Distributed Big Data Analytics with Apache Spark and Apache Flink

T. Draeger, M. Gebert, B. Grasnick, M. Hegner, D. Heller, P. Jung, W. Müller, M. Perchyk, J. Pollak, R. Schüler, D. Wolf and M. Zabel

Hasso-Plattner-Insitute

For analyzing a vast amount of data, currently, platforms are popular that offer distributed and fault-tolerant processing capabilities. They typically follow the map-reduce paradigm and offer high-level operations, such as join or set operations.

However, there are few comprehensive studies of different use cases comparing major platforms in order to select a suitable framework for a specific type of problem.

We present a set of case studies for the use of Spark and Flink in different scenarios from the domain of distributed Big Data analytics and draw conclusions across those scenarios. We find significant differences favoring Flink in scenarios of iterative algorithms, whereas Spark suits memory-intensive challenges better. In addition, we conclude that when only one CPU per node is available, Spark runs faster than Flink. However, Flink profits more than Spark from scaling up to multiple CPUs per node.

1 Introduction

Nowadays, the amount of data that is generated and made available via the internet is increasingly high, clearly following an upward trend. The volume and variety of Big Data leads to new opportunities and perspectives for those capable of analyzing it. Unfortunately, the sheer size of the data poses new challenges for the analysis. The data is simply too big to be stored or too complex to be processed on a regular machine.

In order to cope with these challenges, cluster systems are used to store and process the data in a distributed and parallel fashion. Using systems consisting of multiple machines, also known as nodes, is called scaling out. Adding more CPUs to a board in order to parallelize data processing on a single machine is called scaling up. Scaling out is especially effective in order to solve bigger problems in the same amount of time. A cluster system usually consists of commodity hardware. Adding computational power by purchasing more machines is therefore simple and relatively inexpensive. While there is no real limitation for adding more machines, as it exists during scale-up, the speedup is affected because of necessary synchronization. In addition to that, parallelizing an algorithm for the use in such an environment poses a few challenges. First of all, distributed systems introduce an additional communication overhead that increases, the more machines are added to the system. Secondly, the speedup of

the solution depends on how well the problem is suited for parallelization and how well it fits the solution space of the chosen framework. Furthermore, the implementation needs to be realized using a potentially new framework and its specific expressions and features. Last but not least, proper load balancing is very important: Only if the workload is evenly distributed across the nodes, can parallelization be leveraged.

As already mentioned, different frameworks for parallel data processing in distributed environments exist. A very popular approach in that context is the MapReduce programming paradigm which is implemented by various frameworks.

2 Map Reduce with Spark and Flink

The MapReduce programming paradigm has been introduced in 2008 by Dean and Ghemawat [19]. It has been developed at Google as an additional abstraction layer to simplify the development of programs that distributively process large data sets.

Reoccurring tasks like parallelization, fault tolerance, and data distribution are handled by frameworks like Hadoop or Flink. Developers express computational tasks in the two atomic operations Map and Reduce.

The Map operation takes a key/value pair as input, executes a user-defined processing step on that data and emits a new key/value pair. The resulting values are grouped by their keys and handed over to a user-defined Reduce function which processes the sets of data to produce a typically smaller result, which is the output of the entire MapReduce operation.

The MapReduce concept, e.g. as implemented in Hadoop, has inspired the developer community to extend the paradigm and build frameworks on top of it. Two examples for such frameworks are Apache Spark and Apache Flink, which both pursue similar goals [34, 9].

Traditional MapReduce executes one operation at a time and writes the results back to disk. However, for complex computations with many steps this causes a large overhead in disk usage and also prevents any further steps from starting before previous operations have finished.

The Spark framework tackles those issues among other things using Resilient Distributed Datasets (RDDs) [34]. This concept partitions the data across the different nodes of a cluster in a fault-tolerant way. Furthermore, Spark uses lazy evaluation for optimization and it holds intermediate results in-memory to achieve better performance. In addition to these concepts, Spark is batch-oriented, meaning it operates on chunks of the data rather than on the whole data stream. In contrast to Hadoop, Spark also supports streaming, using a streaming library for real time processing [35].

Similar to Spark's RDDs, Flink uses Datasets to partition the data [9]. In contrast to Spark, Flink focuses on streaming, but it is able to deal with batch data as well. Moreover, Flink supports real-time processing and has its own

implementation for iterations. Likewise, Flink provides an execution plan optimization to achieve better performance [31].

Both frameworks provide additional libraries for specialized tasks, such as machine learning and graph processing. Both frameworks also offer APIs with operations beyond basic MapReduce, such as set operations and joins, SQL, as well as web interfaces to monitor the execution.

Our goal in this paper is to conduct a practical comparison of the two frameworks for a range of distributed Big Data tasks.

3 Experiments

We executed benchmarks for six different problems. We implemented them in Spark and Flink utilizing the specific features of both frameworks. All benchmarks are executed on a cluster with one master node and ten worker nodes. The master is a Dell PowerEdge R310 with 4(8)x2.66 GHz and 8 GB DDR3 RAM. All Workers are Dell OptiPlex 780 machines with 2x2.6 GHz and 8 GB DDR3 RAM.

We measured the scale out of our algorithms to multiple worker nodes. For up to 10 worker nodes, both Spark and Flink used one machine per worker, utilizing 1 CPU core and 4 GB of memory. When using 11 to 20 workers, we apply scale up and all 10 machines are used with one to two cores per machine.

To evaluate the scale out we use the relative speed up S, which is calculated as shown in Formula 1. T_n is the computational time using n cores.

$$S = \frac{T_1}{T_n} \tag{1}$$

To get a broad overview of the advantages and disadvantages of the two frameworks we examine the following six different problems. These problems bear either large underlying data or a high computational complexity.

In Section 4 unique column combinations in the field of data profiling will be examined. The next topic is related to graph mining – detecting highly connected subgraphs, which is described in Section 5. In the field of genetic algorithms, the traveling salesman problem is used to evaluate the two frameworks in Section 7. In Section 8 the detection of duplicated records within a data set is evaluated by finding users with the same movie taste on netflix. The same dataset is also used in Section 9 to predict movie ratings, where association rules are found using the apriori algorithm. The last data intensive task is relation extraction from plain-text documents with minimal human participation. The comparison of the usage of Flink and Spark for this use case is described in Section 6.

Then, we summarize findings about the comparisons of the two frameworks across all six tasks in Section 10. Finally, we draw a conclusion resulting from the evaluation of the different problems in Section 11.

4 Data Profiling

4.1 Motivation

One of the major tasks of data profiling is the finding of key candidates. Key candidates serve the purpose of uniquely identifying tuples in a given data table. In the following, key candidates will be called uniques or unique column combinations.

Definition 1 (Unique). *A unique, also called unique column combination, denotes a column or a combination of columns that uniquely identifies all records in a data set. In a column combination, there are no two records that have the exact same values.*

Definition 2 (Minimal uniques). *A minimal unique is a unique where there exists no subset that is already unique.*

Since keys have the characteristic of identifying single records, they are especially useful for indexing or data integration. Further fields of application are for instance query optimization, anomaly detection or data modeling. Particularly for query optimization it is helpful to know about the information content a certain column carries, since less redundancy in values leads to better selectivity.

Finding key candidates is a complex problem, in fact it has been shown that it is NP-hard [22]. For the data processing this means that the amount of candidates that will have to be checked grows exponentially. The exact number of candidates can be computed as follows, n being the number of columns: $2^n - 1$. In other words, for every additional attribute column, the number of candidates doubles.

Due to the complexity of the problem, tables that can be analyzed in a reasonable amount of time are comparatively small. Using parallelization and the computational power of additional machines, the goal is to analyze bigger tables with more columns in the same amount of time. For the task of finding key candidates, the parallel processing could yield a faster candidate checking, which is a key component of the algorithm. However, the complexity of the problem and the communication overhead will pose a challenge.

4.2 Algorithm

The detection of minimal uniques consists of two parts: the creation of an index structure and an iteration. During the iteration we create new column combinations and check their uniqueness.

With the help of an index structure the uniqueness of a column combination can be detected in linear time compared to the number of duplicates of a column. More concretely, we use a position list index (PLI).

Definition 3 (Position list index). *The PLI is a light-weighted data structure which helps to determine the uniqueness of a column combination. For each column combination, the PLI contains entries of sets with indices having the same value for this column combination. [25]*

Single column PLIs For the calculation of the PLI, we first map each row to a unique index (e.g. a row index). Due to the parallelism of the mapping, we can not only use a global index across all nodes. Otherwise different worker nodes would give the same index to different rows and consequently the index would not be unique. As a solution, Flink offers the possibility to get information about the current runtime context, i.e. the id of the current worker. In contrast those functions are missing in Spark. Therefore, we picked a random index with a minimal chance of a collision.

In the next step, we utilize the unique index to construct the PLI structure. As we defined in Definition 1, unique column combinations do not have redundant values. Therefore the PLI of an unique is an empty list. Hence, we can use this condition to filter unique and non-unique columns.

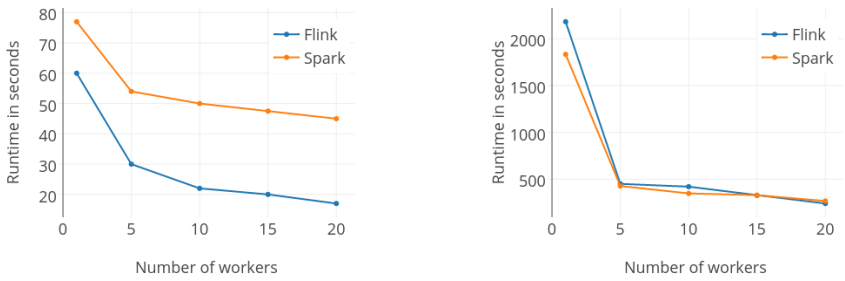
Iteration Then we iterate over the data set of candidates utilizing a bottom up approach. Meaning that at the beginning, the candidate list is composed of the non-unique single columns. In the second level of the iteration our candidates consist of combination of two single columns. In the third level, three columns build one candidate a.s.o. In Spark we implemented the iteration using a simple while-loop. In contrast Flink allowed us to utilize its own *iterate* function.

Candidate generation In the first step of each iteration level, new candidates are generated. We have not parallelized this part because all candidates as well as previous found minimal uniques are required. If we would parallelize the candidate generation, there would be a huge communication overhead: We had to broadcast all candidates and previous found uniques every iteration because these data sets change from iteration to iteration. The disadvantage of this approach is that small data sets have only a minimal communication overhead compared to the bigger computation part. Moreover, the candidate generation does not benefit from multiple nodes.

To generate candidates we use a prefix based algorithm. We take the prefix of the current candidates and combine them with each other. This results in a list with many duplicated new candidates. To complete the candidate generation, we have to remove these duplicates.

PLI intersection and uniqueness check After generating the next level candidates, we determine new PLIs for those candidates. This part is not parallelized for the same reasons like the candidate generation. For the PLI intersection the single column PLIs are needed. That is why we broadcast them once at the beginning. We combine multiple single column PLIs at a time in the following way: We iterate over the smallest PLI and for each duplicate we check whether it remains a duplicate. As soon as we find a duplicate, we break. In this case the candidate is non-unique, we do not need the complete PLI. Afterwards each candidate is mapped to a boolean flag, indicating whether it is a unique or not.

Then a new iteration continues. If no new non-unique candidates are generated, the iteration stops. After the iteration, we filter and collect all minimal non-uniques.



(a) ncvoter-1k (b) ncvoter-10k

Fig. 1: Scale out performance for Flink and Spark on different sized data sets

4.3 Benchmarks and Evaluation

All tests have been run on the cluster system described in 1. In order to test the scale out behavior, the algorithm has been tested with 1, 5, 10, 15 and 20 workers. Up to 10 workers, each worker represents a CPU on separate machines. Since there are only 10 machines in the cluster, when using more than 10 workers, every additional worker will be added to one of the machines where only one CPU has been used op to that point. All evaluations have been run on a subset the ncvoter data set, which gives the numbers of voter registrations by county in North Carolina. It contains personal information such as name, gender, address and others.

Scale out performance On both platforms the algorithm's scale out performance is quite good. Adding more machines to process the same amount of data leads to a visible decrease in runtime, as can be seen in Figure 1. It is noticeable that the first added workers have the highest impact. Between one and five work-

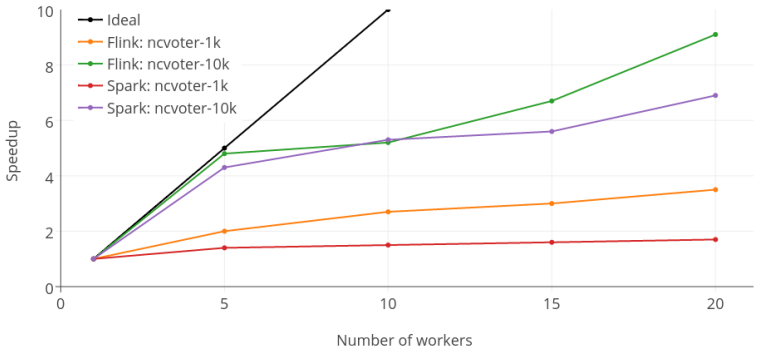
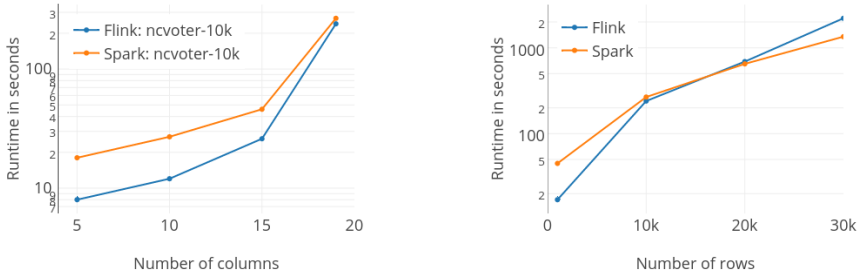


Fig. 2: The relative speedup shows the speedup factors that can be reached using additional workers

ers, the scale out is approximately linear. Adding more than five workers will still lead to a decreased runtime, but the scale out factor is far below linearity.

Speedup The relative speedup, also known as speedup ratio, is illustrated in Figure 2. It is computed by dividing the runtime of the serial execution by the runtime of the parallel execution. The black line denotes a linear speedup, the ideal result. As mentioned before, for up to five workers, the speedup is very close to linear. Afterwards it drops down a bit, but keeps increasing constantly. This result conforms with the fact that runtimes decrease the more workers are added to the execution. Therefore, it can be assumed that adding more workers would lead to an even higher speedup. Eventually a limit will be reached due to an increasing communication overhead within the cluster, but for the present cluster system with 20 workers, this could not be observed yet. The highest speedup measured is of factor 9 for Flink, and factor 7 for Spark. Flink shows the better speedup in all tests. For small files the speedup factor is much lower and appears to stagnate earlier. This might be to the comparatively high overhead of a cluster and the overall setup of the framework's environment for a parallel processing in contrast to the relatively low execution times. Accordingly, we focus on data sets where the setup time and the overhead do not affect the speedup so much, as it is the case in the ncvoter data set with at least 10,000 lines.

Scaling the number of columns and rows Scaling the number of columns leads to an exponential growth in the number of candidates. The illustration 3 shows that the same exponential growth can be observed in the runtime. This is expected behavior as the problem is very complex. The tests have shown that Flink performs better until a certain size of the data set, then Spark takes over and appears to be the faster solution. One reason is that Flink implemented its specialized iterations. Therefore for a small amount of data and a high number of iteration, it is faster than Spark. But as soon as the data set gets bigger, the memory management becomes more important and Spark wins.



(a) Scaling with the number of columns (b) Scaling with the number of rows

Fig. 3: The increase in columns and rows leads to an exponential growth in the runtime

The same effect can be observed when scaling the number of rows. For the given data set, at around 15,000 rows, Spark becomes the faster solution. However, both show an exponential growth in runtime. Adding more rows is especially bad for the runtime if the values that are added to existing columns are highly redundant. Adding redundant values leads to an increase of the cluster sizes of the position list index. The bigger the clusters are, the longer it will take to find a unique column combination where there are no more intersections possible. If only non-redundant values were added, the additional rows should not affect the runtime much, as they would not appear in the position list index. Since ncvoter is a table about voters in the counties of North Carolina, it is very likely that due to the spatial locality there are many redundancies in the values.

5 Graph Mining

5.1 Motivation

Graph mining deals with extracting information from nodes and the edges that connect them. One use case is attempting to find highly connected subgraphs. These may signify groups of friends or families in social networks or be relevant for advertising purposes when looking at website interlinks [28].

We look at trusses, a relaxation of cliques, as introduced by Cohen in [16]:

Definition 4 (Truss). *A k-truss is a non-trivial, one-component subgraph such that each edge is reinforced by at least k–2 pairs of edges making a triangle with that edge.*

Definition 5 (Maximal Truss). *A maximal k-truss is a k-truss that is not a proper subgraph of another k-truss.*

This problem's solution set size is $2^{|V|}$, where V is the set of vertices. Thus, any naive approach will have at least an exponential runtime. Since many relevant data sets contain multiple millions of vertices and edges, it becomes clear that no naive approach will perform sufficiently well.

In [16], Cohen proposed an algorithm for calculating k-trusses in polynomial time. Later, he adjusted this algorithm to run on distributed systems with MapReduce [17]. We implemented this last algorithm in Apache Spark and Apache Flink, while making use of both frameworks' individual strengths.

5.2 Algorithm

To find k-trusses, we first calculate the degrees of every node in our graph, which allows us to filter all nodes whose degree is not sufficient enough for their edges to be contained in k-2 triangles. This filtering step significantly reduces the amount of edges for every non-trivial value of k and is thus worth the additional computational effort. Following this, we calculate all triangles in our remaining graph, see below.

The next step is iteratively removing all edges which are not contained in at least k-2 triangles. For this purpose, we must first annotate each edge with its triangle count. After removing those edges, we need to recalculate the triangle counts, since removing edges can also result in removing triangles, so that remaining edges may no longer be contained in sufficient triangles. In Spark, we solved this by using a simple for-loop, while Flink allowed us to utilize its specialized *iterate* function. When no more edges are being removed, we end the iteration and find the subgraphs that are still interconnected, as described later in this section.

If we do not want to simply find all trusses of a certain size, but instead the maximal truss(es), we must run the algorithm described above multiple times, as described at the end of this section.

Triangle calculation Since finding trusses in a graph relies on the triangle counts of its edges, our algorithm needs to find all triangles in a graph. To do so, we first look for so-called triads, triangles with a missing edge. For this purpose, we key all edges by their node with the smaller degree. We then perform a self-*join* on these edges, which results in a set of triads. This triad set must be filtered to remove duplicate triads and ones which consist of two copies of the same edge. What remains after this *filter* step is one triad per potential triangle. We then key these triads after their missing edge and join this list with all edges to remove all triads whose missing edge does not actually exist. The result is a list of all triangles in the graph.

Connected Components After finding all edges that are contained in a truss the different graph components need to be detected. There are several MapReduce approaches available. For example, the graph mining system PEGASUS [26] detects components via iterative matrix-vector multiplication. The most popular approach was introduced by Cohen in [17]. It is a repeated zone propagational algorithm. We adjusted Cohen's approach to function with Spark and Flink as follows.

The basic idea is to assign a zone (represented by the smallest vertex ID) to each vertex. Then, these zone are merged iteratively along the edges until no further merges are possible. This originally required three MapReduce steps as described by Cohen [17]. Afterwards, each subgraph will have its own zone.

In Spark we followed this main idea. We implemented some of the reduce functions with the *join* operator or *reduceByKey* and *map*. In addition, some intermediate results are cached to avoid multiple redundant calculations.

The Flink implementation uses *delta iterations*. The solution and working set are a vertex-zone map which will be updated after each iteration. This eliminates the last MapReduce step, since this updating is done internally by Flink. In general, the reduce parts are implemented using *joins*.

Maximal Truss As defined in Definition 5 a maximal truss is the truss with the highest k-value. Our algorithm finds the highest k-value due to systematic testing. Given a starting value by the user, k is increased if a truss was found or

decreased if none were found. The increasing and reducing is done according to a binary search strategy.

The trusses with higher k are included in those with lower k, as described in Section 5.1. Using this property, the maximal truss calculation can reuse the results of a previous iteration which significantly reduces the amount of processed data.

The Spark implementation is straight forward since we merely need to count the found trusses. In contrast, Flink has a different *count* implementation. Count is a Data Sink, which means that the Flink plan is executed immediately. The truss result is not available afterwards, if it was not written to disk. A Flink *iteration* could not be used since nested iterations are not supported yet and the truss calculation already relies on iterations. To deal with these limitations in Flink we had two possibilities: Not reusing previous results or saving the intermediate results on disk. A short benchmark showed that writing to disk is about 3-4 times faster, so we use this second solution.

5.3 Benchmarks and Evaluation

We evaluated the previously described graph algorithm with a Wikipedia data set of 2007 [1]. This data set contains the links between the different articles in the English Wikipedia [28].

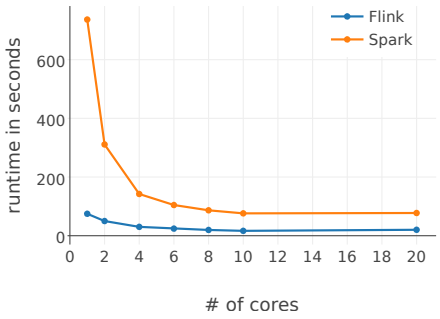
Furthermore we only used edges which are available in both directions. When using all edges, large hubs (e.g. twitter users with millions of followers or websites that are frequently referenced) might indicate relations which do not actually exist.

Unless otherwise noted, we use the setup described in Section 3 and a k value of 20.

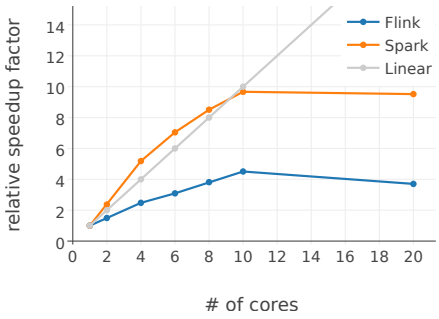
Scaling to multiple workers Our benchmark results with the conditions described in Section 1 showed that both implementations, Spark and Flink, do indeed scale out. Figure 4a shows the scale-out with different numbers of worker nodes for Flink (blue line) and Spark (orange line). In this scenario, Flink is – depending on the number of workers – four to ten times faster than Spark. This time difference is likely the result of the different iteration implementations. The optimum appears to be reached at 10 cores. This is due to the test setup we used, since we only use a maximum of ten worker nodes and adding more cores will only have worker nodes switch from one to two cores. This adds no speed-up to the algorithm, because it is data intensive, rather than calculation intensive.

The relative speed-up depending on the number of workers is shown in Figure 4b. The speed-up factor is calculated as shown in Formula 1. The grey line shows how a perfect linear scale-up would look like. It appears that Spark has a more than linear speed-up. This can be explained with its memory management: Using only one worker, the data does not fit in main memory and needs to be spilled on disk and be read again later, which is a rather slow process. Since the speed-up factor is calculated with one worker as a basis, adding more workers (and thus

[1] http://konect.uni-koblenz.de/networks/wikipedia-growth



(a) Runtime scaling per Cores calculating truss in Spark and Flink

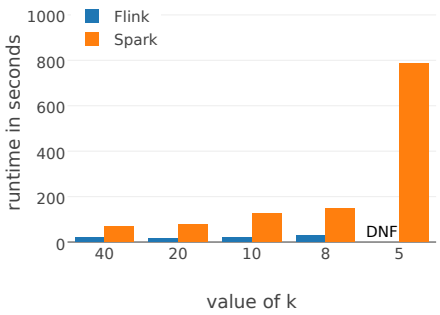


(b) relative speedup calculating truss in Spark and Flink

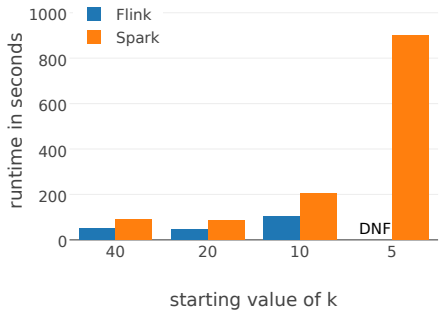
Fig. 4: Benchmarks for truss calculation depending on the number of cores, which represents worker

more combined memory), results in a more than linear scaling until all data fits into main memory. Flink has a slow but steady speed up, as no intermediate results need to be written to disk.

Scaling k-value We also evaluated the two algorithms regarding their scalability of the k value. This gives an impression of how well big data can be handled. Figure 5a shows that Flink is faster than Spark for higher k values. This means that Flink (due to prefiltering) is the better choice for small data sets. When the data does not fit into main memory, Flink continues to run with minimal computational load, not producing any results, even after an extended period of time. This behaviour has already been reported to the Flink community. In



(a) Runtime of calculation truss with different k values



(b) Runtime of calculation of maximal Truss with different starting values for k

Fig. 5: Benchmarks for truss and maximal truss calculation with different k values; DNF stands for did not finish

comparison, Spark is a little slower, but performs much better for lower k values. It can handle a full main memory very efficiently.

Figure 5b shows our benchmarks for attempting to find the maximal truss. For small data sets, Flink is still faster than Spark, although less so. While Flink (unlike Spark) needs to write the intermediate results of each truss calculation on disk, its speed advantage when calculating single trusses outweighs this. When dealing with smaller k and thus larger data sets, Flink runs into the same issues it did when calculating individual trusses.

6 Text Mining

6.1 Motivation

The World Wide Web contains a huge and growing amount of unstructured texts. These documents are formulated in a natural language but hold valuable structured information, such as relations.

Relations are defined as subject-predicate-object triples or, more generally, two entities that are connected by a specific relation type. Examples include Organization (is based in) Headquarter, Soccer Player (plays for) Soccer club, and City (is capital of) Country.

Relation Extraction is an important task in the field of Natural Language Processing. It aims to retrieve tuples of entities (that are in a given relation) from unstructured text and to return them in a structured form. The resulting relations can fill structured data sources and knowledge bases like *DBPedia* or *Freebase* [12, 14]. Additionally, such relations can help answering complex user queries. For instance, the query "*Who is the spouse of Germany's Chancellor?*" can be transformed into "*Y (is spouse of) X*" and "*X (is Chancellor of) Germany*", both of which are queries for simple binary relations.

Due to the large number of texts in the web, a manual extraction of relations is not feasible. Therefore, several machine learning techniques have been applied to the task [13]. Supervised learning algorithms formulate the extraction task as a binary classification problem. They need to be trained with a large sample of labelled training instances (i.e., entity tuples both with and without a real relation). Semi-supervised algorithms on the other hand only require a few manually labelled seed samples (i.e., seed tuples or extraction rules) to launch the training process. Then, they automatically generate new tuples with each iteration and utilize those they are confident with to train the next iteration. This approach is called Bootstrapping and the most notable algorithm of this kind is *DIPRE* by Brin et al. [15]. The *Snowball* algorithm whose implementation is described in this article builds on *DIPRE*.

6.2 Algorithm

Snowball is a bootstrapping approach for relation extraction that was introduced by Agichtein et al. in 2000 [7]. Its goal is the extraction of structured relation data from plain-text documents with minimal human participation.

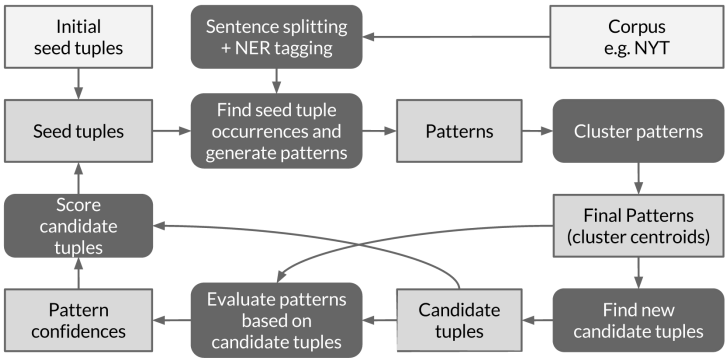


Fig. 6: Schematic overview of the Snowball Algorithm

An overview of the algorithm is shown in Fig. 6. Snowball is given two inputs: the corpus of unstructured texts and a small set of initial seed tuples (typically 5-15). Taking the organization-headquarter example, the seed tuples would be well-known organizations with their correct headquarter. Snowball is an iterative algorithm as can be clearly seen in the figure. Prior to the first iteration, the corpus texts are split into sentences. Moreover, named entities such as organizations, locations, persons, etc. are annotated by a Named Entity Recognition (NER) algorithm.

The first step in each iteration is to find all occurrences of seed tuples in the corpus (i.e., sites in the text where both entities of a seed tuple appear close to each other). From each occurrence a pattern is produced. Snowball patterns are 5-tuples. They consist of the two entity types (e.g., organization and location) and the three surrounding text contexts (left, between, right). Each context is represented by a vector of weighted terms. The weight associated to a term reflects the importance of the term for the context.

As the set of resulting patterns possesses a high level of redundancy, Snowball clusters the patterns in the next step. The resulting cluster centroids are patterns themselves and can be applied to the text corpus. When they match a text segment, they produce a new candidate tuple. Over the entire corpus, numerous candidate tuples are found and need to be validated. For that purpose, two kinds of confidences are calculated. Firstly, pattern confidences are computed based on the set of candidate tuples that each pattern generated. For each candidate tuple that contains an organization from the seed tuples, the correctness of the associated location is checked. Thus, the performance of each pattern can be measured. Secondly, each candidate tuple is assigned a tuple confidence. These confidences are calculated from the confidences of the patterns that generated the tuple. A candidate tuple from numerous high-confidence patterns receives a higher score than a tuple that was generated from only one low-confidence pattern.

Finally, a confidence threshold is applied to obtain the best candidate tuples. These are added to the set of seed tuples for the next iteration. Snowball is expected to find less and less new tuples in each round and halts when no new candidate tuples exceed the threshold.

6.3 Implementation

We implemented the Snowball algorithm both on Apache Spark and Apache Flink. Five main steps in the algorithm can be easily parallelized because they operate on individual paragraphs or sentences independently. Other steps, however, cannot be formulated that easily. Appendix B gives more details.

Distributed pattern clustering As described in Sec. 6.2, the newly generated patterns need to be clustered. For this purpose, the authors of *Snowball*, Agichtein et al., suggest a simple single-pass clustering algorithm [7]. That algorithm passes over the patterns only once and is described in Appendix A.

The challenge in implementing this relatively simple algorithm on Apache Spark and Apache Flink was that the clustering should happen in a parallel fashion. Before the clustering, the patterns are distributed on the nodes. For large data sets, their number can easily exceed 100,000 which makes a centralized clustering unfeasible.

This led us to explore a two-level approach (illustrated in Appendix C): In a first step, the patterns are clustered in parallel on their respective nodes. Next, the resulting cluster centroids (significantly less than original patterns) are all moved to a single node where a second clustering step merges the clusters to final patterns. For subsequent steps, such as the candidate tuple search, the final patterns are later broadcasted back to the individual nodes.

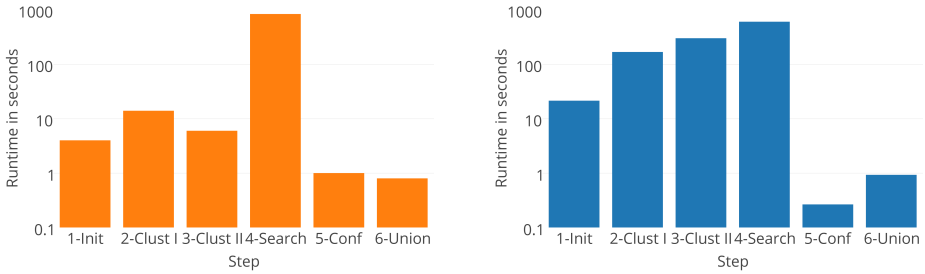
Our clustering utilizes cosine similarity between the term-weights of the patterns as distance metric. In contrast to many other algorithms, such as k-means, it does not require the number of clusters k as input. That is beneficial as the number of clusters in the text corpus cannot be known beforehand. It depends both on the supplied seed tuples and the text corpus.

The downside of the approach is its non-deterministic nature. Depending on the order of input patterns and their distribution on the different nodes, the clustering yields different results. This gives rise to large deviations in runtime and number of result tuples.

6.4 Benchmarks

To evaluate the performance of Apache Spark and Flink, we measured the runtime of Snowball on both platforms. All benchmarks were run on the cluster system described in Sec. 3. As test data set, the years 1987 to 1990 of the New York Times (NYT) archive were chosen. The NYT archive is a text corpus which comprises articles from 1987 to 2007 with in total more than 50 mio. sentences.

Snowball is an iterative algorithm. When measuring the runtimes of the individual iterations, we observed that runtime increases linearly with iteration number. Furthermore, the number of new tuples found per iteration decreases consid-



(a) Apache Spark (b) Apache Flink

Fig. 7: Runtime of individual parts of the Snowball pipeline (10 cores, 4th iteration, 1987-1990). Note that runtimes are on log scale.

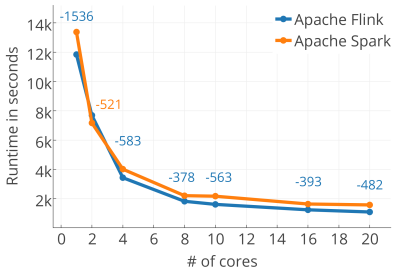
erably after 4 iterations. To achieve meaningful results, the following benchmarks were therefore performed by running Snowball for exactly four iterations.

Runtime of individual parts Figure 7 shows runtime measurements for the individual parts of the algorithm. The parts are: 1 - Initialization + Search for raw patterns, 2 - Clustering I, 3 - Clustering II + Search for text segments, 4 - Search for candidate tuples, 5 - Identifying high-confidence tuples, and 6 - Union with existing seed tuples.

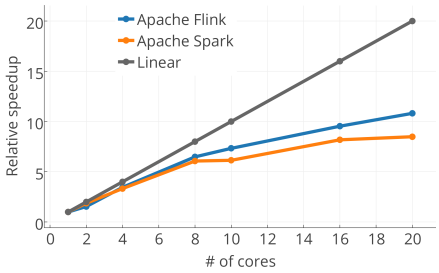
It can be seen that most time is spent searching candidate tuples - a task that involves applying each clustered pattern to every text segment. This step makes up approximately 97% and 54% of the total runtime on Spark and Flink, respectively. On Spark, its dominance on runtime seems to be more distinct than on Flink where the other steps took longer. On Flink, however, measuring the individual parts proved to be very difficult. To gain the measurements, the program had to be split up into parts, which made it impossible for Flink to use streaming. Therefore, the results on Flink might not fully reflect the real workload of each pipeline step.

Total runtime To determine how well both implementations scale with a high number of workers, we measured runtimes for 1, 2, 4, 8, 10, 16, and 20 cores. It was found that generally both implementations possessed a very similar runtime as can be seen in Fig. 8a. The likely reason for this is that Snowball spends most of the time in the search for candidate tuple step. This step is not formulated with framework-specific operations but in basic Java code. Therefore, the properties of the frameworks only have a limited impact on the runtime.

However, Apache Flink is slightly faster than Apache Spark in all but one configuration (with 2 cores). One explanation for this small difference can be found in the distributed pattern clustering algorithm. Its results depend on the distribution of the input patterns on the workers. We observed that the algorithm produces considerably less clusters on Flink compared to Spark, which naturally reduces runtime. This is most likely caused by the different approaches that



(a) Total runtime of Apache Spark and Apache Flink with runtime differences as labels (color indicates faster framework)



(b) Relative speedup of Apache Spark and Apache Flink

Fig. 8: Benchmarks depending on the number of cores (4 iterations, 1987-1990, average of 3 runs per configuration)

Spark and Flink use to determine the number of partitions an input data source is divided in (see [4] and [1], respectively).

Relative speedup Figure 8b shows that both frameworks scale well with increasing number of cores. This is not surprising as the runtime-dominating search for candidate tuples can be parallelized very well. Apache Spark and Flink reach a maximum speedup of 8.5 and 10.8 respectively. Until 8 cores, the relative speedup stays relatively close to linear but slows down considerably for more than 10 cores. This can be explained by the architecture of the benchmark server which possesses 20 cores but only 10 workers.

7 Machine Learning

7.1 Motivation

The Traveling Salesman Problem (TSP) is a NP-complete optimization problem which is well known in computer science [23]. The shortest path connecting all cities of a given set as a round trip is the desired result. To find the optimal route or a good approximation many TSP solvers exist, for instance Concorde [2] which already solved a 85900 city instance [18, p. 161].

Variants of the TSP occur in a broad range of domains such as drilling operations, genome mapping in biology and navigation systems [18, p. 44]. However, the optimization criteria tend to be a lot more complex than just the total length of the path. For example, fuel cost, different types of roads and mandatory breaks are all relevant factors in real navigation scenarios. While some of those aspects can easily be reduced to TSP (e.g. asymmetric journey times in two directions [10, p. 126]), others can only barely be modelled as TSP instances. A more generic approach is required which can handle any kind of parameter determining how suitable a solution is. We use the concept of Genetic Algorithms [30] and

show how it can be applied to the Travelling Salesman Problem (Section 7.3) and how it can be distributed using Spark or Flink (Section 7.4).

7.2 Genetic Algorithms

A Genetic Algorithm (GA) is a heuristic method which searches solutions to a given optimization problem. The quality of a solution is expressed in a single outcome measure which has to be maximized. GAs imitate the process of natural evolution, so the measure is traditionally called fitness and the three main phases are referenced by their biological counterparts: mutation, crossover and selection.

Initially, a random set of valid solutions for the problem is generated. The set is called a population, while the solutions are called individuals.

Mutation is the process that slightly changes each individual randomly.

Selection is the phase in which two random individuals are chosen to be recombined into a new individual for the next generation. The method of selecting the pair determines the evolutionary pressure.

Crossover is the method by which both selected individuals are recombined. A good crossover function should result in valid solutions that combine traits of both parents.

Given an initial population, mutation, selection and crossover are applied in a loop. Due to the evolutionary process of modifying existing solutions and recombining only good ones, the average fitness increases over the time. This loop can be stopped at any time and the individual with the highest fitness can be used as an approximate solution to the initial problem.

7.3 Modeling TSP as Instance of Genetic Algorithms

A path and its length are defined as follows:

Definition 6 (Path). *Given a set of cities* $\{C_1, \ldots, C_n\}$*, a path is an ordered list* $\langle C_1, \ldots, C_n \rangle$*.*

Definition 7 (Length of a path). *With the function distance* : $City \times City \to \mathbb{R}$*, giving the distance between two cities, define length* : $Path \to \mathbb{R}$ *as* $length(\langle C_1, ..., C_n \rangle) = distance(C_1, C_2) + distance(C_2, C_3) + \cdots + distance(C_n, C_1)$.

Using these definitions, we present a GA taking a set of cities and the distance function as input and searches for a path that has a small *length*.

Individual and Fitness Function A valid path connecting the given cities is considered as individual for the GA.

Definition 8 (Fitness). *Let* $fitness : Path \to \mathbb{R}$ *be defined as* $fitness(p) = \frac{1}{length(p)}$.

The fitness function could be much more complex, embedding any number of optimization criteria. This makes the GA approach so flexible. We concentrate on the classical interpretation, however all of the requirements mentioned in the introduction could be applied here.

Mutation As long as a random variable is below the mutation threshold, one of the mutation methods is used. This means each individual could be mutated once, more than once or not at all. The used mutation methods are:

1. Interchanging two randomly chosen sequences of the same length. For instance, $\langle C1, C2, C3, C4\rangle$ could become $\langle C3, C4, C1, C2\rangle$.
2. Reversing the order of cities in a range. We often observed the case that a whole sequence of cities is traversed in reverse order, as shown in 9. The resulting solution is near the optimum, but it is very hard to reach the actual optimum, since each single exchange mutation would decrease the fitness. This kind of classical local maximum poses a general challenge to all kinds of heuristic optimization [20]. This method of mutation allows the GA to solve these often occurring problems with one step. For example $\langle C1, C2, C3, C4, C5\rangle$ could become $\langle C1, C5, C4, C3, C2\rangle$. Experiments showed that adding this second kind of mutation increased the overall performance tenfold.

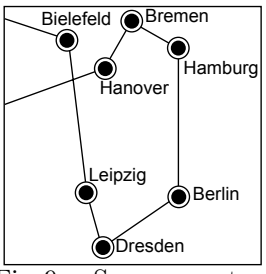


Fig. 9: Sequence traversed in reverse order

Selection We use the Roulette Wheel Selection, which assigns each individual a probability to join a crossover proportional to its fitness (see also [21, p. 63]).

Crossover The crossover step between two paths removes the last k elements of one of the paths and adds them in the order they appear in the other, with k being randomly chosen. This method guarantees a valid new individual that contains each city once.

7.4 Implementation on Apache Spark and Flink

The distribution of the GA on a cluster can generally be seen as distributing the population over multiple "islands" where each cluster node is an island and has its own population that evolves independently [32]. There has to be some kind of exchange between the islands though, to share the information of each island population. The frequency of exchange however is a tradeoff: more exchange leads to a more unified whole population but also increases the communication effort.

Generally, the critical resources are computation time and communication overhead. An optimal population for GAs should only contain a few hundred individuals. Those are in our use case efficiently stored as integer lists and thus, the memory consumption of our GA is negligible. Furthermore, the distances between all cities are precalculated and broadcasted once so the fitness calculation is also very fast.

Iterations The three steps mutation, selection and crossover are implemented as a single partition map operation. In practice, exchanges are only done every n generations to minimize the exchange overhead.

In Spark, a `StackOverflowError` occurred when too many evolution operations were chained. The cause of this was Spark's task serialization which serializes the task chain recursively and thus quickly ran out of memory when we tried to chain thousands of generations. We solved this by merging the n evolution cycles between exchange steps into a single partition map operation.

For Flink, our algorithm uses Flink's iteration concept.

Exchange In Spark the exchange was implemented by simply using the built-in `coalesce` operation, which supports a full shuffle of all objects.

Flink did not support such a shuffle directly. We implemented a full shuffle by creating a custom partitioner that partitioned each individual with the round-robin algorithm. We also implemented an alternative exchange method that only selects a few individuals for exchange instead of all of them.

7.5 Benchmarks

Method The TSPLib [5] serves as main point of reference. It includes a collection of TSP problems together with proven, exact solutions up to a size of 666 cities. Our quality measure is the relative fitness:

Definition 9 (Path). *Given a path and the optimal_path for a problem, we define* $relative\_fitness(path) = \frac{fitness(path)}{fitness(optimal\_path)}$

The shown results are taken from a problem containing 202 African cities, we also applied the experiments on a 96 city instance and the results for scaling and the comparison of Spark and Flink are similar. All benchmarks were run with a population size of 250, parallelism of 10 and an exchange frequency of 10000 if not stated otherwise. All given numbers are the median of at least 4 measurements.

Exchange The left diagram of Fig. 10 shows that in general more exchange between the islands increases the effectiveness of the GA per generation. If you consider the actual runtime the picture is a different one. The right diagram shows the same benchmarks scaled by time and it shows that with too much exchange the communication overhead outweighs the gain. The optimum for the round-robin exchange is at a lower frequency because each round-robin exchange has a lot more overhead than an exchange with selected individuals. On the other hand a round-robin exchange is a complete shuffle and thus more effective per exchange. Altogether the round-robin exchange proved to be the more efficient exchange method.

Distributed scaling The diagrams of Fig. 11 show how the different GA implementations scale with more nodes in the cluster. An important fact to keep in mind is, that the cluster had only 10 physical machines. The measuring points for parallelism of 20 is thus with two parallel cores per machine. The diagrams show that without an exchange the performance only scale logarithmically. With an exchange, spark or flink, the algorithms scale very good. The superlinear scaling is probably an artifact resulting from the large 202 city problem. Our measurements were still in the phase of logarithmic fitness growth where the logarithmic

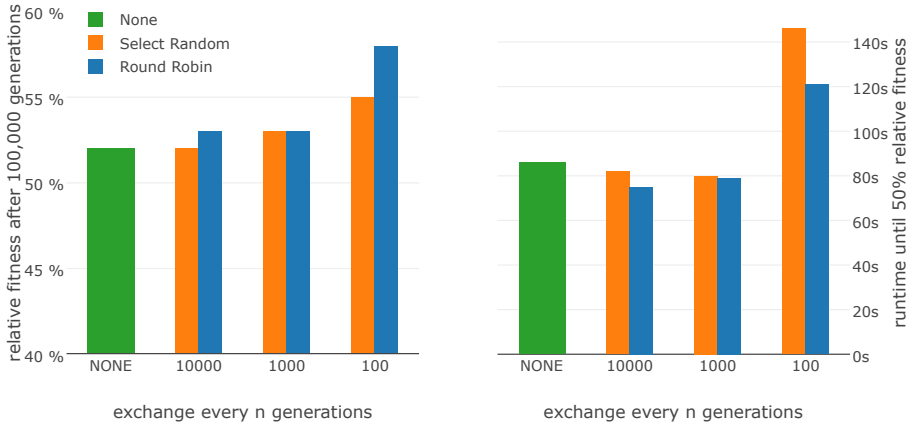


Fig. 10: Performance per generation and time with different exchange methods and frequencies

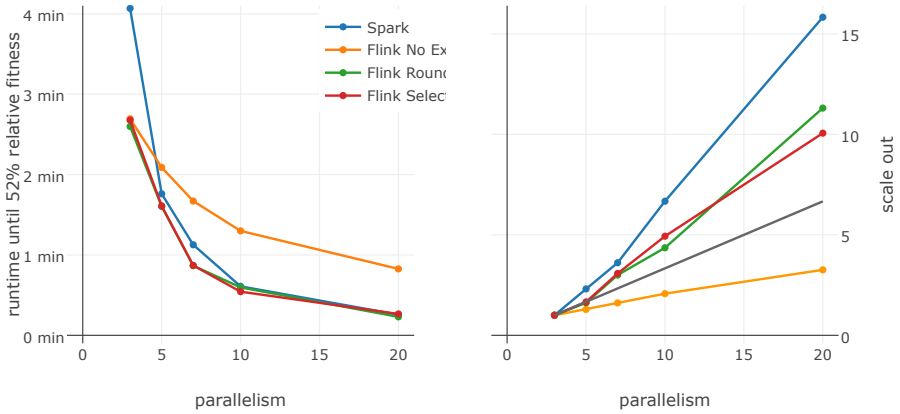


Fig. 11: Performance scaling with different numbers of parallelism

function grows faster than the linear one. If you would run the GA a longer time the scaling should become nearly linear. The evolution itself should scale perfectly, because twice as many nodes means that each node only has to evolve half the number of individuals. The time needed for the exchange should also scale linearly, because each node only needs to send half as much individuals to the other nodes.

8 Data Cleansing

8.1 Motivation

Finding similar tuples in a large data set without missing any pair of similar tuples is a common challenge for recommender systems or data cleansing tasks.

One algorithm for finding similar tuples guaranteeing an exact result is the set-similarity join. While it is a problem of quadratic complexity $\mathcal{O}(n^2)$, there exist a range of optimizations that achieve a practical run-time [11, 33], assuming the algorithm is running on a single node. For tackling this problem in a distributed computing scenario, Vernica et al. [29] proposed an algorithm following the map-reduce paradigm. In the following, we discuss an implementation of this algorithm on Spark and Flink – platforms that allow higher-level operations than the basic map and reduce used in the mentioned work by Vernica et al.[29].

We use the Neflix Prize data set [3] in which we try to find users with similar movie preferences.

8.2 Algorithm

The goal of the presented algorithm is to find the *complete* and *correct* set of similar data entries in large data sets. In the given scenario, this means to find the exact pairs of users having a sufficient number of rated movies in common. As a measure for similarity we use the Jaccard set similarity of the movies as defined in 10. We do not consider the rating value, but only the fact that a movie has been rated.

Definition 10 (Jaccard Similarity). $similar(x, y) = \frac{|x \cap y|}{|x \cup y|} \leq \theta$

We are looking for users who have the majority of movies in common, so the default threshold for our experiments is $\theta = 0.9$. For illustration purposes, let us consider the sample data set presented in Figure 12a.

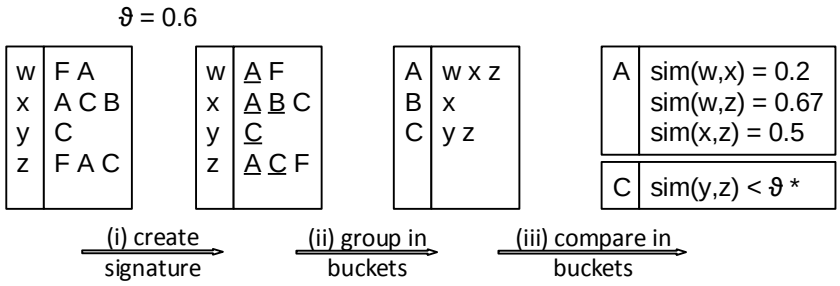


Fig. 12: Steps of the algorithm: (i) Movie popularity is counted to create selective signatures. Ratings are sorted according to popularity, underscored ratings are used for signature. (ii) Users are grouped into buckets by signature. (iii) Candidates are compared inside the buckets. * Users y and z don't have to be compared due to size-based filtering as explained in Section 8.2

A naive approach would require comparing every user with every other user in the data set, which is not a viable solution for large data sets because it lies in the complexity class $\mathcal{O}(u^2)$, where u is the number of users present in the data set – in our case 480,189. Instead of comparing each user with each other user, the idea is to compare only those having certain aspects, namely movie ratings, in common.

Our algorithm follows the approach for executing a similarity join as proposed by Vernica et al. [29] and is composed of the steps displayed in Figure 12, which are discussed in detail in the next paragraphs.

(i-ii) Gather statistics, create signatures and group users in buckets At first we gather statistics of the movie popularity in the data set as suggested in [29]. This knowledge is used for a global ordering of the ratings. To maximize the selectivity of the signature, each user's ratings are ordered by increasing popularity so that the movies with the least ratings come first. That way we decrease bucket sizes, as the algorithm tends to produce buckets from movies with a small number of ratings. This step has an algorithmic complexity of $\mathcal{O}(r)$, where r is the number of ratings, in our data set 100,480,507.

To reduce the number of necessary comparisons, the goal is to form groups of candidates, buckets, with as little as possible similarity candidates while still producing a correct and complete result. A means suggested by [33] is to use a prefix of the ordered user data as a signature and group the users into buckets by that signature. The prefix concept for Jaccard-based comparisons works as follows: Two Users, x and y, are represented by the globally stable ordered list of movies they rated. Users x and y match with a Jaccard-similarity $sim \geq \theta$ if and only if they have at least one of the p first movies they watched in common. For a user x: $p = |x| - \lceil |x| \cdot \theta \rceil + s$, where s is the signature size[2].

It is thus sufficient to place users into buckets according to their prefix only. This step has an algorithmic complexity of $\mathcal{O}(u)$, where u is the number of users, which is 480,189 in the Neflix Prize data set.

With the prefix size usually being larger than one, users appear in multiple buckets. To keep the data duplication to a minimum, we execute the bucket generation only with the user-IDs, not with the ratings. Ratings data is attached back to the user by a join operation just before the comparison, that way we can filter buckets of size one before they take up space unnecessarily.

(iii) Compare Users in Buckets After having formed the buckets of possibly similar users, for all such pairs the similarity (see Definition 10) of their movie taste is computed. This pair-wise comparison has an algorithmic complexity of $\mathcal{O}(b \cdot s_b^2)$, where b is the number of bins and s_b the size of the bins. As we will see in Section 8.3 this step dominates the run-time of this algorithm as it is the only one with quadratic complexity. Yet, our experiments show that the size of the bins is far smaller than the number of users. This is consistent with the related literature (e.g. [33]) as they also observed a growth in execution time quadratic

[2] Throughout our experiments $s = 1$, but later on we will also discuss the effect of using longer signatures.

to the input size, but with a dampening factor allowing this approach of forming candidate buckets being practical for large data sets. Additionally, because bins are distributed among the workers, we argue that for a sufficiently high number of workers the number of bins b is less critical than the size of the bins s_b.

To further decrease the number of comparisons we employ a technique called *size-based filtering* by Arasu et al. [11] which prunes only by the number of ratings each user has made. In the given example (Figure 12), users y and z do not have to be compared because with $\theta = 0.6$, user z can only be similar to users that have at least $\lceil |z| * \theta \rceil = 2$ ratings. y only has one rating. Thus, we use that size filter to save unnecessary calculations of similarity leading to a further reduction of pair-wise comparisons.

8.3 Benchmarks and Evaluation

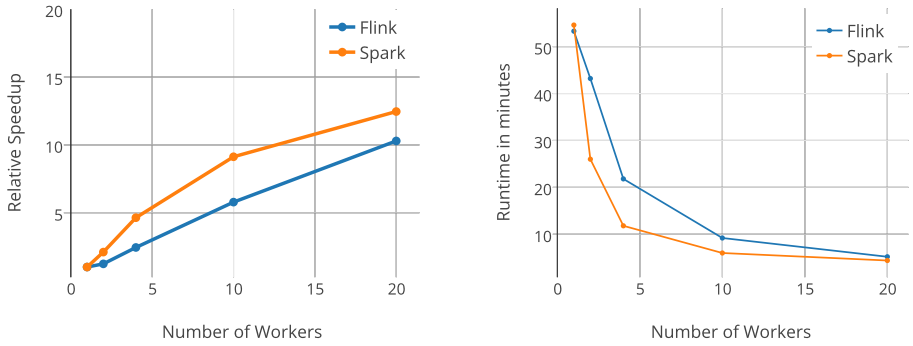
In the following we give an idea of how the algorithm performs in different settings. We look at scale-out performance, bottlenecks and compare run times of the implementations in Flink and Spark.

Scaling to multiple workers Figure 13a compares the run-times of the Flink and Spark implementations for 1 up to 20 workers on our test cluster (as described in Section 1). The data set for all executions were the ratings of the same 10 % of all users. The shown run times are means from 5 iterations. The execution times range from 5.2 minutes with 20 workers to 53 minutes, when run on a single worker. The differences between Spark and Flink are rather small at 1 worker and 20 workers, more noticeable for values in-between. It should be noted that for up to 10 machines one core is used per dual-core machine and when using 20 workers all machines use their available two cores. Therefore the data suggests that Spark has advantages over Flink using only one CPU core per machine, which results in a higher speed of up to 40 % in this scenario. It appears that adding more workers (task slots) to Flink results in a constantly linear scale up, whereas for Spark this only holds true for up to 10 workers. Afterwards, the scale up grows more slowly.

Scaling Data Set Size An analysis with varying data set sizes indicates a quadratic increase in execution time. The dominating component is comparing the users' ratings inside the buckets whereas steps (i) and (ii) (see Figure 12) have less influence on the total execution time. With 10 % of the data, these steps take up 10,0 % of the time, whereas for 20 % of the data it is only 5,6 % of the time.

The Impact of the Signature on execution times The quadratic time complexity of the comparison step dominates the run time so we looked for means to further shrink the buckets that are created from the users' signatures. So far, the buckets are based on a single movie ID, with bucket sizes exceeding 5000 users. That results in up to 12.5 million comparisons that have to be executed to find similar users in only one bucket.

To make buckets more selective, we increased the signature length, which means using more than one movie to create user signatures. We implemented



(a) Relative runtime speedup

(b) Absolute runtime speed up

Fig. 13: Speed up for Spark and Flink on 10 % of the Netflix Prize data set (mean over 5 repetitions).

that approach and compared run times as well as the bucket size distribution for 1 and 2 movies in the signature. Our results indicate that the bucket distribution is far more favorable for signature length of two. Yet, we observed the number of buckets to increase extremely (44,345,399 for signature length of two, 17,737 for signature length of one). Also the run time and memory use is higher for signatures of 2 movies, so that a successful run with more than 10 % of the data was not possible.

We suppose the reason is that data of a user is duplicated to every bucket that the user is in. Therefore, we conclude that depending on the algorithm, both, the number of buckets as well as the bucket size, can be dominating factors in the observed run times.

To decrease the overall run time we plan to further improve balancing the number of buckets and bucket sizes by de-duplicating data among sufficiently small buckets. Another approach is concentrating on selective signatures for movies with many ratings to assure evenly small buckets. This we could achieve by using signature size of 2 for frequently rated movies and a signature of 1 rating for less popular movies. In addition we might use the idea of PartEnum as discussed by [11]

We suspect that for more workers a greater number of smaller buckets would be beneficial, which are distributed among the nodes. The nodes can than profit from the smaller bucket sizes meaning less candidate pairs to compare.

9 Data Mining

9.1 Motivation

A well studied problem in data mining is association rule mining, analysing the co-occurrences of entries in a number of data sets. Common applications

include market basket analysis, data cleansing and intrusion detection. Among other algorithms for association rule mining, like Eclat[36] or FP-Grows[24], the best known is the Apriori algorithm[8], which in its original form is designed to run on a single machine. In this chapter we will present our adaptions of the Apriori algorithm for the distributed Map-Reduce-Frameworks Apache Spark and Apache Flink. As a sample application of the adapted algorithms we will show how they can be used to predict user ratings for movies for the Netflix price data set[3]. Based on our implementations we will then discuss the practical differences between Spark and Flink.

9.2 Algorithm

Outline of the Apriori Algorithm The Apriori algorithm identifies frequent individual items and extends them to larger and larger item sets as long as those *itemsets* appear sufficiently. These frequent item sets can thereupon be used to determine association rules. The Apriori algorithm deals with so called *transactions*, which are sets containing one or more *items*. In our application of the Apriori algorithm, every movie of the Netflix data set is considered an item and for each user all the movies he/she has rated form one transaction. Given a set of transactions, the algorithm finds all sets of items that are part of at least s transactions. s, the number of transactions an itemset has to be present in, is called *support* and is chosen by the user. Item sets that appear often enough are called *large itemsets*. From every of these large itemsets association rules of the form $X \implies Y$ are derived where $X \cap Y = \emptyset$ and $X \cup Y$ is the itemset. For every association rule its *confidence* is calculated and all association rules whose confidence is smaller than a user chosen value c are discarded. The confidence of an association rule $X \implies Y$ is defined as the number of transactions containing $X \cup Y$ divided by the number of transactions containing X. The association rules that were not discarded form the result set of the algorithm.

Single-Machine Implementation The single-machine implementation begins by counting the support of the individual items. From the items whose support is higher than s, itemsets of size two are generated. This are candidates for large itemsets for the next round. The Apriori algorithm is based on the fact that large itemsets of size n can only be generated from other large itemsets of size $n-1$. In the next step, the support of all candidates of size two is counted and itemsets with too small support are discarded. From the remaining itemsets the candidates of size three are generated – i.e. all itemsets of size three for which all subsets of size two have a large enough support. This is repeated for increasing set sizes until no more candidates can be generated. Once all large itemsets are found, the Apriori algorithm derives the association rules from them. To this end, a straight-forward approach is used to derive association rules from each of the large itemsets: Let Z be the large itemset, then for each smaller, non-empty subset X of Z the association rule $Z \backslash X \implies X$ is output if and only if its confidence is at least c.

Adapted Algorithm for Spark The problem of a distributed association rule mining was addressed e.g. in "Apriori-based frequent itemset mining algorithms on MapReduce." [27]. Our implementation uses the main ideas of this paper for adapting Apriori algorithm for a distributed Map-Reduce-Framework and finding a way to parallelize (parts of) the algorithm so that it can be distributed across the available machines.

For our Spark implementation we parallelized the support counting for various itemsets. In the beginning, the input data set consisting of all transactions is distributed in the cluster. The support for all itemsets of the same size are counted concurrently using a map and a reduce task: In the map task, each worker counts the occurrences of each of the itemsets in its own share of the transactions. In the reduce task, the number of occurrences for each itemset are summed up across the different workers. So the counting is parallelized by having each worker do the counting only on a subset of the input transactions.

The generation of candidate itemsets from previously found large itemsets was not parallelized. Instead, this is done locally on the master node for two reasons. For one, the candidate generation is less well suited for parallelization because the access to all large itemsets of the previous round is needed. Sending them to all workers in the cluster instead of keeping them only on the master node leads to increased network traffic and therefore computational overhead. The seconds reason for not parallelizing the candidate generation is that the overall runtime is vastly dominated by the support counting so that a parallel execution of the candidate generation would only lead to a very small speedup, if any.

For the same two reasons the association rules derivation from the found large itemsets is also done locally on the master node. This step takes even less time than the candidate generation.

To achieve reasonable runtimes, we – apart from using parallelization – also payed attention to an efficient implementation of the algorithm. For example we use a prefix tree to store all current candidates as described by . This on the one hand reduces bandwidth when distributing the candidates in the cluster and on the other hand speeds up the comparison of each transaction to the set of all candidates.

Adapted Algorithm for Flink One can think of our Spark implementation as running on the master node of the cluster, with some selected parts of the algorithm being run distributed. The generation of new candidates and counting their support until no new candidates can be found is expressed with a *while-loop* in the Apriori algorithm. Unfortunately there is no Spark equivalent for loops, thus the algorithm can't be fully expressed using Spark operators, but relies on manual loop unrolling. In Flink on the other hand, there are two equivalents for a while loop, called Bulk Iteration and Delta Iteration, respectively [9]. Using a Delta Iteration operator, which was better suited for our use case, we could express the entire Apriori algorithm with Flink operators. In comparison to expressing only parts of the algorithm using Flink operators this made it possible for the Flink framework to optimize the given task as far as possible. To express

the whole algorithm in Flink, we also had to express the candidate generation as a Flink operator. Since we decided to not parallelize this task, we simply created a Reduce task that reduced *all* the large itemsets of a certain size by collecting them and generating candidates from them. Since we define that there is only one group of large itemsets to reduce which contains all itemsets, this Reduce task run only on one of the workers and therefore was not parallelized, just as before, although it's now expressed as a Flink operator.

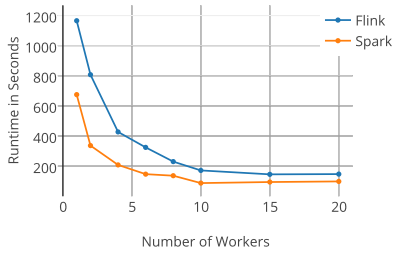
9.3 Benchmarks and Evaluation

We evaluated the behaviour of our Spark and Flink version of the Apriori algorithm in several different settings to gain insights into the practical differences between the Spark and the Flink framework. For the evaluation we used the mentioned Netflix price data set. This dataset has a size of 1.3 GB and consists of nearly 152000 transactions, containing 40 items on average. All measurements were done using the cluster described in the introduction of this paper. The runtimes presented here are in each case averages of three runs with identical parameters.

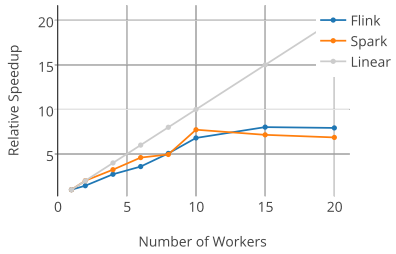
The choice of the support has an enormous impact on the performance of the algorithm, since it affects the size of the generated candidates as well as the size of the candidate tree. A smaller support leads to a higher number of itemsets that satisfy the support constraint. Therefore the frequency of the appearance of more candidates has to be counted, which results in a longer execution time. Moreover, there is a higher memory consumption, since the tree has to be kept in memory.

To achieve reasonable runtimes, we chose a minimum support value of 350, which corresponds to 2% of all transactions. This value was found empirically, thus for instance the execution of the algorithm on Flink with a support less than 320 leads to a deadlock due to continuous garbage collection 10. Since in Spark the memory is managed more efficiently, the algorithm can be executed with the support value smaller than 320, yet higher than 290.

Scaling to multiple workers In Figure 14a the runtimes for both Spark and Flink implementation are shown for 1, 2, 4, 6, 8, 10, 15 and 20 workers. As is clearly visible in the graph, the Flink implementation is always notably slower than the Spark implementation by an approximately constant factor. It can also be seen that for both Spark and Flink, an increase in the number of workers leads to a decrease in the runtime up until ten workers. From there on adding more workers doesn't have much influence on the runtime anymore. As explained in the introduction, the reason for this behaviour is that the used cluster consists of ten machines. So as long as adding workers leads to more machines being used, the runtime decreases. However, the algorithm doesn't benefit from running multiple worker nodes on a machine. This behavior can be explained by the fact that all worker nodes running on a machine have to fetch the data to process (i.e. the transactions) from the same hard drive. This apparently is the bottleneck here. The fact that an increase in the number of workers leads to a decrease in runtime up until ten workers shows that both implementations are scalable,

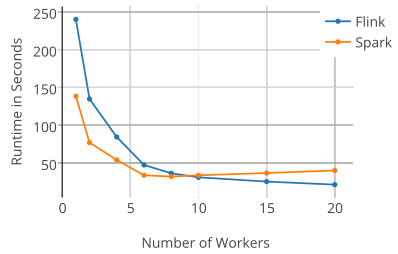


(a) Absolute Runtime in Spark and Flink

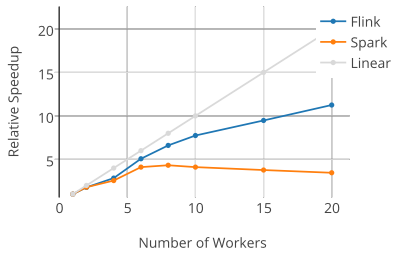


(b) Relative Speedup for Spark and Flink

Fig. 14: Runtime and speedup on the initial Netflix data set



(a) Absolute Runtime of Apache Spark and Apache Flink



(b) Relative Speedup of Apache Spark and Apache Flink

Fig. 15: Runtime and speedup on a data set with a greater length of transaction

which is the prerequisite to benefit from distributed Map-Reduce-Frameworks. Figure 14b shows the relative speedup for both implementations, as well as the ideal linear speedup. As could be expected, neither implementation reaches the ideal speedup. This is due to the communicational and organizational overhead. With an increasing number of workers the speedup also increases up until ten workers. From this point on, the speedup is approximately constant also due to the number of cluster machines.

There are minor differences in the relative speedup of both algorithms for one to ten workers. However for a higher number of workers, Flink has a higher relative speedup. This means that Flink seems to be able to leverage high numbers of workers better than Spark, although the difference is quite small on this data set.

Furthermore we examined the scaling behaviour of both frameworks on another data set. The second data set consists of a smaller number of transactions, namely 35533. However the transactions consist of a higher number of items, since in this case there were only two transactions per user. Thus, all movies with the ratings one and two were put into one transaction, as well as movies with the ratings three till five accordingly. The growth in the length of the transactions lead to a higher number of itemsets that reached the minimum support and therefore bigger candidate trees and extra computations.

Figure 15a demonstrates that with a small number of workers, this data set shows similar behaviour as the first one. Yet, Flink seems to be able to exploit several cores on one machine more efficiently, than Spark, which reaches its speedup maximum with ten cores. There is a visible speedup on more than ten cores with the Flink's version of the algorithm as seen in Figure 15b, which verifies the assumption, that Flink is able to leverage multiple workers on a machine more efficiently than Spark. Furthermore Flink performs better with algorithms that rely more on computations than on IO. Still, both frameworks scale well and show only little difference in the relative speedup.

10 Synthesis of Findings

Although Apache Spark and Apache Flink have a similar scope and syntax, and also share many concepts, we experienced several notable differences between the two frameworks.

Execution plan Apache Spark's execution plans consist of many subplans. This design enables flexible composition of atomic subplans and makes complex dependencies between subplans easy to implement, because these subplans are synchronized with the master.

In contrast, execution plans in Apache Flink are monolithic and executed only once. They can be inspected with a dedicated plan visualizer tool. On the one hand, this makes automatic optimization of the execution tree possible and allows Flink to skip some steps and stream chunks of data through every step of the pipeline. For example, sometimes only part of the data that is required for the next step is deserialized by Flink. On the other hand, such monolithic plans are less flexible. Flink tries to compensate for this by including an iteration loop feature in its plans.

Since the data is streamed between the different steps, it is very difficult to isolate the run time and resource consumption of a single operation in Flink. Still, this can be achieved by implementing an accumulator for benchmarking. The methods `print()` and `collect()` that Flink offers contradict its monolithic plan strategy. They should not be used because they trigger immediate plan execution and thus prevent the optimization intended by Flink.

Iterations To implement iterations, Apache Spark requires the use of the native loops defined in the respective programming language. Spark executes them by manual loop unrolling. In contrast, Apache Flink possesses iteration operators and employs streaming in order to keep data moving for as long as possible.

This approach led to performance improvements over Spark as is described in the Sections 4.3, 5.3 and 7.5. The concept of explicit iterations in Flink is particularly advantageous as native loop unrolling in an application with high memory usage results in tedious garbage collection. On the other hand, Flink's approach also complicates benchmarking and debugging because operations and iterations often overlap.

Memory usage Apache Flink uses its own way of processing data in memory to avoid memory over-allocation. Instead of putting numerous objects on the heap, Flink serializes them into a fixed number of pre-allocated memory segments. The data is thus stored in a binary form to avoid storage overhead and enable efficient binary operations. If more data needs to be processed than can be kept in memory, Flink's operations partially spill data to disk. Therefore, `OutOfMemoryErrors` can be avoided.

Apache Spark on the other hand has a memory-centric approach and profits hugely from more memory as heap space. Similarly to Flink's DataSets, RDDs are serialized and written to disk when the memory is full.

Due to its memory management techniques, Spark can use the given memory more efficiently. It can perform tasks with a higher memory usage whereas Flink seems to reach a deadlock through garbage collection. This could be observed in the Data Profiling, Graph Mining, and Data Mining tasks as mentioned in Sections 4.3, 5.3, and 9.3, respectively.

Serialization Serialization has an enormous impact on the performance of applications. Formats, which are too slow to serialize or which consume much memory, can slow down the computation remarkably. Apache Spark aims to strike a balance between convenience and performance by using the default Java serialization library. This library is flexible since it works on every class that implements `java.io.Serializable`.

In contrast, Apache Flink includes its own custom serialization framework in order to control the binary representation of data and enable various optimizations. Unfortunately, this approach does not support for recursive structures like prefix trees used in the Data Mining algorithm, 9.2, thus a own implementation of the serializer for these structures is necessary. Both frameworks allow users to alternatively use different serialization libraries, such as the fast and compact Kryo library.

Performance We conducted comprehensive benchmarks to determine performance differences between the two frameworks for all of the six use cases. The results show that, for all of them, Spark and Flink scale well with an increasing number of workers.

For the use cases Graph Mining, Text Mining, and Machine Learning, Flink was found to be faster overall - particularly because of the more efficient implementation of iterations. In contrast, for at least one data set of the Data Mining task, Spark was observed to be faster than Flink. For the Data Profiling scenario, Flink outperforms Spark on small data sets whereas Spark is faster on larger data

sets. Only in the Data Cleansing use case, both frameworks performed equally well.

Another observation, which was made in the Graph Mining and the Data Mining use cases, is that optimal speedup is reached with exactly one core per physical computer. The reason behind that is obviously that memory- and IO-intensive applications do not benefit from multiple CPU cores per machine. Moreover, in the case of Data Cleansing and Data Mining we observed that Flink profits from using multiple workers (task slots) per node more than Spark does. Due to the lack of asynchronous message passing, both frameworks are conceptually not perfectly suited for Genetic Algorithms.

11 Conclusion and Future Work

In this paper, we applied Spark and Flink to multiple use cases. We conducted benchmarks for each of these scenarios and compared the two frameworks according to a number of aspects. Neither Spark nor Flink excel at every use case and thus, users must choose their framework carefully with regards to their intention.

We observed noticeable performance differences between the frameworks for all use cases except Date Cleansing. In the cases of Graph Mining, Text Mining, and Machine Learning, Flink was found to be faster overall, whereas Data Mining ran faster in Spark. Generally, all scenarios benefited from running on more workers and scaled well with more CPU cores. However, memory- and IO-intensive applications, such as Graph Mining and Data Mining, did not benefit from multiple CPU cores per machine. They reached their optimal speedup at 1 core per physical computer.

We conclude that, for cases of iterative algorithms, Flink is the more efficient framework. This became particularly visible in the scenarios of Graph Mining and Machine Learning. However, our measurements also suggest that Spark might be a more appropriate candidate for memory-demanding applications. This was observed in the use cases of Data Profiling, Graph Mining, and Data Mining.

Motivated by our results, we plan to investigate in the future whether other iterative algorithms perform better on Flink than on Spark and thus confirm our observation. Moreover, we would like to confirm our finding that the execution of memory-intensive applications is faster on Spark with more experiments. It will be also interesting to inspect the impact of memory tuning approaches suggested by the Spark developers for algorithms with heavy memory usage [6]. To verify our findings, we plan to increase both the number of nodes and the memory per node to be able to process larger data sets.

Appendices

Appendix A

Text Mining: Single-pass clustering algorithm

The authors of *Snowball*, Agichtein et al., suggest a simple single-pass algorithm for clustering a large number of patterns (see Sec. 6.3) [7]. This algorithm shall be described here. It begins by assigning the first pattern P_1 as the centroid for cluster C_1. For each subsequent P_i it calculates the similarities S_j with each existing cluster (i.e. its centroid). If the greatest of these similarities $max(S_j)$ is greater than a threshold value T, the item is added to the corresponding cluster with the index $argmax_j(S_j)$ and that cluster centroid is recalculated. Otherwise, P_i is used to initiate a new cluster.

Appendix B

Text Mining: Implementation details

Parallel text processing Five main steps in Snowball can be easily parallelized because they operate on individual paragraphs or sentences independently: splitting the paragraphs into sentences, NER tagging, filtering out dispensable sentences, finding seed tuple occurrences and generating patterns, and finding new candidate tuples. Filtering out dispensable sentences ensures that only sentences that possess both entity tags (e.g. organization and location) are processed, reducing the data size by 95%. The remaining sentences are filtered again by retaining only the ones that contain any of the seed tuples, which reduces the amount of data again by over 90%.

Pattern confidences Other steps cannot be formulated that easily. Here, the calculation of pattern confidences will serve as an example (again, using the organization-headquarters example). It has two inputs: the new tuples generated by a pattern

```
<tuple_organization, <pattern_id, tuple_location>>
```

and the seed tuples

```
<seed_organization, seed_location>
```

Firstly, these datasets are joined on the organization and yield

```
<organization, <<pattern_id, tuple_location>, seedtuple_location>>
```

Next, the two locations are compared. When the tuple location matches the seed tuple location, it is counted as a positive, if not, as a negative. This is realized with a map-transformation that yields

```
<pattern_id, <#positives, #negatives>>
```

where the two last fields are either `<1,0>` or `<0,1>`. With a reduceByKey operation, the total counts of positives and negatives for each pattern are computed. Finally, a map transformation calculates the pattern confidence based on these counts as the fraction of positives.

Appendix C

Text Mining: Two-level clustering

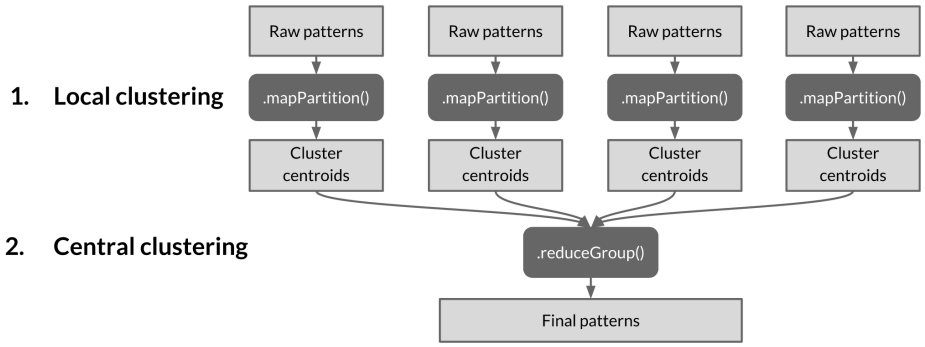


Fig. 16: Schematic overview of the two-level clustering approach.

Bibliography

[1] Apache Flink documentation: Configuration. https://ci.apache.org/projects/flink/flink-docs-master/setup/config.html#configuring-taskmanager-processing-slots#configuring-taskmanager-processing-slots. Accessed 2015-08-31.

[2] Concorde tsp solver. http://www.math.uwaterloo.ca/tsp/concorde.html. Accessed 2015-08-31.

[3] Netflix prize data set. http://www.netflixprize.com. Accessed 2015-07-21.

[4] Spark programming guide. http://spark.apache.org/docs/latest/programming-guide.html#external-datasets. Accessed 2015-08-31.

[5] Tsplib, a library of sample instances for the tsp. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Accessed 2015-08-31.

[6] Tuning spark. https://spark.apache.org/docs/0.8.1/tuning.html#memory-tuning. Accessed 2015-08-31.

[7] Eugene Agichtein and Luis Gravano. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the fifth ACM conference on Digital libraries*, pages 85–94. ACM, 2000.

[8] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[9] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal—The International Journal on Very Large Data Bases*, 23(6):939–964, 2014.

[10] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

[11] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 918–929. VLDB Endowment, 2006.

[12] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. *DBPedia: A nucleus for a web of open data*. Springer, 2007.

[13] Nguyen Bach and Sameer Badaskar. A review of relation extraction. *Literature review for Language and Statistics II*, 2007.

[14] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structur-

ing human knowledge. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 1247–1250. ACM, 2008.

[15] Sergey Brin. Extracting patterns and relations from the world wide web. In *The World Wide Web and Databases*, pages 172–183. Springer, 1999.

[16] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis.

[17] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engg.*, 11(4):29–41, July 2009.

[18] William Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[20] Tobias Friedrich, Timo Kötzing, and Andrew Sutton. Heuristic optimization. University Lecture, Lecture Notes, 2015.

[21] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[22] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)*, 28(2):140–174, 2003.

[23] Michael Hahsler and Kurt Hornik. Tsp—infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 23(2):1–21, 12 2007.

[24] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM-SIGMID International Conference on Management of Data*, pages 1–12, Dallas, 2000.

[25] Arvid Heise, Jorge-Arnulfo Quiane-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combincations. 2013.

[26] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proceedings of the Ninth IEEE International Conference on Data Mining*, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[27] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh. Apriori-based frequent itemset mining algorithms on MapReduce. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, 2012.

[28] Alan E Mislove. *Online social networks: measurement, analysis, and applications to distributed information systems*. ProQuest, 2009.

[29] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 495–506, New York, NY, USA, 2010. ACM.

[30] Thomas Weise. *Global Optimization Algorithms – Theory and Application*. Germany: it-weise.de (self-published), 2009.

[31] Mike Wheatley. Will the mysterious Apache Flink find a sweet spot in the enterprise?, 2015.

[32] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–47, 1998.

[33] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15:1–15:41, August 2011.

[34] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[35] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[36] Mohammed J. Zaki. Scalable algorithms for association mining. volume 12, pages 372–390. IEEE, May/June 2000.