this will require further investigation in the future.

Nevertheless, this small difference should not change the tests for time analysis in a significant manner. For that we compared the two approaches in respect to their computation time. The Table 3 shows the outcome for n = 3:

Data set	#triples	#properties	Graph Creation	Connected Components	Total Time	SAKey[2]
DB:Website	8506	66	16s	151s	172s	1s
DB:OAEI_2011	1130	7	14s	144s	168s	1s
DB:OAEI_2013	1744	11	13s	104s	120s	1s
YA:SportSeaso	83944	35	>30min	-	-	5s
YA:Building	114783	17	>30min	-	-	5s

Table 3: Time analysis of Spark-SAKey

The processor used in the experiment is Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz (4 cores, simulating 8) and 8gb RAM.

While reading Table 3 one can notice, that our approach creates a big overhead while computing the connected components. The idea behind the approach was to justify this overhead by using parallelization and therefore reducing its time on the cluster. For example, if you would run the algorithm on a cluster with 180 cores (more than 20 times than the number used here), one would assume that this reduces the runtime by a factor of 20. That puts the runtime from 100-170 seconds down to just 5-9 seconds. Since it is hard compete against run times ranging in seconds, the next step was to test it on larger files. However, testing the new approach on those sets, led to the process being stuck in the step of creating the graph in GraphX. There are different conclusions one can draw from this.

One would be that the used function (SANSA function .asGraph()) is not performant enough and there are better ways to create the graph. Another one is that with increasing size, the overhead increases faster than the time won by parallelization. In addition, after creating the graph, getting the connected components is even more time consuming. It runs in time O(|V|*|E|), therefore in around squared time of the graph creation.

As long as there is no faster way to receive the connected components, the time used in this operation exceeds the total time of SAKey [2] and no improvements can be made with our approach.

6 Future Work and Conclusion

In this report, we presented an approach trying to parallelize parts of the SAKey approach to improve its performance. The idea behind the approach was to insert two extra steps into the algorithm. Namely, creating the graph in Spark's GraphX and determine the connected components (in parallel) with the in-build function. Afterwards, we hoped to improve the performance of finding max cliques by running this algorithm on each connected component in parallel. The same thing would be done afterwards for computing the n-non keys. Since the connected components are disjoint, this does not reduce the effect of the used pruning techniques like "Antimonotonic Pruning", "Intersection Pruning" and "Inclusion Pruning". Therefore, it should result in a reduced run time by the factor equal to the number of used workers.

However, the experiments show, that the use of the connected components leads to an overhead which can not be balanced out by the improvements in the other sectors. In the future, one could try other ways to parallelize the SAKey algorithm. The main areas to target are the non key algorithm and the key derivation, as those two consume by far the most time.

A way to do this could be, to run the non key algorithm in parallel for each found clique. The drawback to this would be that, cliques in general do not have to be disjoint. Thus, it could

happen that the algorithm considers the same candidates multiple times for different workers, as running in parallel means each worker has its own sets for seen intersections, stored non Keys, and potential non keys. However, this loss could potentially be balanced out by using more workers and utilizing the fact that it runs in parallel.

7 Project Timeline

Table 4 shows the overall plan to finish the four segments of the project by week.

Week	Date	Work Description
1	By May 23	Project assignment
2	By June 7	Pre-presentation
3	By June 14	New paper assignment and reading Planning for implementation and meeting with intructors
4	By June 21	Implementation of final map
5-6	By July 5	Implementation of GraphX and Connected Components
7-8	By July 19	Implementation of Max Cliques for PNK
9	By July 26	Implementation of n-non Key derivation
–	July 27 - August 10	Break for final examination
10	By August 16	Implementation of almost key derivation Evaluation of the code and start report writing
11	By August 24	Project report submission
12	By August 29	Final presentation

Table 4: Weekly plan for the project completion

References

[1] Rina Dechter and David Cohen. *Constraint processing*. Morgan Kaufmann, 2003.

[2] Danai Symeonidou, Vincent Armant, Nathalie Pernelle, and Fatiha Saïs. Sakey: Scalable almost key discovery in rdf data. In *International Semantic Web Conference*, pages 33–49. Springer, 2014.