ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

MASTER SEMESTER PROJECT – REPORT

PROJECT ADVISOR: PROF. ANNE-MARIE KERMARREC

PROJECT SUPERVISOR: DR. RAFAEL PIRES

DISTRIBUTING kNN GRAPH CONSTRUCTION

ANALYZING THE CLUSTER AND CONQUER ALGORITHM AND ITS PERFORMANCE

SCALABLE COMPUTING SYSTEMS LABORATORY – SACS

BY PAULINE ISABELA CONTI (DATA SCIENCE)

2021 Winter Semester - 7th January 2022

CHAPTER 1

INTRODUCTION

A k-Nearest-Neighbors (kNN) graph is a widely used data-structure in Machine Learning and is especially relevant in the case of recommendation systems [1][2]. They provide a way of connecting a dataset's points with each other based on a chosen criteria of similarity. Nowadays, many social media companies, online shops, websites as well as streaming services use them to tailor their content to each user [4][7]. However, computing an exact kNN graph is very costly computationally.

Approximate kNN graphs can often be used instead of exact ones without hurting the application's performance (when they have a good quality) [1], and tend to be less expensive computationally. As the quantity of data at disposal for these applications continues to grow, so does the need for an efficient way of computing them, especially in a distributed setting. Finding new ways of producing rapidly high-quality kNN graphs is exactly what the Cluster and Conquer algorithm [5] is intended to do. Reaching state-of-the-art results in terms of computational time, and graph quality, the algorithm uses multiple innovative techniques.

The objective of this project is to implement the Cluster and Conquer algorithm in a distributed setting, as well as try to understand the impact on the results of the techniques used. We will first present the background, with some core definitions. Then, the implementation and measurements of our distributed version of Cluster and Conquer will be explained. Finally, we will explore more specifically some aspects of the algorithm by reverting to more naive approaches, including specific elements and analyzing their impact on time and quality.

CHAPTER 2

BACKGROUND

2.1 K-NEAREST-NEIGHBORS GRAPHS

In a kNN graph, two vertices *p* and *q* will be connected by a directed edge *(p, q)* if *q* is part of *p*'s k most similar vertices according to a similarity metric. Hence two neighbor vertices will be two entities considered as being similar in a specific context.

In many – but not all – recommendation systems, as well as in this report, the vertices represent the set of users of a given application. In this context, kNN graphs often are a tool to better understand or predict a user's preferences on the application, based on what similar users do.

2.1.1 SIMILARITY

The similarity metric is a very important aspect of kNN graph construction. Being application-dependent, it is responsible for encapsulating the relevant information from each node into a neighborhood relationship. In order to do so, the metric is often defined as a function of another set of entities – referred to as items – which are related to the vertices. Here in the MovieLens dataset [6], the items are movies for which users have given a rating, meaning that users will be neighbors if they have similar tastes in cinema. Given those sets of users and items, the choice of function to link pairs of users is made according to the application's needs.

Given sets $U = \{u_1, ..., u_n\}$ of users and $I = \{i_1, ..., i_m\}$ of items, we can define for each user $u \in U$ a profile P_u corresponding to the set of items this user has rated. This allows us to define a similarity function on pairs of profiles – which directly encapsulates the relationship between the items and the users. Hence the similarity $sim(u, v)$ between two users $u, v \in U$ can be expressed as a function of their profiles: $sim(u, v) = f_{sim}(P_u, P_v)$.

The similarity used here is the *Jaccard similarity*. It is defined as:

$$sim(u, v) = Jaccard(P_u, P_v) = \frac{|P_u \cap P_v|}{|P_u \cup P_v|}$$

This similarity metric is positively correlated to the number of common items in two users' profiles. While being computationally expensive, it does not take into account the value of the rating, making it relatively simple to compute and is applicable to a wider range of applications than other similarities.

2.1.2 APPROXIMATE KNN GRAPHS & QUALITY

An exact kNN graph is very expensive to compute (requiring $O(|U|^2)$ similarity computations) and therefore scales poorly, motivating the search for computationally cheaper alternatives. As a result, approximate kNN graphs [1][3] are often used in practice. They define for each $u \in U$ a neighborhood $\widehat{knn(u)}$ with as little differences as possible to u's exact neighborhood $knn(u)$, while being more scalable than the brute force approach. We can measure how close an approximate kNN graph $\widehat{G_{kNN}}$ is to the exact kNN graph G_{kNN} through its $quality$.

An approximate kNN graph's quality is computed using its average similarity, and comparing it to the average similarity of its corresponding exact kNN graph. A graph's average similarity is simply the average of similarities over its $k \times |U|$ edges. We then define an approximate graph's quality as:

$$quality(\widehat{G_{kNN}}) = \frac{avg_{sim}(\widehat{G_{kNN}})}{avg_{sim}(G_{kNN})}$$

An exact graph having a quality of 1, an approximate graph with a quality close to it can be used without compromising most applications' results.

CHAPTER 3

CLUSTER AND CONQUER

3.1 THE ALGORITHM

3.1.1 OVERVIEW

In this section, we will go over the main steps of the Cluster and Conquer algorithm. If more details are needed, it is recommended to refer to the original paper [5], where each step is explained in rich detail. The algorithm can be split into three main steps: Clustering, Scheduling & KNN graph computation, and Merging.

During the *clustering* step, the dataset is clustered in $t \times b$ different clusters using a clustering scheme called *FastRandomHash*, with *t* the number of hash functions and *b* the number of clusters per hash function. An additional parameter, *N* allows to set a maximum capacity of users per cluster. Any cluster with more than *N* users will be recursively split until all clusters respect that condition. The clustering and recursive splitting will be further discussed in the following sections. The *scheduling and KNN graph construction* step is where the partial KNN graphs are computed in parallel, for which the authors used greedy scheduling. Finally, the *merging* step turns the partial KNN graphs into the final KNN graph.

3.1.2 CLUSTERING STEP & RECURSIVE SPLITTING

FAST RANDOM HASH SCHEME

The clustering step and its recursive splitting are the focal points of this algorithm. In particular, some characteristics of this step are central to reaching very low runtimes while still producing high-quality kNN graphs. The clustering of the users is done through one scheme: *Fast Random Hash*. Given a number *t* of distinct generative hash functions $h_x : I \rightarrow [\![1, b]\!]$, $1 \leq x \leq t$, with *I* the set of all items, and *b* the initial number of buckets, all items are hashed into one of *b* values by each of the hash functions. With $|I| = M$, $1 \leq r_{xy} \leq b$, $\forall\ x,\ y$ such that $1 \leq x \leq t$ and $1 \leq y \leq M$, we have the following:

For each of the t generating functions, the resulting hash $H_x(u)$, $u \in U$ and $1 \leq x \leq t$ is defined as the minimum hash value (for this generating hash function) among u's items:

$$H_x(u) = \min_{i \in P_u} h_x(i),\ with\ 1 \leq x \leq t$$

item id	h_1	h_2	$\cdots$	h_t
i_1	$\mathrm{h}_1(i_1) = r_{11}$	$\mathrm{h}_2(i_1) = r_{21}$	$\cdots$	$\mathrm{h}_t(i_1) = r_{t1}$
i_2	$\mathrm{h}_1(i_2) = r_{12}$	$\mathrm{h}_2(i_2) = r_{22}$	$\cdots$	$\mathrm{h}_t(i_2) = r_{t2}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
i_M	$\mathrm{h}_1(i_M) = r_{1M}$	$\mathrm{h}_2(i_M) = r_{2M}$	$\cdots$	$\mathrm{h}_t(i_M) = r_{tM}$

Note that $H_x(u)$'s result corresponds to u's cluster (out of the b possibilities) for the generating hash function h_x, and that u is assigned to clusters for each of the t generating hash functions, resulting in a total of $b \times t$ clusters.

For instance, given two generating hash functions h_1, h_2, $b = 3$, three users and five items, the *Fast Random Hash* scheme would work as follows. With $U = \{u, v, w\}$, $I = \{i_1, i_2, i_3, i_4, i_5\}$, and the following results for the hash values of the items by the two functions:

item id	h_1	h_2
i_1	$\mathrm{h}_1(i_1) = 3$	$\mathrm{h}_2(i_1) = 2$
i_2	$\mathrm{h}_1(i_2) = 2$	$\mathrm{h}_2(i_2) = 1$
i_3	$\mathrm{h}_1(i_3) = 1$	$\mathrm{h}_2(i_3) = 3$
i_4	$\mathrm{h}_1(i_4) = 2$	$\mathrm{h}_2(i_4) = 1$
i_5	$\mathrm{h}_1(i_5) = 3$	$\mathrm{h}_2(i_5) = 3$

Considering the following profiles for the three users: $P_u = \{i_2, i_4, i_5\}$, $P_v = \{i_1, i_2, i_3\}$ and $P_w = \{i_3, i_5\}$. We then would have the following:

user	H_1	H_2
u	$\mathrm{H}_1(u) = min\{2, 2, 3\} = 2$	$\mathrm{H}_2(u) = min\{1, 1, 3\} = 1$
v	$\mathrm{H}_1(v) = min\{3, 2, 1\} = 1$	$\mathrm{H}_2(v) = min\{2, 1, 3\} = 1$
w	$\mathrm{H}_1(w) = min\{1, 3\} = 1$	$\mathrm{H}_2(w) = min\{3, 3\} = 3$

Resulting in the following $b \times t = 6$ clusters:

cluster index $\longrightarrow$	1	2	3
H_1	v, w	u	
H_2	u, v		w

RECURSIVE SPLITTING

By assigning the minimum value of its hashed items to be the index of a user's cluster, the clustering algorithm causes a bias towards low indices. Indeed, the configuration where a cluster is significantly larger than the others is easy to encounter as long as a popular item is hashed to a low value. However, dividing the data into similar sized clusters is one of the main objectives of

the scheme. As the complexity of the similarities' computation grows exponentially with the number of users, larger clusters significantly impact the performance. As a result, the authors added a recursive splitting step.

Given a cluster C, with index η_C corresponding to hash function h_x, a second value $H_x \setminus \eta_C(u)$ is assigned to all $u \in C$. This new cluster index will be computed in the same way as with the *Fast Random Hash* scheme, only ignoring the current value η_C:

$$H_x \setminus \eta_C(u) = \min_{i \in P_u,\ h_x(i) > \eta_C} h_x(i),\ with\ 1 \leq x \leq t$$

Any user with only one item in its profile (and thus with η_C as the only index to choose from), or hashed alone into a new cluster will remain in the original cluster C. This splitting step repeats recursively until no cluster's size exceeds the maximum N, including newly created clusters. After each splitting, the original cluster is disregarded and up to b new clusters are created.

Considering the previous example, with an additional user z, with $P_z = \{i_3\}$. The maximum number of users per bucket is set to $N = 2$, meaning that any bucket with 3 or more users would have to be split through recursive splitting. Remembering that $h_1(i_3) = 1$ and $h_2(i_3) = 3$, we would have the following buckets:

cluster index $\longrightarrow$	1	2	3
H_1	v, w, z	u	
H_2	u, v		w, z

As bucket 1 of hash function H_1 exceeds the maximum number of users per bucket, it would need to be split. All the users that were assigned the bucket 1 for H_1 would need to be re-assigned a new bucket using the formula above. We would thus have:

$$H_1 \setminus 1(v) = \min_{i \in P_v,\ h_1(i) > 1} h_1(i) = min\{3, 2\} = 2$$

$$H_1 \setminus 1(w) = \min_{i \in P_w,\ h_1(i) > 1} h_1(i) = 3$$

$$H_1 \setminus 1(z) = 1,\ as\ P_z = \{i_3\}$$

And the following new split buckets:

cluster index $\longrightarrow$	1	2	3
H_1	~~v, w, z~~	u	
H_2	u, v		w, z
$H_1 \setminus 1$	z	v	w

IMPACT OF CLUSTERING AND SPLITTING ON PERFORMANCE AND RESULTS

The *Fast Random Hash* clustering scheme uses multiple techniques to yield really high-quality approximate kNN graphs fast. While the "Divide and Conquer" approach of mapping users into independent clusters to compute local kNN graphs is not new, the clustering scheme used in this algorithm is and yields state-of the art results. The first noticeable aspect of the clustering is the way in which users are assigned to a bucket for each hash function. Indeed, the fact that the minimum hash value of the items becomes the bucket id for a given generating function, allows the authors to prove that more similar users will have a higher probability of being hashed in the same bucket. Secondly, this scheme creates at least $t \times b$ buckets, in which each user is represented t times. This gives t opportunities to users to be mapped in the same bucket as potential nearest neighbors, and thus increases the quality of the resulting graph. Lastly, the recursive splitting step allows to counterbalance the bias introduced by the minimum when clustering. While this step lowers slightly the quality of the resulting graph, it speeds up the algorithm as it prevents buckets from being too large which makes the local graphs very costly to compute.

3.2 IMPLEMENTATION & EVALUATION

3.2.1 IMPLEMENTATION

The initial goal of this project was to implement the Cluster and Conquer algorithm in a distributed setting, and to evaluate its performance. The paper's authors implemented the algorithm in Java, with multithreading on 8 threads. The distributed version was implemented with Scala 2.11.12 and Apache Spark version 3.1.2. The implementation of steps of clustering, computing the local kNN graphs and merging them into the final graph had been started prior to the beginning of this project. As a result, the implementation that was performed in the scope of this project was focused on the recursive splitting step, integrating it with the rest of the code and correcting potential implementation errors.

Spark is a very useful and powerful framework for distributed computing. However, it is also more restrictive in terms of available functions and data structures than Java or Scala. While the Cluster and Conquer algorithm is very efficient, it is a complicated algorithm that involves a lot of expensive operations like joins and groupings. As will be discussed later, the present implementation works, but is not optimally adapted to the needs of the framework. As a result, we were only able to test it on the MovieLens ML1M dataset [6], and not the larger ones.

3.2.2 MEASUREMENTS & RESULTS

After extensive testing and comparison of the results, the authors chose as parameters $t = 8$ for the number of generative hash functions, $b = 4096$ as the number of buckets per hash function and $N = 2000$ as the maximum bucket size.

With ML1M, the recursive splitting step is not necessary when using a maximum number of users $N = 2000$. However, as it is one of the important steps of the algorithm and the main focus of the implementation, it was decided to also run measurements with $N = 1500$ where the step was indeed needed. As when using $N = 2000$, no cluster needs splitting, comparing the algorithm

with the two values chosen for N will indeed allow to measure the effect of the splitting on time and quality, and in particular, if it matches the findings of the Cluster and Conquer paper.

Finally, the impact of the number of generative hashing functions was also tested using the values $t = \{2, 4, 6, 8\}$. As this parameter reflects the number of times each user was represented across the buckets, it is interesting to see how varying it affects both the time needed to generate an approximate kNN graph and its corresponding quality. For the measures, each version of the sets of parameters was run five times to obtain both the average and standard deviation.

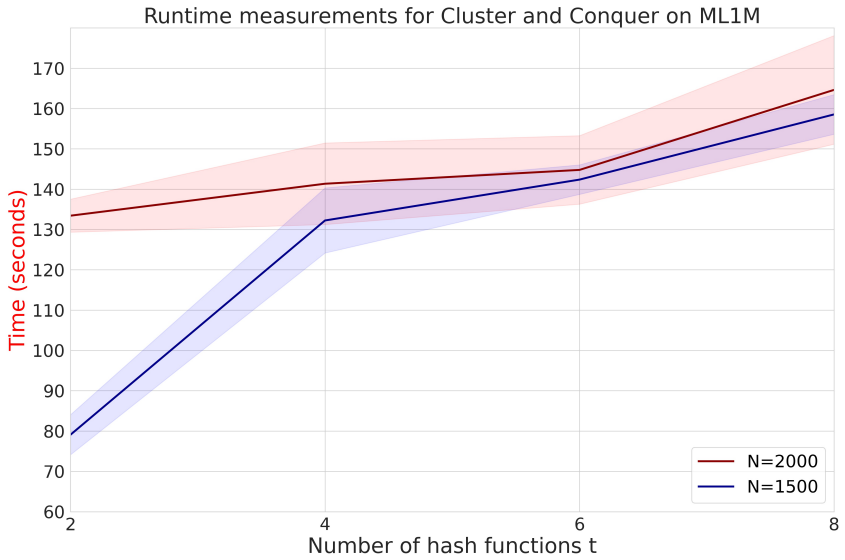


FIGURE 3.1
Time taken by Cluster and Conquer on ML1M in function of the number of hash functions

Figure 3.1 shows the resulting times measured when running our implementation of Cluster and Conquer in a distributed setting with varying values of t and N. The plotted lines are the average over five runs, while the shaded areas represent the standard deviation of the time measurements. The first observation is the fact that our times largely exceed the times reported in the paper (2.64 seconds with $N = 2000$ and $t = 8$). While this is deceiving, the impact of modifying parameters corresponds more to the expectations. As discussed previously, the main interest of recursive splitting is to reduce the computation cost of the algorithm, at the expense of quality. Indeed, as buckets exceeding a certain size are split into at most b new buckets, it separates users that potentially have a high similarity score. On the other hand, as the computation of the similarities is the main bottleneck of the algorithm, the local kNN graphs of the newly created buckets will be much faster to compute than their bigger counterpart, thus reducing the overall runtime of the algorithm. This corresponds to what we see in Figure 3.1. Indeed, for $t = 2$ in particular, there is a significant difference in the time taken by the algorithm between $N = 1500$ which involves the splitting step and $N = 2000$ which doesn't. This difference is much less striking for larger values of t, even though the runs with $N = 1500$ are consistently faster.

The fact that the difference is much larger with $t = 2$ than with other values can also be explained by the implementation. In particular, the step where all the local graphs are computed and then merged involves many costly operations. As each new generative hashing function used represents 4096 new buckets, it may be that due to the implementation, using more generative

hashing functions has an exaggerated impact on the performance compared to the findings of the paper.

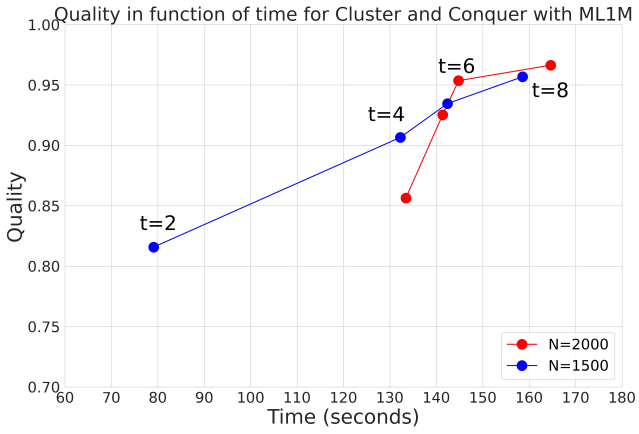


FIGURE 3.2
Quality in function of time for Cluster and Conquer on ML1M

t	Quality $N = 1500$	Quality $N = 2000$	Δ
2	0.8158	0.8564	0.0406
4	0.9066	0.9252	0.0186
6	0.9346	0.9536	0.0190
8	0.9568	0.9664	0.0096

TABLE 3.1
Quality of the resulting kNN graphs on ML1M with Cluster and Conquer

Figure 3.2 and Table 3.1 show how the modification of t and N impacted the quality of the resulting kNN graphs. The quality measures were made using the formula presented in Chapter 2, and the exact kNN graph generated by the code written by the authors of the Cluster and Conquer paper [8]. It was chosen to evaluate the quality with their graphs and not the ones we generated to ensure the reproducibility of the results on other exact kNN graphs than our own.

The results reported in Table 3.1 show that indeed, quality decreases when the recursive splitting step is used. However, this effect's scale decreases as the value of t grows, going from 0.04 for $t = 2$ to 0.0096 for $t = 8$. This observation is expected; when more hash functions are used, the importance of a specific bucket in the overall quality decreases, as similar users have more opportunities of being mapped into the same bucket.

Figure 3.2 maps the quality of the kNN graphs generated for different values of t in function of the time needed to generate them. Each dot represents a new value for t, from 2 to 8. While this graph also shows the points discussed above, it shines lights on the limits of our implementation. Indeed, the curve for $N = 1500$ should resemble more the one of a logarithm (like in Figure 6(a) of the paper [5]), whereas it looks like it follows a linear trend. In fact, while the qualities obtained are very satisfying, generating the graphs for $t = 4$ and $t = 6$ should take less time in comparison to the other values of t than the results we have. This shows that the present implementation doesn't scale as it should, and that augmenting the number of generating hash functions greatly hurts the performance.

Overall, while the results were satisfactory in terms of quality, the implementation doesn't meet the expectations in terms of runtime or scalability. The Cluster and Conquer algorithm being complex and having many steps, we wanted to "zoom in" on some of its components to understand better their impact on the overall performance and quality when faced with the task of generating approximate kNN graphs.

CHAPTER 4

BRUTE-FORCE-LIKE APPROACHES

4.1 OVERALL IDEA & GOAL

As seen in the previous chapter, the authors of the cluster and conquer algorithm were able, through some specific clustering and splitting schemes, to improve the quality of the approximate kNN graphs generated while greatly reducing the time needed to compute them. In order to truly understand the impact of those choices, we decided to "deconstruct" some of the steps of the algorithm and try to measure their impact on performance and quality.

We decided to come back to the basic brute force approach, choosing specifically what aspects of the algorithm to include. As we wish to understand how this algorithm might be adaptable to a distributed setting, we naturally started by focusing on the partitioning. Hence, we started by implementing a random splitting of the users across different numbers of partitions, measuring the quality of the resulting graph, as well as the computation time. We then decided to measure the impact on those same metrics of duplicating the users across different partitions.

4.2 SPLITTING ACROSS PARTITIONS

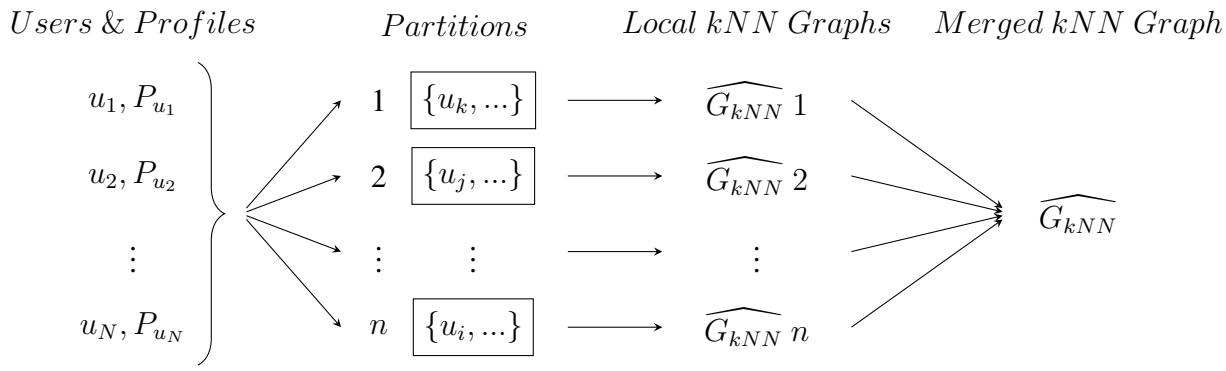
In Cluster and Conquer, the *Fast Random Hash* splitting scheme hashes two users in the same bucket according to a probability proportional to their Jaccard similarity. This allows for the initial partitioning of the users to indeed assign similar users to the same bucket. While this method is certainly conclusive (as the Cluster and Conquer algorithm reaches a quality of approximately 90% on different datasets) it is possible that simpler approaches also reach a high quality.

4.2.1 PARTITIONING RANDOMLY

IMPLEMENTATION

This led us to explore the impact on the quality of using a naive approach. In particular, to measure the impact of hashing similar users together, this approach needed to cluster the users independently of their profiles. As a result, the most simple approach; randomly splitting the users into a certain number of clusters, was used at first.

Once the profiles P_u, $\forall u \in U$ were obtained, only keeping ratings above 3, the users were simply randomly split to one of the partitions. All the partitions have an approximately equal number of users. Then, the Jaccard similarity of each pair of users within each partition was computed. From the similarities, like in the Cluster and Conquer algorithm, a local kNN graph was constructed in each of the partitions. Finally, all the local kNN graphs are used to create the final approximate kNN graph output during a merging step. With $|U| = N$ users and n partitions, the algorithm (starting from the profiles already determined) is thus comprised of the following steps:



This method presents multiple differences compared to the *Fast Random Hash* splitting. First of all, the size of each bucket is approximately the same. This prevents the issue of having one bucket significantly larger than the others, which motivated the random splitting step. As a result, while the final number of partitions is much lower than with Cluster and Conquer, no further splitting or re-arranging is needed and the number of local kNN Graphs to compute is fixed.

Another major difference with Cluster and Conquer's approach is that each user is mapped to only one bucket. In the paper, each user is present in *t* of the $t \times b$ final buckets. This allows to maximize the chances of each user to be compared to every possible neighbor. In this naive approach, however, the users are randomly mapped to one of the partitions. This causes the final kNN graph's quality to be dependent on whether similar users were partitioned together.

The implementation was also done using the same versions of Scala and Spark, however, RDD's were used as main data structure, as opposed to the DataFrames used for Chapter 3's implementation of Cluster and Conquer.

EVALUATION

Performance evaluation was done using the runtime and quality on MovieLens Ml1M and Ml10M. For Ml1M time was measured both including and excluding the steps of formatting and outputting the graph to a CSV file. For Ml10M, as the resulting dataset was much larger and induced a lot of variance in the measures (as the step can only be performed by one executor), the times reported were measured excluding these last steps. This allowed to not only measure more precisely how much time was needed to obtain the graph, but also to mitigate some of the variance in performance due to Spark's inner workings.

All the time measurements were obtained by averaging five runs with fixed parameters. The quality was measured in the same way as in Chapter 3, allowing to be certain that the resulting quality was independent of our own outputs.

For MovieLens ML1M, the measures were made for numbers of partitions n ranging from 2 to 12. As the dataset is small, it was possible to measure the evolution of the time taken and the quality on a linear scale, allowing us to visualize better the evolution of the monitored metrics, time and quality.

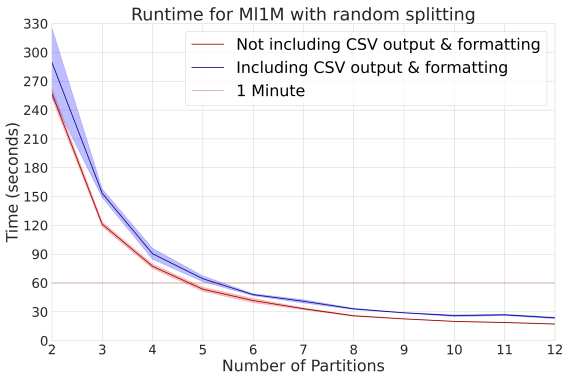


FIGURE 4.1
Resulting runtime of the algorithm compared to total time to get output

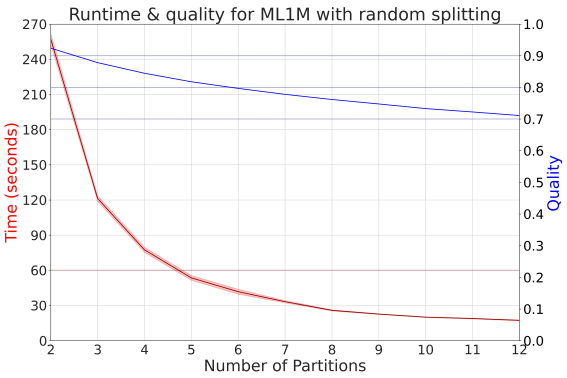


FIGURE 4.2
Runtime & Quality on ML1M in function of the number of partitions

Figure 4.1 displays the measured runtimes averaged, as well as their standard deviation, with and without the steps of formatting the data and outputting it to a CSV file. As we can see, while the time added by the formatting is a fraction of the total time, it adds uneven variance to the results. Both those steps take place once the graph is already computed, and could be used as-is if need be. Omitting this time also allows to measure more precisely the time required for the all the steps from reading the data form the database until having the kNN graph at our disposal. As a result, these steps were not included when measuring the time for the results in Figure 4.2 and later in the report.

Figure 4.2 shows the runtimes and their standard deviation (in red) of the algorithm on ML1M as well as the quality (in blue) of the resulting kNN graphs. We can see that the runtimes form a strongly decreasing curve as the number of partitions go up. As the number of similarities to compute for N users grows in $O(N^2)$, it is expected that such a graph would arise. A light red line is drawn at one minute to help with visualisation. With ML1M, we have that the approximated kNN graph can be generated in less than a minute when splitting the data over five partitions or more, reaching 17.31 seconds with twelve partitions. With a number of partitions lower than five, the time needed to generate the graph quickly augmented up to 4 minutes 17 seconds on average with two partitions.

Figure 4.2 also displays how the quality of the resulting kNN graphs evolves as the number of partitions changes. As the right axis' ticks don't align with the left's, light blue lines were added at 0.7, 0.8 and 0.9 for increased readability. We can see that while the quality does indeed decrease with the growth of the number of partitions, its slope is much more gradual than time's. With twelve partitions, we have a quality of 0.7111 which is expected given that each user is only compared to $1/12$th of the users in the dataset. With two partitions, the quality reaches 0.9238, which is much better, but still low taking into account the time necessary to generate the graph. With five partitions, this algorithm reaches a quality of 0.8177 in 53.51 seconds on average. We can clearly see that the number of partitions uncovers a trade-off between the time

taken by the algorithm to produce the graph and its quality. Given the two curves shown on Figure 4.2, the impact of the number of partitions on the runtime is much greater than on the quality, until $n = 8$, above which the two curves have similar slopes. Starting at seven, the lower the value of n, the larger of a step on runtime with relatively low increases in terms of quality.

For MovieLens ML10M, the values of n follow two linear scales concatenated. The first, consisting of intervals of 5 between 10 and 30 and the second, with intervals of 20 between 40 an 120. It was chosen to do a fine-grained evaluation of the evolution of the metrics we observe. This changing scale allowed to adapt the measurements to the growth of the time taken as the number of partition decreases.

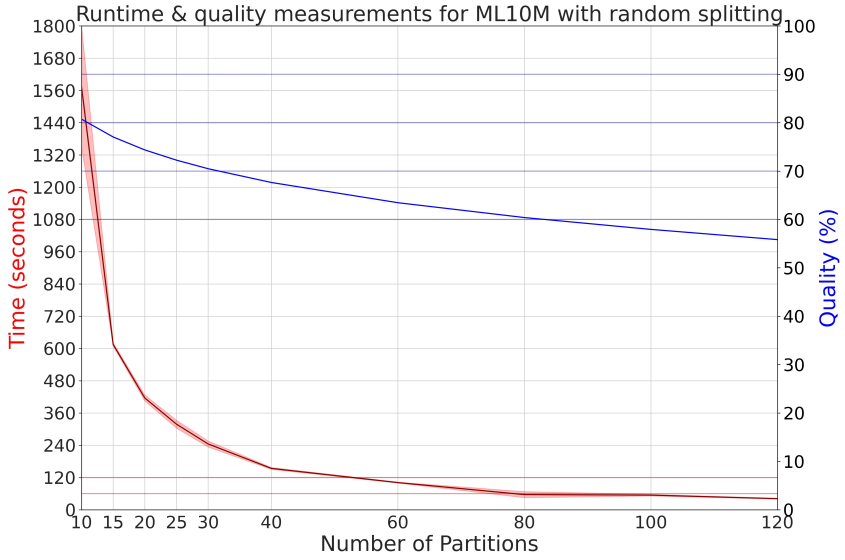


FIGURE 4.3
Runtime & Quality on ML10M in function of the number of partitions

Figure 4.3 shows the resulting times and qualities measured for the algorithm on MovieLens ML10M. Just like before, the red line shows the average over five runs, surrounded by the standard deviation over those runs. Two light red lines highlight where 1 and 2 minutes lie on the graph. Lines corresponding to qualities from 0.6 to 0.9 are in light blue, for increased readability of the blue quality curve.

The pattern of the runtimes is very similar to the one found when running the algorithm on MovieLens ML1M, only that the augmentation of the time is more abrupt. This is expected as ML1M counts around 6'038 users, whereas ML10M has 69'816. As a result, we can see much more clearly the square complexity of the similarity computations causing the jump in the runtime as the number of partitions decrease. We also see that past $n = 80$ partitions, augmenting n does not improve much the runtime, compared to the cost in quality. In addition to that, while the quality does increase when the numbers of partitions goes down, it only reaches 0.8073, at the cost of a runtime of around 26 minutes with $n = 10$. On the other end of the spectrum, while it takes 41 seconds to compute the graph with 120 partitions, the quality of the result is only 0.5583.

Overall, we can clearly see the limitations of this naive approach: while randomly splitting the users across the partitions still yields good quality kNN graphs for small datasets, it does not

scale well. This was expected, as the intent is not to yield better results than more complex and extensively thought through algorithms, but rather to understand the singular impact of certain choices and techniques.

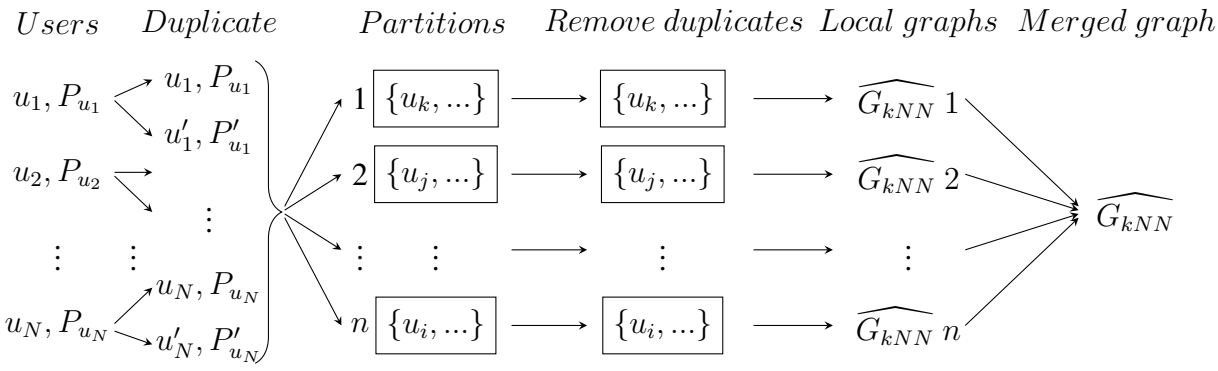
With a high number of partitions, while distributing and parallelizing the computations can allow to reduce considerably the time of computation, it comes at the cost of quality. As each user is only mapped to one partition, the probability of it being the same as its optimal neighbors decreases rapidly as the number of partitions grows. This issue is addressed by the *Fast Random Hash* scheme in multiple ways. First, clustering based on the profiles allowing users with similar profiles to have a higher chance of being in the same cluster. Second, repeating the clustering method with multiple hash functions to maximize the chance of two similar users to be clustered together at least once. This lead us to the next step that was investigated: replicating data across partitions.

4.2.2 ADDING DUPLICATION – IMPLEMENTATION

As we saw in the previous section, the naive Divide and Conquer approach of simply splitting the users into several partitions and computing local kNN graphs is quickly met with limitations. While the time needed to perform the algorithm is one of the limitations, quality is the main focus here.

From our naive approach, we can easily try to measure the impact of this repetition in a relatively controlled setting; where the presence or absence of repeated users across partitions is the only variable. In the *Fast Random Hash* scheme, each user is represented *t* times across the final buckets. The results from Chapter 3 also show that a higher value for t (meaning users replicated more times) yielded better qualities. However, duplicating the users to represent each one twice still allows us to grasp to what extent the quality improvement was due to the data being replicated, versus other techniques (like the minimum to choose the bucket index). Of course, more occurrences of each user mean a higher probability of grouping similar users together, but unless the number of clusters grows alongside it, it also reduces the initial benefit of clustering, as the amount of data the similarities are computed on is not reduced as much.

This new algorithm is very similar to the previous random clustering. A step that duplicates all users and their profiles is added before the shuffling and random partitioning, as well as a step that removes the duplicated users inside each partition before computing the similarities. The removal of duplicates before computing the local kNN graphs was not only necessary (as one user could have had multiple times the same user in its final nearest neighbors) but also ensured that no similarity computation was done redundantly. Once the similarities are computed, local graphs are then computed in the same way and merged into the final kNN Graph. Referring back to the previous diagram with $|U| = N$ and n partitions, we get:



4.2.3 EVALUATION & COMPARISON OF THE TWO ALGORITHMS

APPLIED TO MOVIELENS ML1M

The performance evaluation on MovieLens ML1M was done using the same range of number of partitions n as with the previous algorithm.

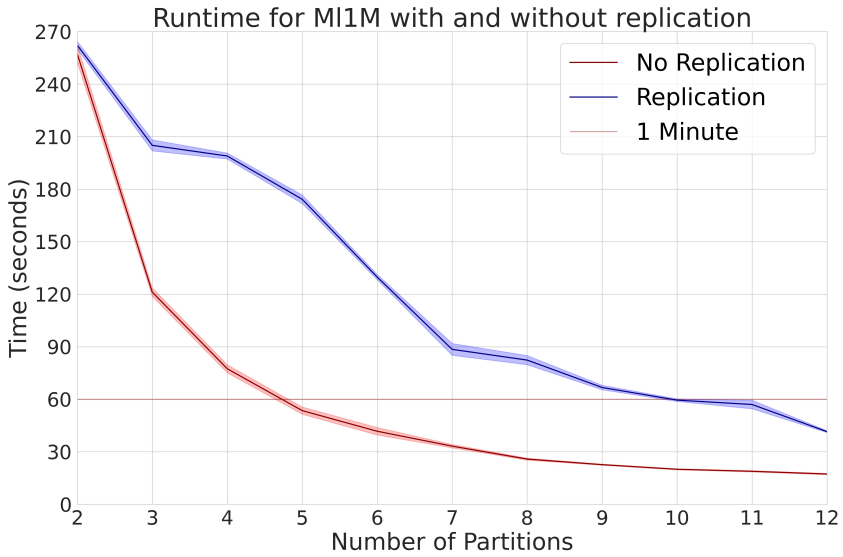


FIGURE 4.4
Time taken on ML1M in function of the number of partitions with and without replication

Figure 4.4 shows the evolution of the time taken to generate the kNN graph when duplicating or not the data. The red curve is the same as in the previous section, and the blue shows the impact of duplication. As expected, the replication does lengthen the computation, especially for $n \in \{3, 4, 5, 6\}$ for which the difference is the greatest. We can also notice a slight increase in the standard deviation of the times, which can be attributed to the shuffling of the partitioning with the duplicates, having an impact on the number of similarities being computed. While the curve keeps the overall shape; lower times for $n \geq 7$ and steeper slope towards lower values, it does not follow exactly the same pattern as the red curve.

In the algorithm, each user is represented twice, before being randomly partitioned. The larger the value of n, the lower the probability of the same user being mapped to the same partition

twice. As a result, few of the duplicated elements are removed before computing the similarities, yielding a similar curve for larger numbers of partitions. For $5 \leq n \leq 7$, we can see the steep augmentation in runtime expected by the presence of approximately twice the records when computing the similarities.

However, the slope starts flattening towards lower values of n, which is counter-intuitive considering the number of users mapped into each partition continues to grow. The number of collisions within a partition grows as n decreases, and with it the number of duplicated users removed, but the slope observed doesn't seem to match the expectations. Another explanation might lie in the merging process. Indeed, the algorithm without duplication simply needs to "concatenate" the local kNN graphs into the final merged graph, whereas replicated users across partitions force to recompute part of the graph. If more collisions happen, the merging of the local kNN graphs might also require less computation. However, while sorting an already ordered list might be marginally faster than sorting a random list, both computations have the same complexity. In addition to that, spark RDD sorting has $O(Nlog(N))$ complexity [9][12] and isn't nearly as costly as computing more similarities per partition, hence its impact would be more tamed and gradual across the values of n. As those explanations were not satisfactory, we decided to explore more precisely what might cause it. The means taken to understand and explain this curve, as well as the results, are presented in the next section.

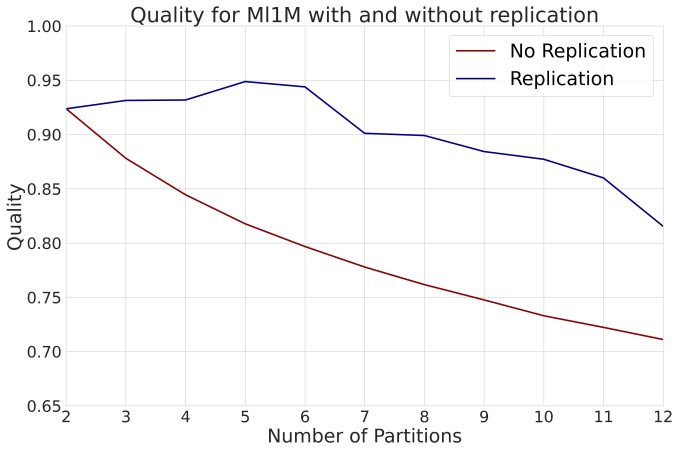


FIGURE 4.5
Quality on ML1M in function of the number of partitions with and without replication

n	Quality		
	No duplication	Duplication	Gain
3	0.8781	0.9314	+0.0533
4	0.8445	0.9318	+0.0873
5	0.8177	0.9488	+0.1311
6	0.7968	0.9439	+0.1471
7	0.7779	0.9011	+0.1232
8	0.7617	0.8991	+0.1374
9	0.7476	0.8843	+0.1367
10	0.7330	0.8772	+0.1442
11	0.7223	0.8600	+0.1377
12	0.7111	0.8154	+0.1043

TABLE 4.1
Quality of the resulting kNN graphs on ML1M with and without duplication

Figure 4.5 displays the quality of the resulting approximate kNN graphs obtained with the duplication (in blue), alongside the qualities obtained with the initial algorithm (in red). The y-axis starts at 0.65 for increased readability. Two things can be noticed immediately; first, the quality did improve significantly, sometimes reaching an almost 0.15 increase, and second, the curve is no longer strictly increasing.

The first observation shows the large impact of the duplication of the users across partitions on the final quality. Indeed, as we can see, the quality for all numbers of partitions exceeded 80% when duplication was done. Table 4.1 shows the numerical values, as well as the gain in quality for all values of n. The gain quality for $n \geq 5$ all exceed 0.10, which is very promising. Indeed, it seems that while a higher number of partitions is detrimental to the graph's quality, simply having two occurrences of each user already presents a significant increase in the results. In

addition to that, qualities obtained with values of $n \leq 6$ exceed the reported quality of Cluster and Conquer of 0.91. While it is certain that the complex algorithm performs better in terms of speed and scalability, it hints to the fact that the duplication of the users is a very important element to incorporate when designing such algorithms.

The second observation is however a bit more surprising, especially the results for $n \leq 5$ where the quality starts decreasing. Corresponding to the values of n for which the time measurements were surprising, it confirms the idea that the code doesn't behave as anticipated for values of $n \in \{2, 3, 4\}$.

UNDERSTANDING THE RESULTS FOR LOW VALUES OF n

Like mentioned above, the results in Figures 4.4 and 4.5 show unexpected results for values of $n \leq 4$, both in terms of quality and time. While the augmentation of the number of collisions might partly explain them, it seemed that the results need further explanations. In particular, we can see that the quality of the generated kNN graph for $n = 2$ partitions is the same with and without duplication of the users prior to partitioning. This suggests that all pairs of replicated users might have been mapped into the same partition, and thus having been removed before computing the local graphs.

This is probably due to how Spark shuffles the data when the function *repartition* is called. As stated in the documentation [10] and the source code [11], the function does indeed trigger a shuffling of the data when calling *coalesce*, with a random hashing of the old index, from which the new index is extracted with a modulo function. In addition to that, the increase in quality and runtime for all other numbers of partitions, especially when $n \geq 5$, suggests that a shuffling does take place. The shuffling of the users across partitions had been verified by hand on a smaller dataset prior in the semester, but for a higher number of partitions. This same examination was re-iterated with $n = 2$ and $n = 3$. It showed that indeed, when $n = 2$, all records stayed in the same partition before and after *repartition* was applied. For $n = 3$, while some records where indeed shuffled, part of them stayed in their original partition.

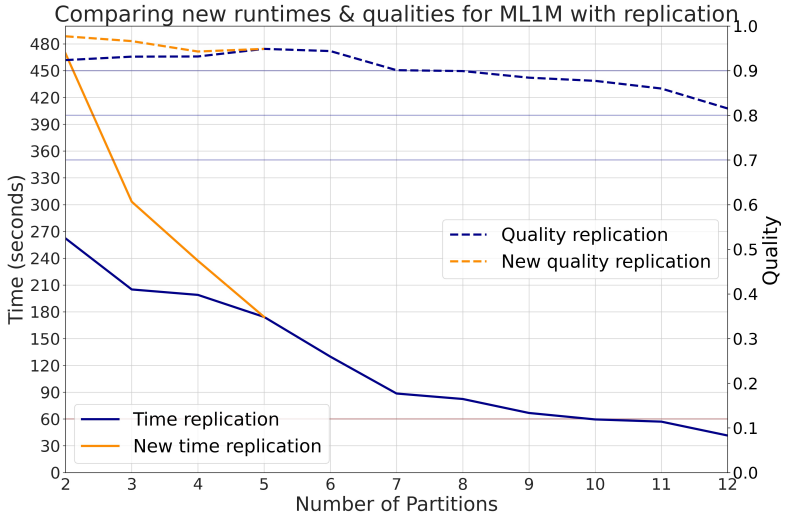


FIGURE 4.6
Time and quality measurements on ML1M with the new replication method for low values of n

This might be explained by the way Spark internally represents RDDs and schedules operations, reducing as much as possible the impact of expensive operations requiring a lot of data movement. As a result, an extra step was added to try to "force" the full shuffle of the values across the partitions when $n \leq 4$. This step consisted of using *repartition* to map the data to a higher number of partitions, before using *coalesce* in order to revert back to the desired number of partitions.

Figure 4.6 shows the new runtimes and qualities measured for $n \in \{2, 3, 4\}$, connected to the previous curve shown above. The new results correspond better to what was expected, both in terms of quality and time. The computation times did increase considerably, matching the quantity of users per partition, and the qualities reflect that, with 0.977 for $n = 2$. While those new results show that replication allows to construct graphs of very high quality for low values of n, the same limitations of high runtimes keep arising. The fact that an extra step was needed to ensure proper shuffling also suggests that the opacity of internal computations can impact results. Comparing this result to the quality obtained when using Cluster and Conquer with $t = 2$ (0.8564), and $t = 8$ (0.9664), we see that quality does depend a lot on the replication of the users across partitions. As Cluster and Conquer has much more buckets, it also suggests that quality depends less on the number of users per partition than them being replicated. This leads to question the impact of the specific clustering used in their algorithm.

APPLIED TO MOVIELENS ML10M

Coming back to the original random splitting with replication, For ML10M, n took values from $\{16, 32, 64, 128\}$. Those values were chosen because the duplication augmented considerably the time taken by the algorithm. Powers of 2 gave, with only a few values, a better overview of the targeted metrics' trends than a linear scale would. The objective for this section was primarily to compare the resulting qualities more than evaluate the exact evolution of the runtimes, which was done in more detail in the previous section.

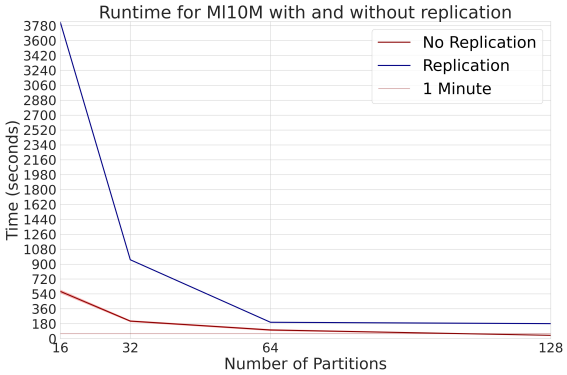


FIGURE 4.7
Time taken on ML1M in function of the number of partitions with and without replication

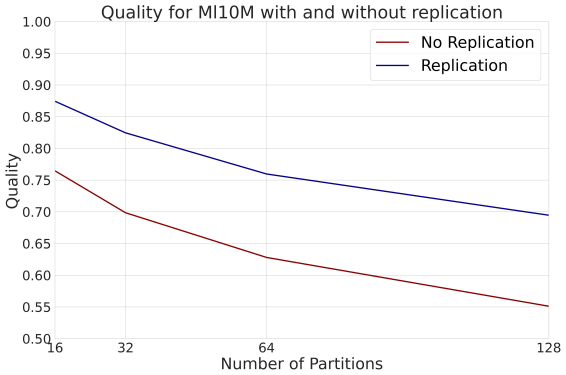


FIGURE 4.8
Quality on ML10M in function of the number of partitions with and without replication

Figure 4.7 displays the evolution of the runtime for the new algorithm, with the corresponding measures on the first algorithm. As we can see, the runtimes exploded compared to the version without replication when used on more data. The blue curve does not have the standard deviations, as the algorithm was not run five times with $n = 32$ and $n = 16$. Even though lower runtimes

n	Quality		
	No duplication	Duplication	Gain
16	0.7646	0.8743	+0.1097
32	0.6985	0.8245	+0.1260
64	0.6280	0.7569	+0.1289
128	0.5511	0.6946	+0.1435

TABLE 4.2
Quality of the resulting kNN graphs on ML10M with and without duplication

had been achieved earlier in the semester, the application on this larger dataset once again shows the limitations of such naive approaches. With $n = 128$ resulting in approximately 3 minutes and $n = 16$ reaching 1 hour, duplicating the data before partitioning with only a few partitions is very costly. The ML10M dataset contains 69'816 users with ratings above 3, [5] meaning that with $n = \{16, 64, 128\}$, each bucket respectively contains approximately 8727, 2181 and 1090 users before removing the duplicates. Hence, the number of users per partitions for $n = 64$ corresponded roughly to the limit chosen for Cluster and Conquer. While the runtime is very close when using 128 and 64 partitions, it does not hold for the smaller values studied. The authors of the paper similarly found that starting at $N = 3000$, the quality in function of the runtime plateaued in the shape of a logarithmic function (see figure 7 [5]). A limit of approximately 2000 users per partition seems to also be the best regarding the trade-off between time and quality.

Figure 4.8 shows the resulting qualities of the kNN graphs generated with those same number of partitions. As we can see, supported by the numbers in Table 4.2, the gain in quality very faintly diminishes, as the number of partition decreases. Still, simply having two instances of each user in the dataset allows to improve by more than 0.1 the quality of the resulting kNN graph. However, while the improved quality does reach almost 0.9 with 16 partitions, the quality of the graphs generated using 64 only reaches 0.75. This value seems to coincide with the result in quality that was found by the authors of Cluster and Conquer; with *t=2* hash functions, the quality they measured seemed to fall between 0.7 and 0.75. While this algorithm reached very high qualities for ML1M, it seems that replicating multiple times each users would be necessary to reach similar qualities.

CHAPTER 5

CONCLUSION

5.1 DISCUSSION & LESSONS LEARNED

The initial objective of this project was to adapt the Cluster and Conquer algorithm to a distributed setting. While an algorithm successfully recreating it was indeed implemented and tested, the results measured did not get close to reaching the performances reported by the authors. Spark is a very powerful framework for distributed computing, but its inner workings can be hard to grasp at first. In particular, given the resulting runtimes measured and the lack of scalability, it is clear that the current implementation is not optimal for the framework and would need some refactoring. The fact that it may better to restart the implementation from scratch instead of trying to fix specific aspects is amongst the main lessons learned during this project. The results obtained in Chapter 4 are a good start, but it would have been interesting to also study how the choice of clustering scheme impacted the results. In particular, if using the most occurring hash value from a user's profile instead of the minimum when choosing its bucket would have yielded similar results in terms of quality. Finally, another lesson has been to anticipate better the inconsistencies of the cluster when running the experiments, by performing more rigorous experiments throughout the semester and not be faced with the knowledge that better results were obtained in the past, but not reachable when the final measures where made.

5.2 CONCLUSION

To conclude, the results presented by the Cluster and Conquer authors are very impressive. And while this distributed implementation of the algorithm did not achieve close to similar runtimes, the quality obtained in our experiments were satisfactory. This algorithm presents characteristics making it both easier and harder to distribute. On one hand, the many independently computed local kNN graphs on fractions of the data are very adapted to parallel computing. On the other hand, the recursive splitting step limits the size of the buckets but calls for very expensive computations and a lot of data movement.

The two simple algorithms that were implemented and tested allowed to understand better the impact of some of the choices made for the Cluster and Conquer algorithm. Using a "divide and conquer" approach for the kNN graph construction problem can yield very satisfying results,

but some aspects of the chosen algorithm seem to be key to generating high-quality graphs without the constraint of time. Splitting the users across partitions and computing local graphs can help reduce considerably the runtime if the number of users per cluster is below a certain value. However, in order to maintain a certain quality in the final graph, the users need to be represented multiple times across the partitions. This creates a trade-off between time and quality that both depend on the number of clusters.

It also seems like the way the users are clustered in the first place might have an important role in the final graph's quality. This may be what helped the authors of Cluster and Conquer mitigate the impact of the trade-off. Measuring the impact of the clustering and the minimum function inside it, similarly as it was done for duplicating, would be the next step in trying to understand what are the main factors behind the impressive results.

BIBLIOGRAPHY

[1] Antoine Boutet et al. 'HyRec: leveraging browsers for scalable recommenders'. In: *Middleware* (2014). URL: https://infoscience.epfl.ch/record/208035?ln=en.

[2] Pedro Campos et al. 'Simple time-biased KNN-based recommendations'. In: *ACM International Conference Proceeding Series* (Sept. 2010). DOI: 10.1145/1869652.1869655.

[3] Wei Dong, Charikar Moses and Kai Li. 'Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures'. In: 2011. DOI: 10.1145/1963405.1963487. URL: https://doi.org/10.1145/1963405.1963487.

[4] 'E-commerce recommendation applications'. English (US). In: *Data Mining and Knowledge Discovery* 5.1-2 (Jan. 2001), pp. 115–153. ISSN: 1384-5810. DOI: 10.1007/978-1-4615-1627-9_6.

[5] George Giakkoupis et al. 'Cluster-and-Conquer: When Randomness Meets Graph Locality'. In: *CoRR* abs/2010.11497 (2020). arXiv: 2010.11497. URL: https://arxiv.org/abs/2010.11497.

[6] F. Maxwell Harper and Joseph A. Konstan. 'The MovieLens Datasets: History and Context'. In: *ACM Trans. Interact. Intell. Syst.* (2015). DOI: 10.1145/2827872. URL: https://doi.org/10.1145/2827872.

[7] G. Linden, B. Smith and J. York. 'Amazon.com recommendations: item-to-item collaborative filtering'. In: *IEEE Internet Computing* 7.1 (2003), pp. 76–80. DOI: 10.1109/MIC.2003.1167344.

[8] Olivier Ruas. *Sampling KNN*. 2020. URL: https://gitlab.inria.fr/oruas/SamplingKNN.

[9] *Spark Scala documentation for Range Partitioner*. 2021. URL: https://spark.apache.org/docs/latest/api/scala/org/apache/spark/RangePartitioner.html.

[10] *Spark Scala documentation for ShuffledRDDs*. 2021. URL: https://spark.apache.org/docs/latest/api/scala/org/apache/spark/rdd/ShuffledRDD.html#repartition(numPartitions:Int)(implicitord:Ordering[T]):org.apache.spark.rdd.RDD[T].

[11] *Spark Scala source code for RDDs*. 2021. URL: https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala.

[12] *Spark Scala source code for sortBy*. 2021. URL: https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/OrderedRDDFunctions.scala.