Programme ANR VERSO

Projet VIPEER

Ingénierie du trafic vidéo en intradomaine basée sur les paradigmes du Pair à Pair

Décision n° 2009 VERSO 014 01 à 06 du 22 décembre 2009
T0 administratif = 15 Novembre 2009
T0 technique = 1er Janvier 2010

Deliverable 4.3
Report on CDN/dCDN design

Auteurs:
C. Bothorel (Telecom Bretagne), Z. Li (Telecom Bretagne), P. Michiardi (Eurocom),
Y. Hadjadj-Aoul (INRIA), J. Garnier (NDS Technologies France)
Edited by:
J. Garnier (NDS Technologies France)

February 2012

Telecom Bretagne; Eurocom; INRIA; NDS Technologies

Abstract

This document aims to present the report on the CDN / dCDN design. As you can see in the architecture and the work done during the previous deliverables, VIPEER will manage a huge amount of data, regarding: 1. network: routing, numbers of routers (cache configuration), bandwith... 2. chunks: each movies is split in many chuncks depends on the quality, the aspect ratio... 3. users informations: region, devices, historique. That's why we introduce the Parallel Processing as we need a method for processing large amount of data in a efficient way. For exemple, if you want to distribute the chuncks based on network information traffic or you want to predict the video consumption of a user based on the analysis of all the sytem log files, this represents a huge amount of data. On the second part and based on the preliminary report, we have identified that a recommender engine can be used in the dTracker once distributing the chuncks. Finally, we will see how to arrange the placement of video content in a large-scale distributed Content Delivery Network (dCDN) system in order to have the optimal placement using the previous work on the prediction and recommendation and also the MapReduce framework.

Keywords: Prefetching, Map reduce, Genetic Algorithm.

Contents

1 Preface 7
1.1 Purpose of this document 7
1.2 Referenced ViPeeR deliverables 7

2 Parallel Processing in VIPEER 9
2.1 MapReduce Background 9
2.2 Scheduling MapReduce Jobs 10
2.2.1 Introduction 10
2.2.2 The Scheduling Protocol 12
2.2.3 Implementation details 16
2.2.4 Conclusions and Future Work 18

3 Prefetching 19
3.1 Introduction 19
3.2 Related work in the VoD distribution context 20
3.3 Scientific and technological challenges 21
3.4 Recommender systems 22
3.5 MapReduce Recommendation 23
3.5.1 VoD downloads dataset 23
3.5.2 Apache Mahout boolean recommender 25
3.6 Evaluation 27
3.6.1 Realistic chronological scenario 27
3.6.2 Netflix scenario: fixed k recommendations, testing with the K latest downloads 28
3.6.3 Adaptative scenario: variable k above a threshold, testing with the K latest downloads 28
3.7 Is recommendation relevant for Vipeer? 29
3.8 Conclusion 30

4 Optimal placement of video content 33
4.1 Introduction 33
4.2 Problem Formulation 34
4.3 Introduction of Genetic Algorithm 35
4.3.1 Basic Elements 35
4.3.2 Typical Procedure 35
4.4 Modeling k-PCFLP by Genetic Algorithm 36
4.4.1 Encoding 36

4.4.2 Fitness Function 36
4.4.3 Initialization and selection 38
4.4.4 Crossover and Mutation 38
4.4.5 Replacement and Termination 39
4.5 Perspective of Using MapReduce 39

5 Conclusion 41

42

Bibliography 43

1 Preface

1.1 Purpose of this document

This document aims to present the report on the CDN / dCDN design. As you can see in the architecture and the work done during the previous deliverables, ViPeeR will manage a huge amount of data, regarding: 1. network: routing, numbers of routers (cache configuration), bandwith... 2. chunks: each movies is split in many chuncks depends on the quality, the aspect ratio... 3. users informations: region, devices, historique. That's why we introduce the Parallel Processing as we need a method for processing large amount of data in a efficient way which is implemented in Hadoop. But why did we decide to use it? Traditional distributed systems have to deal with: - far more data than was the case in the past (Facebook: over 70 PB of data) - generate data at a rate of terabytes per day - data was stored on a SAN and copied to the compute nodes (transfer and synchro issues) - spend a lot of times to design system for failure rather than investigation on solving the problem with data - data becomes the bottleneck Hadoop: the proposal * system must support partial failure * when a component fails, its workload should be assumed by another functioning unit in the system * a failed component that recovers can rejoin the system (no full system restart) * partial failure during job execution shall not affect the results of the job * adding / removing component resources shall result in proportional decline or increase of load capacity

On the second part and based on the preliminary report, we have identified that a recommender engine can be used in the dTracker once distributing the chuncks. Finally, we will see how to arrange the placement of video content in a large-scale distributed Content Delivery Network (dCDN) system in order to have the optimal placement using the previous work on the prediction and recommendation and also the MapReduce framework.

1.2 Referenced ViPeeR deliverables

Table 1 lists documents and other reference sources containing information that may be essential to understanding topics in this document.

No	Designation	Title
1.	D4.1	State of the Art
2.	D4.2	Preliminary report on the CDN/dCDN design

2 Parallel Processing in VIPEER

In this Section we first provide a short backround description of MapReduce, a popular parallel processing framework, underlying the components that are relevant for our work on scheduling analytic jobs, which we present in Sec. 2.2.

2.1 MapReduce Background

Before delving into the details of the inner functoning of MapReduce, we introcuce this Section by discussing the applications that we intend to study in the context of VIPEER. Essentially, the project includes components that operate on large amounts of data (e.g. the prefetching component, that may include a recommender engine) or that are required to perform several iterations over large inputs to compute the near-optimal allocation of content – or fractions thereof – to be stored in the distributed CDN (the dCDN). To this end, MapReduce, which we describe below, constitutes a reasonable approach to address such setting. In what follows, we will focus on our frist attempt at working with MapReduce in the context of VIPEER.

MapReduce, popularized by Google with their work in [6] and by Hadoop [1], is both a programming model and an execution framework. In MapReduce, a job consists in three phases and accepts as input a dataset, appropriately partitioned and stored in distributed file system. In the first phase, called Map, a user-defined function is applied in parallel to input partitions to produce intermediate data stored on the local file system of each machine of the cluster; intermediate data is sorted and partitioned when written to disk. Next, during the Shuffle phase, intermediate data is "routed" to the machines responsible for executing the last phase, called Reduce. In this phase, intermediate data from multiple mappers is sorted and aggregated to produce output data which is written back on the distributed file system. Note that complex jobs may require several iterations or combinations of Map-Shuffle-Reduce phases.

In this Section we gloss over several details of MapReduce and focus on the key ingredients that define the performance of a job, in terms of execution time. Simply stated, disk and network I/O are the main culprits of poor job performance. The task of a job designer is then to optimize the amount of memory allocated to mappers and reducers, so as to minimize disk access. Moreover, a job may include an optional *Combiner* phase in which intermediate data is pre-aggregated before it is sent to reducers, to minimize the amount of data to be transmitted over the network. Job optimization is generally a manual process that requires the knowledge

of the size of intermediate data sent to each reducer, and the characteristics of the cluster, including number of nodes, number of processors and cores and available memory.

A key component of the MapReduce framework is the scheduler, which is the subject of this work. The role of the scheduler in MapReduce is to allocate resources to running tasks: Map and Reduce tasks are granted independent slots on each machine. The number of Map and Reduce slots is a configurable parameter.

When a single job is submitted to the cluster, the default scheduler simply assigns as many Map tasks as the number of machines in the cluster. Note that the total number of Map tasks is equal to the number of partitions of input data: as such, the scheduler may need to wait for a portion of Map tasks to finish before scheduling subsequent mappers. Similarly, Reduce tasks are scheduled once all intermediate data output from mappers is available: reducers may receive data from potentially all mappers. Note that Hadoop implement an optimization called "pipelining": if all reducers were to be scheduled once all mappers complete, the network would suffer from a burst in data transmission and congestion may arise. Instead, with pipelining, reducers are scheduled before mappers are done[1] such that they can start copying intermediate data as soon as some is available. However, the user-defined Reduce function is only executed once all mappers are done.

When multiple jobs are submitted to the cluster, the scheduler decides how to allocate available task slots across jobs. The default scheduler in Hadoop implements a FIFO policy: the whole cluster is dedicated to individual jobs in sequence; optionally, it is possible to define priorities associated to jobs. Despite the well known problems of FIFO scheduling, which starves short jobs that sit in a queue waiting for a long job to finish, jobs can be optimized as if they were executing alone in the cluster, which is a desirable feature.

2.2 Scheduling MapReduce Jobs

This Section is devoted to describe in detail our work on one of the main components of a parallel processing framework in general and of MapReduce in particular. In the VIPEER project, it is reasonable to believe that the parallel processing framework will be used by several other components, especially those designed in WP4; the following material is also relevant to WP2.

2.2.1 Introduction

The advent of large-scale data analytics, fostered by parallel processing frameworks such as MapReduce [6] and Dryad [17], has created the need to organize and manage the resources of clusters of computers that operate in a shared environment. Initially designed for few and very specific batch processing jobs, data-intensive scalable computing frameworks are nowadays used by many companies (e.g. Facebook, LinkedIn, Google, Yahoo!, ...) for production, recurrent and even experimental data analysis

[1] Precisely, a configuration parameter indicates the fraction of mappers that are required to finish before reducers are awarded a slot.

jobs. Within the same company, many users *share* the same cluster because this avoids redundancy (both in physical deployments and in data storage) and may represent enormous cost savings.

In this work, we study the problem of resource *scheduling*, that is how to allocate the (computational) resources of a cluster to a number of concurrent jobs submitted by the users, and focus on the open-source implementation of MapReduce, namely Hadoop [1]. Despite scheduling is a well known research domain, the distributed nature of data-intensive scalable computing frameworks makes it particularly challenging. In addition to the default, first-in-first-out (FIFO) scheduler implemented in Hadoop, recently, several solutions to the problem have been proposed to the community [28, 5, 8, 18, 23, 25]: in general, existing approaches aim at two key objectives, namely *fairness* and *performance*.

For example, [28] propose a scheduler that in principle is equivalent to processor sharing; in addition, the authors note that data locality, which imposes computation to be moved toward the data rather than vice-versa, plays an important role in the efficiency of a particular job schedule. By simply waiting for a suitable resource to be available, the "delay scheduler" [28] greatly reduces the amount of data that has to be moved within the shared cluster. In another work [5], scheduling is cast as an optimization problem: using an abstract system model, the authors focus on finding an approximate solution to an on-line scheduling problem whose objective function to minimize is the total time required to complete all jobs served by the system, while fairness is not considered. The work presented in [8] considers a non-cooperative scenario in which users may be inclined to "game" the scheduler in order to receive more than their fair share of the cluster, and proposes a new, strategy-proof metric to be enforced by the cluster scheduler.

Given the state-of-the-art, it is natural to question the need for another approach to scheduling cluster resources. In this work we observe that fairness and performance are non-conflicting goals, hence there is no reason to focus solely on one or the other objectives. We proceed with the design of a scheduling protocol that can be implemented in practice, and that caters both to a fair and efficient utilization of a shared cluster.

First, in Sec. 2.2.2, we give an high level overview of our algorithm, and compare it to traditional fair-sharing approaches. When the demand for cluster resources is high, *i.e.*, jobs necessitate all cluster machines, our scheduler "focuses" the resources to an individual job, instead of partitioning the cluster to accommodate all jobs. When the workload of a cluster is composed by heterogeneous jobs, in terms of resource requirements, our scheme accommodates the possibility of running small jobs in parallel, so that the cluster is fully utilized.

In Sec. 2.2.3, we delve into the implementation details of our scheduler. In contrast to previous works, we pinpoint the key aspects of the MapReduce framework that need to be considered when implementing a scheduling policy: as a result, we show that locality constraints are important both for input data and for intermediate results generated during the job execution flow. Furthermore, we show that our approach to scheduling supports job optimization techniques that aim at minimizing I/O operations. This is very important as MapReduce job designers usually spend a non-negligible amount of time in optimizing their jobs: this effort can be nullified

by a scheduler that is oblivious to such optimization techniques.

Sec. 2.2.4 concludes this article with a series of further improvements that can be applied to our scheme, and with our research agenda toward the implementation and evaluation of our scheme.

2.2.2 The Scheduling Protocol

Scheduling in a distributed system like Hadoop represents a challenging problem: theoretical results for such a complex environment are hard to derive. For this reason, we adopt a top-down approach. We first consider the distributed system as a single processing resource: according to the measurements in [28], jobs arrive according to a Poisson process, while job size distribution is unknown. Therefore we can look at the general results for the M/G/1 queueing system and select the most promising scheduling policy. Such scheduling policy would be subsequently adapted to the specific context of Hadoop.

2.2.2.1 Scheduling Disciplines

Scheduling has been a subject of many studies. Here we focus on the theoretical results for scheduling policies in case of M/G/1 systems. We consider the mean response time (i.e. the total time spent in the system, given by the waiting and service time), and the fairness. While fairness is a complex subject, we consider the notion of fairness as equal share of the resources[2].

Among all the scheduling disciplines proposed in the literature (for a general overview, see [11] and the references therein), we consider only two policies that are relevant to our case: a policy that minimizes the mean response time and one that provides perfect fairness.

In the system we consider, the job size is known *a priori*: in this case the optimal preemptive scheduling policy that minimize the mean response time is the Shortest Remaining Processing Time (SRPT), where the job in service is the one with the smallest remaining processing time. Since the focus is on the mean response time, the fairness is not guaranteed, i.e., long jobs may starve. SRPT represents an interesting solution, since recent measurements [28] have shown that the job size distribution in MapReduce clusters belongs to the category for which SRPT may provide fairness; nevertheless, it is also true that there are many short periodic jobs – a case where SRPT may perform poorly [7].

If we consider only the fairness, Processor Sharing (PS) represents the policy that guarantees a fair share of the resources. In PS, if there are N jobs in the system, each job receives a $1/N$th fraction of the server resources. Unfortunately, the mean response time is higher (especially for high loads) than SRPT.

Given the two objectives, good performance (in terms of mean response time) and fairness, is it possible to obtain both with a scheduling policy? The solution of this problem is represented by the Fair Sojourn Protocol (FSP) [7], described in the following section.

[2] Alternatively, the fairness can be defined through the total time spent in the system normalized to the job size, which should be proportional to $1/1-\rho$, where ρ is the server utilization.

2.2.2.2 How Fair Sojourn Protocol Works

The main idea of FSP is to run jobs in series rather than in parallel. In practice, assuming a PS policy, where each job has its fair share, it is possible to compute the completion time. The order at which jobs complete in PS is used as a reference to schedule jobs in series. In the basic single server configuration, this means that at most one job is served at a time, and the job can be preempted by a newly arrived job.

In order to show how FSP works, we make an example. Assume that there are three jobs, j_1, j_2 and j_3, which require 100% of the cluster. The jobs arrive at time $t_1 = 0$s, $t_2 = 10$s and $t_3 = 15$s respectively, and it takes 30 seconds to process job j_1, 10 seconds to process job j_2 and 10 seconds to process job j_3. For simplicity, assume that the processing time is independent from the location of the tasks, i.e., even it some tasks need to be moved within the cluster, the processing time remains the same.

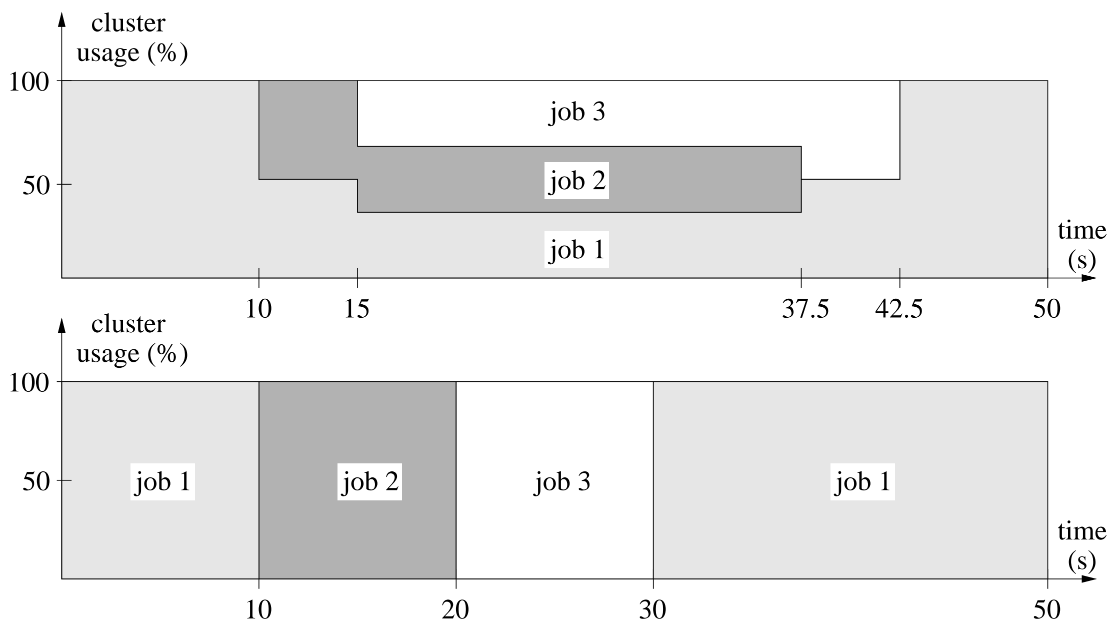

Figure 2.1: Comparison between PS (top) and FSP (bottom).

Figure 2.1 (top) represents the cluster usage over time in case of PS: when job j_2 arrives, the cluster is shared between j_1 and j_2, and, when job j_3 arrives, the cluster is shared among the three jobs. The job completion order is j_2, j_3 and j_1. The bottom part of the figure shows the FSP approach. When job j_2 arrives, since it would finish before job j_1 in case of PS, it preempts job j_1. When job j_3 arrives, it does not preempt job j_2, since it would finish after it in case of PS; when job j_2 finishes, job j_3 is scheduled since it would finish before job j_1 in case of PS.

The FSP scheme is able to assure that each job receives the fair amount of resources as in the PS scheduling. At the same time, the scheme is able to decrease the mean job completion time. In particular, long jobs tends to have the same completion time as in PS, while short jobs finish before.

While the formulation of FSP is simple in case of single server, when we take into accounts a cluster of servers, we should adapt the scheme to this specific context. In particular, it may happen that a job is composed by few tasks, and it is sufficient to use a portion of the cluster to process such job at the maximum possible speed. We illustrate this situation in the following example. Assume that jobs j_1, j_2 and j_3 require 100%, 55% and 35% of the cluster respectively. The arrival times are $t_1 = 0$s, $t_2 = 10$s and $t_3 = 13$s and the processing time (if the required percentage is given) is 30 seconds for job j_1, 10 seconds for job j_2 and 10 seconds for job j_3.

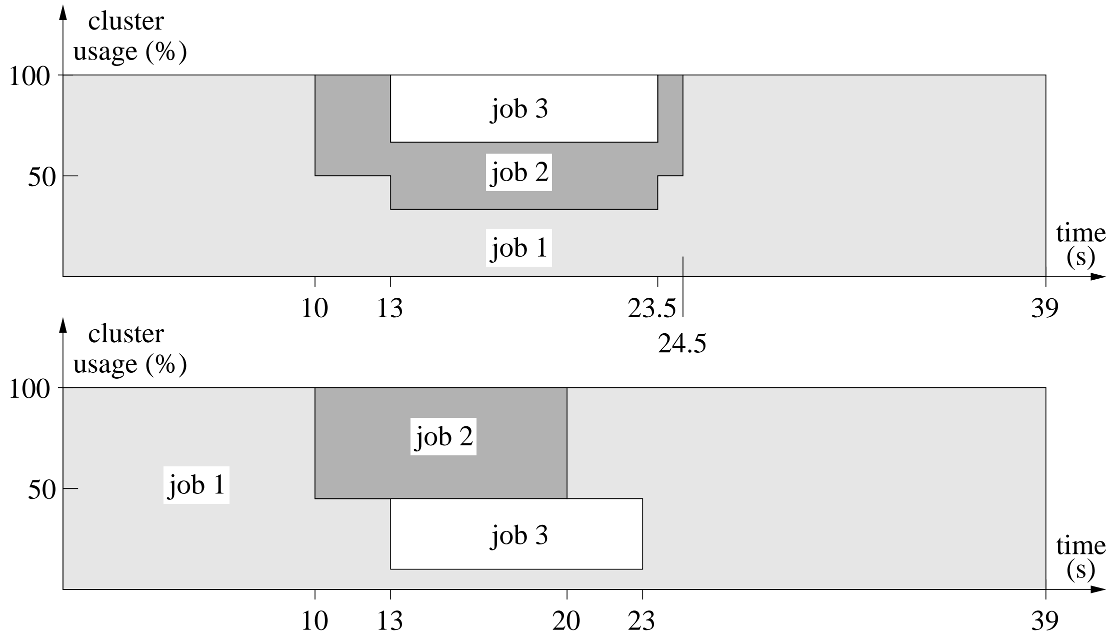

Figure 2.2: Comparison between PS (top) and FSP (bottom), with jobs that do not require the full cluster.

Figure 2.2 compares the processing in case of PS (top) and FSP (bottom) approach. With FSP, job j_2 would preempt job j_1; since j_2 requires only 55% of the cluster, the remaining 45% can still be used by j_1. When job j_3 arrives, it would preempt job j_1 (but not job j_2), but it is sufficient to allocate 35% of the cluster to serve it, leaving 10% of the cluster to job j_1. Even in this case, FSP is able to decrease the mean job completion time, yet maintaining the fair allocation of the resources. Note that the final order of job completion with FSP is different from the PS case (j_2, j_3 and j_1 instead of j_3, j_2 and j_1): in this case job j_2 finishes before the corresponding completion time in case of PS, therefore the fair allocation of the resources is not compromised.

Having described with some examples the general behavior of the scheme, we provide the basic algorithm for task allocation implemented in the scheduler. As observed in [28], when the scheduler needs a resource, it can either kill some existing running tasks or wait for task completion. Since killing task would be a waste of resources, it is preferable to wait for available task slots: thanks to the high rate at which task finish, this does not represent a major problem. In our specific case, if a

newly submitted job preempts the existing one (or ones), we simply let the current running task complete and launch the new job as task slots become available.

The general scheme is therefore composed by two parts which perform different actions for the two possible events: (i) job submission or completion, and (ii) task completion (see Algorithm 1).

Algorithm 1 Basic Task Allocation

```
1.  while a job is submitted / finishes do
2.    for all jobs do
3.      compute the fair share
4.      compute the finish time
5.    end for
6.    sort jobs according to their finish time
7.  end while
8.
9.  while a task slot is available on machine M do
10.   for j in jobs do
11.     if j has an unlaunched task t then
12.       launch t on M
13.       return
14.     end if
15.   end for
16. end while
```

When a job is submitted or finishes, the algorithm computes the fair share for each job and updates the finish times. These values are used to order the jobs, i.e. priority is given according to the completion time. When a task slot becomes available, the scheduler considers the ordered list of current jobs and assigns the task to the higher priority job that has at least one unlaunched task – a job may still be running, but all its tasks (e.g., all the Reduce tasks) may have been already allocated.

The proposed basic scheme does not take into account different aspects, e.g., task locality or user fair share (as opposed to job fair share) which are discussed in the next section.

2.2.2.3 Improving basic FSP scheme

The scheme summarized in Algorithm 1 is divided into two parts: job sorting and task assignment. These two building blocks can be modified independently, since they solve different problems.

The job ordering can be done taking into account multiple aspects: for instance, we may assume that jobs belong to different classes, and each class has different shares of the cluster. In this case the job completion times are computed considering the Generalized Processor Sharing (GPS), where every job i has a weight ϕ_i and it receives a share $\phi_i / \sum_j \phi_j$ (the sum over j is done considering the current jobs) of the resources. Another modification may take into account users rather than jobs, *i.e.*, jobs submitted by the same user are served with FIFO, and the resources are shared among users.

The task assignment mainly tries to solve problems related to locality. Given the ordering of the jobs, the task assignment may skip some jobs if the locality requirement is not met. The implementation of this approach can be borrowed directly from the Delay Scheduler [28].

As a final remark, we should mention that FSP needs to estimate the job finish time. While, as a first approximation, we may consider the job size, we will evaluate more detailed mechanisms (that take into account, for instance, the difference between CPU-intensive and memory-intensive jobs) as future work.

2.2.3 Implementation details

In Sec. 2.2.2 we assume a simplified version of the MapReduce framework to outline the key ideas behind FSP: for instance, we do not differentiate between Map and Reduce tasks. We now drop such assumptions and delve into the implementation details of FSP.

2.2.3.1 Preempting jobs

The solution we propose is based on job preemption. How such preemption should be managed? In order to understand this aspect, let's focus on a detailed example, shown in Fig. 2.3, where we consider two jobs, j_1 and j_2: j_1 operates on a dataset split in 10 blocks, involving 10 Map tasks and 3 Reduce tasks; j_2 is smaller, and consists in 4 blocks, 4 Map tasks and 4 Reduce tasks. In the example we consider a cluster of 4 machines, with 1 Map and 1 Reduce slot each. Fig. 2.1 illustrates, on a time-line, the task allocation per machine, per slot and per task, with salient points annotated on the time-line.

Note that the ingredients of this example are similar in spirit to that of Fig. 2.1 (although with two jobs instead of three). Both jobs require all cluster resources (at least in the Map phases) since input blocks are located on all the cluster machines: as such, j_2 *preempts* j_1.[3]

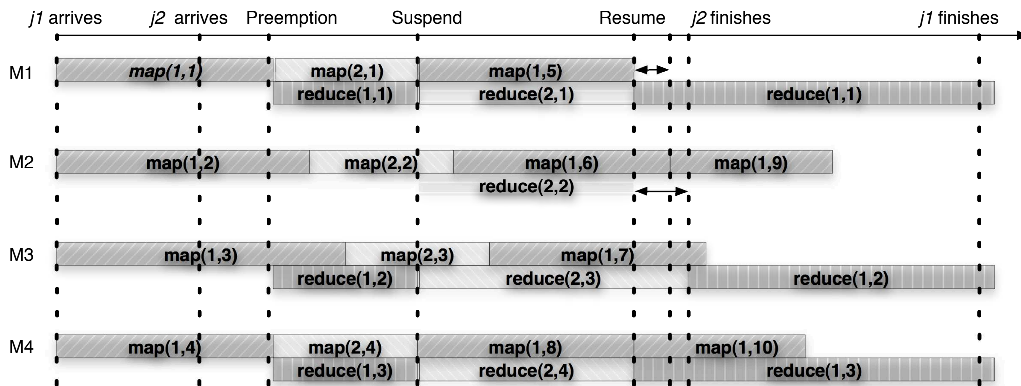

Figure 2.3: A detailed scheduling example with FSP.

[3]A close look at Fig. 2.3 reveals that job "serialization" is not as sharp as illustrated in Fig. 2.1.

In Fig. 2.3, *map*(i, j) indicates Map task number j of job i; similarly *reduce*(i, j) indicates a Reduce task number j of job i. First, j_1 arrives and occupies the whole cluster: the first 4 Map tasks of j_1 are scheduled. After some time, j_2 arrives: based on Algorithm 1, j_2 preempts j_1. Preemption of Map tasks is not immediate: the FSP scheduler waits for mappers from j_1 to finish to schedule mappers from j_2.

It is now important to note how FSP handles Reduce task scheduling: as soon as[4] Map tasks from j_1 finish, Reduce tasks from the same job are scheduled on the available slots: in the figure, the three Reduce tasks from j_1 are scheduled on machines M_1, M_3, M_4.

Now, when mappers from j_2 are done[5], FSP schedules the reducers. To do so, two options are available: Reduce tasks from j_1 can be killed or, as we suggest, *suspended*. Killing Reduce tasks, as implemented in [28], is convenient because the scheduler is simplified: reducers are simply tagged as not completed and will be scheduled subsequently. However, this simplification comes at a price: all work done by the reducers, including disk and network I/O is wasted.

Since I/O is the major factor that determines the job execution time, we instead suggest to suspend reducers and save their state to disk. Reduce task suspension can be implemented as follows: the scheduler can force the TaskTracker in charge of a reducer to *spill* data to disk; once this is done, the scheduler pushes the suspended reducers in the queue of pending Reduce tasks. One key observation is necessary: as intermediate (and sorted) data is materialized on disk, suspended Reduce tasks must be resumed on the same machines they were suspended on. In practice, *data locality* applies to both Map and Reduce tasks. We show this in Fig. 2.3: once mappers of j_2 are done, reducers for j_2 are scheduled, while reducers from j_1 are suspended. In the meanwhile, new Map tasks from j_1 are scheduled on available Map slots. When reducers from j_2 complete (and j_2 finishes), the reducers from j_1 can be resumed: however, note that although machine M_2 has an available Reduce slot, this cannot be used by any of the suspended reducers from j_1, due to data locality. When the Reduce slot on machine M_3 becomes available, the Reduce task for j_1 can be resumed. Fig. 2.3 illustrates (through an example) also the effect of data locality on mappers: the 9-*th* mapper from j_1 is not scheduled on machine M_1 as it becomes available, but it is delayed to wait for machine M_2, which is hosting an un-processed block of data.

2.2.3.2 Optimized jobs

MapReduce job optimization consists in tuning the parameters of the framework, and in using *combiners*, so as to minimize I/O. Simply stated, given the characteristics of the machines of the MapReduce cluster, an optimized job rarely writes on (the local) disk, reduces the amount of intermediate data to be *shuffled*, and performs sorting operations in memory. As such, each job must be properly tuned in order to use at best the resources available to a machine.

[4]The default behavior in Hadoop is that reducers are launched when 5% of the mappers have completed. As such, reducers can initiate intermediate data transfers; however, the Reduce function is only executed once all mappers from a job have completed.

[5]Precisely, when one mapper is finished

With the default scheduler, namely a FIFO approach, jobs are serialized and can thus fully exploit the resources of each machine. Alternative scheduling approaches appeared in the literature, instead, tend to parallelize as much as possible jobs, so as to achieve a fair sharing of cluster resources. This has the negative impact of hindering job optimization because it is impossible to predict the number (and the nature) of concurrent jobs scheduled on each machine. As such, from the practical perspective, a scheduling approach that achieves fairness while granting the job designer with the possibility of tuning each job, is a desirable property.

Our approach to scheduling achieves exactly what discussed above: when jobs require all cluster resources, they have access to the full capabilities of each machine they are scheduled on, in contrast to having the "illusion" of running alone in the cluster. When jobs are small, they share the cluster and can be processed in parallel, as shown in Fig. 2.2.

2.2.4 Conclusions and Future Work

In this work we presented the design of a new scheduling algorithm for a shared cluster dedicated to MapReduce jobs. In contrast to previous works, that abstract away parts of or the whole system, we pinpoint the impact of scheduling on the inner components and phases of MapReduce jobs, paying attention to current best practices in job optimization. Such an approach allowed us to extend the concept of data locality and apply it to the Reduce phase, which is a desirable feature especially when reducers can be suspended instead of being simply killed, as done in previous works.

The scheduling algorithm presented in this article, labelled FSP, achieves two objectives that have only been considered separately in the literature: fairness in resource allocation and job performance. Currently, we focus on the implementation of FSP: inspired by the work in [28], we plan to issue a JIRA and contribute to the Hadoop community with a "contrib" FSP module. Subsequently, we will proceed with a thorough experimental evaluation of FSP and compare its behavior with other available schedulers, for a variety of job types (short and long) and composition (number of mappers and reducers).

3 Prefetching

The Vipeer project proposes an evolutionary and pragmatic method to efficiently deploy a novel video distribution architecture. This architecture is based on the collaboration between service providers —or traditional CDNs— and peer-assisted CDNs —or distributed CDNs, called dCDNs— operated by ISPs.

This architecture leads to a neat delineation between the role of a CDN and that of a dCDN. A CDN is in charge of disseminating content at a wide scale (say, at the international level) while a dCDN aims at delivering content within a medium size network (say, regional, typically an ISP's network). In such architecture, the network operator engineers the video traffic delivered to its customers.

3.1 Introduction

The main idea of Vipeer dCDNs is to decrease the load of the origin content server by serving clients from ISP managed caches that have been strategically placed close to the clients. The peers in the peer-assisted dCDN may be network elements such as network nodes or boxes located at customers premises.

The delivery system consists of a set of content caches which deliver content replicas to end-users. Among others, managing the dCDN comes with the management of the caches. The placement of the video content replicas among the different peers is described in Chapter 4.

The question here is which video content replicate among the peers so that the downloading process keeps been managed by the dCDNs (and not the traditional CDN). In order to choose the content, define the number of replicas of each content, and organize their placement close to the end-users, we design a Prediction and Recommendation (P&R) system.

When end-users start downloading their videos, their requests are registered and delivered to the P&R module. The aim is to analyze the collected downloading behavior and generate for each user k potential downloads during the next period of time. The dCDN prefetches one copy of the predicted pieces of content from the CDN, if necessary. The dCDN then optimizes the number of replicas and their placement according to the predicted behavior.

This chapter describes a MapReduce-based P&R system. We will mainly show here that it is actually relevant to use such a prediction system in Vipeer. The Recommendation engine is trained with a real VoD dataset provided by our ISP partner.

3.2 Related work in the VoD distribution context

P2P file downloading and streaming architectures dramatically reduce the server loading and provide scalable content distribution (cf. state of the art, Vipeer deliverable D4.1). When a file is downloaded by many users, systems like BitTorrent or Emule allow users to help each others: their storage unit maintains a replica which is distributed to others. Each peer stores a small amount of storage compared to CDNs. In this context, users manage their own cache which contribute to the whole set of caches. Popular content delivery or crowd behaviors are then intrinsically managed by the users themselves. But for unpopular content, it may be difficult to find any peer able to deliver it: users may be disconnected, only few replicas —if any— are available.

P2P is also used to distribute live streaming content (IPTV for example). Since the size of such a cache on each peer is limited, it is imperative that an appropriate cache replacement algorithm is designed. In this context again, the content to be delivered is related to a program schedule and is then similar to mass phenomena. *Passive* caching such as LRU or LFU is here effecient.

According to [15], P2P-VoD "is a new challenge for the P2P technology. Unlike streaming live content, P2P-VoD has less synchrony in the users sharing video content, therefore it is much more difficult to alleviate the server loading and at the same time maintaining the streaming performance." They monitored a real system deployed by PPLive in 2007, and performed various performance measures. In 2008, this PPLive P2P-VoD was already supporting more than 150K simultaneous users. The system passively managed the caches with a LRU-like policy. They did not perform any prefetching. Without any prefetching, only the VoD already downloaded within the system are stored in the caches. While prefetching may improve performance, and reduce server load, it may also unnecessarily waste precious peer uplink bandwidth. PPLive P2P-VoD did not choose to *actively prefetch*, especially when the system is deployed on ADSL peers with low uplink capacities.

[15] brings also a framework to study and measure different design choices from file segmentation strategy to content discovery and transmission. A real deployed system is analyzed, which is intrinsically interesting. They provide clues to understand the users behavior, metrics to measure their satisfaction in terms of network fluency, or metrics to quantify the content replication strategy. But unfortunately, the replication strategy is not compared to others, and the choice of no-prefetching is not discussed.

Wu and Li [26] provide a theoretical analysis of passive peer caching (used in real world such as PPLive) and demonstrate an optimal cache replacement policy. They compare it with different cache replacement algorithms (LRU, LFU). They argue from extensive simulations than in most cases, the simplest cache replacement policies are actually effective enough. Their optimal strategy decreases only marginally the average server load. In their study, they do not consider the use of active prefetching. They do not either describe in which cases the server load is not reduced. But they highlight the lack of robustness of passive policies against the churn. Yet it would be interesting to confront these two issues on the utility of the prefetching. In particular, for non- very popular content not enough replicated by

the logic of the LFU or the LRU.

Prefetching may be employed to optimize the content searching step. [13] aggregates seeking statistics and proposes an optimal prefetching scheme and an optimal cache management to minimize the seeking delay to find a position in a VoD content. They propose a seeking guidance. The guidance is obtained from collective seeking statistics of other peers who have watched the same title in the previous and/or concurrent sessions. From the collected seeking statistics, they estimate the segment access probability. They show the efficiency of the guidance, but also that it is very challenging to aggregate the statistics efficiently, timely and in a completely distributed way.

VOVO [12] models users' behavior at the peer level. Each peer collects its logs and propagates to its neighbors (gossip). Each peer predicts its own future downloads using on-line association rule mining. The tracker may be used to store large volume of user logs. Reinforcement learning techniques are used to maintain a model of user behavior. Each peer receives the learnt model as a bootstrap and then uses it to prefetch contents. Hybrid prefetching is then performed. A classic LRU policy is used to cache the lastest 5 minutes of video played. In parallel extra cache is used to prefetch related pieces of content based on the user's bahavior.

As a matter of fact, predictive prefetching reflects the whole group of users. There is no personalization. APEX [27] presents a personalization framework able to discover each user's preference. We may have this kind of issue if we detect that the popularity ranking of VoD depends on the users geographic area.

3.3 Scientific and technological challenges

In Vipeer, the peers are managed by the ISP, not by the users themselves (as in Emule for example). The problem is to anticipate future downloads so that the maximum of the VoD can be searched, retrieved and viewed in acceptable conditions for the end-users: small start-up latency, sustainable playback rate, etc. State of the art shows that popular content is effectively cached. We aim in this project, to improve the less popular content management.

- how to choose the content to be prefetched in order to provide a large coverage of content —not only popular ones— and then minimize the requests that go out of the ISP domain?

- how to distinguish profiles of users? Prefetching may not reflect the whole set of users, but may be adapted to communities of users? Or regional areas with specific behavior? How to take into account community/geographic zones preferences?

As mentionned in the previous section, we still have the issue of small but numerous caches, with limited bandwidth and storage capacities. The challenges are: which content to cache and where? The placement problem will be addressed in the Chapter 4. It will be seen as the optimal allocation as a k-product capacitated facility location problem.

3.4 Recommender systems

In order to prefetch content, we propose a recommender engine. The recommendation customizes the access to information for a given user, and thus facilitates the choice of content in a catalog too large for her/him to get an overall idea. In practice, recommendation systems, from knowledge about a user, filter a set of content and produce a list, often ordered, with these few selected content and deemed appropriate for her/him.

The research focuses on designing algorithms for selecting content for a given user with a known profile of tastes and/or purchases. The prime example is the e-commerce site Amazon[1] that directs the visitor to the content that other visitors have appreciated. Recommendation methods are used on other e-commerce websites (Fnac[2], Virgin[3], etc.), on music platforms (Lastfm[4], Pandora[5], etc.) or VoD merchant such as Netflix[6].

[22] reveals two main categories: techniques based on *collaborative filtering* which compute similarities between user profiles based on ratings (a number of stars, the list of past purchases) or similarities of profiles on the basis of content descriptors. The second family of techniques is based on *content filtering*, and it searches a thematic fit between user profiles and content profiles. [4] compares these different methods with different similarity measures and different assessment methods on two sets of reference data, including a very modern, the Netflix challenge dataset[7].

The state of the art of recommender systems algorithms lists quantity of calculation of similarity, with possibly the combination of techniques. The Netflix contest has offered a motivating challenge to researchers who rapidly improved the state of the art. Researchers like [14] wonder wether the quality gain of recommendation are now visible to the users: there is indeed a kind of "magic barrier" that prevents the system to achieve perfection as the ratings by users themselves are inconsistent or even contradictory.

Yet there are still scientific obstacles in the field, including the recommendation of not popular content, also called content in the Long Tail.

[3] compares different collaborative filtering algorithms and reports the weakness of many of them when considering sparse data. The sparsity of ratings leads to unrelevant similarities between users (or items). In our case, we address non popular content, i.e. VoD for which the "light" history of downloads does not allow confident recommendation. Each user downloads only a few VoD, and only few users download non-popular VoD, so most of the cells of the user-VoD matrix are empty. They also show that SVD-based techniques (involving matrix factorization) are the

[1] http://www.amazon.com
[2] http://www.fnac.com
[3] http://www.virgin.com
[4] http://www.lastfm.com
[5] http://www.pandora.com
[6] http://www.netflix.com
[7] Competition launched by Netflix, Inc.. an algorithm to find recommendations for films surpass theirs significantly: gain of more than 10 % in the relevance of the predicted scores (RMSE measure) on the test set provided. Netflix has rewarded the winners of a premium of one million dollars in 2009.

most efficient of the collaborative filtering techniques. They also propose their own method addressing large volume of data.

[21] demonstrate that it is difficult to predict a score for the Long Tail: using 9 different learning methods, they show that the prediction error grows for content for which there is only few notations. To overcome this problem, they propose a method to combine the content of the Long Tail into groups of similar content and then applying predictive methods for groups of items and not on each one. They combine the low ratings in an exploitable critical mass of notations. They also suggest ways to detect the boundary between the head and tail of the distribution, as well as to discover the number of clusters of rare items. These two parameters are difficult to choose, vary from one data set to another and affect the outcome significantly.

We have also explored a Social Network Analysis-based method [2] which improves the recall of predictions. The social network links help in computing similarities between users, and thus allow a better capture of users profiles. The method actually increases the portion of good items retrieved (by improving the recall, we miss less good results). The accuracy of the recommendations (precision) remains low, but at a level comparable to the other tested techniques on our dataset (less than 0.50 % of recommendations are among the movies annotated users, the main portion of prediction are unexpected results). Unfortunately, we will see that our Vipeer dataset is poor and does not allow such an approach.

We are still conducting experimentations with different techniques, and we describe here only the first and common collaborative filtering approach in order to justify the use of recommendation techniques in the Vipeer project.

3.5 MapReduce Recommendation

We test here the well-kown collaborative filtering approach. We used the concept of *item-based* recommender engine. The principle is simple: initially we determine the similarities between items based on usage data from users. Then in a second step, from the items already "preferred" by each user, we recommend $k-$items corresponding to her/his tastes.

There is another model called *user-based* recommender engine which takes things the other way, by calculating the similarities between users to recommend items. We turned to the item-based recommenders for two reasons: one being that in general, new items are added to the system less often than users. The similarities between items should not be calculated frequently (when the catalog is updated). The second reason is that it is more robust against the sparsity of the user-item matrix [24, 16, 3].

3.5.1 VoD downloads dataset

Our ISP partner provides us an extraction of the VoD downloading history from a regional zone. The commercial VoD service is legal and comes with the ISP offering.

For each download, the logs give the timestamp, the user ID and the film ID. 8,935 customers downloaded 108,108 VoD during 6 weeks (February-March 2010).

5,777 different movies were requested. Our previous deliverable D4.2 provides a detailed description of the downloaders profiles.

	Orange	Orange > 20	MovieLens
Items	5745	4812	1682
Users	8903	1200	943
Ratings	84905	41961	1000000
Density	0.1%	0.72%	6.3%

Table 3.1: User-Item matrix, comparison with the well-known MovieLens dataset.

The MovieLens dataset[8] contains real data corresponding to movie ratings. As this dataset is well-known in recommendation, we may compare our results with this dataset. Users with less than 20 ratings have been removed, but the User-Item matrix is still sparse with a low density (Table 3.1). Our own dataset is more sparse, with or without the occasional users (the ones with less than 20 downloads).

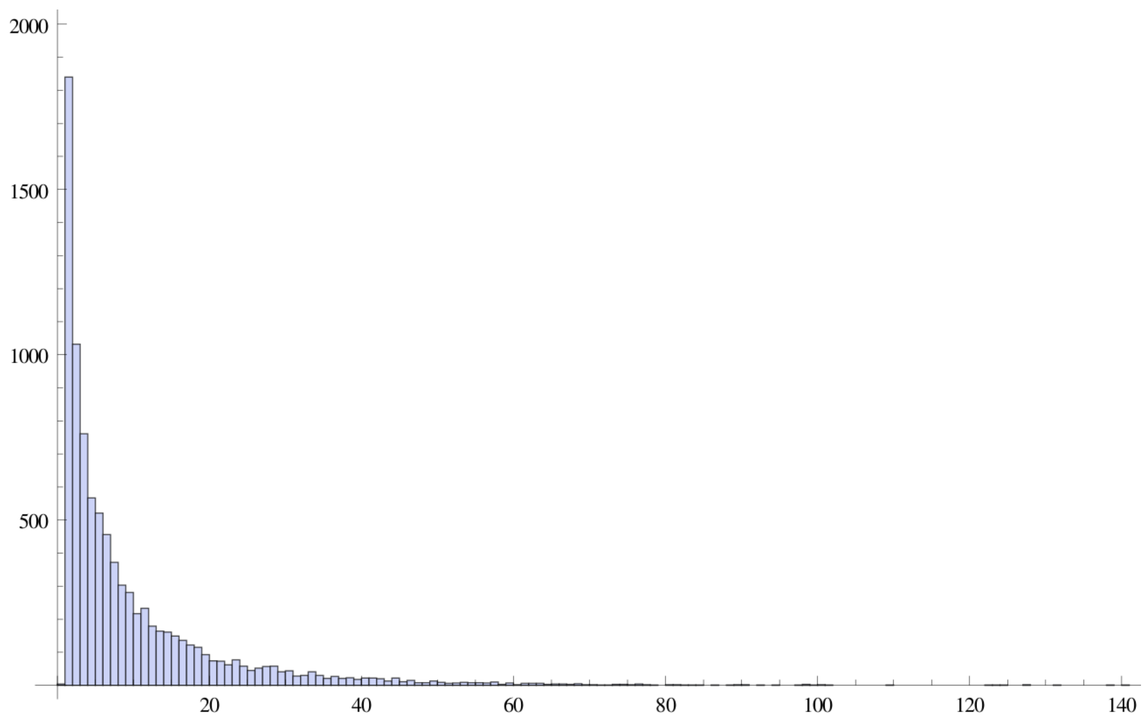

Figure 3.1: Great diversity of Orange VoD popularity: distribution of VoDs regarding the number of downloads. The 3 most popular VoDs were downloaded more than 700 times (in 1 month).

When we compare the popularity of VoDs between our original dataset and the one without the occasional users (Figures 3.1 and 3.2), we observe that occasional users contribute to very popular VoDs. We will show later that occasional users

[8]http://www.grouplens.org/node/73#attachments

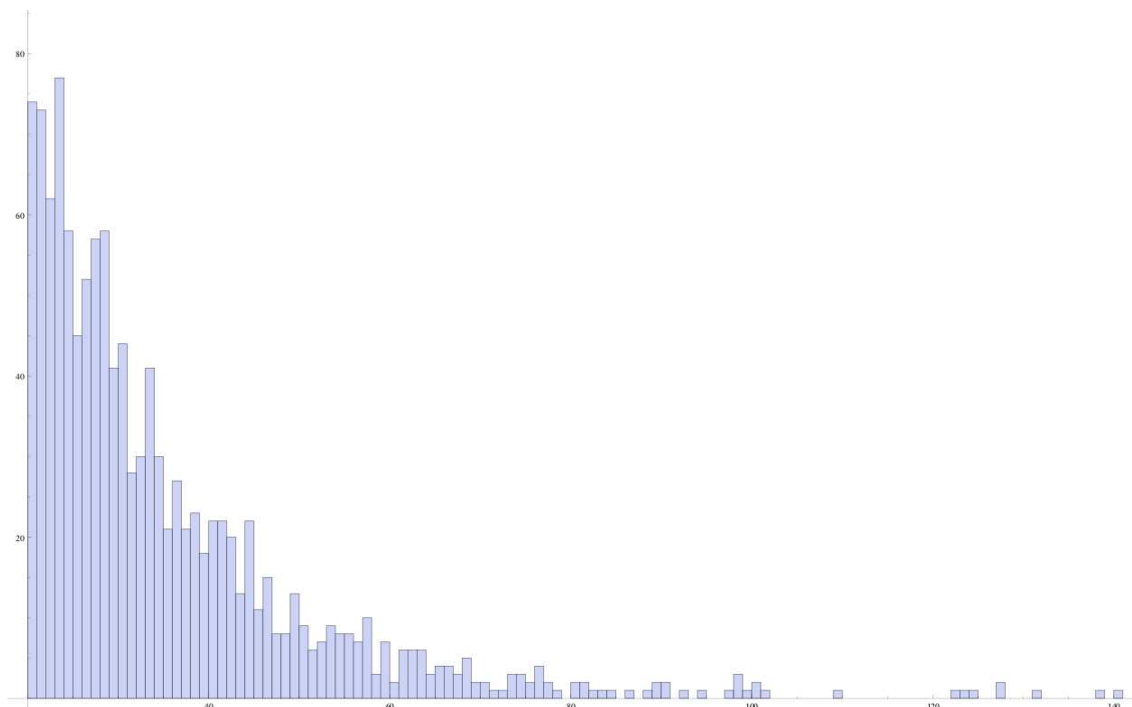

Figure 3.2: Distribution of VoDs regarding the number of downloads, when occasional users are removed (Orange > 20). The 3 most popular VoDs were downloaded more than 77 times in 1 month.

bring noise in the data and lead to irrelevance. Removing those users does not remove any VoD in the logs, and we keep the relative popularity between the films. Our aim is to build a similarity matrix of items, the more relevant, the better.

3.5.2 Apache Mahout boolean recommender

The Apache Mahout machine learning library's goal is to build scalable machine learning libraries. The core algorithms for clustering, classification and batch based collaborative filtering are implemented on top of Apache Hadoop using the MapReduce paradigm. The platform[9] supports recommendation mining tools. Written in Java, it provides a solid base although it is still under development. Recommender Systems comes from the part of Taste project, now included in Mahout. It can be deployed both on simple and distributed Hadoop-based architectures.

The Vipeer project is intended to cover the entire traffic of VoD distribution from an ISP. Thus the use of MapReduce is required if we need results fast. However, given our available sample dataset —the number of user-item associations do not approach 100 million— it is not necessary to study the distributed algorithm for this first experimentation. We use here the *ItemBasedRecommender* classes[10] from the

[9] http://mahout.apache.org/

[10] `https://cwiki.apache.org/confluence/display/MAHOUT/Itembased+Collaborative+Filtering`

```
//Data Model creation, here a log file
DataModel model = new FileDataModel(new File(filePath));

//Item-Item matrix comutation using Tanimoto Coefficient
ItemSimilarity similarity =
 new TanimotoCoefficientSimilarity(model);

//Creation of the boolean preference item based recommender
ItemBasedRecommender recommender =
new GenericBooleanPrefItemBasedRecommender(model, similarity);

//Generation of k recommendations for a user
List<RecommendedItem> recommendedItems =
recommender.recommend(userID, k);
```

Figure 3.3: Mahout java code to create and activate a boolean recommender.

Mahout version 0.6-SNAPSHOT (downloaded mid 2011). As shown in an exemple of Java code, Figure 3.3, Mahout recommender is ready to use.

Our dataset does not include any ratings nor additional information about downloaded movies. Thus we only have a binary data: the user has downloaded the Orange VoD or not. Each line of the available file is a *download*, with the following fields: userID, itemID (the movie), timestamp (date of download).

For binary data, the choice is not well off in terms of algorithms [10]. It is recommended to calculate the similarities between items with the coefficient of Tanimoto (Jaccard coefficient, cf. equation 3.1):

$$s_{ij} = \frac{|U_i \cap U_j|}{|U_i \cup U_j|} \tag{3.1}$$

with U_i the set of users who have downloaded the VoD_i.

Note that it may be interesting, in addition, to consider another matrix of similarities calculated from the movies' metadata, which may improve the accuracy of recommendations. The metadata may be the type of the movie, the actors, the film maker, etc. In the present case, this is impossible since we have no information on the films, not even the title.

$$r_{ui} = \sum_{j \in I_u} \frac{s_{ij}}{|I_u|} \tag{3.2}$$

with U_i the set of users who have downloaded the VoD i.

The recommended items are sorted according to the predicted rating (equation 3.2). The "recommender" outputs the k best items for each user with knowledge of the similarities between items s_{ij} and the list of items I_u previously downloaded by the user u.

3.6 Evaluation

MAE and *RMSE* metrics are excluded in the boolean case (these two types of measures need a rating. MAE is the absolute mean deviation between the predicted rating and the real rating, see the equation 3.3). [14] provides an exhaustive list of metrics.

$$MAE = \frac{\sum_{i=1}^{N} |predicted_i - real_i|}{N} \tag{3.3}$$

$$precision = \frac{|predicted \cap real|}{|predicted|} = \frac{TP}{TP + FP} \tag{3.4}$$

$$recall = \frac{|predicted \cap real|}{|real|} = \frac{TP}{TP + FN} \tag{3.5}$$

with TP the number of true positive (correct result), FP the number of false positive (unexpected result) and FN the number of false negative (missing result).

We use here the *precision* (equation 3.4) and *recall* (equation 3.5) measures. The precision measures the part of true positive predicted items among all the predictions. The recall measures the ability of the system to predict all of the relevant items (true positive added to false negative items). Thise measures are really relevant in our context. With a bad precision, the risk to prefetch items which will not be downloaded is high. We may here select only the very confident predictions. A bad recall means that there exist content that are not (can not be?) predicted by the method. Only "mean" behavior is really captured, this is a well-know weakness of collaborative filtering methods.

Note that we are more in a context of prediction than a context of recommendation. [14] suggests that the recommendation of novelty and surprise is expected by the user. Such "non-obvious" recommendations are appreciated, even so these items would never have been in the dataset (the item would not have been discovered by the user without any recommendation). In this case, non optimal precision and recall may not be a problem. A contrario, in our prefetching context, with limited storage space, a bad precision is a critical point. We aim to predict only true positive items, or been more realistic, reduce the false positive ones. We aim also a good recall, so that the caches provides the highest part of future downloadings and keep the traffic within the ISP network.

The class "GenericRecommenderIRStatsEvaluator" from Mahout includes evaluation of boolean recommendation systems. Unfortunately, at the time we have tested it, it was not functional. We have developed our own classes with different strategies, described in the next section.

3.6.1 Realistic chronological scenario

The dataset is sorted by downloading date, it is then splitted into a training dataset (typically 80% of all the data) and a testing dataset (20%). The *predicted* items calculated for every user from the training dataset are compared to the *real* items downloaded.

This model is realistic because, at time t, the Vipeer P&R component recommends items based on the data then available. The recommendations are checked with the downloads actually made beyond that date t.

However, this algorithm cannot in our current situation provide convincing and reliable results: in fact, by selecting the test set, we must consider in the training set only the few users who belong to both sets. Not forgetting the fact that we eliminate also the users who have downloaded less than $2 * k$ VoD in the training set (practical rule to keep "valuable" users).

This algorithm is therefore limited for our current Orange dataset, where only some of the users appear in the test set. But we keep in mind that it will be relevant to evaluate this way the Vipeer prefetching component under realistic conditions.

3.6.2 Netflix scenario: fixed k recommendations, testing with the K latest downloads

Lots of research on recommendation systems simply do not take into account the chronology when splitting the dataset. It was the case during the Netflix challenge. Small dataset are evaluated more easily this way since we keep more users under consideration.

For each user, we extract the K latest (dated) VoDs from the downloading log. We recommend k items for each users. This algorithm leads to a precision of about 0.11, which is reasonable. 11% of the predictions are real future downloads. This is a good score for a recommendation system. We will discuss later on this chapter if the approach is suitable as a prefetching method.

To measure the impact of keeping occasional users in the dataset, we tested different values of the filtering threshold (note that this threshold must be greater than K). The F1 score (equation 3.6) measures the compromise between precision and recall. F1 score reaches its best value at 1 and worst score at 0.

$$F_1 = 2\frac{precision.recall}{precision + recall} \tag{3.6}$$

We see on these the graph Figure 3.4 that the F1 score varies only a little with the power of filtering. However, this teaches us that it is risky to keep occasional users and also dangerous to remove too many users (high filtering threshold). In our case, a value of 6 or 8 could be interesting —and not 20 as it was filtered in MovieLens. Again, this is the result of a single dataset, this kind of calibration must be performed with any new kind of dataset.

3.6.3 Adaptative scenario: variable k above a threshold, testing with the K latest downloads

A peculiarity of the first two scenarios was that the number of recommended films was constant: k for each user. The idea for the third method is to keep the principles of selection of training and test dataset of the second senario, while offering a dynamic number of recommendations.

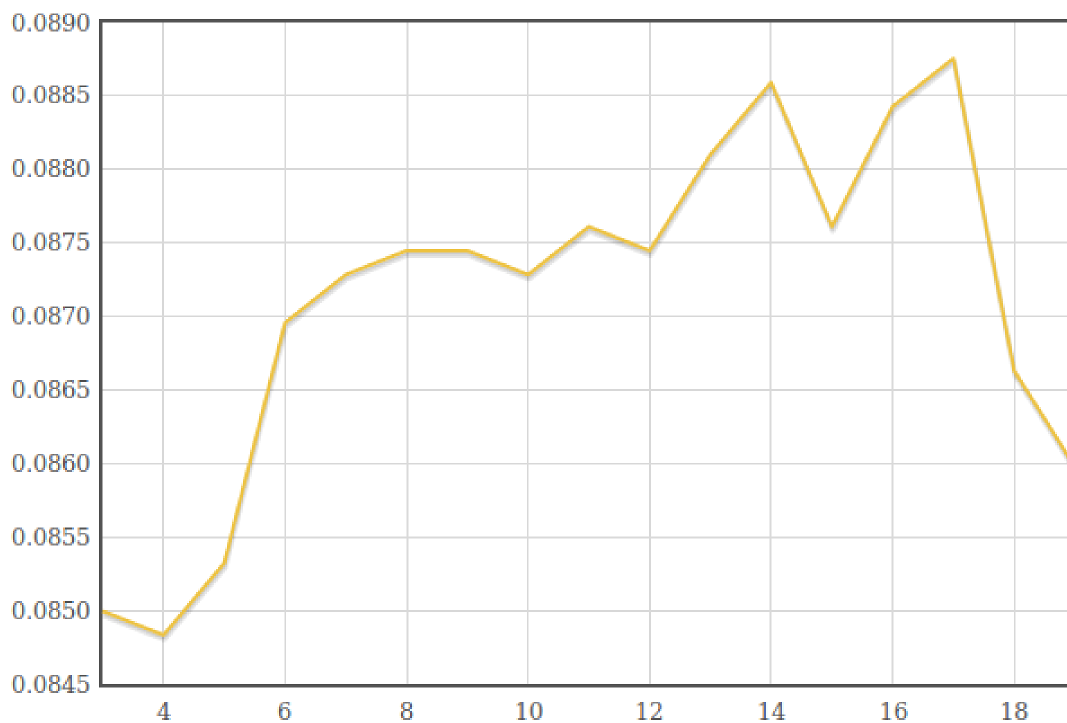

Figure 3.4: The removal of occasional users and its impact on the F1 score. The X axis is the minimum number of downloadings below which a user is occasional.

For each user, at least $MinRecNum$ items are recommended, maximum $MaxRecNum$, and the score of the items selected must be greater than $ValidRecThreshold$. For example, with a minimum of 3 to a maximum of 10 and a threshold of 1.0 (the best possible rating), we obtain a precision of 0.15, which becomes acceptable. A series of tests by varying these three parameters will of course be performed, especially with a more relevant dataset (Orange logs during one year).

This algorithm is interesting since it is adapted to the project: as the ultimate objective is to build a list of VoDs for the caches, we are not interested in the items whose recommendation score is too weak (below $ValidRecThreshold$). It is therefore logical to include this selection in the evaluation.

3.7 Is recommendation relevant for Vipeer?

After describing the data available, the recommendation system and evaluation methods of the recommendations, it is time to return to the heart of the Vipeer project and evaluate the real contribution of the recommendations, by comparing the two types of caching policies:

- The *Popularity cache*, which is simply constituted of the most popular films (type of LFU method),
- The *P&R cache*, from the recommendation engine.

Several methods exist to create a list of the best recommendations. The one we selected is based simply on an array of recommendations made with a dynamic number of recommendations per user, as established in the previous section. We then select the VoDs that appear most in this table. Another possibility would be to modulate this ranking with the scores given by the recommendation algorithm.

We developped the class $VipeerBooleanRecommenderEvaluator$, based on the third evaluation algorithm, described in the section 3.6.3 with the suggested parameters. This class fills the two types of cache, and then produces statistics by counting the number of real downloads that have been satisfied from the caches, as if they had been deployed into the peers. The Table 3.2 shows the results for two global sizes of cache: 200 or 1000 VoD spread among the peers.

Downloads	Caches capacity 200 items		Caches capacity 1000 items	
	nb true positive	% true positive	nb true positive	% true positive
Popularity	10258	34.2	20750	69.1
P&R	8639	29	16192	53.9
Both	6468	21	14537	48.45

Table 3.2: The number (and %) of downloads served by each types of caches. Some of the downloads may be served by both methods.

These results teach us several things. First, we show that with a cache of 1000 movies (1/5 of the whole!), the Popularity cache covers 70% of the downloads, which is very good because this method is easy to implement.

On the other hand, the P&R cache is globally less effective than the Popularity cache. It misses 20% of the downloads that the Popularity approach catches (12% for a cache size of 200). But what is interesting is that it can cover the downloading of movies that are actually downloaded regularly but which do not appear in the most popular films: 5% of the downloads that were not refering to the top-1000 most popular VoD were predicted by the P&R component (7% with the top-200). It is precisely this aspect we are looking for: P&R prefetching is actually complementary to LFU-like caching, especially with global low storage capacity. The recommendation system can potentially improve the performance of the final cache.

An additional step is needed: we have to determine the rank of popularity above which LFU is efficient, and the rank below which the items are mostly recommended by the P&R component.

3.8 Conclusion

In Vipeer WP4, we study an alternative approach to not cache exclusively the most downloaded movies. We aim to provide a large coverage of content, not only popular ones, and then minimize the requests that go out of the ISP domain. With huge storage capacities, such as Google CDNs, with a popularity-based caching policy (k-LRU for Google), the limit of 90% for the hit ratio is the current challenge. Here, our first tests show that with popularity-based caches, filled with the top-20% of the

most popular VoDs, 70% of the downloads may be managed by the ISP. The results are produced from a real VoD dataset provided by Orange, with 100000 downloads, collected in 2010, during one month, in one region, and involving 5000 VoDs.

This chapter also demonstrates that the P&R recommendations can provide an improvement to predict the download of non-blockbuster VoDs, especially when the caches have a low storage capacity.

The contribution of this chapter (the use of a simple recommender engine, based on the collaborative filtering approach) has to be integrated in the whole dCDN architecture (see Chapter 4). We have to test other recommendation algorithms. We also need to conduct further tests to compare the accuracy of caching with and without prefetching. We have to compare to other caching policies (e.g. LRU or LFU only, combination?). These experimentations will be conducted on an extended dataset collected during one year by Orange on a dozen of geographic regions.

Finally, one of the objective has not been yet addressed: can the geographic information in the logs improve the prefetching? Are mid-popular contents different from one region to another?

Beyond this recommendation method and its utility in Vipeer, another objective was to focus on the tool MapReduce. This programming model is designed to make it possible to run algorithms on a large scale, what is needed here since the project covers the entire traffic of VoD distribution from an ISP. The Apache Mahout machine learning library provides a solid base to develop new algorithms or test existing ones.

4 Optimal placement of video content

4.1 Introduction

In this chapter, we intend to use the MapReduce framework to arrange the placement of video content in a large-scale distributed Content Delivery Network (dCDN) system. We introduce a push server, which is responsible for dynamically calculating the optimal placement of requested videos onto a set of servers. The push server is in fact a MapReduce cluster. The decision made by the push server should take into account several factors, including the predicted user requirement, the intra-domain network environment and the servers' capability. The idea is that the MapReduce framework quickly compute an optimal placement, despite the NP-hardness of the content placement problem. Some previous works have demonstrated the potential of MapReduce in related contexts [19].

The factors are offered by the prediction and recommendation (P&R) system at the client side, and the statistical engines at the network and server side. The structure of the whole distribution system is shown in Figure 4.1.

When end users start downloading their videos, their requests are registered by the P&R module. The collected information is eventually analyzed in order to predict the requirement of content in the next period of time. The P&R associates each client with a set of movies that are most possible to be demanded. At the same time, the statistical engines at the network side and the server side estimate the cost for pushing videos into servers, the delay for sending videos from a server to a client, *etc.* Then the push server compute an optimal allocation of contents based on these predicted and statistical results.

The typical configuration of VoD service is described as follows. Assume the ISP need to provide the VoD service for 1 million clients. Several clients share one box with limited bandwidth of 28 Mbps (the bandwidth of a normal orange livebox). Since the playback bit rate of the standard-definition TV is 3.5 Mbps, one box can serve at most 8 clients. The ISP deploys $2 \cdot 10^5$ boxes so that there are enough boxes to support all clients. On the other hand, we assume that there are 500 films in the playlist, and each client is associated with 5 films. The average length of a film is 120 minutes, thus its average size is 3 GB. The storage capacity of one box is 200 GB, which means it can store around 60 movies.

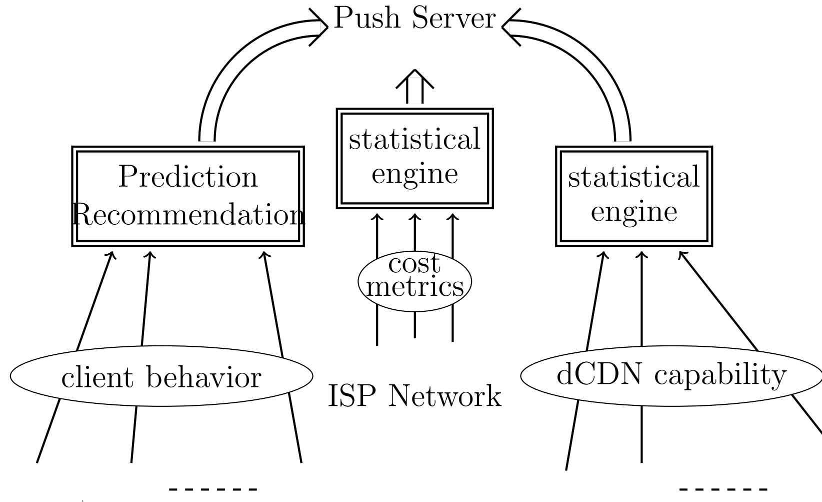

Figure 4.1: Distribution System

4.2 Problem Formulation

We formulate the decision of the optimal allocation as a k-product capacitated facility location problem (k-PCFLP). Given a set of clients $i \in I, (n = |I|)$, servers $j \in J, (m = |J|)$ and families of products $k \in K, (l = |K|)$, we define a binary variable $x_{jk} = 1$, if product k is assigned to server j, and $y_{ijk} = 1$ indicating that the demand from user i for product k is served by server j. The assignment cost and service cost are denoted as a_{jk} and e_{ij}. The service capability of each server is restricted by storage and bandwidth capacity s_j and b_j. Moreover, we introduce an integer constant $p_{ik} \in \{0, 1\}$ indicating whether a chunk k is recommended to client i or not.

$$\text{Minimize} \sum_J \sum_K a_{jk} x_{jk} + \sum_I \sum_J \sum_K e_{ij} p_{ik} y_{ijk}$$

$$\text{subject to} \sum_J y_{ijk} = p_{ik}, \qquad \forall i \in I, \forall k \in K \tag{4.1}$$

$$x_{jk} \geq p_{ik} \cdot y_{ijk}, \qquad \forall i \in I, \forall j \in J, \forall k \in K \tag{4.2}$$

$$\sum_K x_{jk} \leq s_j, \qquad \forall j \in J \tag{4.3}$$

$$\sum_I \sum_{k \in K} p_{ik} \cdot y_{ijk} \leq b_j, \qquad \forall j \in J \tag{4.4}$$

Since k-PCFLP is NP-complete and we are facing a huge amount of data, the approach to solve the problem should be easily parallelizable in MapReduce. Genetic algorithm, which is widely used to solve complex optimization algorihtms, fits well our requirement.

4.3 Introduction of Genetic Algorithm

4.3.1 Basic Elements

Generally, a Genetic Algorithm consists of the following elements [20].

4.3.1.0.1 Encoding method is used to interpret a feasible solution of the problem into an individual or a chromosome in the Genetic Algorithm. A certain number of individuals (commonly 100 to 200) constitute a generation. The evolution of generations yields the optimal or nearly optimal solution for the problem. Binary encoding is the most common encoding method in Genetic Algorithm. Other encoding methods are Many-Character encoding, Real-Valued Encoding and Tree encoding. For any optimization problem, choosing an appropriate encoding method is a central factor in the success of the corresponding genetic algorithm.

4.3.1.0.2 Fitness function estimates the value of an individual. Usually, a fitness function assigns a score to each individual in the current generation. Higher score means that the individual is nearer to the optimal solution.

4.3.1.0.3 Genetic Algorithm operaters lead to the evolution of generations. A typical Genetic Algorithm involves three operators: selection, crossover and mutation.

- The selection operator choose individuals in the current generation for reproducing individuals in the next generation. The fitter individual has more chance to be selected to reproduce offspring. Usually, the selection is done with replacement, that is, the same individual can be selected more than once to be a parent.
- The crossover operator randomly decides a locus and exchanges the subsequences before and after that locus between two individuals to create two offspring. For example, the two binary encoded individuals 1001001 and 1101110 can be crossed over after the second bit, and produce two offspring as 1001110 and 1101001. In the example, the crossover occurs only once. However, there are also multi-point crossover strategy in which the crossover takes place several times.
- The mutation operator randomly flips some of the bits in an individual. For instance, the offspring after crossover is 1001110. The mutation happens in its third bit, and finally the offspring in the new generation becomes 1011110. Mutation can occur at each bit position in an individual with very small probability (*e.g.,* 0.001).

4.3.2 Typical Procedure

When the encoding method for feasible solution of a problem is determined, a simple Genetic Algorithm executes as follows:

1. Start with a randomly generated population of n individuals.

2. Compute the fitness $f(x)$ of each individual x in the generation.
3. The following steps are the process of reproduce offspring in the next generation, they should be executed n times to ensure that the number of population does not change.

   a Select a pair of parent individual from the current generation

   b With probability P_c, cross over the parents at a randomly chosen point (chosen uniformly in all locus) to create two offspring. If no crossover happens, the produced offspring are exact copies of the parents.

   c Mutation takes place at each locus of the two created offspring with probability P_m. Then include the new individuals into the next generation. If n is odd, one offspring can be discarded randomly.
4. Replace the current generation with the new generation.
5. Go to step 2 until some convergence condition is reached.

Typically, a Genetic Algorithm is iterated for 50 to 500 generations. Usually, one or more highly fit individuals can be found in the last generation.

4.4 Modeling k-PCFLP by Genetic Algorithm

We now concentrate on our particular k-PCFLP. As it is observed in the linear program in section 4.2, in our special case there is no fixed cost for a facility, instead, we introduce an assignment cost for each product. Since establishing more facilities does not increase the total cost, we assume that all the servers are on duty. On the other hand, the service capability of each server is restricted by its storage and bandwidth capacity, so these conditions should be reflected in the encoding method or fitness function.

4.4.1 Encoding

We use the real value encoding to form an individual. The length of each individual is the sum of the storage capacity of all servers. The value of each gene is the sequence number of a product. For example, given an instance with $|K| = 6, |J| = 3$, and the storage capacity of every server is 3. Then, the individual $(1, 2, 3, 4, 5, 6, 0, 3, 5)$ means that products (k_1, k_2, k_3) are on server 1, (k_4, k_5, k_6) are offered by server 2, and (k_3, k_5) are stored by server 3.

4.4.2 Fitness Function

The fitness value is calculated by the objective function in the linear program. Since the value of x_{jk} is already determined by an individual, in the fitness function we have to decide the value of each y_{ijk}. In other words, for every product we assign suitable servers to the clients requiring the product. More concretely, the assignment should find the optimal solution based on the given individual.

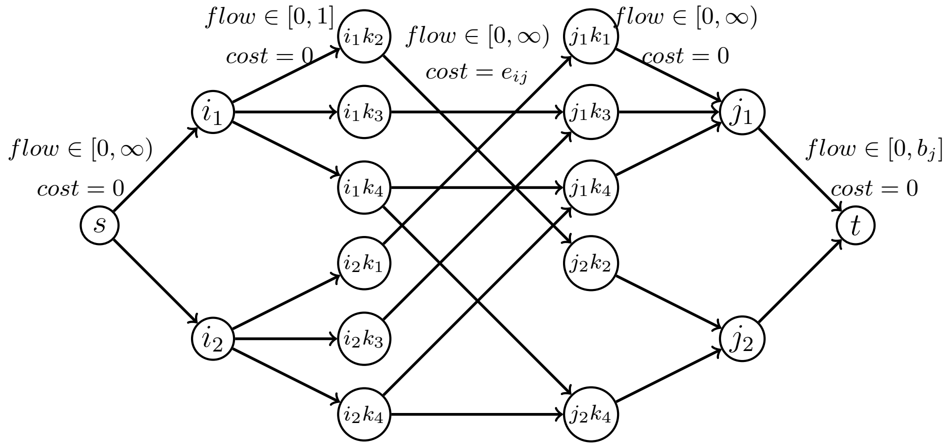

Figure 4.2: Example of MCMF

We model the assignment as an other optimization problem, Minimum Cost Maximum Flow (MCMF) Problem. There are five classes of vertices in the flow graph. A virtual source s and a virtual sink t are used for flow computation. Client node i and server node j represent the clients and servers in k-CPFLP. We use a type of nodes called cli-pro ik to indicate that client i is interested in product k. Similarly, a class of nodes serv-pro jk implies that server j stores product k. An example of the flow graph with 2 clients, 2 servers and 4 products is shown in figure 4.2.

In the example, client i_1 is asking products $\{k_2, k_3, k_4\}$, and i_2 is demanding $\{k_1, k_3, k_4\}$. The products k_1 and k_3 are on server j_1, k_4 is offered by j_2 and k_4 is stored by both j_1 and j_2. There is one link between the source node and each of the client node. Then a client node is connected with every cli-pro node where the client presents. To satisfy the constraint (4.2), we impose that the flow can be passed from a cli-pro node to a serv-pro node, only if the product indicated by the cli-pro is stored by the server denoted by the serv-pro node. Thereafter, every serv-pro node should transmit the flow to the corresponding server node. Finally, the flow goes from each server node to the sink. The flow constraints and the costs of links between two classes of nodes are also given in the graph. The link from source to client node has no flow constraint and costs nothing. The flow from client node to cli-pro is restricted by one, since the total flow should not overpass the value of $\sum_{i \in I} \sum_{k \in K} p_{ik}$. Together with the objective of maximum flow, the constraint (4.1) is ensured. The service cost defined in k-CPFLP is now associated with the cost of each link between cli-pro and serv-pro. It takes e_{ij} for a unit of flow to be transmitted from ik and jk. At last, the flow on the link from server node and sink is limited by the bandwidth of the server b_j, so that the constraint (4.4) of k-CPFLP is guaranteed in the solution of MCMF.

The optimization of MCMF can be divided into two parts. The first part is to find the max flow in the graph. Using the well known $Fold-Fulkerson$ algorithm, the complexity of this part is bounded by $\mathcal{O}(nml^2 \cdot \sum_{i \in I} \sum_{k \in K} p_{ik})$. If the resulting maximum flow is less than $\sum_{i \in I} \sum_{k \in K} p_{ik}$, the fitness value of the given individual is set to ∞, since the bandwidth given by the individual is insufficient for the

requirement on one or more products. Otherwise, the maximum flow is regarded as a parameter of the second part, a minimum cost flow problem. This problem can be solved by the minimum mean cycle canceling method proposed in [9], which runs in $\mathcal{O}((m+n)nml^2 \cdot \log((m+n)l) \cdot \min\{\log((m+n)l \cdot C), nml^2 \cdot \log((m+n)l)\})$, where C is the maximum absolute value of a link cost. Since the constraint (4.3) is ensured by the encoding, we obtain the optimal solution for the MCMF as well as the k-CPFLP based on the given individual.

4.4.3 Initialization and selection

The size of population and the initial generation give a great impact on the performance of the Genetic Algorithm. We should find a trade-off between the efficiency and the completeness of the search region. In our case, if MapReduce significantly ameliorates the performance of Genetic Algorithm, we may potentially have a large number of population. The completeness is guaranteed when all products as well as the empty storage space appear in the research process. We deploy all products initially, and the gene representing empty storage space is given by the mutation operation described later. Moreover, the initial generation should be carefully designed because the products in our problem are not uniformly requested, therefore popular products should have more replicas in the servers. The number of replicas of a product depends on the total storage capacity of servers and the popularity of the product. Specifically, the number equals $\left\lceil \sum_{j \in J} s_j \cdot g(k) \right\rceil$, where $g()$ is the probability density function of the popularity distribution. Then, we randomly choose the location for each replica. Unpopular products have higher priority to be located, so that the chance that they disappear in an individual becomes lower. Moreover, two replicas of one product should never exist on the same server.

To ease the parallelization of our Genetic Algorithm, we use the Tournament Selection. Every parent is chosen from two individuals that are randomly picked up from the current generation. A random number r_1 and a threshold thr_1 are used to decide which individual becomes the parent. If $r_1 < thr_1$, we select the individual with lower fitness value (the fitter individual), otherwise, the one with higher fitness value is chosen as the parent. Then the two are returned to the population, they are allow to be selected again.

4.4.4 Crossover and Mutation

Remind that in our Genetic Algorithm the form of an individual should strictly obey constraint (4.3), the common uniform crossover for binary encoding is not suitable, instead, we use a merge operation. Besides constraint (4.3), the result of the mergence should also ensure that no two replicas of the same product are located on one server. Therefore, the operation is implemented on each section of genes representing one server. For a server j, the first step of the mergence may produce a multiset with u genes, where $s_j \leq u \leq 2s_j$. If $s_j < u$, we randomly choose s_j genes from the multiset, so that the resulting section still follows (4.3). But in this case, the gene representing one product may appear twice in the same section. So preselection is necessary to treat the gene that exists in both parents

sections	j_1	j_2	j_3
parent	1,2,3	4,5,6	0,3,5
parent	3,4,5	6,0,1	2,3,5
r_2	0.7	0.5	0.4
preselection	non	6	3,5
mergence	1,2,3,4,5	0,1,4,5	0,2
offspring	1,2,4	6,0,5	3,5,2

Table 4.1: Crossover by mergence

before the mergence. Particularly, we use another random number r_2 and threshold thr_2. If $r_2 < thr_2$, the repeated genes in the multiset are picked out, and the corresponding product is located on the server. Otherwise, one of the repeated elements is eliminated so that the multiset becomes a normal set, where we randomly select genes for the offspring. Take the same example described in section 4.4.1, we show our special crossover operation in table 4.1, in the instance, we set $thr_2 = 0.6$.

The mutation operation is reserved especially for the gene "0", since in the initializing stage we fulfill all the storage space with products. After the crossover operation, each gene has the probability $p_{mut} = 0.001$ to mutate to 0.

4.4.5 Replacement and Termination

Considering the parallelization, we implement the elitist strategy in the generation revolution. Assume the number of current population is N_p, then each generation must produce N_p offspring as well. In the N_p offspring, the ones with less fitness value than the minimum fitness value in the current generation are qualified to enter the next generation. If the number of qualified offspring is N_q, then the N_q individuals with highest fitness value (less fit individuals) are replaced by new individuals to form the new generation. On the other hand, the algorithm terminates if N_p offspring are generated, however, no qualified individual can be found.

4.5 Perspective of Using MapReduce

MapReduce will be used to ameliorate the performance of the most time-consuming parts in the Genetic Algorithm. Namely, the evaluation of the initial population and offspring, as well as the production procedure of offspring. We should develop two different MapReduce algorithms to tackle the initial population and offspring.

5 Conclusion

To conclude this deliverable, we prove the need of the MapReduce framework inside the ViPeeR project and more especially for the WP4. On one hand, we need to improve and continue to work on the MapReduce framework and so the Apache Hadoop project. On the other hand, we need to apply this framework in the Genetic Algorithm to ameliorate the performance of the most time-consuming parts. In parallel, we will continue on the prefetching study by applying other algorithms in order to make some comparisons and fine tune the method.

Bibliography

[1] Hadoop: Open source implementation of mapreduce. `http://lucene.apache.org/hadoop/`.

[2] C. Bothorel. Analyse de réseaux sociaux et recommandation de contenus non populaires. Revue des nouvelles technologies de l'information (RNTI), A.5, 2011. ISBN 9782705682217.

[3] F. Cacheda, V. Carneiro, D. Fernández, and V. Formoso. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. ACM Trans. Web, 5:2:1–2:33, February 2011.

[4] L. Candillier, K. Jack, F. Fessant, and F. Meyer. State-of-the-art recommender systems. Collaborative and Social Information Retrieval and Access: Techniques for Improved User Modeling, pages 1–22, 2009.

[5] H. Chang, M. Kodialam, R. R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee. Scheduling in MapReduce-like Systems for Fast Completion Time. In Proc. of IEEE INFOCOM, 2011.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In Proc. of OSDI, 2004.

[7] E. Friedman and S. Henderson. Fairness and efficiency in web server protocols. In Proc. of Sigmetrics, 2003.

[8] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resources types. In Proc. of NSDI, 2011.

[9] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. J. ACM, 36:873–886, October 1989.

[10] M. Hahsler. Developing and testing top-n recommendation algorithms for 0-1 data using recommenderlab. Technical Report, 2011.

[11] M. Harchol-Balter. Queueing disciplines. In Wiley Encyclopedia Of Operations Research and Management Science. John Wiley & Sons, 2009.

[12] Y. He and Y. Liu. VOVO: VCR-Oriented Video-on-Demand in Large-Scale Peer-to-Peer Networks. Parallel and Distributed Systems, IEEE Transactions on, 20(4):528–539, 2009.

[13] Y. He, G. Shen, Y. Xiong, and L. Guang. Optimal prefetching scheme in p2p-vod applications with guided seeks. IEEE Transactions on Multimedia, 11(1), 2009.

[14] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. Riedl. Evaluating collaborative filtering recommender systems. ACM Trans. Inf. Syst., 22(1):5–53, 2004.

[15] Y. Huang, T. Z. Fu, D.-M. Chiu, J. C. Lui, and C. Huang. Challenges, design and analysis of a large-scale p2p-vod system. SIGCOMM Comput. Commun. Rev., 38:375–388, August 2008.

[16] Z. Huang, H. Chen, and D. Zeng. Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering. ACM Transactions on Information Systems (TOIS), 22(1):116–142, 2004.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In Proc. of ACM EuroSys, 2007.

[18] K. Kc and K. Anyanwu. Scheduling Hadoop jobs to meet deadlines. In Proc. of CloudCom, 2010.

[19] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

[20] M. Michell. An introduction to genetic algorithms. The MIT Press, 1996.

[21] Y.-J. Park and A. Tuzhilin. The long tail of recommender systems and how to leverage it. In RecSys '08: Proceedings of the 2008 ACM conference on Recommender systems, pages 11–18, New York, NY, USA, 2008. ACM.

[22] P. Resnick and H. R. Varian. Recommender systems - introduction to the special section. Commun. ACM, 40(3):56–58, 1997.

[23] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In Proc. of Workshop on Job Scheduling Strategies for Parallel Processing, 2010.

[24] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th international conference on World Wide Web, pages 285–295. ACM, 2001.

[25] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: A slot allocation scheduling optimizer for MapReduce workloads. In Proc. of International Middleware Conference, 2010.

[26] J. Wu and B. Li. Keep Cache Replacement Simple in Peer-Assisted VoD Systems. In IEEE INFOCOM 2009 - The 28th Conference on Computer Communications, pages 2591–2595. IEEE, Apr. 2009.

[27] T. Xu, B. Ye, Q. Wang, W. Li, S. Lu, and X. Fu. Apex: A personalization framework to improve quality of experience for dvd-like functions in p2p vod applications. In Quality of Service (IWQoS), 2010 18th International Workshop on, pages 1 –9, june 2010.

[28] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In Proc. of ACM EuroSys, 2010.