Data Analysis at Scale: Systems, Algorithms and Information

Vincent Leroy

THÈSE

Pour obtenir une

HABILITATION À DIRIGER DES RECHERCHES

Spécialité : **Informatique et Mathématiques Appliquées**

Présentée par

Vincent Leroy

Data Analysis at Scale: Systems, Algorithms and Information

Thèse soutenue publiquement le 26/09/2017,
devant le jury composé de :

M. Éric GAUSSIER, Président du jury
Professeur, Université Grenoble Alpes
Mme Ioana MANOLESCU, Rapporteur du jury
Directrice de recherche, INRIA Saclay-Île-de-France
M. Volker MARKL, Rapporteur du jury
Professeur, Université Technique de Berlin
M. Tamer ÖZSU, Rapporteur du jury
Professeur, Université de Waterloo
Mme Sihem AMER-YAHIA, Membre du jury
Directrice de recherche, CNRS délégation Alpes
M. Mokrane BOUZEGHOUB, Membre du jury
Professeur, Université de Versailles - UVSQ
Mme Esther PACITTI, Membre du jury
Professeur, Université Montpellier 2

Data Analysis at Scale: Systems, Algorithms and Information

Vincent Leroy

Contents

Chapter 1

Introduction

Over the past 10 years, I have been involved in research related to different aspects of data analysis. As a Ph.D candidate, my work started in an INRIA research group called **A**s **S**calable **A**s **P**ossible, whose primary focus what the design of large-scale peer-to-peer systems. During the second year of my Ph.D, I had the privilege to do an internship at Yahoo! Research Barcelona, on data mining applied to probabilistic graphs. Since then, this combination of large-scale systems and data mining has been at the center of my research. After my Ph.D, I returned to Yahoo! Research Barcelona as post-doctorate researcher in the distributed systems group. I developed large-scale systems for data processing, often borrowing techniques from information retrieval and data mining to optimize the behavior of the systems by leveraging properties of the data. Since 2012, I am an associate professor at the university of Grenoble, and a permanent member of the **S**calable **I**nformation **D**iscovery and **E**xploitation group at LIG (Laboratoire d'Informatique de Grenoble). My research is *data-driven*, and generally starts by the study of a new dataset, from which research challenges emerge.

I have kept a broad interest in topics related to data processing. They can be summarized as follows:

- Systems design
- Algorithms development
- Information discovery

This versatility allows me to contribute to many steps of a data analysis problem, starting with scaling the processing, developing more efficient algorithms, up to the problem of better targeting relevant information. This thesis presents some of my contributions to each of these three research topics (Chapters 2, 3 and 4). Then, I give an outlook on my future research in Chapter 5.

Systems design

Systems are the muscles of data analysis. The goal of systems research is to build hardware and software architectures that provide users with an increasing amount of processing power while hiding the underlying complexity behind simple paradigms. Although this definition applies to high-performance computing community in general, data analysis is currently one of

the largest consumer of computing resources, which makes it a primary target. For the past years, systems research has focused on scale-out (horizontally) architectures that provide parallel processing through the use of multi-core CPUs and distributed systems spanning hundreds of servers. Map-Reduce [35], published by Google, is a seminal work that has had a huge impact in the systems community. It has also unlocked new possibilities for the users of these systems by allowing any organization to build a data processing cluster from commodity hardware. More recent work, such as Heron [62] and Flink [27] illustrate the trend of switching from a batch processing model to analyzing continuous streams of data.

Systems are considered as the underlying layer of data processing architectures. Hence, they aim at being general purpose and support many data processing applications. Even more specialized systems, such as Arabesque [89] that targets graph analysis, offer a flexible programming model supporting different algorithms. This layered approach means that systems are generally oblivious to the properties of the workload they execute. Their main concern is load balancing, i.e. ensuring that all processing resources are used equally. In Chapter 2 of this thesis, I present my work on optimizing distributed systems for data processing. I advocate the idea that the distribution of data in a system should account for its properties: pieces of data which are correlated in the application's workload should be placed in close proximity. I apply this design principle to distributed search engines, deployed over several data-centers, and key/value-based applications, deployed in a single location.

Algorithms development

Algorithms are the articulations of data analysis. Algorithms are at the core of Database research to produce more efficient ways of querying data. Although the relational model [32] remains a cornerstone of databases, a large fraction of the recent work is related to XML [18] and RDF [82], with contributions to query optimization and rewriting [24], or the creation of indexes for accelerating query execution [94]. The Data Mining community also creates many algorithms that compete to extract results efficiently as possible.

Algorithms research often builds on systems work, and accounts for the processing model exposed by the system to optimize algorithms. Hence, the publication of Map-Reduce lead to a large number of follow-up databases publications related to porting existing queries (e.g. theta-joins [100]) to this new programming model. In Chapter 3 of this thesis, I tackled two specific types of algorithms: temporal joins, and pattern mining. The challenge in developing these algorithms for a Map-Reduce execution model is that they rely on a dynamic exploration of a result-space, which is particularly hard to optimize with a distributed system.

Information discovery

Information discovery is the brain of data analysis. This research area deals with identifying information relevant to the data analyst. In the standard Information Retrieval model, a user submits a query to a search engine, and the goal is to assign a relevance score to documents in a collection. Unsupervised Data Mining aims at discovering information in a dataset without the need for a specific query. With the democratization of massive datasets, identifying and ranking the most important results is of crucial importance. Indeed, data processing systems can scale to hundreds of machine and generate millions of results, but the ability of analysts to read those results and identify by themselves the interesting ones remains limited. Thus,

building relevance models for varied data types and modeling user intent remain popular topics [69]. In addition, learning to rank has been a very active field of research [95].

Chapter 4 of this thesis deals with the problem of identifying relevant results. The first part considers association rules mining, a popular algorithm of unsupervised data mining. A major challenge is to identify, among all interestingness measures defined in the literature, which should be used to rank the rules produced in a specific domain (retail in this case). The second part of this chapter considers complex information needs, and goes beyond returning individual results by building composite items.

Chapter 2

Data placement in distributed systems

The resources needed to execute an application in a timely manner increase with its workload. From a single-threaded execution on a single server to a parallel execution over thousands of servers, each step taken to scale an application requires the developer to fragment the processing tasks into smaller independent pieces, and distribute memory over the resources available. In this chapter, we focus on the problem of distributing memory.

The question of where to allocate a data structure arises as soon as an application becomes parallel. Computer architectures are not uniform, they are hierarchical. Most multi-core systems implement a Non-Uniform Memory Access architecture, which means that computing resources (CPU cores) pay a different cost for accessing different parts of the memory. The High Performance Computing community has been investigating the issue of optimizing memory management in NUMA architectures for over 10 years, with an increases interest since the advent of multi-core CPUs [46]. Distributed systems are, to some extent, another step in the hierarchical memory structure. Each server has a privileged access to its own memory, and can access the memory of other servers with a penalty varying with the performance of the network. A server located in the same rack can be accessed efficiently, while exchanging data with a server located in a different data-center halfway across the world is less efficient and more costly.

Currently there are two main strategies for placing data in distributed systems. When communication costs are high (e.g. data centers in different regions) and data is relatively static (read dominated workload), data is replicated to avoid remote accesses. This is notably the case of Web search engines: multiple data centers are necessary to scale with the number of queries, but the full Web index is replicated in each of them so that they operate independently. When data is modified frequently (significant amount of writes), data is split among servers using a pseudo-random assignments. This is the case if most key-value based applications, in which a hash function is used to assign each key to a server. Replicating data consumes memory, and thus has a monetary cost (additional servers and energy), while a pseudo-random partitioning does not account for the hierarchical nature of data center architectures. I advocate for a different solution: the distribution of data across servers should account for its properties, i.e. pieces of data which are correlated in the application's workload should be placed in close proximity.

Section 2.1 considers the case of a search engine distributed across multiple datacenters.

We show that, by dividing the index according to the queries of users in each region, we can deploy a distributed search engine that executes most queries locally while replication remains minimal. In Section 2.2, we address the problem of key-value based applications and show that we can drastically reduce network usage, and thus increase performance, by leveraging correlations in key access patterns.

2.1 Multiple data centers: Distributed Search Engine

2.1.1 Context

Search engines are of critical importance for navigating the Web. Their goal is to help users identify relevant content by answering complex queries. Even in the case of a simple navigation operation, people would rather type "facebook" in a search engine than `www.facebook.com` in the address bar of their browser. Hence, search engines continuously receive a huge amount of traffic that must be answered with low latency to preserve user experience.

A search engine is typically deployed in a single location, or data center, and comprises a large number of servers hosting an full index of the Web. When a query is received, it is processed locally using the Web index and results are returned to the client. This *local* search engine architecture scales with the number of queries *horizontally*, i.e. by replicating the search engine over multiple standalone sites (data centers). These sites can be located in different regions of the world, which has the advantage of reducing network latency with respect to users. The drawback of local search engines is that each site is a replica of the original search engine, and must be dimensioned to host a full index of the Web to answer any query. When the size of the index increases, each data center must be upgraded: this is referred to as *vertical* scalability.

An appealing solution to scale Web search is to avoid the Web index replication by having search sites collaborate to answer queries. *Distributed* search engines rely on multiple sites deployed in distant regions across the world, and each site specializes in serving queries issued by the user of its region. Thus, each site hosts a fraction of the Web index specifically selected to serve queries originating from its region. When a site receives a query it is unable to answer accurately (e.g. a query in Japanese at a European site), it forwards the query to the appropriate site to preserves the quality of the results. All sites of the distributed search engine collectively provide a full index of the Web, so this architecture enables *horizontal* scalability with the index size: adding a new site to a geographic area allows other data centers to eliminate part of their Web index as they no longer receive queries from this area. A smaller index means faster processing [21], so distributed search engines have the potential to be answer queries faster than their centralized counterpart. However, if a query cannot be answered locally, forwarding it increases both latency and the workload of all search sites. Thus, a major challenge in deploying a distributed search engine is to carefully select which documents are indexed by each search site, and optimize the trade-off between memory usage and locality.

The work presented in this section was done during my post-doc at Yahoo! Research Barcelona, and is detailed in the following publications:

- **Assigning documents to master sites in distributed search**
  Roi Blanco, B. Barla Cambazoglu, Flavio P. Junqueira, Ivan Kelly and Vincent Leroy
  In Proceedings of the ACM Conference on Information and Knowledge Management (CIKM), 2011, pages 67–76.

- **Reactive index replication for distributed search engines**
  Flavio P. Junqueira, Vincent Leroy and Matthieu Morel
  In Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR), 2012, pages 831–840.

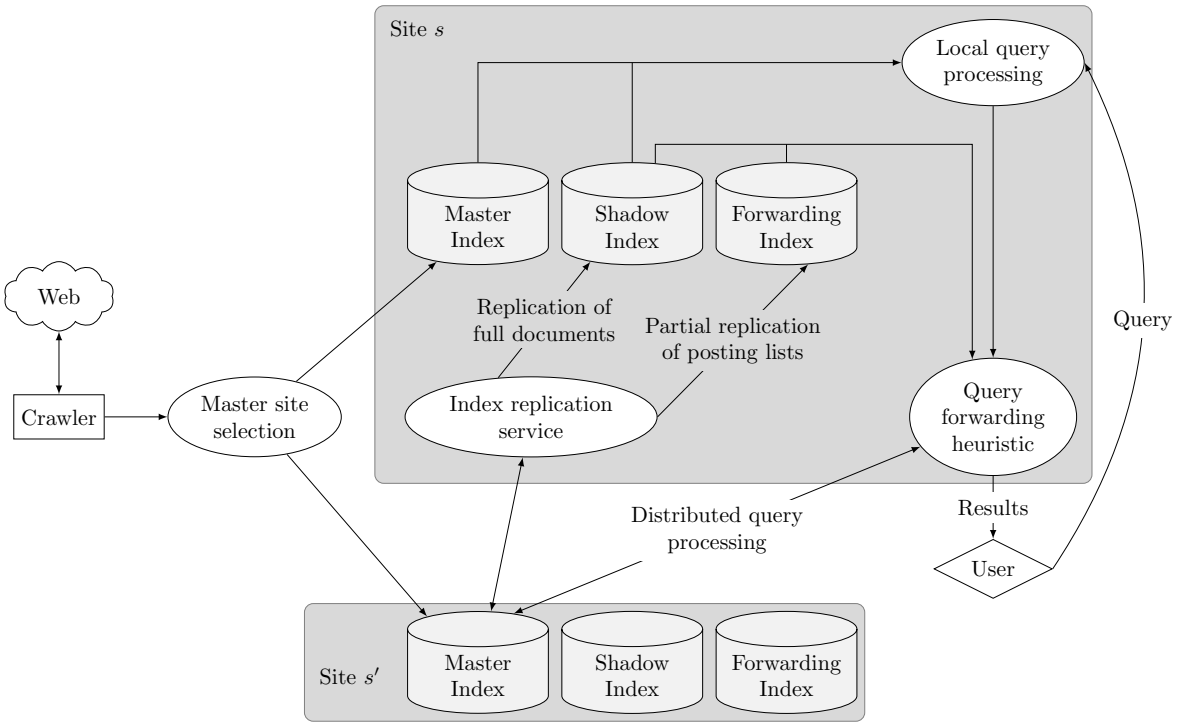


Figure 2.1: Overview of the distributed search engine architecture

In this thesis, I focus on the data placement policies and omit systems details. Section 2.1.2 gives an overview of the search engine architecture. In Section 2.1.3, we consider the problem of placing new documents, that have just been discovered by the Web crawler. Then, in Section 2.1.4, we describe the index replication protocol, that learns from user queries to dynamically adjust the content of the index and minimize query forwarding. Finally, Section 2.1.5 concludes.

2.1.2 Architecture

A search engine consists of three main components:

- The *crawler* fetches documents from the Web and discovers new content by following hypertext links.
- The *indexer* processes the collection of documents fetched by the crawler to generate an inverted index. For each term t present in the collection, the inverted index contains a posting list, i.e. a list of the documents that contain t.
- The *query processor* receives user queries and processes them against the index to identify relevant results. Web search generally relies on conjunctive query processing [88], which means that a document has to contain all the terms in the query to be in the result set. The search engines then identifies the top-k (typically $k = 10$) documents having the highest relevance score and returns them to the user.

We consider a distributed search engine comprising a set of search sites $\mathcal{S}$ deployed across geographically different regions. Collectively, the sites form a search engine indexing a collection of documents $\mathcal{D}$. We assume that each search site has a fixed index capacity, either due to limited resources, or arbitrarily chosen to reduce query processing time. We express this limit as a number of postings, i.e. the sum of the length of all the posting lists in the index. Users are very sensitive to the quality of a search engine. Thus, we decided to not make any trade-off on the relevance of search results: the distributed search engine must return the same results as its centralized counterpart for any query submitted to any search site.

Figure 2.1 gives an overview of the architecture of the distributed search engine. To remain concise, this figure only represents two sites $s, s' \in \mathcal{S}$ and details s. As explained in Section 2.1.1, the main goal of this work is to specialize the index of each search site to answer the queries of the users in its region and avoid query forwarding. Hence, while geographically distributed crawling is an interesting research problem [45], we focus on indexing and query processing. For the sake of simplicity, we consider the case of a single *crawler*. The distribution of the search engine architecture starts after the crawler, with the *master site selection*. We now detail the *indexer* and *query processor* of the distributed search engine.

Assignment of documents to search sites In a distributed search engine, each search site has its own index and processes the queries of the users in its region. It is therefore important to carefully select documents to index in each site and tailor each search site for its users. For a short response time, a search site must be able to process most of the queries it receives using its local index alone. Indexing documents that are popular in a region enables locality. However, it is also important to limit the number of documents indexed at each site for scalability reasons, and to reduce query processing latency [21]. Upon its discovery by the crawler, each document $D \in \mathcal{D}$ is assigned to a single master site $s \in \mathcal{S}$ and is fully indexed in the *Master Index* of this search site MI_s. The master selection generates a partitioning of $\mathcal{D}$, and results in a minimal index, with each document indexed in a single location. The presence of the master index guarantees the search results quality: it is always possible to execute a query on all search site and merge the results to obtain the same results as a centralized search engines. Evidently, broadcasting search queries to all sites is not desirable, as it incurs a large processing cost and increases response time. Hence, our goal is to select for each document the site where it is most likely to be requested as a master. The challenge is that master assignment is proactive: a document is assigned upon its discovery by the crawler, before it appears in search results. We detail different methods for the master selection process in Section 2.1.3 and evaluate their performance.

Full documents replication The popularity of Web pages typically follows a power law: while most Web pages are unpopular, a few of them are requested very frequently. A Web page might present a high locality, being popular in a single region (e.g. Le Monde in France), or be popular across many regions (e.g. IMDb). Distributed search engines work better in a context where documents have a strong locality. Indeed, this means that each document only needs to be indexed by the search site located in the region where it is popular. Fortunately, a large fraction of Web pages exhibit a high divergence in their popularity across regions [58]. There are still documents which are popular across region boundaries and are requested by users of different search sites. When a document D is frequently requested at a search site s, and $s' \neq s$ is the master of D, s can replicate the document by indexing it in its *Shadow*

Index SI_s. Contrary to the master site selection process, replicating documents in the shadow index can be reactive as it does not affect the quality of search results, only likelihood that all results can be found locally. Section 2.1.4 describes our document replication solution.

Distributed query processing and partial index replication In a distributed search engine, a query is submitted by a user to the search site of her region. This query is first processed locally, on the documents indexed in the master and shadow indexes (i.e. on the content of MI_s and SI_s), to obtain local results. A *query forwarding heuristic* is then in charge of deciding whether local results are sufficient and can be returned immediately, or if another search site could provide better results, in which case the query is forwarded. As explained previously, our goal is to always return exact results. Consequently, the query forwarding heuristic considered in this work is conservative: it can generate false positives (i.e. forwarding when local results were already exact), but no false negatives (i.e. not forwarding when a document not indexed locally should be added to the results). Hence, the heuristic at site s must answer the following question: is there a possibility that a document can score higher than the k best results computed from MI_s and SI_s? To answer this question, the heuristic leverages properties of the search engine ranking function to compute an upper bound on the score of documents which are not indexed locally. Equation 2.1 presents the function $s(D|Q)$ that computes the score of a document D given a query Q.

$$\begin{aligned} s(D|Q) &= w_f f(D) + \frac{w_g}{|Q|} \sum_{t \in Q} g(D|t) \\ r(D|t) &= w_f f(D) + w_g g(D|t) \\ s(D|Q) &= \frac{1}{|Q|} \sum_{t \in Q} r(D|t) \end{aligned} \tag{2.1}$$

This simple function was introduced in [13] and expresses the fact that a document D has a query independent quality score $f(D)$ (e.g. PageRank [22]) and a relevance score $g(D|t)$ for each term of the query $t \in Q$. The partial score $r(D|t)$, combining both quality and relevance, is maintained in the posting list of t. Each search site maintains a *Forwarding Index* that contains portions of the posting lists of other search sites. This partial data is then used by the forwarding heuristic to bound the score of results of other search sites. We show in Section 2.1.4 how fragments of posting lists are selected for replication.

2.1.3 Proactive placement: Master selection

Problem definition We consider the problem of selecting exactly one site for each document fetched by the crawler. This site is referred to as the *master* of the document, and is responsible for keeping it in its *Master Index*. Master site selection relies on a scoring function *mScore* such that:

$$\begin{aligned} master &: \mathcal{D} \rightarrow \mathcal{S} \\ mScore &: \mathcal{D} \times \mathcal{S} \rightarrow \mathbb{R} \\ master(D) &= \operatorname*{argmax}_{s \in \mathcal{S}} mScore(D, s) \end{aligned} \tag{2.2}$$

Hence, our objective is to design a *mScore* function that produces higher scores when it predicts that the document D matches the interests of the users of site s.

Document assignment by language The first feature we consider for *mScore* is the language of the document. Regional documents are more likely to be requested from the countries whose language is the same [20]. Using past search engine activity, we define *mScore* as the the conditional probability $p(s|l)$: given that a document in language l is returned to a user, what is the probability that the user is querying site s? This solution is rather simple, as the only requirement is to provide the crawler with a language classifier and the distribution $p(s|l)$.

Document assignment by likelihood While predicting the language leverages the content of the document, it remains a coarse grain information. Many terms, including names and locations, can be used to make a more accurate estimation. Hence, we propose to compute *mScore* using a probability estimation based on the distribution of terms. The optimal placement for a document D at a master site s depends on the queries that will be issued to s in the future $\mathcal{Q}_s^f$ in which D appears among the top results. We make use of two sources of information: C_s is a random vector that represents the content of site s, and $\mathcal{Q}_s$ is a random vector representing the most recent query stream processed by s. Both these information can be extracted from the cache of the search engine or its query logs. We want to select the most probable site maximizing the following log-likelihood:

$$\begin{aligned} \log p(s|D) &= \int \log p(s|D, \mathcal{Q}_s^f) p(\mathcal{Q}_s^f|D) d\mathcal{Q}_s^f \\ &= \int \log p(C_s, \mathcal{Q}_s|D, \mathcal{Q}_s^f) + \log p(\mathcal{Q}_s^f|D) d\mathcal{Q}_s^f \end{aligned} \tag{2.3}$$

In order to estimate $\mathcal{Q}_s^f$, we make use of $\mathcal{Q}_s$, i.e. we estimate that a recent query stream is representative of future queries.

$$\begin{aligned} \log p(s|D) &\approx \int \log p(C_s, \mathcal{Q}_s|D, \mathcal{Q}_s) + \log p(\mathcal{Q}_s|D) d\mathcal{Q}_s \\ &\approx \log p(C_s|D) + \frac{1}{|\mathcal{Q}_s|} \sum_{Q \in \mathcal{Q}_s} \log p(q|D) \end{aligned} \tag{2.4}$$

This formulation has two components: one that uses the content of a given site and its relatedness to the document $p(C_s, D)$, and another one that uses the available query stream $\mathcal{Q}_s$ as a source of evidence. We relate to the former probability with the *term distribution* of the site, and the latter as the fraction of queries that are *invalidated* by a given document.

The *term distribution* probability can be estimated using a similarity function, in the same way standard language models score documents with respect to their likelihood to a query. The rationale is that if the contents of the documents returned by a site matches the queries it receives, we should assign a new document to the search site that has the closest term distribution. To estimate $p(C_s|D)$, we employ the language-model-based KL divergence [98], which measures how similar a term distribution (query) is with respect to a reference distribution (document). We consider two different ways to obtain C_s, the content of search site s. The first one uses the content of the documents returned as results to the users of the site, while the second approach directly uses the term distribution of user queries. It is important to notice that both of these approaches are directly driven by user activity, and not influenced by the master selection process. This avoids propagating errors based on previous erroneous master assignment decisions.

Probability $p(Q_s|D)$ represents the likelihood of the query stream to the document. This has been studied in the past in the context of *cache invalidation*. Search engines cache query results to avoid computing them multiple times when a query is repeated. Cache entries need to be invalidated when indexing new documents to avoid results staleness. This can be done by estimating which queries present in the cache would return different results given the presence of the new document [17]. Master site assignment takes place upon the discovery of the document, and thus cannot directly rely on the appearance of the document in search result. However, cache invalidation gives us knowledge on which past queries would have returned the new document in their top-k results if it had been in the index at that time. This is expressed as:

$$p(\mathcal{Q}_s|D) = \frac{1}{|\mathcal{Q}_s|} \sum_{Q \in \mathcal{Q}_s} I(Q, D) \tag{2.5}$$

where $I(Q, D)$ is 1 if D would be in the top-k results for Q (i.e. indexing D invalidates the cached results for Q) and 0 otherwise. Contrary to the probability based on *term distribution*, cache invalidation considers exact queries that were submitted to s, and the impact of D on their results. This information is more precise than considering all queries simultaneously as a bag of words, since is maintains the co-occurrence of terms in queries. However, not all documents cause cache invalidation, so this method does not indicate a more important region for all documents.

Evaluation We use a dataset $\mathcal{D}$ of 31M Web pages obtained from a Web crawl and a workload $\mathcal{Q}$ of 7M queries from a commercial search engine. In our experiments, we assume that the top-10 documents returned by the search engine to a query are all relevant ($k = 10$). Let $R_Q \subseteq \mathcal{D}$ be the set of 10 results for query Q. Each query is assigned to one of 5 search sites based on its country of origin. $\mathcal{Q}_s^f$ are test queries of site s, and $\mathcal{D}_s$ the documents assigned to s by the master selection method evaluated. Our evaluation measure is the locality of the results returned by each site, i.e.:

$$\mathit{locality}(s) = \frac{1}{|\mathcal{Q}_s^f|} \sum_{Q \in \mathcal{Q}_s^f} \frac{R_Q \cap \mathcal{D}_s}{|R_Q|} \tag{2.6}$$

Given that some popular Web pages are requested by queries originating from different sites, it is impossible to achieve 100% locality at all sites. We first define an optimal baseline for master assignment that assigns each document to the site that requests it the most in the test queries. This gives us a performance upper bound of 72.9%.

In spite of being a simple feature, master selection by language achieves a performance of 38.1%. In our dataset, 58% of the documents are in English, and over 30% of the results of each site are in English. When performing assignment by language, all these documents are sent to a search site assumed to be in North America (80% results in English). This significantly limits performance. However, for less popular languages, this feature performs quite well.

Term distribution improves on language by considering specific words present in the document. Our experiments show that learning the term distribution of a site from the queries gives better result than learning it from the documents it returns. While documents are richer, they also contain noise, while queries describe more specifically the intent of a site's users. Overall, using the term distribution of queries we achieve an average locality of 45.3%.

Cache invalidation mimics query processing to determine which recent query results would be affected by the presence of the new document. Hence, it is not surprising to see that it gives the best results, with 52% locality. However, it can only be applied to 24% of the documents, since most of them do not trigger any invalidation. On this subset of documents, term distribution achieves 48.7% and language 39.5%. Since term distribution and cache invalidation can be interpreted as probabilities, it is possible to combine them. This leads to a performance of 52.6% on documents that trigger a cache invalidation, and 46% overall.

These experiments indicate that it is possible to improve query processing locality in a distributed search engine by predicting which search site is most likely to request a document. Measuring the similarity between a new document and terms used in the queries of each site leads to good results, which can be further improved by using information from the cache invalidation protocol. However, predicting where a new document will be popular remains a difficult task. In addition, many popular documents are requested at several sites. Hence, in addition to the master index, each search site is given additional information in a shadow index and a forwarding index. Contrary to master assignment, these indexes are maintained in a reactive manner, so they adjust to changes in user queries. The protocol maintaining them is presented in Section 2.1.4.

2.1.4 Reactive placement: Index replication

Problem definition Following the master selection process (Section 2.1.3), the collection of documents $\mathcal{D}$ is partitioned and indexed across search sites. This ensures the quality of the results through a conservative query forwarding process. Our goal is now to reduce query processing cost and latency by increasing the number of queries for which a search site can provide exact results locally, without resorting to query forwarding. As we did previously, we estimate $\mathcal{Q}_f^s$, the future queries received at site s, with a recent query stream $\mathcal{Q}_s$. At any point in time, we aim at maximizing the locality of queries in $\mathcal{Q}_s$ to increase the probability that future queries are answered locally.

$R_Q \subseteq \mathcal{D}$ is the set of the k documents obtaining the highest scores for query Q according to the search engine's ranking function $s(D|Q)$ (Equation 2.1). A site $s \in \mathcal{S}$ has to fulfill two conditions to answer Q locally.

1. The documents of R_Q must all be indexed an copied locally. The search site needs to compute an exact score for each document to display them properly ranked. In addition, it needs a copy of the document to generate the snippet presented on the results page, and also in the case that it uses a two-phase ranking [26]. This requirement can be expressed as follows:
$$\forall\, D \in R_Q, D \in MI_s \cup SI_s$$

2. The search site must be able to determine, using local data only, that no other document could potentially score higher than the lowest score of the results:
$$\forall\, D \in (\mathcal{D} \setminus R_Q), \forall\, D' \in R_Q, sBound(D|Q) \leq s(D'|Q)$$
where $sBound(D|Q)$ is the function that computes an upper bound on the score of document D for query Q using only local information, i.e. information from MI_s, SI_s and FI_s

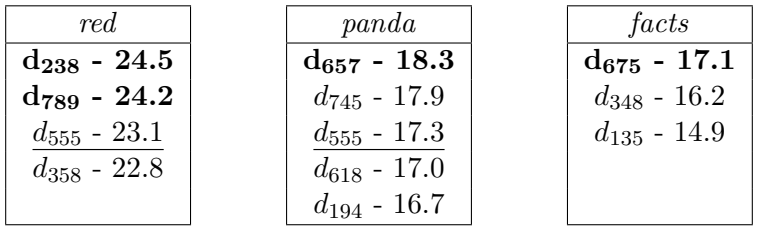


Figure 2.2: Forwarding heuristic on FI_s

Document replication using a Knapsack model The cost of indexing a document is equivalent to the number of posting list entries required. To simplify the problem, suppose that all documents contain the same number of terms and therefore have the same cost for full indexation (SI_s). The problem we are trying to solve is a particular form of the knapsack problem in which objects selected are queries. The utility of selecting a query is proportional to its frequency, while its cost is equal to the indexing of the results (SI_s), as well as the partial information ensuring the quality of results (FI_s). Given that each query has k results, we could make the simplifying assumption that all queries have the same cost: indexing k documents. The knapsack problem is NP-hard, but has greedy heuristics that perform well. The most common approach consist of selecting objects in a decreasing order of $\frac{value}{cost}$. However, in our case, the complexity arises from the fact that many queries share results, so the cost of selecting a query depends on the other queries selected. This makes these heuristics unusable in our setting.

Document replication using a Graph model We can model the dependencies between queries and documents as an hypergraph: documents are vertices, and a query Q is an edge connecting its results R_Q. Replicating a set of documents becomes equivalent to selecting a subgraph, and it allows a search site to meet the first local-answer criterion for all queries present in the subgraph. Hence, we can formulate selection of documents to be replicated in SI_s as a densest subgraph selection, where the weight of an edge is linked to the frequency of the query. Chlamtác [30] et al. found approximation bounds for this problem in the case where each edge is connected to exactly 3 vertices. This is however not the case in our setting, as generally $k \geq 10$.

Document replication using online caching Search engine workloads can vary over time, with new search topics becoming popular overnight (e.g. death of a celebrity, political scandal) and new documents being continuously discovered by the crawler. This consideration indicates that an online approach is favorable, as it allows for faster adaptation of the index. Furthermore, a practical solution should use a minimal amount of computation and memory, so as to ensure that as many resources as possible are dedicated to query processing. Inspired by previous work on Web caches [87], we propose a Reactive Indexing Protocol (RIP) that continuously monitors query execution to trigger replication decisions.

To understand RIP, it is first necessary to describe in details what triggers a query forwarding decision, and more specifically how $sBound(D|Q)$, the function that computes an upper bound on the score of documents that are not indexed locally, works. The score function of the search engine is a linear combination of partial scores, for each query term. Thus, the search engine can rely on the seminal NRA top-k processing algorithm [55], with a few

adaptations to deal with partial knowledge caused by the distributed setting. In NRA, posting lists are sorted by score and processed from top to bottom. NRA maintains a sorted heap of potential top-k results with upper and lower bounds on their scores. These bounds are updated as the processing progresses down the posting lists. As soon as the upper bound of the $k+1^{th}$ document is lower than the lower bound of the k^{th} document, the top-k results are identified an the algorithm terminates. In the worst case, NRA has to process the full posting lists, but, in most situations, it terminates early and only processes a small fraction of the index. When processing a query, the search engine starts by computing the top-k results on documents that are indexed locally (MI_s and SI_s) and obtains a first version of the top-k. The query forwarding heuristic then executes NRA over the Forwarding Index FI_s, that represents all other documents, and obtains computes upper bounds on scores. The posting lists of FI_s are only partial, but they are continuous. A posting list replicated by s for a term t down to the score value v contains all the documents of $\mathcal{D}$ whose master is not s and whose partial score $r(D|t)$ is higher than v. Therefore, for a given term, FI_s provides either an exact partial score, or an upper bound v. While processing, the forwarding heuristic ignores the documents present in the shadow index SI_s, as they are already evaluated.

We illustrate the forwarding algorithm with the example of Figure 2.2. The query Q is *red panda facts* and the figure displays the posting lists of FI_s for those terms. The top documents for *red* are replicated in SI_s (in bold), so they are not considered. The following document is d_{555}, so we know its exact partial score for this term. This document is also present in the posting list of *panda*, so we will also find its exact partial score for *panda* as NRA progresses. However, d_{555} is not present in the posting list of *facts*. The last known document in this posting list is d_{135}. As a consequence, we use its partial score as an upper bound of $r(d_{135}, facts)$. This gives us $sBound(d_{555}|Q) = \frac{23.1+17.3+14.9}{3} = 18.4$. We can also compute *sBound* for any document absent from these posting lists using the scores of the last entries ($\frac{22.8+16.7+14.9}{3} = 18.13$ in this example).

Using FI_s, the forwarding heuristic computes the highest possible score for a document that is not fully indexed locally and compares it with the score of local documents (MI_s and SI_s). If this score bound is higher, the query is forwarded to other search sites. To avoid this situation, RIP adjusts which posting lists are prioritized in FI_s. More specifically, RIP controls replication by selecting, for each term t, two replication thresholds expressed in partial score values: the document replication threshold td_t and the postings replication threshold tp_t.

$$\forall\, D \in \mathcal{D}, r(D|t) \geq td_t \wedge master(D) \neq s \implies D \in SI_i$$

$$\forall\, D \in \mathcal{D}, r(D|t) \geq tp_t \wedge master(D) \neq s \implies D \in FI_i[t]$$

In the example described on Figure 2.2, $td_{red} = 24.2$ while $tp_{red} = 22.8$. By lowering td_t, RIP decreases the highest scores associated to t from a non local document. Lowering tp_t decreases the lowest score associated to t in FI_i. Both these actions increase the information related to the term t and decrease the amount of query forwarding. However, their impact and cost can vary significantly. Fully replicating a document is costly, as it generates one posting entry per unique term in the document. On average, a Web page contains 250 unique terms [99], so replicating a document is 250 times more costly than replicating a posting entry. Given that the difference in partial scores between entries are, in most cases, higher among high quality documents, fully replicating a document often has a higher positive impact on

query forwarding. RIP achieves a good balance between documents and postings replication to use the replication budget as efficiently as possible.

Evaluation We use the same distributed search engine setup and dataset as in the evaluation of Section 2.1.3. As explained in the problem definition, our goal is to maximize the number of queries that can be answered locally, which is the equivalent of minimizing query forwarding. Following the master assignment phase, each document is indexed by a single search site. Then, each site has a replication budget that can be used to either fully index documents it is not the master of (SI) or replicate fragments of posting lists (FI). We compare 3 different replication policies:

- Static Documents replication (SDR): Using the training queries, we determine which documents are globally popular and replicate this static set across all sites [13, 25]. In this case, FI contains an upper bound per term for non replicated documents, which corresponds to the thresholds of Baeza-Yates et al [13].

- Reactive Documents Replication (RDR): Each search site reactively determines which documents are most frequently part of the results of their users and replicates the most popular ones. This setup computes a different set of replicated documents for each site to match the activities of their users. As with SDR, FI only contains one bound per term, dynamically adjusted to reflect document replication.

- Reactive Indexing protocol (RIP): Each site reactively replicates blocks of documents and posting lists, as well as individual documents when they generate false positives in query forwarding. This is the approached based on *online caching* described in this section. Each site replicates data matching the needs of its users, while preserving continuity on information in posting lists.

We evaluate the impact of each approach on the proportion of local queries for varying replication budgets and report results on Figure 2.3. SDR replicates the same documents on all search engines, so it does not optimize replication based on the behavior of the users of each specific search site. Hence, it obtains the lowest performance, as users of different regions are generally interested in different documents. For a low replication budget (below 100k documents), we observe that simply replicating the results of the queries is more efficient than replicating to FI, so RDR performs best. However, as the budget increases, RIP clearly outperforms RDR. With a replication budget of 1M documents, each search site has an average indexing capacity of 7,119,982 documents (22.5% of the total collection), which represents an overhead of 14% over a setup without any replication. In this configuration, RIP raises the amount of queries processed locally by 25% while RDR raises it by 13%.

These experiments show that it is necessary to integrate the index replication process with the query processing function. RIP takes into account the properties of top-k algorithms to ensure not only that relevant documents are replicated locally, but also that the search engine is able to lower the score bound on unseen documents sufficiently to avoid query forwarding.

2.1.5 Conclusion

Different people from different regions of the world request different content. While this simple fact seems obvious, based on language differences alone, using it to design more scalable search

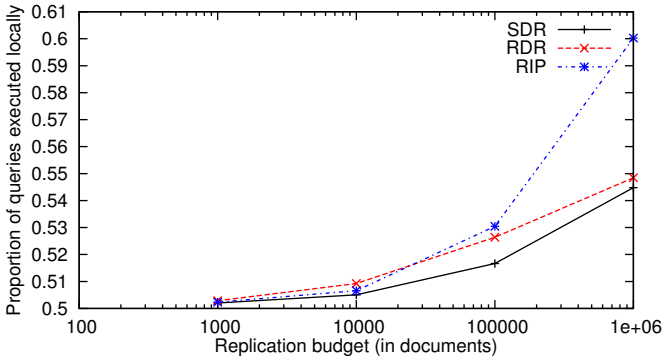


Figure 2.3: Query locality wrt. replication budget

engine architectures is quite challenging. The main reason is that search results quality is of the utmost importance, so every request must be answered as if each search site was hosting a full copy of the Web index. Whenever a search site is missing content to ensure the accuracy of its results, the query is forwarded to other search sites, which incurs a high processing cost and latency.

We divide the problem of selecting on which search site a document should be indexed in two phases:

- In the case of new documents that have never been returned as search results before, we compare the distribution of terms in the documents to the queries of the users in each site, and select for each document a single master site that is responsible for indexing it. When the search engine has a cache, we can also rely on the cache invalidation protocol to estimate the proportion of queries that would have returned this document if it had been indexed previously.

- In the case of a document that is already indexed at a master site, a different search site can replicate this document is its absence causes a significant amount of query forwarding. The main difficulty is that having all correct documents is not enough to avoid query forwarding, the search site must also ensure that no unknown document can score higher. This is done by integrating with top-k processing algorithm and replicating score bounds from other search sites.

This work integrates existing search engine architectures (indexer, query processor, cache ...) with a fully data-driven content assignment policy. As each search site specializes for the queries of its users, search engines become more scalable and less costly to operate. Our approach was validated using a real dataset consisting of a Web crawl and user queries from different countries. While experimental results were promising, there was unfortunately no real deployment within Yahoo! due to a change of focus in the company.

2.2 Single data center: Key/Value-based applications

2.2.1 Context

Distributed applications are deployed in clusters of servers that collaborate to provide a service. As an application becomes more popular and its workload increases, more severs are added to the pool of resources available. In this section, we are interested in the problem of dividing the workload among these servers. Many applications are modeled following the key/value paradigm. Pieces of data (values) are uniquely identified by a key, and can be updated or retrieved. For instance, in a social network such as Twitter, the publications of a user (value) can be stored using the identifier of the user as a key. When the user posts a status update, the value associated to her identifier is updated. When she browses her social feeds, the values associated to her friends are retrieved. Another example of key/value application is a URL shortener service such as Bitly. Each time a user clicks one of the shortened URLs (key), the popularity counter associated to it (value) is updated.

In this work, we consider two building-block operations of these key/value applications:

- **Multi-get query**: This query consists in retrieving content associated to multiple keys: $multiget(k_1, k_2)$ fetches content associated to keys k_1 and k_2. This type of query represents a significant proportion of the workload of social networks such as Twitter or Facebook [91]. The multi-get query can be used to construct the social feed of a given user by retrieving content generated by all of her friends. Hence, a user following 50 twitter accounts triggers a multi-get to 50 different keys whenever the social feed is updated.

- **Streaming sequence**: This operation consists in streaming a message to different keys sequentially: $stream(m, k_1, k_2)$ sends message m first to key k_1 and then to k_2. Note that the sequence of keys is not always known in advance from m alone: k_2 can depend on the combination of m and the current value associated to k_1. This operation is common in stream processing applications. For instance, when a customer buys products in a store, her receipt can be first routed to the user identifier to update her purchase history, then to the store identifier to update its stock, and depending on the stock count, to the appropriate provider.

The common feature of these two operations is that they access the data associated to several keys, in parallel or sequentially. As the application is deployed on a varying number of servers, it requires an application level routing service able to determine which server is currently hosting data associated to a given key. The most common strategy to implement this type of routing is a hash-based pseudo-random assignment: $server = hash(key)$. The main advantage of this policy is that routing is stateless: any server can directly route to the recipient server of any key without requiring a consistent view of a routing table. The main drawback is that this deterministic yet random assignment leads to a significant spread of keys accessed by each operation. Indeed, the likelihood that two keys are located on the same machine is $\frac{1}{\#servers}$. As the application is scaled to a larger number of servers, the average number of servers each operation needs to access increases, up to, in the worst case, one per key.

Servers within a data-center are typically connected following a hierarchical tree-shaped network structure. Figure 2.4 provides an example of a three-level tree of switches, with a core tier at the root of the tree, an intermediate tier, and an edge tier at the leaves of the tree. The core tier consists of the top-level switch (ST), which connects multiple intermediate

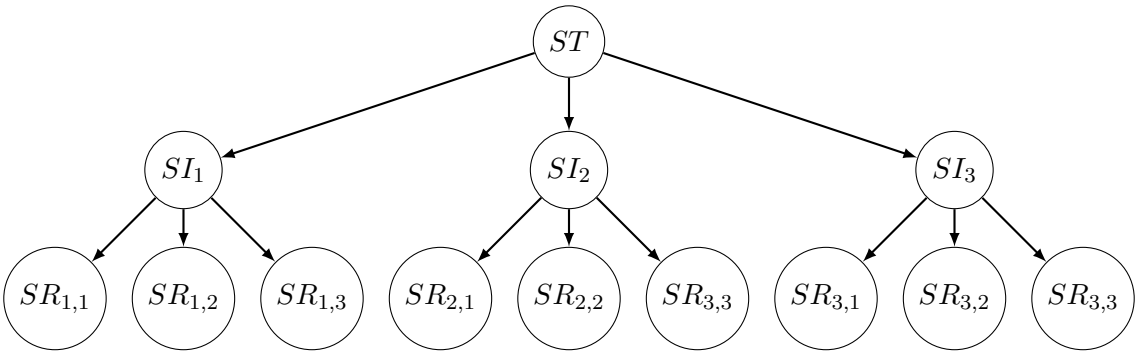


Figure 2.4: Hierarchical network architecture

switches. The intermediate tier consists of intermediate switches (SI), and each of them connects a subset of racks. Finally, the edge tier consists of racks, and each rack is formed by a set of servers connected by a rack switch (SR). This hierarchical structure leads to heterogeneous communication costs between servers. Messages exchanged between servers of the same rack only need to traverse the rack switch. However, messages between servers located in different branches of the tree also require going through intermediate switches, and potentially all the way to the top-level switch. As the resources needs of the application grow, the number of servers increases, and an application originally deployed in a single rack spreads to multiple racks. In this situation, hash-based key assignment generates communications between random pairs of servers, which tends to saturate top-level and intermediate switches, leading to degraded performance and limiting scalability.

The solution we propose to counteract this effect is to account for the heterogeneity of network architecture, and replace pseudo-random data placement by a stateful routing policy designed to optimize the likelihood that keys accessed by an operation are located at close network distance, ideally on the same server or within the same rack.

The work presented in this section was started during my post-doc at Yahoo! Research Barcelona, and was continued at the University of Grenoble in collaboration with the ERODS research group. It is detailed in the following publications:

- **DynaSoRe: Efficient In-Memory Store for Social Applications**
  Xiao Bai, Arnaud Jégou, Flavio P. Junqueira and Vincent Leroy
  In Proceedings of the 14th International Middleware Conference (Middleware), 2013, pages 425–444.

- **Locality-Aware Routing in Stateful Streaming Applications**
  Matthieu Caneill, Ahmed El Rheddane, Vincent Leroy and Noel De Palma
  In Proceedings of the 16th International Middleware Conference (Middleware), 2016, pages 1–13.

In Section 2.2.2, we first consider the case of an offline optimization for a static workload and model it as a graph partitioning problem. We then move on to the online optimization problem. Section 2.2.3 presents the case of a social network application using multi-get queries. Then, Section 2.2.4 presents the case of streaming applications relying on sequentially routing messages to keys. Finally, we conclude in Section 2.2.5.

2.2.2 Offline approach: data placement as a graph partitioning problem

Problem definition Given an application deployed on a set of servers $\mathcal{S}$ and maintaining data associated to a set of keys $\mathcal{K}$, our goal is to assign each key $k \in \mathcal{K}$ to a server $s \in \mathcal{S}$. $K_s \subseteq \mathcal{K}$ represents the set of keys assigned to s, and $serv(k) = s$ indicates that k was assigned to s.

Our first goal is to optimize *locality*: given an operation of the application, we want to minimize the cost of messages exchanged between servers. We assume the existence of a $netCost(s, s')$ function aware of the network topology and indicating the network cost of communicating between servers s and s'. By definition, $netCost(s, s) = 0$, while $netCost(s, s') < netCost(s, s'')$ if s and s' are in the same rack, while s and s'' are on different racks.

In the case of $multiget(k_1, \ldots, k_n)$ executed from a broker b, we want to minimize the cost of contacting all servers hosting keys:

$$\sum_{s \in \bigcup_{i=1}^{n} \{serv(k_i)\}} netCost(b, s)$$

This can be done by ensuring keys are co-located on the same server to reduce the number of servers reached by the query, but also by ensuring that these servers are in the same rack as the one executing the query, b.

Similarly, in the case of a streaming operations $stream(m, k_1, \ldots, k_n)$, our goal is to minimize the cost of transitions between keys:

$$\sum_{i=1}^{n-1} netCost(serv(k_i), serv(k_{i+1}))$$

In this case, the cost is reduced when consecutive keys k_i, k_{i+1} are located on the same server or, failing that, close servers.

Each key is associated a weight $weight(k)$ representing the resources a server spends to serve queries related to k. Depending on the application's bottleneck, this weight can be related to the size of the data associated to k, or the frequency at which k is accessed. Hence, the load of a server is given by:

$$load(s) = \sum_{k \in K_s} weight(k)$$

Our second objective is to ensure *load balancing*, i.e. ensure that the load of all servers is comparable. This secondary objective is necessary to ensure that no server is overloaded, as the trivial solution of assigning all keys to the same server would maximize locality. We express load balancing using a maximum imbalance factor ε:

$$\forall\, s \in \mathcal{S}, load(s) \leq (1 + \varepsilon) \min_{s' \in \mathcal{S}} load(s')$$

i.e. no server is assigned a load higher than $(1 + \varepsilon)$ times the minimum load.

Modeling as a graph When the workload is steady, it is possible to gather statistics on queries executed, compute an offline assignment, and deploy the application using this optimized configuration. In our context, the natural solution to describe the workload is a

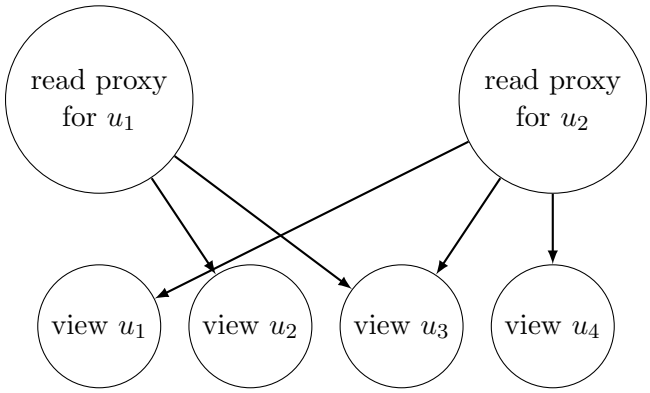


Figure 2.5: Graph obtained with multi-get queries in a social network. u_1 follows u_2 and u_3 while u_2 follows u_1, u_3 and u_4.

graph of keys with edges representing communications between servers handling these keys. For instance, Figure 2.5 presents a graph resulting from multi-get queries. In the case of a social network, the structure of the graph is given by friendship relations between users, while observing the real workload gives insight on the frequency of queries and thus the weight of edges. Indeed, it is more important to optimize queries for users that refresh their twitter feed every 10 minutes than for users that consult it once a week. Figure 2.6 is an example of graph obtained from streaming sequences. In this context, the graph structure is discovered from application traces, as it relies on correlations between different fields of messages.

Once the graph is obtained, assigning keys to a set of servers $\mathcal{S}$ consists in partitioning the graph into $|\mathcal{S}|$ subgraphs, where the keys of each subgraph are assigned to a single server. Graph partitioning aims at minimizing the cost of edges traversing partitions, which is consistent with our *locality* objective. Furthermore, several partitioning algorithms, including the ones from the METIS library [59], support constraints on the balance of clusters, and thus meet our *load-balancing* objective. Hence, we can implement an offline algorithm based on METIS that gathers traces from previous executions of the applications, and generates a static assignment of keys to servers. This is the approach used by Schism [33] to improve the locality of transactions in a distributed database. In our case however, it only constitutes a starting point. Indeed, we aim at supporting workloads that vary over time. This requires an online approach, able to monitor the workload and reconfigure the system dynamically without interrupting the application. In Section 2.2.3, we detail the online configuration process for multi-get queries, and in Section 2.2.4, we present our approach for streaming applications.

2.2.3 Online multi-get

Architecture In the context of multi-get queries, the system we developed, DynaSoRe (Dynamic Social stoRe), is specifically designed for handling social feeds (e.g. Twitter). Publications are stored as producer-pivoted views, i.e. all publications of a user are stored in a single data structure keyed by the user identifier. The API provides two functions:

- `read` retrieves the views of all the friends of the user, according to the social network. This function relies on the *multiget* query introduced earlier, and aims at generating the social feed of the user.

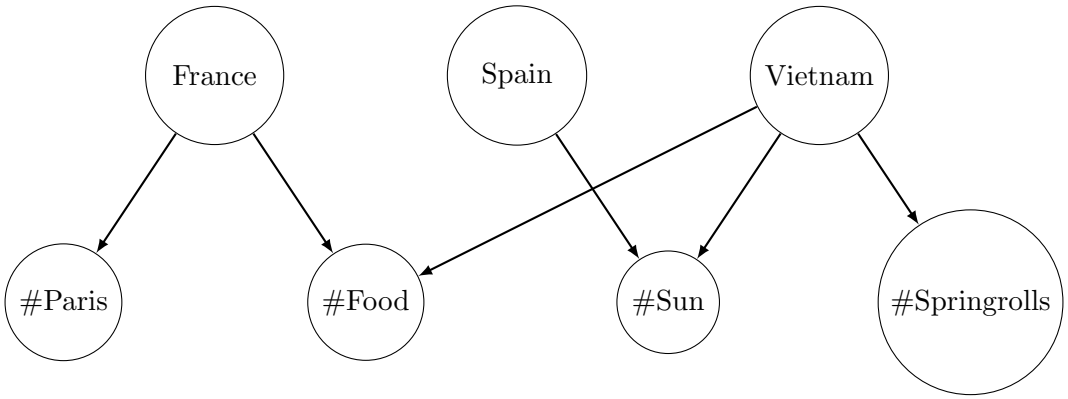


Figure 2.6: Graph obtained with routing sequences. The first key corresponds to a location, while the second key is a hashtag. Tweets posted in France are associated with hashtags #Paris and #Food.

- `write` appends a publication to the view of the user. DynaSoRe is designed as an in-memory cache, so adding a message to a view evicts the oldest publication to keep its size constant. We assume that publications are simultaneously persisted to a stable storage.

Each user is assigned two proxies deployed on broker servers and in charge of executing her `read` and `write` operations. The rationale behind separating the read proxy and the write proxy is that these functions access different views (friends' views against user view), and so their optimal positioning with respect to the hierarchy of servers can differ. The view of the user can be replicated to several servers. This further improves locality, as a read proxy can chose to access the closest replica of the view to further reduce the load of the top-level switch. The trade-off is that writes must be sent to all replicas of the view, and that memory consumption increases.

Gathering of statistics DynaSoRe starts from an initial data placement obtained from a clustering algorithm based on previous execution traces, as described in Section 2.2.2. Then, DynaSoRe continuously monitors the workload to continuously adjust and improve locality. DynaSoRe maintains statistics about the frequency and the origin of each access to a view. This information is stored on the servers, along with the view itself. The origin of an access to a view is the switch from which the request accessing this view originates. Consequently, two brokers directly connected to the same switch correspond to the same origin. The writes to a given view are always executed from the write proxy. However, reads can originate from any broker in the cluster, so their origin should be tracked.

To reduce the memory footprint of access recording, DynaSoRe makes the granularity coarser as the network distance increases. Considering a tree-shaped topology, a server records accesses originating from all the switches located between the server and the top-level switch, as well as their siblings. For example, in Figure 2.4, a server located in the rack of $SR_{1,1}$ records accesses from $SR_{1,1}$, $SR_{1,2}$, $SR_{1,3}$, SI_2, SI_3. In this way, in a cluster of m intermediate switches and n rack switches per intermediate switch, every replica records a maximum of $m - 1 + n$ origins instead of $m \times n$. While significantly reducing the memory footprint, this solution does not affect the efficiency of DynaSoRe, as the algorithm still benefits from precise

information when making the final adjustments on the assignment of a view. In addition, to adjust to variations of the workload over time, we use rotating counters to focus on recent activity.

Data migration policy Each server has a fixed memory capacity, expressed as the number of views it can store. DynaSoRe ensures that the view of each user is stored on at least one server. Each server stores several views, some of them being the only instance in the system, while others are replicated across multiple servers and therefore optional. Using the view access statistics, DynaSoRe evaluates the utility of a view on a given server, i.e. the impact of storing the view on this server in terms of network traffic. In the absence of the view, the overhead for `read` requests is computed by assuming the request would be sent to the closest replica of the view instead. The cost of updating the view (`write`) is then subtracted. If the utility is negative the cost of maintaining the replica outweighs its benefits for `read` requests, so the view is immediately eliminated. Each server maintains a utility admission threshold equal to the 10^{th} percentile of the views they host (or 0 if the server has over 10% free capacity). If the server reaches 95% memory usage, it eliminates the view replicas having the lowest utility. Upon receiving a `read` for a view, the server evaluates the possibility of replicating it to a server closer to the proxy performing the request. Similarity to the utility computation presented above, the server evaluates the benefits of this additional replica by simulating a "what-if scenario" in which the replica exists. If the utility computed is higher than the admission threshold of the server hosting the new replica, the view is replicated. When replicating or evicting data associated to a given key, the write proxy of the associated user acts as a coordination point to ensure that one copy always remains available, and that writes are propagated to all replicas. Note that while DynaSoRe does not implement a "view migration" operation per-se, they occur by first replicating the view to a new location, and then deleting the original view because of its low utility.

Evaluation We consider a social network extracted from Facebook [96] that consists in 3M users connected by 47M friendship links. Each user to the social graph is assigned real user traffic extracted from Yahoo! News Activity for a total of 17M writes and 9.8M reads. Since this Yahoo! application allows users to share news with their Facebook friends, this combination of datasets represents a realistic setup. When establishing a correspondence between Yahoo! users and the Facebook graph, we assign the most active profiles to the users with most friends to account for the correlations observed by Huberman et al. [52]. The data-center considered is simulated. It has a top switch, 5 intermediate switches, each connected to 5 rack switches, for a total of 25 racks containing 10 servers each. In each rack, 1 server is a broker while the 9 others are used for storage of views associated to keys. The capacity of the servers is tailored to the dataset, with an additional 50% for replication.

We compare 4 different data placement policies:

- Random: This policy is a baseline that corresponds to a hash-based pseudo-random assignment of keys to servers. This policy is used by many distributed data-stores. Each key is stored on a single server, so this policy does not leverage additional storage capacity.

- SPAR: This algorithm replicates all the views in a user's social network on the same server to implement highly efficient read operations [81]. SPAR does not consider write

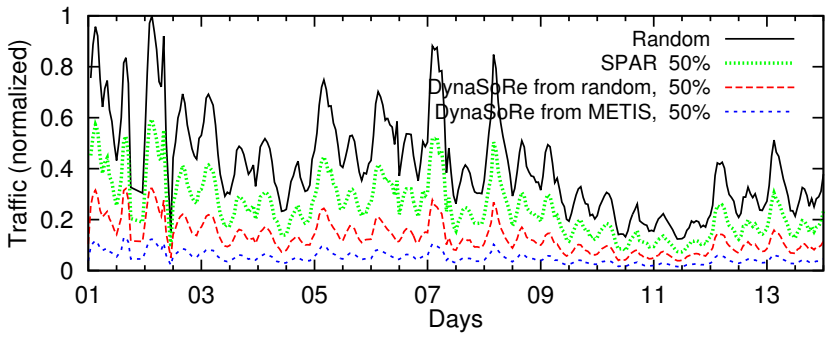


Figure 2.7: Top switch traffic with a Yahoo! News Activity query log

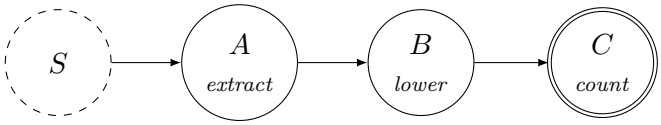


Figure 2.8: A simple wordcount stream application. S sends sentences, operator A extracts words, B converts them to lowercase, and C counts the frequency of each word.

costs when creating additional replicas, and does not assume bounds on the capacity of servers. To obtain a fair comparison, we adapt SPAR to avoid replication when a server is out of capacity.

- DynaSoRe from random: This is a fully online version of our algorithm described in Section 2.2.3, where the initial data placement is random.
- DynaSoRe from METIS: This is the algorithm described throughout Section 2.2.3, where the initial assignment of keys to servers is obtained using a clustering algorithm (Section 2.2.2).

Figure 2.7 shows the traffic of the top-level switch throughout the 14 days of the traffic log duration. This figure shows that DynaSoRe is able to converge to an efficient view placement configuration, even in the case with high variance traffic. DynaSoRe clearly outperforms the random baseline, but also SPAR, despite its efforts to improve locality. This is caused by three main factors: (i) DynaSoRe handles limited server capacity by prioritizing replicating views that have a high impact on traffic, (ii) DynaSoRe accounts for the network hierarchy to place data in the vicinity when a server is full, (iii) DynaSoRe evaluates the read and write costs when replicating data, to avoid high update costs when the read benefits are low. We can see that DynaSoRe performs better when the initial configuration is obtained through clustering, while it gets stuck in a local minimum when starting from a random configuration. These results only show the traffic of the top-level switch, which is where DynaSoRe has the highest impact. However, the network traffic is reduced at all levels, albeit in a smaller proportion.

2.2.4 Online stream sequence

Architecture Stream processing was developed to continuously execute operators on potentially unbounded streams of data tuples. Apache Storm [90], Flink [27], S4 [73], Samza [11],

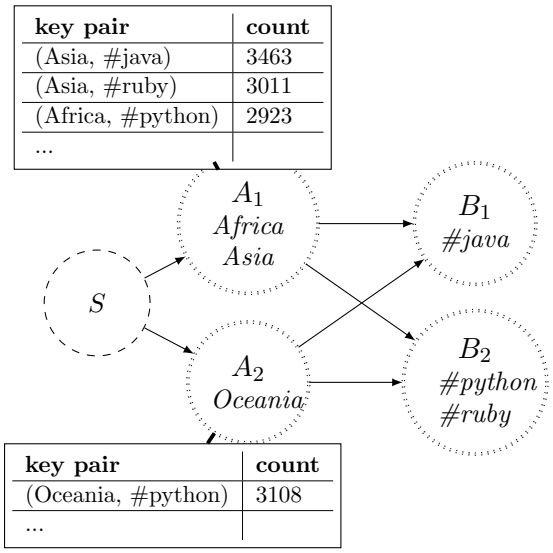


Figure 2.9: PO instrumentation: every instance counts the key pairs it receives and sends, and keeps the most frequent pairs in memory.

and Twitter Heron [62], are examples of popular stream processing engines. Following the dataflow programming paradigm, a stream processing application can be described as a Directed Acyclic Graph (DAG). Vertices represent processing operators (POs) that consume and produce data streams, which flow along edges. A source constitutes the entry point of the DAG, and streams data tuples, such as posted tweets or uploaded pictures, to POs. Figure 2.8 represents a simple wordcount application for streams of sentences. POs A and B are stateless, as they do not update any internal state when processing data, while C is stateful as it maintains frequency counts. In this work, we are interested in stateful POs, since they maintain state associated to keys.

The specification of the application DAG indicates which PO is the recipient of another PO's output stream. To scale stream processing, each PO is executed in parallel on multiple instances (POIs) deployed on servers. In the case of a stateful PO, the state maintained is partitioned horizontally across POIs. Hence, it is important to select which particular POI receives each tuple of data. This is handles by the *fields grouping* routing policy, where the developer selects the field of the tuple that is used as a key in the stateful PO. As mentioned in Section 2.2.1, the default implementation for *fields grouping* relies on hash functions. Our contribution is the addition of routing tables in *fields grouping* that optimize data placement. These routing tables indicate, for some keys, the POI they should be routed to. If a key is absent from the routing tables, the default hash-based policy is used.

Gathering of statistics Data streams often fluctuate over time, particularly when they are generated by human activity. For example, *#breakfast* is associated to *America* and *Europe* at different moments of the day. In addition to diurnal and seasonal patterns, flash events can occur, generating temporary correlations between keys. It is necessary to detect these correlations at run time to perform an online optimization of stream routing without interrupting the execution of the application. For this purpose, we add an instrumentation tool to stateful POs. For each passing message, a POI extracts the input key, which was

used to route the data tuple to this instance, and the output key, which decides towards which POI the message is routed next. Pairs of keys are stored in memory along with their frequency, as depicted in Figure 2.9. Computing the frequency of pairs of keys online can be done with the *SpaceSaving* algorithm [70]. Using a bounded amount of memory, it maintains an approximated list of the n most frequent pairs of keys. This limitation on the collection of statistics is, fortunately, not problematic for most large-scale datasets. Indeed, many real datasets follow a Zipfian distribution [9], with few very frequent keys, and many rare keys. Identifying the pairs containing the most frequent keys captures most of the potential for optimization, so the loss compared to an exact offline approach is limited. Whenever the routing of keys is updated, the statistics are reset to only take into account recent data and detect new trends.

Data migration policy State migration in the case of stream processing is more complex than in the case of multi-get queries. Stream workloads are significantly more write intensive, so data is not replicated and a key is only handled by a single POI. Furthermore, in DynaSoRe (Section 2.2.3), we know that write messages for a given key all go through its write proxy. Hence, we can use it as a coordinator to control state migration while ensuring data consistency. In the case of stream processing, a write to a given key can originate from any POI of the previous PO, so there is no obvious single coordinator.

Every stateful POI holds the state of the keys to which it is associated. When a key is assigned to a different POI in the updated routing tables, its corresponding state needs to be transferred between POIs. Moreover, after POIs migrate the state of their previous keys, they should no longer receive any message related to this key. This means that preceding POs in the DAG must have proceeded to their reconfiguration first, and route messages according to the new routing tables. To this purpose, we opt for synchronized reconfiguration cycles, instead of a fully on-the-fly approach as in DynaSoRe. A global *coordinator* gathers all statistics and generates a new optimized configuration which is deployed regularly. The configuration is computed using graph partitioning, as described in Section 2.2.2. The challenge is to deploy this new configuration and migrate data without interrupting stream processing or losing data. The coordinator synchronizes key migration in a progressive reconfiguration following the PO order specified by the DAG. This protocol is shown in Figure 2.10.

The reconfiguration protocol is executed by the coordinator C. The coordinator first asks every running POI to send the collected statistics ①. Upon receiving them all ②, it builds the bipartite graph of the key pairs (see Figure 2.6), partitions this graph with METIS, and computes the new routing tables. It sends these tables to the respective POIs ③, and waits for all acknowledgements ④. It then enter the propagates phase, and tell the instances of the first PO to proceed to the reconfiguration ⑤. The two instances update their routing table and exchange the state of the keys whose assignment has changed ⑥. After this operation, they forward the propagation instruction to the instances of the second PO ⑤, which in turn update their routing tables and exchange their states if necessary ⑥.

The data stream is not suspended during reconfiguration, so it is possible that a POI receives a tuple associated to a key while it has not yet received the state associated to it. In this case, tuples are buffered and are only processed once the state of their key is received. This solution is preferable to suspending the stream as some stream sources do not support back pressure and would lose messages. To handle fault tolerance, the coordinator saves all routing configurations to stable storage before starting reconfiguration. If a POI crashes,

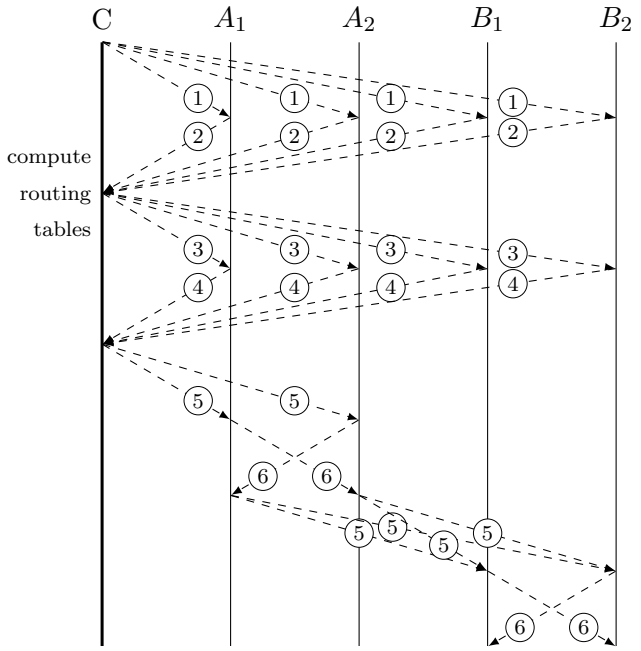


Figure 2.10: Reconfiguration protocol, forwarding routing tables and key states between POIs. ① Get statistics. ② Send statistics. ③ Send reconfiguration. ④ Send ACK. ⑤ Propagate. ⑥ Exchange keys.

the guarantees are the ones provided by the streaming engine and are not impacted by state migration.

Evaluation We implement our algorithm in Apache Storm to evaluate its performance. For evaluation purposes, we consider the case of a streaming application composed one source, S, and two stateful POs, A and B. The first PO computes statistics based on the first field of the tuples by counting the number of occurrences of its different values, and the second PO executes the same operation on the second field. Hence, fields grouping is used to route data tuples to both POs. Each PO is deployed as 6 instances, and we ensure that every instance of the first PO (A) has a local instance of the second (B).

Our workload consists of tweets crawled from October 2015 to May 2016 using the API of Twitter. Twitter provides for each tweet a location identifier which can be either the location of the user at the moment of the tweet, or a location associated to the content of the tweet. Locations can be countries, cities, or points of interests. Overall, our dataset contains 173 million associations between locations and hashtags. We set our application to first route using the location, and then the hashtag.

We compare 3 different data placement policies:

- Random: This is the default pseudo-random key assignment policy based on hash functions.
- Offline: This consists of a single data-placement configuration step, corresponding to an offline approach which is then never updated.

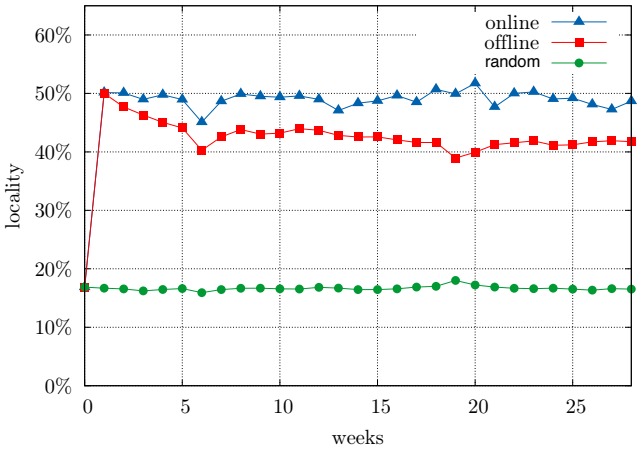


Figure 2.11: Locality obtained with different data placement policies

- Online: This consists a regularly optimizing data placement to continuously improve performance and adjust to changes in the characteristics of the workload.

Our performance measure is locality, i.e. the proportion of data tuples that are processed on the same server for both POs.

Figure 2.11 shows the evolution of locality over time, *Hash-based* achieves a locality of 16.6%, which corresponds to a random assignment with 6 servers. After one week, *online* and *offline* both obtain a sufficient amount of data to perform locality-aware routing, which raises the locality to 49%. However, this value decreases over time in the case of *offline*, and stabilizes around 40%. *Offline* preserves locality for stable associations, but fails to leverage transient ones. *Online* however maintains a locality in the vicinity of 50%. This shows that to capture volatile correlations, reconfiguration should be triggered on a regular basis. Our throughput experiments indicate that a 10% difference in locality can lead to a throughput gain of 25%. This demonstrates the benefits of the online optimization process in the case of fluctuating workloads. Note that when generating routing tables, METIS reports an expected locality of 75%. This difference in performance is explained by two reasons. The first one is that we only record the most frequent associations between keys, so METIS does not have perfect information. The second is that data varies between consecutive weeks. It contains a significant proportion of new hashtags and locations that were not observed previously and are thus routed using hash functions.

2.2.5 Conclusion

When distributing applications across several servers co-located in a data-center, the most common strategy is a hash-based pseudo-random assignment of data. However, this simplicity has a cost: diminishing returns when scaling applications. As more servers are involved, the probability that pieces of data accessed are located on different servers increases, which leads to degraded performance as the number of servers increases.

We consider the case of Key/Value-based applications, such as social networks and stream processing, as their execution cost is generally dominated by network communications, and not processing costs. By analyzing execution traces, we can discover correlations between accesses to specific keys, and place them on the same servers to avoid communications. We used the

Schism approach [33] as a starting point for an offline optimization using graph clustering. Our contribution is an extension to online reconfiguration. This allows the optimization of systems without interrupting their execution, and is also necessary to deal with workloads that vary over time.

- For social applications relying on the multi-get query, we opt for a fully online-approach. The data of a given key can be replicated to several servers to improve the locality of accesses, while replicas with low utility are eliminated. We rely on one coordinator per key, so the protocol is event-based and is triggered by performance measures triggered by data accesses.

- In the case of stream processing, an access to a given key can originate from multiple machines, which makes it more difficult to maintain the consistency of routing tables. In this case, we opt for a cycle-based optimization orchestrated by a coordinator on a regular basis. The computation of an optimal configuration is similar to the offline approach, and the main focus is on deploying this new configuration through the application without interrupting it or losing consistency.

In the case of social applications, our system was evaluated on real datasets through the use of an event-driven simulator. Subsequently, when working on streaming application, we were able to implement our solution in the storm processing framework using a real deployment. In both cases, our experiments demonstrated significant gains, and I the next step will be releasing our implementation to an open-source stream-processing platform to benefit as many applications as possible.

2.3 Summary of contributions

During my Ph.D, on large-scale P2P networks, I was dealing with the problem of organizing networks by connecting users interested in the same data. In the context of my post-doc, I remained in the domain of distributed systems, but considered applications running on dedicated hardware. The main difference is that dedicated hardware means that I was in control of the data, and could decide which machine should be storing a specific information. Hence, instead of establishing a link between two servers, I could instead move the data to avoid network communications altogether.

When dealing with applications deployed on multiple data-centers, such as Web search engines, regional trends in data accessed can be easily observed by examining the query log of each search engine. These differences can then be analyzed to predict, using the content of each Web page, in which region it is likely to be accessed. In the case of single data-centers, all queries are processed from the same location, and discovering locality in data accesses can be more difficult, as trends are not naturally partitioned in different query logs. Fortunately, graph clustering algorithms applied to access patterns are able to uncover correlations. The application can then be configured to place data according to these clusters in order to maximize the locality of data accesses.

Data processing frameworks generally consider data-centers as a flat infrastructure, and advertise easy scalability by simply adding servers ("Kill It With Iron"). But without paying attention to data placement, communication costs increase significantly with the number of servers, so each new server brings diminishing returns. We showed, through simulations and live deployments using real datasets, that improving the locality of distributed applications has a large impact on their throughput and scalability. By better leveraging the properties of data, we can actually divide large applications running on hundreds of machines into many small clusters of machines collaborating, with few communications between them, and get much better performance with the same hardware.

Chapter 3

Distributed algorithms for top-k processing

Several frameworks have been developed for batch processing on large-scale datasets. Hadoop Map-Reduce and Spark are two of the most popular open-source options available. These frameworks deploy the execution over executors distributed across servers in a data center. The execution of a program is divided into *stages*. Within a stage, each executor processes data locally, without communicating with other executors. Between stages, the framework can operate a *shuffle* that re-distributes the data between executors. This is most notably the case before *reduce* phases, that require pieces of data having the same key to be stored on the same executor. Shuffling is expensive, as it generates a significant amount of I/O operations, so these frameworks aims at limiting shuffles to a minimum. In addition, a *barrier* synchronizes executors at the end of a phase, so each phase should encapsulate as much processing as possible, to avoid wasting execution time. To sum up, the only time at which data can be exchanged is between processing phases, which does not occur frequently.

Given these constraints, we can see that algorithms in which the workload can be partitioned into independent jobs while requiring very rare data exchange between jobs are natural candidates for being ported to these frameworks. This is for instance the case for join processing algorithms, that aim at outputting all data tuples satisfying a predicate. Generally, the challenge when distributing these algorithms lies in ensuring each executor gets a balanced load while minimizing data transfers (I/O) [100]. This becomes increasingly difficult in cases where the presence of a data tuple in the output does not only depend on a input query, but also on other results candidates discovered at runtime. A common class of such problems is top-k processing [41]: the score of a data tuple is defined by the query, but its presence in the output depends on the score of all other result candidates. Top-k queries require a *dynamic exploration* of the results space in order to discover promising results first and prune as many candidates as possible to reduce the workload. The first challenge in distributing top-k processing is that it is difficult to predict the cost of processing a workload partition: all candidates may be pruned from the beginning, leading to a small execution time, or all candidates may be explored, leading to a high execution time. The second challenge comes from the lack of communications between executors during processing phases: if an executor discovers a candidate with a very high score, the other executors are not aware of it until the next shuffle phase, which can lead to the exploration of candidates that would have been pruned otherwise.

In this chapter, we propose solutions for distributing two specific top-k processing applications. In Section 3.1, we consider the case of processing joins on interval data. Using scored temporal predicates, we assign a score to each data tuple, and aim at finding the top-k best results. Our distributed algorithm, TKIJ, relies on statistics on the data to bound the scores of results present in each partition of the workload. We then ensure that each executor receives a fair share of high-scoring tuples so that it can prune efficiently locally despite the lack of communications between them. Section 3.2 considers the case of mining patterns in large-scale retail datasets. We propose item-centric mining, an approach that aims at finding the top-k patterns of each product in the dataset. We then present TopPI, an algorithm that solves this problem by combining methods from pattern mining and top-k processing. We show that TopPI can be implemented on Map-Reduce by splitting the mining phase into only two stages, without altering the pruning efficiency of the algorithm.

3.1 Temporal joins

3.1.1 Context

Temporal data is pervasive. Store receipts, tweets, traffic data, and temperature measures generated by weather sensors or by wearables, are just some examples. Such data is best represented as intervals with start and end timestamps. For example, in network traffic, a connection between machines forms an interval. In social media, the lifespan of a discussion topic is represented as an interval. Temporal data analysis requires the ability to compare intervals. These comparisons are expressed as predicates that reflect the chronological relationship between intervals, such as *before, meets, starts* or *overlaps* from Allen algebra [5].

As datasets become larger, these simple binary predicates show their limitations. The number of results of *before* query on two collections of intervals C_1 and C_2 is $O(|C_1| \times |C_2|)$. This is not practical when dealing with millions of intervals. Not only does generating all results consume a large amount of processing time, but it is also unlikely that an application actually requires every single results. Furthermore, binary predicates rely on strict equality of endpoints, which is illusory in a distributed environment of smart devices where clocks can be slightly desynchronized. The ability to evaluate predicates *approximately* and assign scores to resulting interval pairs appears as a natural requirement to finding interesting results. We formalize Ranked Temporal Joins (RTJs) that feature any number of interval collections and temporal predicates and return the k best results. We illustrate RTJ with the following example on network traffic monitoring. Consider the two interval collections C_1 and C_2 in Figure 3.1. Assume that each collection gathers requests from a different country. An analyst interested in monitoring traffic between two countries would seek (x, y) pairs, $x \in C_1$, $y \in C_2$, where x ends *just before* y starts. A Boolean semantics would compute 9 (x, y) pairs satisfying *before*(x, y), including pairs where x ends *long before* y starts. An appropriate scoring function and top-k semantics would select $\{(x_1, y_1), (x_2, y_2), (x_2, y_3)\}$, that best satisfy x ends *just before* y starts for $k = 3$.

The challenge we tackle is to devise an efficient query evaluation strategy that guarantees to return the best answers for a variety of temporal predicates. The first difficulty here is to develop a general-purpose algorithm that works with a variety of predicates and ranked semantics and yet, that is able to exploit the nature of those predicates to devise an efficient evaluation.

The efficient processing of interval joins has been studied before [28,36,39,68]. The closest to our work is the one by Chawda et al. [28] with a focus on processing interval joins on Map-Reduce [35]. However, proposed algorithms are not directly applicable in our case because they focus on a Boolean semantics. In our work, scores constitute both a challenge and an opportunity. They are a challenge because, unlike Boolean semantics, every combination of

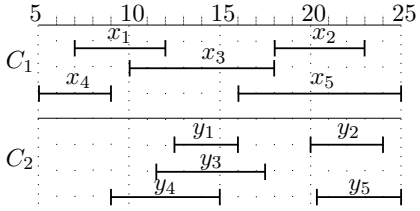


Figure 3.1: Motivating Example

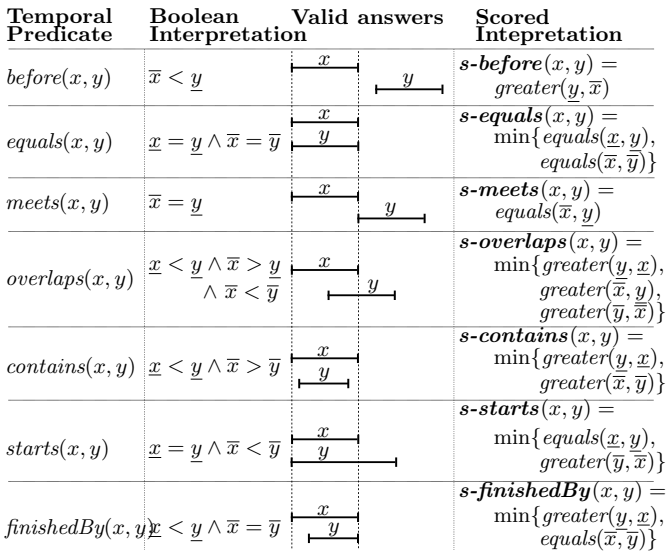


Figure 3.2: The Allen algebra with Boolean and scored temporal predicates.

intervals is potentially an answer. The opportunity lies in the ability to leverage statistics on input data in order to avoid computing low-scoring results. The second difficulty of this work is to design a top-k processing algorithm able to deal with the limitations of distributed processing frameworks.

The work presented in this section was part of Julien Pilourdault's Ph.D work, co-supervised with Sihem Amer-Yahia. It is detailed in the following publication:

- **Distributed Evaluation of Top-k Temporal Joins**
  Julien Pilourdault, Vincent Leroy and Sihem Amer-Yahia
  In Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2016, pages 1027–1039.

Section 3.1.2 describes our data model and defines the problem of evaluating RTJ queries. *TKIJ* is presented in Section 3.1.3. Experiments are detailed in Section 3.1.4. We conclude in Section 3.1.5.

3.1.2 Data model and problem definition

We are given m collections of intervals $C_1, \ldots, C_m$. Each interval x has a unique identifier, a start time $\underline{x}$ and an end time $\overline{x}$.

Boolean temporal predicates The general form of a temporal predicate between two intervals x and y is denoted $p(x, y)$ and is expressed as a Boolean conjunction of equalities and inequalities between their endpoints $\underline{x}, \overline{x}, \underline{y}, \overline{y}$. This allows to capture a wide range of predicates among which the seminal Allen algebra [5]. The first 3 columns of Figure 3.2 summarize Allen temporal predicates and their semantics. For example, $meets(x, y)$ imposes that y starts when x finishes while $starts(x, y)$ requires that x and y start at the same time and that x ends before y.

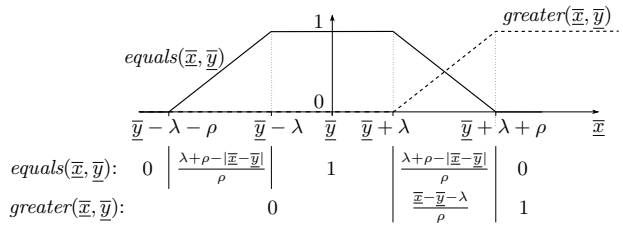
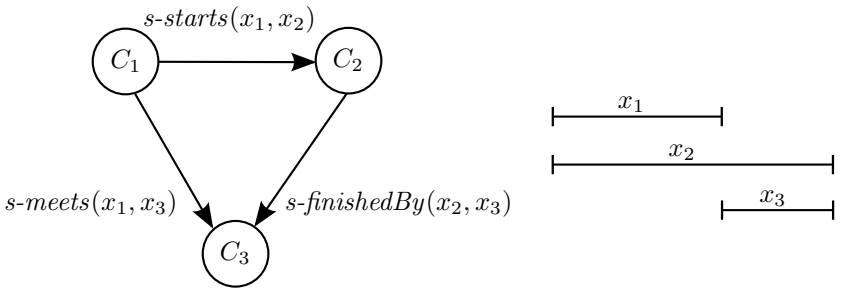


Figure 3.3: Approximating *equals* and *greater*. Here, $\overline{x} \in \{\underline{x}, \overline{x}\}$, $\overline{y} \in \{\underline{y}, \overline{y}\}$.



(a) Query graph (b) High-scoring result

Figure 3.4: Example of RTJ query

Scored temporal predicates Since we are interested in capturing the degree at which a temporal predicate is verified by a pair of intervals, we propose to associate a score to each predicate. Here again, we aim to be general and we adopt the flexible approach for scoring Allen predicates [38] and adapt it to our settings. This approach relies on two primitive approximation comparators on intervals' endpoints. Those comparators, $\mathit{equals}(\overline{x}, \overline{y})$ and $\mathit{greater}(\overline{x}, \overline{y})$, are used to express the degree of equality or inequality of intervals' endpoints $\overline{x}$ and $\overline{y}$, where $\overline{x} \in \{\underline{x}, \overline{x}\}$, $\overline{y} \in \{\underline{y}, \overline{y}\}$ as a graded value in $[0, 1]$. They rely on two parameters λ and ρ that provide flexibility in controlling the tolerance degree when comparing intervals' endpoints. Figure 3.3 shows how $\mathit{equals}(\overline{x}, \overline{y})$ and $\mathit{greater}(\overline{x}, \overline{y})$ are used with λ and ρ to express that tolerance. For instance, by defining that whenever $|\overline{x} - \overline{y}| \leq \lambda$, $\mathit{equals}(\overline{x}, \overline{y})$ returns 1, λ sets a tolerance for exact endpoint equality. ρ, on the other hand, is used to define score values. A large ρ value defines a wide range of score values and a small ρ produces a more abrupt curve and fewer possible score values.

Since temporal predicates are expressed as equalities and inequalities on intervals' endpoints, their approximation can be achieved using a conjunction of *equals*() and *greater*() with appropriate λ and ρ values. This allows us to *associate a scored variant to each temporal predicate*. We denote that variant $\mathit{s\text{-}p}(x, y)$ and refer to it as *scored temporal predicate*, abusing the term "predicate" to mean "function". Indeed, while $p(x, y)$ returns a Boolean value, $\mathit{s\text{-}p}(x, y)$, returns a score in [0,1]. For example, we can define the scored version of $\mathit{starts}(x, y)$ as $\mathit{s\text{-}starts}(x, y) = \min\{\mathit{equals}(\underline{x}, \underline{y}), \mathit{greater}(\overline{y}, \overline{x})\}$ (last column of Figure 3.2).

Temporal join queries We are interested in expressing n-ary join queries on interval collections $C_1, \ldots, C_m$. We express a query Q as a weakly connected oriented simple graph of the form (V, E). Each each vertex $v_i \in V$ is mapped to a collection C_i. Each edge $(i, j) \in E$ between two vertices v_i and v_j is labeled with a scored temporal predicate $\mathit{s\text{-}p}_{(i,j)}()$ between

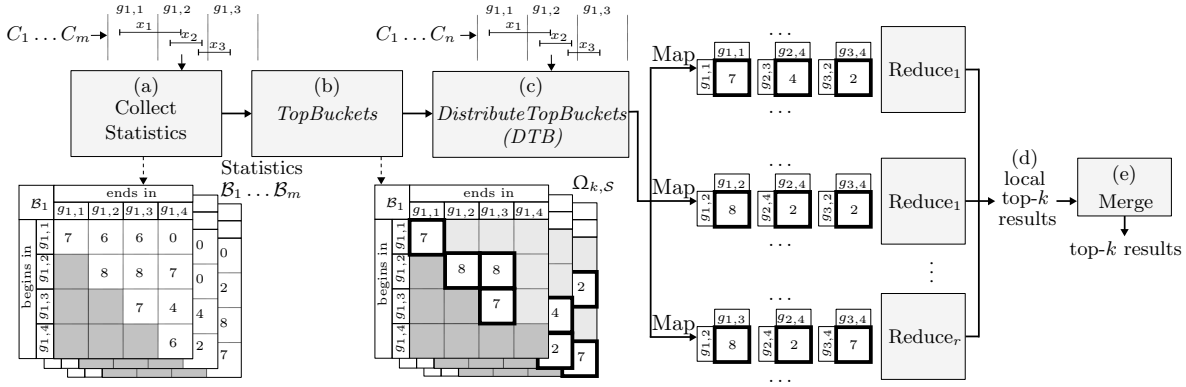


Figure 3.5: Overview of *TKIJ*

the two collections C_i and C_j corresponding to v_i and v_j. Figure 3.4 presents a sample query.

The evaluation of an n-ary join query Q returns a set of tuples of the form $(x_1, \ldots, x_n)$ where $x_i \in C_i$. The score of each tuple in the query result is computed using a function $\mathcal{S}$ that aggregates the partial scores assigned by each predicate $s\text{-}p_{(i,j)}()$ associated with each query edge $(i,j) \in E$. $\mathcal{S}$ could be any monotone function such as the weighted sum as it is commonly the case in ranked aggregation [41, 42, 54, 74].

For example, we can express a 3-way query that returns a tuple (x, y, z) where $x \in C_1$, $y \in C_2$ and $z \in C_3$ and the score of (x, y, z) is computed as an aggregation of its partial scores for query predicates $s\text{-}starts(x, y)$ and $s\text{-}meets(y, z)$.

Although we use the term "join" to refer to our queries, their expressivity goes beyond traditional relational joins. Our queries are not compositional in the sense of a relational join since their results are not intervals but tuples of any length (corresponding to the number of vertices in the query). Our queries can express any combination of interval collections with any scored predicates including chain queries and queries containing cycles.

Ranked Temporal Join (RTJ) problem Given an n-ary temporal join query $Q = (V, E)$ expressed over a set of collections $C_1, \ldots, C_m$ corresponding to query vertices in V and temporal predicates $s\text{-}p_{(i,j)}()$ associated to each edge $(i,j) \in E$, our problem is to find a top-k set of tuples of the form $(x_1, \ldots, x_n)$, $x_i \in C_i$, ranked by (descending) order of $\mathcal{S}_{(i,j)\in E}(s\text{-}p_{(i,j)}(x_i, x_j))$.

3.1.3 TKIJ algorithm

We present TKIJ, our approach for evaluating Top-K Interval Joins, that efficiently finds the set of k *best* results for an RTJ query Q. We first provide an overview of TKIJ, then we give each step in detail.

Overview

Figure 3.5 summarizes our approach, TKIJ. Given a set of interval collections $C_1 \ldots C_m$, TKIJ executes a query-independent pre-processing phase to collect statistics on intervals' distribution. This phase partitions time into granules and computes *buckets* for each collection

(a). A bucket associated to a collection C_i corresponds to a pair of granules, and contains the number of intervals of C_i starting at one granule and ending at another. Given a query Q, these statistics are used to evaluate *bucket combinations* that should be processed in order to obtain top-k results (b). TKIJ relies on a constraint programming solver to compute score bounds for each bucket combination and uses those bounds to prune combinations that do not contain top-k results. The third phase is the actual join processing which relies on two Map-Reduce jobs. The first job assigns a subset of buckets to each reducer r_j (c) which then processes locally the RTJ query, returning local top-k results (d). This assignment aims at reducing data replication to limit I/O, and leverages score bounds to distribute high-scoring results evenly so that each reducer can quickly prune low-ranking results. The second Map-Reduce job merges all local results into a single query output (e).

Statistics collection

TKIJ pre-processes each dataset once in order to collect statistics which are then used to optimize the execution of any RTJ query on this dataset. These statistics maintain a matrix $\mathcal{B}_i$ representing the distribution of endpoints of intervals in each collection C_i. TKIJ partitions the time range of each C_i into a set of contiguous *granules*. We adopt a uniform partitioning which has been shown to be appropriate for temporal joins [28, 36, 44].

As illustrated in Figure 3.5a, each matrix entry records the cardinality of a bucket, where a bucket $b_{i,l,l'} = (g_{i,l}, g_{i,l'})$ contains all intervals of C_i that start in $g_{i,l}$ and end in $g_{i,l'}$:

$$\mathcal{B}_i[l][l'] = |b_{i,l,l'}| = |\{x \in C_i, \underline{x} \in g_{i,l} \wedge \overline{x} \in g_{i,l'}\}|$$

As an example, given $g_{1,1} = [10, 20]$ and $g_{1,2} = [20, 30]$, the matrix entry for $b_{1,1,2} = (g_{1,1}, g_{1,2})$ indicates 6 intervals starting in $[10, 20]$ and ending in $[20, 30]$.

Range partitioning is a common approach in temporal join processing [28, 36, 44, 68]. The rationale is that intervals having similar endpoints are likely to satisfy similar join predicates. For example, most previous studies that focus on *intersection* joins leverage partitions to avoid pairs of intervals that are guaranteed not to intersect. Similarly, TKIJ relies on these statistics to obtain information on the distribution of intervals within buckets and prune the search space of *any* RTJ query.

Statistics are computed in a single Map-Reduce phase. Each mapper reads a fraction of the data and maintains a local matrix per collection. Matrices are then aggregated in the reduce phase, and the reducer responsible for collection C_i outputs a final matrix $\mathcal{B}_i$.

Selection of bucket combinations

We now describe how TKIJ uses pre-computed statistics to estimate score bounds on candidate results. Then, we present how score bounds are used to avoid computing unnecessary results while we guarantee to return the exact top-k results.This constitutes a preliminary pruning step of the top-k processing: using a summary of the dataset, the idea is to identify part of the results space that are promising and should be further explored.

Estimating score bounds Processing an RTJ query Q requires to return the top-k tuples $(x_1, \ldots, x_n)$, $x_i \in C_i$ according to a scoring function $\mathcal{S}$. Since any tuple $(x_1, \ldots, x_n)$ is a potential answer, we investigate how to reduce the amount of data processed using scores. We use $\omega = (b_{1,l_1,l'_1}, \ldots, b_{n,l_n,l'_n})$ to denote a bucket combination, $\omega.nbRes = \prod_{i=1}^{n} |b_{i,l_i,l'_i}|$ the

total number of results that can be obtained from a bucket combination ω, and Ω the set of all combinations.We define score upper and lower bounds in each ω, denoted $\omega.UB$ and $\omega.LB$.

Definition 1. *The score upper-bound (resp. lower-bound) $\omega.UB$ (resp. $\omega.LB$) of a bucket combination $\omega = (b_{1,l_1,l'_1}, \ldots, b_{n,l_n,l'_n})$ is the upper-bound (resp. lower-bound) of $\mathcal{S}_{(i,j)\in E}(s\text{-}p_{(i,j)}(x_i, x_j))$ where $\underline{x_i} \in g_{i,l_i}, \overline{x_i} \in g_{i,l'_i}, \forall i \in 1 \ldots n$.*

As an example, suppose that query Q features a predicate $s\text{-}meets_{(1,2)}(x, y)$ where $x \in C_1$ and $y \in C_2$, using scoring parameters $(\lambda_{equals}, \rho_{equals}) = (4, 8)$. Collected statistics show 6 intervals in bucket $b_{1,1,2} = ([10, 20], [20, 30])$ for C_1 and 7 intervals in bucket $b_{2,2,3} = ([20, 30], [30, 40])$ for C_2. We build the bucket combination $\omega = (b_{1,1,2}, b_{2,2,3})$. Then, we can derive bounds on the score $\mathcal{S}(x, y) = s\text{-}meets(x, y)$ of a result $(x, y) \in \omega$. The maximum possible score is 1 (e.g. with $(x, y) = ([12, 25], [25, 35])$), and the minimum score is 0.25 (with $(x, y) = ([15, 20], [30, 35])$). Hence, $\omega.UB = 1, \omega.LB = 0.25$. Thus, 42 results in ω have a score in $[0.25, 1]$.

TKIJ relies on a constraint programming solver as a generic approach to compute score bounds for any combination of predicates.

Pruning bucket combinations TKIJ leverages computed score bounds to reduce computation cost by eliminating results that are guaranteed not to be in the top-k. To do so, it computes $\Omega_{k,\mathcal{S}} \subseteq \Omega$, a subset of the search space that is sufficient to guarantee correctness. We define $\Omega_{k,\mathcal{S}}$ as follows:

Definition 2. *The set of Top Buckets $\Omega_{k,\mathcal{S}}$ is a subset of Ω satisfying the following conditions:*

- $\forall \omega \in \Omega \setminus \Omega_{k,\mathcal{S}}\ \exists \Psi \subseteq \Omega_{k,\mathcal{S}}$:
  - $\forall \omega' \in \Psi\ \omega'.LB \geq \omega.UB$
  - $\sum_{\omega' \in \Psi} \omega'.nbRes \geq k$

This definition ensures that whenever a bucket combination ω is pruned, there are at least k results from $\Omega_{k,\mathcal{S}}$ with a score higher than results generated from ω. Pruning unnecessary results is a two-step process, coined *TopBuckets*. A first step computes score bounds for bucket combinations using a solver. Then, a second step uses those bounds to eliminate unnecessary results. In our setting, all $(x_1, \ldots, x_n)$ combinations are potential answers, and we cannot employ traditional top-k techniques to prune the search space. *TopBuckets* addresses this challenge using pre-computed statistics to locate high-scoring answers.

Distributed Top-k Join Processing

The *TopBuckets* process generates $\Omega_{k,\mathcal{S}}$, a set of bucket combinations that are sufficient to accurately compute the top-k results. We now describe how TKIJ computes the top-k results (Steps (c)-(d)-(e) in Figure 3.5). We implement TKIJ on Map-Reduce [35]. Given a set of r reducers, TKIJ assigns each bucket combination $\omega \in \Omega_{k,\mathcal{S}}$ to a single reducer $r_j, j \in 1 \ldots r$, that processes results in ω. The main challenge in distributed join processing is to devise an efficient workload assignment function. When performing large-scale joins, I/O often constitutes a major bottleneck. We first review existing assignment algorithms, then we consider the specifics of distributed top-k computation and show that it is essential to take scores into account when dividing the workload. Indeed, during top-k join processing, each executor

leverages high scoring results to prune candidates on-the-fly. We present *DistributeTopBuckets*, a novel function that focuses on assigning high-scoring results to each reducer, while minimizing I/O cost as a secondary objective. Finally, we present how an RTJ query is processed using appropriate Map-Reduce algorithms.

Existing I/O optimizations When different reducers require the same chunk of data, this data is replicated in the shuffle phase of Map-Reduce, which increases input cost. Several distributed join algorithms, such as *RCCIS* [28] and the work of Afrati et al. [1] specifically aim at reducing that cost. In TKIJ, this corresponds to different reducers being assigned bucket combinations involving the same bucket. Other approaches focus on assigning a balanced load to each reducer [28, 75]. This ensures that the number of results generated by each reducer is comparable, so that no reducer will have a larger workload in output dominated tasks. Finally, some algorithms optimize both input and output costs simultaneously [100]. All these approaches are not directly applicable to our settings. They achieve optimizations for specific queries (equi-join [1], 2-way θ-join [75], m-way θ-join [100]). One close related work to ours [28] reduces I/O cost by leveraging the Boolean interpretation of Allen predicates. That is not directly applicable to scored predicates.

Top-k optimizations TKIJ significantly differs from standard Map-Reduce-based join processes due to its ranked semantics. In TKIJ, each reducer processes a full RTJ query locally using the bucket combinations it receives (Figure 3.5d). Hence, it is important to ensure that each reducer quickly identifies high-scoring results as it is usually the case in top-k processing [40, 41, 42, 85]. Therefore, the assignment of bucket combinations to reducers favors an even distribution of high-scoring results to ensure that each executor can execute on-the-fly pruning efficiently.

DTB algorithm TKIJ relies on the *DistributeTopBuckets* process to assign bucket combinations from $\Omega_{k,\mathcal{S}}$ to reducers. Following the principles described above, *DTB* increases the probability that each reducer receives a fair share of high-scoring results. This step relies on the knowledge, for each bucket combination, of the number of results generated, as well as their score bounds computed during the pruning step by the solver. *DTB* first sorts $\Omega_{k,\mathcal{S}}$ by descending order of score upper-bound to access them according to their likelihood of generating high-scoring results. It then assigns each bucket combination to the least loaded reducer. Furthermore, *DTB* opportunistically optimizes I/O cost by giving priority to reducers that already process some of the buckets of the combination considered while limiting imbalanced assignments.

Join Processing The final phase of TKIJ first runs *DTB* using $\Omega_{k,\mathcal{S}}$ to determine data distribution among reducers. Then, a Map-Reduce phase processes the RTJ query locally. For each input interval x, a mapper computes the bucket b_{i_x,l_x,l'_x} in which x falls. Then, x is communicated to all reducers r_j that received b_{i_x,l_x,l'_x}. That way, each reducer r_j receives its share of input data, and a list of bucket combinations $\Omega_{r_j} \subseteq \Omega_{k,\mathcal{S}}$ whose results are potential top-k candidates. Once each reducer has processed locally the RTJ query, we run an additional Map-Reduce phase (Step (e) in Figure 3.5), that merges local results and returns the final top-k answers.

Id	Scored Temporal Predicates In $\mathcal{Q}$. $x_i \in C_i \ \forall i \in 1 \dots n$.
$\mathcal{Q}_{b,b}$	$s\text{-}before(x_1, x_2)$, $s\text{-}before(x_2, x_3)$
$\mathcal{Q}_{f,f}$	$s\text{-}finishedBy(x_1, x_2)$, $s\text{-}finishedBy(x_2, x_3)$
$\mathcal{Q}_{o,o}$	$s\text{-}overlaps(x_1, x_2)$, $s\text{-}overlaps(x_2, x_3)$
$\mathcal{Q}_{s,f,m}$	$s\text{-}starts(x_1, x_2)$, $s\text{-}finishedBy(x_2, x_3)$, $s\text{-}meets(x_1, x_3)$
$\mathcal{Q}_{s,s}$	$s\text{-}starts(x_1, x_2)$, $s\text{-}starts(x_2, x_3)$

Table 3.1: Queries

3.1.4 Evaluation

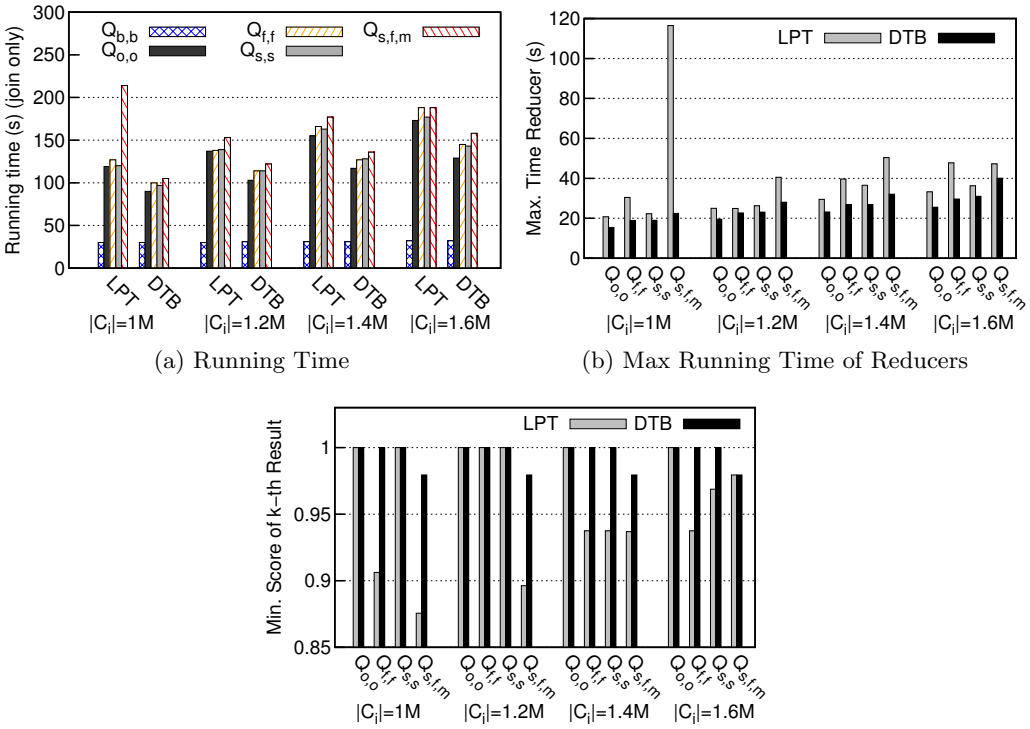
We evaluate TKIJ on a 8-node cluster with 6 executors having 8 cores each. To vary dataset parameters, we generate synthetic data following the approach of previous work [28]. We use a pseudo-random uniform generator to get intervals' startpoints and lengths in specified ranges (respectively $s = [0, 10^5]$ and $w = [1, 100]$). Intervals' endpoints are integers. We vary the number $|C_i|$ of intervals per collection, and the number n of collections. For our evaluation, we define varied queries described in Tables 3.1. We use $\mathcal{S} = \frac{\sum_{(i,j)\in E} s\text{-}p_{(i,j)}(x_i,x_j)}{|E|}$ to compute the score of a query result $(x_1, \dots, x_n)$.

In this thesis, I focus on the workload distribution aspect of TKIJ. The challenge in executing top-k queries over Map-Reduce is to preserve the top-k pruning efficiency and load balancing in spite of the lack of communications between executors. We compare two workload distribution algorithms and evaluate their impact on TKIJ:

- *DTB*: This is the approach described in Section 3.1.3. *DistributeTopBuckets* ensures that each executor gets high scoring results in order to process its local top-k efficiently and benefit from early pruning decisions. In addition, *DistributeTopBuckets* opportunistically optimizes for I/O.

- *LPT*: The *LPT* (*Longest Processing Time*) heuristic aims to minimize scheduling time on parallel machines [37]. *LPT* executes tasks in descending order of processing time. In our context, a naive approach would minimize the maximum number of candidate join results that a reducer has to process, so as to reduce the duration of the longest task. We sort the set of bucket combinations by descending order of number of results ($\omega.nbRes$) and assign each one to the least loaded reducer.

Figure 3.6a presents the running time of the join phase on all queries, where $|C_i|$ varies from 1M to 1.6M and $k = 1000$. On $\mathcal{Q}_{b,b}$, running time is identical for *LPT* and *DTB*, since a single bucket combination is selected in $\Omega_{k,\mathcal{S}}$ and a large number of results with maximum score can be quickly found during the join phase. On other queries, *DTB* outperforms *LPT* for two reasons. Firstly, *LPT* incurs a higher shuffle cost (on average 43% higher). When assigning a bucket combination to a reducer, *DTB* favors assignments that lessen shuffle cost. *LPT* favors the assignment of bucket combinations with a large number of results to the least loaded reducers. Hence, buckets have a higher probability to be sent to several reducers with *LPT* than with *DTB*. Secondly, *LPT* does not necessarily give a fair share of high-scoring results to each reducer. Figure 3.6b shows the running time of the longest reducer task (we omit $\mathcal{Q}_{b,b}$ where *LPT* and *DTB* perform equally for the reason exposed above). *DTB* always outperforms *LPT* because it increases the probability that all reduce tasks terminate early since they can all find high-scoring results. This difference is exacerbated on query $\mathcal{Q}_{s,f,m}$



(a) Running Time

(b) Max Running Time of Reducers

(c) Minimum Score of k-th Result

Figure 3.6: Synthetic Data - Workload Distribution

with $|C_i| = 1M$. Here, the few results that satisfy best all 3 predicates featured in $\mathcal{Q}_{s,f,m}$ are better distributed using DTB. On Figure 3.6c, we represent the minimum score of the k^{th} result among the results returned by reducers. These results support our observation: the score of returned results is higher when distribution is defined using DTB, while unnecessary results with lower scores are returned with LPT.

3.1.5 Conclusion

Temporal data contains contains valuable that can be explored to discover relations between temporal events. We introduce Ranked Temporal Joins as a query model for extracting relevant interval tuples from collections. RTJs generalizes Allen temporal Boolean predicates by introducing scores that can be parametrized by the analyst to better express the relations they seek.

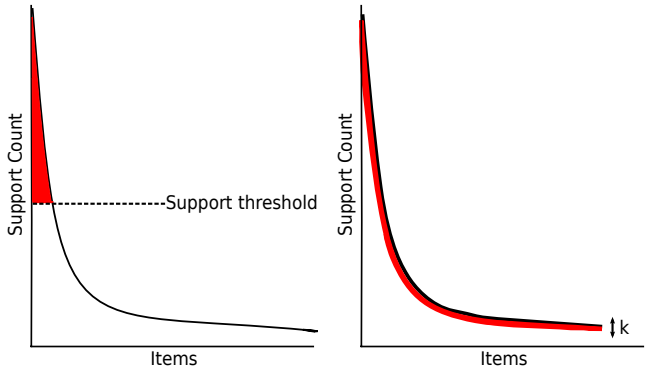
We propose TKIJ, a 3-stage query evaluation approach for RTJ queries that identifies the top-k tuples scoring the highest. TKIJ performs the following steps:

1. Compute query-independent statistics about the dataset. Each collection of intervals is represented as a matrix that indicates the distribution of interval endpoints.

2. Execute a preliminary pruning using the properties of the query and the statistics. Using a solver, TKIJ determines which portions of the dataset contain promising results, and which ones can be safely ignored. This eliminates a large fraction of the search space.

3. Process the top-k queries locally on executors. At this step, each executor performs on-the-fly pruning using its local top-k results. TKIJ ensures that each executor is assigned high scoring result candidates that can be discovered early in the execution in order to improve the pruning performance.

TKIJ was evaluated on synthetic datasets, as well as a real dataset containing a network traffic log [79] provided by an industrial partner. In the future, we aim at performing RTJs on streams of data. A preliminary analysis of this problem can be found in Julien Pilourdault's Ph.D thesis [78].



(a) Frequent Itemset Mining (b) Our approach, TopPI

Figure 3.7: Schematic distinction between frequent itemsets mining and TopPI. The area in red represent each method's output.

3.2 Pattern mining for long-tailed datasets

3.2.1 Context

Pattern mining algorithms have been applied successfully on various datasets to extract frequent itemsets and uncover hidden associations [3, 66]. Frequent itemsets mining (FIM) was popularized by the famous "beer and diapers" association [3] extracted from a retail dataset. 20 years after this publication, FIM is still very appreciated by marketing analysts, as it gives them access a product's sales trends and associations with other products. This allows managers to obtain feedback on customer behavior and to propose relevant product bundles. In the context of a research project, we collaborated with the french retailer Intermarché with the goal of modernizing FIM to scale to massive datasets: 2 years of sales recorded all over France and totaling 290M receipts.

Scaling FIM initially appears as a distributed systems problem that can simply be solved by porting the current state-of-the-art algorithm to a parallel processing framework such as Map-Reduce [35]. However, when executing FIM on large-scale dataset, a fundamental problem appears, and its cause is the definition of FIM. FIM is designed to discover itemsets in order of decreasing support (i.e. frequency), whether it is using a minimum support threshold, or a top-k definition. The worst-case complexity of FIM is exponential in the number of items in the dataset, so analysts use high support threshold to control execution time, thus restricting the mining to the globally most frequent itemsets. Many large datasets today exhibit a long-tailed distribution, characterized by the presence of a majority of infrequent items [48]. Mining at high thresholds eliminates low-frequency items, thus ignoring most of the data (Figure 3.7a). In the context of retail, this means that standard FIM generates hundreds of millions of itemsets related to the most frequent products (i.e. top 5% products sold), and no result for all other products (95%). Online stores such as Amazon have demonstrated that a significant proportion of revenue originates from the long-tail, so it is important to ensure that FIM can produce results for all items.

Our first contribution to address these issues is item-centric mining, a new semantics for FIM that is more appropriate for mining large-scale long-tailed datasets. Item-centric mining solves the problem of items coverage by ensuring that each item is described by at least k

itemsets (Figure 3.7b). Table 3.2 contains examples of item-centric itemsets we generated from a retail dataset, along with their frequencies. Each item is associated to its 3 most frequent itemsets, whether it sells a lot (e.g. milk) or not (e.g. wasabi). With this new definition, our mining problem becomes an exploration of itemsets in which the goal is to fill the list of top-k itemsets of each item. Our second contribution is TopPI, an algorithm that performs item-centric mining efficiently. While TopPI leverages lessons learned from previous FIM algorithms, we show that our new definition significantly alters the properties used by traditional FIM, and requires a different pruning strategy. We design both parallel and distributed version of TopPI, ensuring the completeness of the results while minimizing redundant computations.

The work presented in this section was part of Martin Kirchgessner's Ph.D work, co-supervised with Sihem Amer-Yahia. It is detailed in the following publication:

- **TopPI: An efficient algorithm for item-centric mining**
  Vincent Leroy, Martin Kirchgessner, Alexandre Termier and Sihem Amer-Yahia
  Information Systems (IS), 2017, 64: 104-118.

In this thesis, I focus on the distributed version of TopPI and omit the details of the pruning strategy. The main challenges in scaling TopPI is preserving the efficiency of traditional top-k search-space pruning, while dealing with the limitations of distributed processing environment that do not allow communications between executors during a mining stage.

Section 3.2.2 presents the data model and the definition of item-centric mining. Section 3.2.3 gives an overview of TopPI's algorithm in its centralized version. Section 3.2.4 details the distributed version of TopPI. Finally, Section 3.2.5 provides some experimental results.

3.2.2 Item-centric mining

The data contains *items* drawn from a set $\mathcal{I}$. Each item has an integer identifier, referred to as an index, which provides an order on $\mathcal{I}$. A *dataset* $\mathcal{D}$ is a collection of *transactions*, denoted $\{T_1, ..., T_n\}$, where $T_i \subseteq \mathcal{I}$. An *itemset* P is a subset of $\mathcal{I}$. A transaction T is an *occurrence* of P if $P \subseteq T$. Given a dataset $\mathcal{D}$, the *projected dataset* for an itemset P is the dataset $\mathcal{D}$ restricted to the occurrences of P: $\mathcal{D}[P] = \{T \mid T \in \mathcal{D} \wedge P \subseteq T\}$. In the example dataset shown in Table 3.3a, $\mathcal{D}[\{0,1\}] = \{T_0, T_1, T_2\}$. To further reduce its size, all items of P can be removed, giving the *reduced dataset* of P: $\mathcal{D}_P = \{T \setminus P \mid T \in \mathcal{D}[P]\}$. Hence, in the example, $\mathcal{D}_{\{0,1\}} = \{\{2\}, \{2\}, \{\}\}$.

The number of occurrences of an itemset in $\mathcal{D}$ is called its *support* and denoted $support_{\mathcal{D}}(P)$. More formally, $support_{\mathcal{D}}(P) = |\mathcal{D}_P|$. For example, in the dataset shown in Table 3.3a, the

item (support)	top-3 (itemset,support) pairs
milk (682,288)	({milk, grated cheese}, 40,890) ({milk, cola}, 40,846) ({milk, carrier bag}, 40,675)
wasabi (2132)	({wasabi, nori seaweed},352) ({wasabi, sushi rice},244) ({wasabi, nori seaweed, sushi rice},163)

Table 3.2: TopPI results for $k = 3$ on a retail dataset.

TID	Transaction
T_0	$\{0, 1, 2\}$
T_1	$\{0, 1, 2\}$
T_2	$\{0, 1\}$
T_3	$\{2, 3\}$
T_4	$\{0, 3\}$

(a) Input $\mathcal{D}$

item	$top(i)$: P, $support(P)$	
i	1^{st}	2^{nd}
0	$\{0\}, 4$	$\{0, 1\}, 3$
1	$\{0, 1\}, 3$	$\{0, 1, 2\}, 2$
2	$\{2\}, 3$	$\{0, 1, 2\}, 2$
3	$\{3\}, 2$	

(b) TopPI results for $k = 2$

Table 3.3: Sample dataset

itemset $\{1, 2\}$ has a support equal to 2. To avoid redundant information, when a superset of items also has the same support, we do not output this itemset. For instance, {1,2} is replaced by {0,1,2} since they both have a support of 2. The largest superset having the same support is called the *closure* of the itemset.

Mining algorithms generally target the most frequent itemsets. As explained in Section 3.2.1, this leads in the case of long-tailed datasets to extracting millions of itemsets only related to the most frequent items. In this work, we propose a new mining objective called item-centric mining: given a dataset $\mathcal{D}$ and an integer k, our goal is to return, for each item in $\mathcal{D}$, the k most frequent closed itemsets containing this item. Table 3.3b shows the solution to this problem applied to the dataset in Table 3.3a with $k = 2$. This new definition of relevant patterns avoids the problem of traditional approaches by ensuring coverage, as each item is described in the results by k itemsets.

3.2.3 TopPI algorithm

In this section, we present an overview of TopPI, an algorithm that solves the problem of finding the top-k itemsets of each item. We first describes how TopPI recursively enumerates itemsets, an approach inherited from state-of-the-art FIM algorithms. Then we present how TopPI prunes the search-space of itemsets to efficiently perform item-centric mining. A more detailed description of TopPI and its algorithm is given in [64]. We address the problem of scaling TopPI in Section 3.2.4.

Tree-shaped recursion

Several algorithms aim at mining itemsets present in a dataset [50, 76, 92]. For efficiency reasons, TopPI borrows some principles developed for the LCM algorithm [93] and enumerates itemsets by recursively adding items to a previous itemsets. In this context, pruning the solution space means avoiding recursions. In Table 3.3a, $\{0, 1, 2\}$ is an extension of both $\{0, 1\}$ and $\{1, 2\}$. To avoid redundant computation, extensions are restricted to items smaller than those already contained in the itemset. An additional "first-parent" criterion ensures that even in the case of a closure, each itemset is only enumerated once.

The extension enumeration order shapes the extensions lattice as a tree. Figure 3.8 shows the itemsets tree for the dataset in Table 3.3a. $\langle\{2\}, 1\rangle$ is the first parent of $\{0, 1, 2\}$, but $\langle\{2\}, 0\rangle$ is not. Therefore the branch produced by $\langle\{2\}, 0\rangle$ is pruned. These enumeration principles lead to the following property: by extending P with e, TopPI can only recursively generate itemsets Q such that $max(Q \setminus P) = e$.

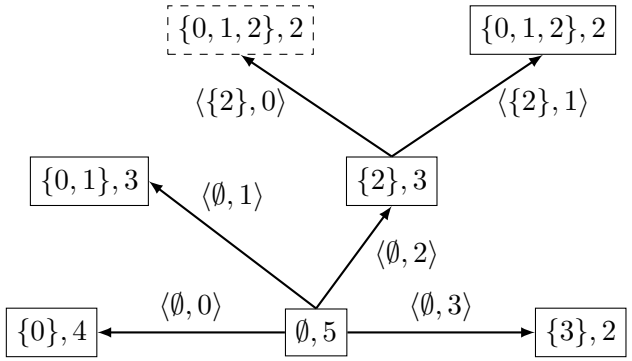


Figure 3.8: CIS enumeration tree on our example dataset (Table 3.3a). $\langle P, i \rangle$ denotes extension to closed ItemSest operations.

Pruning strategy

The core recursive structure provides TopPI with the basic ability to enumerate itemsets. The number of possible itemsets in the tree is $2^{|\mathcal{I}|}$. What makes FIM possible in practice is the ability to prune this enumeration tree to focus on itemsets that are useful to the analyst. In the case of item-centric mining, we only target $k \times |\mathcal{I}|$ of the $2^{|\mathcal{I}|}$ itemsets. Hence, the efficiency of TopPI relies on a specifically designed pruning strategy.

TopPI maintains, for each item $i \in \mathcal{I}$, $top(i)$, a heap that stores the most frequent itemsets containing i enumerated so far. Top-k algorithms have been studied for years in the database community [41]. The lowest support value in the top-k results constitutes an entrance threshold to be a result candidate. The goal is then to obtain an upper-bound on the score of results of fractions (branches in our case) of the results space and eliminate them when this upper bound is lower than the entrance threshold. The specificity of TopPI is that it computes $|\mathcal{I}|$ top-k simultaneously, which makes pruning the tree of itemsets significantly more complex than the one of traditional FIM algorithm. We now give an overview of the main features of TopPI:

- TopPI aims at mining very large datasets while preserving their long-tailed distribution. FIM generally handles these cases by raising the frequency threshold ε, which allows the elimination of a large, infrequent, fraction of the data, to increase performance. With TopPI, we want to be able to output itemsets appearing with very low frequency in the dataset, so we start at $\varepsilon = 2$. To this end, TopPI incorporates a *dynamic threshold adjustment* to improve performance without eliminating relevant data. In each branch of the enumeration tree, TopPI adjusts the threshold to the lowest bound of the top-k itemsets of each item that can be recursively produced in the branch.

- Standard FIM algorithms rely on a simple pruning strategy to drastically reduce the number of itemsets enumerated. Algorithms targeting the most frequent itemsets, using a threshold or a global top-k, directly rely on the anti-monotony property of itemsets' support [3]. Given two itemsets P and Q, $P \subset Q$, if P is not a valid result, then Q isn't one either, since $support_{\mathcal{D}}(P) \geq support_{\mathcal{D}}(Q)$. Hence, reaching an itemset whose support is too low to be a valid result means the early termination of a recursion, with the pruning of the sub-tree. This is, however, not applicable for TopPI: it is possible for the itemset $\{0, 1, 2\}$ to be in the top-k of the item 0 while $\{1, 2\}$ is not in the top-k

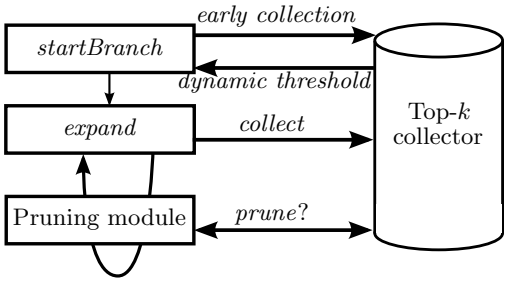


Figure 3.9: Overview of TopPI

of 1 or 2. This makes the early termination of the enumeration of a branch harder to decide. TopPI introduces a new *pruning heuristic* to tackle this issue.

- Efficient top-k processing generally relies on the early discovery of high ranking results. This allows, using heuristics, the pruning of large fractions of candidates without having to perform their exact computation. TopPI boosts the efficiency of its pruning algorithm by introducing the *early collection* of promising results, and optimizing the *order of itemsets enumeration*.

The key components of TopPI are depicted in Figure 3.9. TopPI executes the *startBranch* function for each item and recursively calls *expand* to enumerate itemsets. In both these functions, TopPI takes advantage of an indexing of items by frequency to optimize the *order of itemsets enumeration*. The *top-k collector* stores the results of TopPI by maintaining for each item the current version of its top-k itemsets. TopPI transmits itemsets to the collector using *collect* in *expand*, but also in *startBranch* with an *early collection* of *partial itemsets*. Using the current status of the top-k of each item, the collector provides a *dynamic support threshold* at the beginning of each branch that allows TopPI to heavily compress the dataset without losing any potential result. The state of the collector is also used in *expand* through the *prune* function to determine whether a recursive execution may produce itemsets part of the top-k of an item.

3.2.4 Scaling TopPI

We first present the multithreaded version of TopPI, designed to take full advantage of the multi-core CPUs available on servers. Then, to scale beyond the capacity of a single server, we present a distributed version of TopPI designed for Map-Reduce [35]. The goal is to divide the mining process into independent subtasks executed on executors while *(i)* ensuring the output completeness, *(ii)* avoiding redundant computation, and *(iii)* maintaining pruning performance.

Shared-memory TopPI

The enumeration of itemsets by TopPI follows a tree structure, described in Section 3.2.3. As shown by Négrevergne et al. [72], such enumeration can be adapted to shared-memory parallel systems by dispatching branches of the tree (i.e. *startBranch* invocations) to different threads. This policy ensures that reduced datasets materialized in memory by a thread are accessed by the same thread, which improves memory locality in NUMA architectures and makes better

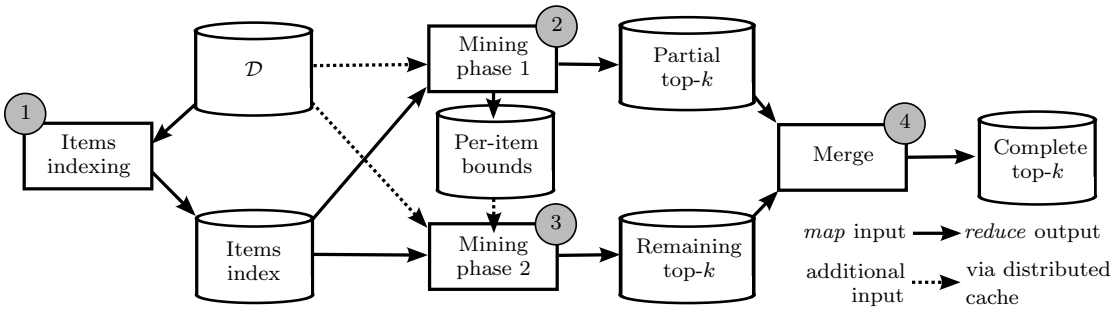


Figure 3.10: Hadoop implementation of TopPI

use of CPU caches. Some branches of the enumeration can generate much more itemsets than others. Towards the end of the execution, when a thread finishes the exploration of a branch and no new branch needs to be explored, TopPI relies on a work stealing policy to split the recursions of *expand* over multiple threads. Threads take into account the NUMA topology to prioritize stealing from other threads that share caches or are on the same socket. The top-k collector is shared between threads using fine-grained locks for the top-k of each item. Hence, the behavior of TopPI when using top-k entrance thresholds for pruning is similar to the sequential implementation, as accurate bounds can be accessed at any time.

Distributed TopPI

When executing TopPI on a cluster of executors, each executor runs one instance of the multi-threaded version of TopPI described above. We consider the case of a distributed processing framework implementing the Map-Reduce primitives. We first describe how the enumeration of itemsets is split between the executors, and then describe a two-stage mining approach designed to preserve the efficiency of pruning for distributed top-k processing. Figure 3.10 gives an overview of our solution.

Partitioning the itemsets enumeration In a distributed setting, enumeration branches are dispatched among executors. Each executor is assigned a partition of items $G \subseteq \mathcal{I}$, and restricts its exploration to the branches starting with an element of G. Following the example of Table 3.3, a executor that is assigned the partition $G = \{0, 2\}$ outputs the itemsets $\{0\}$,$\{2\}$ and $\{0, 1, 2\}$. TopPI's itemset extensions follow a strictly decreasing item order, so an executor generates itemsets P such that $max(P) \in G$. Thus, all executors generate different itemsets, without overlap nor need for a synchronization among them. This partitioning ensures that itemsets are only generated once, so that no processing time is wasted executing redundant operations.

Given the restrictions on the enumeration tree, an executor only requires the transactions of $\mathcal{D}$ containing items of its partition G. Executors must agree on the ordering of items as it determines valid extensions of itemsets. Should two tasks obtain a different assignment of items identifiers, several itemsets would be generated multiple times, and others would be lacking in the output. Consequently, indexing items by decreasing frequency is performed jointly by all executors on the original dataset $\mathcal{D}$ (Figure 3.10 ①). Once the items have been sorted by frequency and indexed, they are assigned to groups in a round robin fashion. This

Partition	Partial top-k (phase 1)	Bounds	Comp. top-k (phase 2)
G_0	$top(0) \to \{0\}, 4; \{0,1,2\}, 2$	$0 \to 2$	$top(1) \to \{0,1,2\}, 2$
$\{0,2\}$	$top(2) \to \{2\}, 3; \{0,1,2\}, 2$	$2 \to 2$	$top(3) \to \emptyset$
G_1	$top(1) \to \{0,1\}, 3$	$1 \to 0$	$top(0) \to \{0,1\}, 3$
$\{1,3\}$	$top(3) \to \{3\}, 2$	$3 \to 0$	$top(2) \to \emptyset$

Table 3.4: 2-phase mining over the sample database (Table 3.3a) with 2 workers, $k = 2$.

ensures that the most frequent items, which are more costly during the mining stage, are assigned to different groups. This balances the load of executors.

Two-stage mining The partitioning of the enumeration tree introduces the drawback that the top-k itemsets of an item may be generated by any executor, without the possibility of predicting which ones. A naive solution is for each executor to compute a local top-k for all items, and then merge all local top-k into the exact top-k. This is causes each executor to maintains a local top-k for each item, instead of a globally shared structure in the case of the shared-memory version. Since executors cannot communicate during the execution of a processing stage, each executor significantly underestimates the entrance thresholds for the global top-k of the item as it only benefits from results produced locally. Consequently, TopPI ends up enumerating up to $k \times |\mathcal{I}|$ itemsets *per executor* instead of $k \times |\mathcal{I}|$ *overall*, significantly limiting the scalability.

Instead, we rely on the following idea: given an item $i \in G$, the executor responsible for G collects a partial version of $top(i)$ close to the complete one. Indeed, the branch of the itemset tree rooted at i contains itemsets that combine i with smaller items of the dataset (*i.e.* more frequent items, thanks the pre-processing). Even though these may not all be in the actual top-k itemsets of i, they are likely to have high support. Consequently, we run distributed TopPI as a two-stage mining process. In the first stage (Figure 3.10 ②), the executor only collects itemsets for items $i \in G$. This step outputs a first partial version of each item's top-k, as well as a lower bound on the support of their complete top-k. After this first stage, these bounds are broadcasted to all executors. In the second stage (Figure 3.10 ③), the executor only collects itemsets for items $i \notin G$ to generate the complement top-k. A final Map-Reduce job is executed to merge the partial top-k and the complement top-k (Figure 3.10 ④). We illustrate this process with the example in Table 3.4.

Overall, this two-stage process is scalable. The first stage of mining completely splits the enumeration and the collection among groups, without any impact on the accuracy of pruning since the top-k of each item is maintained by a single executor. If we compare it to the naive version, we can see that the second stage is the exact complement of stage one, to overall achieve the same goal. Even though the second stage apparently suffers from the same problem of underestimating entrance thresholds as the naive version, the bounds generated at stage one ensure that pruning remains efficient. This second mining stage is extremely short and accurately targeted to simply complete the results of stage one. We confirm the overall scalability of two-stage TopPI in Section 3.2.5.

3.2.5 Evaluation

We evaluate TopPI on a retail dataset provided by Intermarché in the early stages of our collaboration. It consists of receipt data collected over 27 months in 87 supermarkets. Overall,

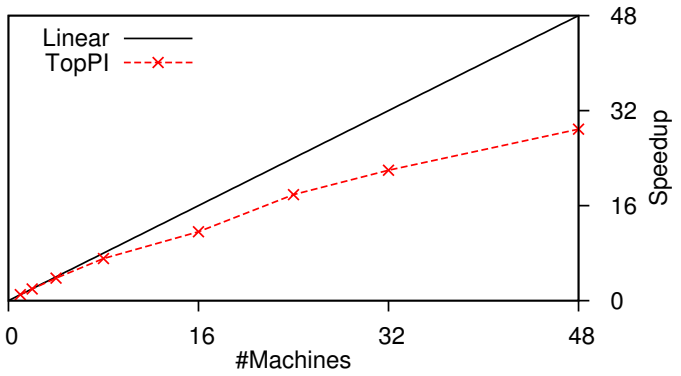


Figure 3.11: TopPI speedup when running on Hadoop with on *Supermarket* ($k = 1000$)

the dataset contains 54.8M transactions and 389k distinct items. The multithreaded version of TopPI is able to process this dataset in 20 minutes on a single server (31 threads) for $k = 1000$. Our main goal in this section is to highlight the scalability of the two-stage mining distributed version.

Figure 3.11 shows the performance of TopPI when running on a cluster of 1 to 48 machines. TopPI shows a perfect speedup from 1 to 8 machines (64 cores), and steadily gets faster with the addition of executors. Overall, the total CPU time (summed over all machines) spent in the mining stages remains stable: from 35,000 seconds on average from 1 to 8 machines, it only raises to 38,500 seconds with 48 machines (using their 384 cores). Adding a task to TopPI incurs I/O costs, such as the time spent reading the initial dataset. Hence there is a trade-off between the execution time of the mining stages and the I/O overhead. In this configuration, the sweet spot is around 8 executors. Should the workload increases, TopPI would achieve optimal speedup on larger clusters, as the overall mining time increases and compensates the I/O costs. This validates the distribution strategy used for the mining stages: the load is well partitioned, and the top-k pruning efficiency does not diminish significantly with increased partitioning.

3.2.6 Conclusion

Pattern mining is one of the key tools of data analysts, especially in the retail industry. As datasets get larger, finding patterns is not only about speeding up algorithms and implementations, but also about better identifying what patterns will be most useful to analysts. In this work, we collaborate with Intermarché to propose a new item-centric mining objective that guarantees the coverage of all products with k patterns in the results. The core of our algorithm, TopPI, is derived from state of the art standard FIM algorithm. This is combined with a top-k pruning strategy with an additional twist, as TopPI computes one top-k per product simultaneously. When executing in a distributed environment, top-k processing becomes challenging, since executors do not exchange information during mining phases, which could lead to sub-optimal pruning decision. We propose a two-stage mining approach that alleviates this issue by first assigning each product to a single executor to generate an approximate version of its top-k results. This produces bounds for each top-k structure are broadcasted, and can then be used in a second stage to generate the missing results while preserving pruning efficiency.

TopPI was mainly designed for a retail application, but was also evaluated on a variety of Web datasets. TopPI was released as an open-source project and can be found on github. Following this collaboration, Intermarché deployed TopPI on their own infrastructure to further analyze their data.

3.3 Summary of contributions

Over the past years, datasets have become increasingly large. As a reaction, distributed processing frameworks, such as Hadoop and Spark have been developed to scale data processing applications by deploying them on several executors running in parallel. The ability of human analysts to interpret results however has not benefited from the same improvement. Hence, as datasets grow, it is important to improve data analysis to better identify the most important results, rather than returning more.

Top-k processing [41] in general is well suited for this qualitative data analysis process. Given a scoring function representing how relevant a result is to the analyst, it aims at returning the k highest scoring results. By letting the analyst select k, the number of results displayed, this approach aims at saving execution time by avoiding computing answers that will never be read. Top-k processing has been studied in a variety of contexts, and the pruning techniques associated to it are well known. Still, implementing efficient top-k algorithms in distributed processing frameworks remains challenging because the programming model significantly restricts communications between executors, which is detrimental to pruning algorithms as results discovered cannot be shared.

In this chapter, we consider two specific data analysis problems and see how a top-k algorithm can be efficiently implemented to solve them. The first issue considered is temporal data processing. Using statistics on the dataset, our algorithm executes a preliminary pruning step that eliminates a large fraction of the results candidates. We then optimize the distributed join processing by ensuring that each executor is able to discover high scoring results locally, which is key to perform on-the-fly pruning. Then, we consider the case of pattern mining. In this context, we show that the mining stage can be split into two stages, which allows almost optimal pruning despite only using one communication phase between executors. In both cases, we validate our approaches with industrial partners on real datasets.

Chapter 4

Information discovery for large and heterogeneous datasets

The previous chapter presents my work on developing scalable algorithms for data analysis. Distributed systems provide computational resources, and top-k processing limits the number of result candidates by pruning the search space. Building these algorithms requires a good understanding of the mindset of the analyst. Indeed, they rely on a scoring function that represents the likelihood of a result to be of interest to the analyst.

On smaller datasets, simple signals such as frequency are sufficient to select results that the analyst can then browse. As datasets become larger, the number of potential results increases. Basic ranking functions are insufficient, as the most relevant results are drowned in an increasing amount of less insightful results. Web search engines have been dealing with this issue for years: they index billions of Web pages, and users will only read the first 10 results that are returned. Hence, a lot of work has been done to build relevance models specifically for text documents [63]. However, other data mining applications, including pattern mining, have received less attention, and thus require additional efforts to identify refined relevance measure. In addition to the ranking problem is the issue of identifying what constitutes a result. While some simple queries are satisfied with individual relevant item, others are more exploratory, and require an overview of the dataset. This is achieved by clustering algorithms [16], but also by returning a set of items with diversity constraints [84]. A new paradigm that has emerge recently is the concept of *composite items* [8]. Composite items bundle together individual items that, together, provide a relevant answer to a query.

Section 4.1 is a continuation of the work on pattern mining presented in Section 3.2. We evaluate, in collaboration with marketing specialists, which measure is the most appropriate to rank retail patterns. Over 30 different mathematical formulas have been proposed in the literature to assess the interestingness of patterns. Hence, in practice, analysts are faced with an embarrassingly large number of options. We first measure empirically the behavior of these measures, and then compare them in a user study. In Section 4.2, we consider the case of complex information needs that require the use of composite items. We propose a new approach based on fuzzy clustering to build high quality representative composite items.

Table 4.1: Top-5 demographics association rules, according to different interestingness measures. Rules are denoted {age, gender, department} → *product category*.

by confidence		**by Piatetsky-Shapiro [77]**		**by Pearson's χ^2**	
$\{>65, F,$ Aube$\}$	→ *Dairy*	$\{*, *, Nord\}$	→ *Liquids*	$\{*, *, Somme\}$	→ *Cut cheese*
$\{>65, F,$ Aveyron$\}$	→ *Dairy*	$\{*, *, Nord\}$	→ *Soft drinks*	$\{*, F, Somme\}$	→ *Cut cheese*
$\{>65, F,$ Val de Marne$\}$	→ *Dairy*	$\{*, *, Nord\}$	→ *Beers*	$\{>65, *, Morbihan\}$	→ *Fresh milk*
$\{>65, F,$ Seine S^t Denis$\}$	→ *Dairy*	$\{*, *, Nord\}$	→ *Spreads*	$\{>65, *, Somme\}$	→ *Cut cheese*
$\{>65, F,$ Haute Saone$\}$	→ *Dairy*	$\{*, F, Nord\}$	→ *Soft drinks*	$\{*, *, Finistere\}$	→ *Canned pork*

4.1 Comparison of interestingness measures

4.1.1 Context

Ever since databases have been able to store basket data, many techniques have been proposed to extract useful insights for analysts. One of the first, association rule mining [2], also remains one of the most intuitive. Finding association rules in long-tailed dataset is challenging, and we developed the algorithm TopPI (Section 3.2) to solve this problem. TopPI performs *item-centric mining*, which ensures that each product is represented in the result by k itemsets. From these itemsets, association rules can be built and presented to the analysts. While TopPI solves computational problem of extracting these rules, a follow-up challenge is to identify, for each item, the most *interesting* rules. TopPI finds, for each product, k association rules, with k typically varying from 10 to 1000. While k can be set to a reasonably low value, a marketing analyst browsing these results is unlikely to read all of them, so ranking is crucial. Unfortunately, there is a lack of thorough studies of which of the many interestingness measures for ranking rules [47] is most appropriate for which application domain. For instance, Table 4.1 shows a ranking of the top-10 rules of the form *customer segment* → *product category* according to 3 different interestingness measures proposed in [47]. If we denote rules as $A \rightarrow B$, *confidence* is akin to precision and is defined as the probability to observe B given that we observed A, i.e., $P(B|A)$. *Piatetsky-Shapiro* [77] combines how A and B occur together with how they would if they were independent, i.e., $P(AB) - P(A)P(B)$. *Pearson's* χ^2 measures how unlikely observations of A and B are independent. This example shows that these measures result in different rule rankings.

In collaboration with Intermarché, one of the largest retailers in France, we developed CAPA, a framework to compare the rankings generated by 34 different interestingness measures. CAPA loads the results generated by TopPI, ranks them according to each measure, and identifies correlations between these rankings. Following this step, CAPA is able to automatically scale down the problem from 34 measures to 6 different groups of measures. We then conduct a user study with two experienced domain experts from Intermarché in to answer the following question: out of the 6 families of interestingness measures, which ones are meaningful?

The work presented in this section was part of Martin Kirchgessner's Ph.D work, co-supervised with Sihem Amer-Yahia. It is detailed in the following publication:

- **Testing Interestingness Measures in Practice: A Large-Scale Analysis of Buying Patterns**
  Martin Kirchgessner, Vincent Leroy, Sihem Amer-Yahia, Shashwat Mishra and Intermarché Alimentaire International Stime

In Proceedings of the International Conference on Data Science and Advanced Analytics (DSAA), 2016, pages 547–556.

4.1.2 Problem definition

Table 4.2: Our mining scenarios and example association rules.

Target Associations	Desired association rules
`demo_assoc`: *segment* → *category*	A customer segment tends to purchase products in a category. $\{< 35, F, *\} \rightarrow$ *Baby food*
`prod_assoc_t`: *product(s)* → *product*	Products purchased simultaneously (1 visit to the store). {*vanilla cream*}→ *chocolate cream*
`prod_assoc_c`: *product(s)* → *product*	Customers' product associations over time (multiple visits). {*Pork sausage, mustard*}→ *dry Riesling*

Mining Scenarios We consider three mining scenarios described in Table 4.2. Each scenario leads to the construction of a collection of transactions, on which TopPI is executed. Each itemset P returned in the top-k of an item i implies an association rule of the form $P \setminus \{i\} \rightarrow i$. $P \setminus \{i\}$ is the antecedent of the rule, and i its consequent. In `demo_assoc`, the antecedent is a customer segment and the consequent is a single product category. In `prod_assoc_t` and `prod_assoc_c`, the antecedent is a set of products and the consequent is a single product. Analysts generally focus on particular products or product categories. This can be done by specifying a list of items for TopPI to focus on.

Interestingness Measures

Large datasets often contain millions of frequent closed itemsets, and each of them may lead to several association rules. The ability to identify valuable rules is therefore of the utmost importance to avoid drowning analysts in useless information. Association rules $A \rightarrow B$ were originally selected using thresholds for support ($support_{\mathcal{T}}(A \cup B)$) and confidence ($\frac{support_{\mathcal{T}}(A \cup B)}{support_{\mathcal{T}}(A)}$) [2]. However using two separate values, and guessing the right threshold is not natural. Furthermore, support and confidence do not always coincide with the interest of analysts. Hence, a number of interestingness measures that serve different analyses were proposed in the literature [47,67,71]. Table 4.3 summarizes the measures we use in this work. The first column contains the name of the measure. The second column will be referred to later. A more complete version of this table including the expression of each measure can be found in [61].

Goal

Our goal is to help analysts test and compare the rankings produced by different interestingness measures on rules extracted from their data. An analyst can specify one of 3 mining scenarios, `demo_assoc`, `prod_assoc_t`, and `prod_assoc_c`, and one or several targets (categories in the case of `demo_assoc`, products in the case of the other two), and CAPA generates as many rule rankings as the number of interestingness measures.

Table 4.3: Interestingness measures of a rule $A \rightarrow B$. $*$, $\triangleright$ indicate measures producing identical rule rankings when B is fixed. $\Diamond$, $\dagger$, $\ominus$, $\otimes$ indicate measures that always produce the same rule ranking. $|\mathcal{T}|$ is the number of transactions. $P(A) = support(A)/|\mathcal{T}|$.

Measure	**Group and description**	
One-Way Support	G_1	Highest confidence Very low recall Favors frequent targets
Relative Risk		
Odd Multiplier		
Zhang		
Yule's Q $\Diamond$		
Yule's Y $\Diamond$		
Odds Ratio $\Diamond$		
Information Gain $*\ominus$		
Lift $*\ominus$		
Added Value $*$		
Certainty Factor $*$		
Confidence / Precision $*\otimes$		
Laplace Correction $*\otimes$		
Loevinger $\dagger$		
Conviction $\dagger$		
Example and Counter-example Rate		
Sebag-Schoenauer		
Leverage		
Least Contradiction	G_2	Very high confidence Very low recall
Accuracy		
Pearson's χ^2 $\triangleright$	G_3	High confidence Low recall Low sensitivity (to target frequency)
Gini Index $\triangleright$		
J-measure		
Φ Linear Correlation Coefficient		
Two-Way Support Variation		
Fisher's exact test		
Jaccard		
Cosine	G_4	Average confidence Average recall, Low sensitivity
Two-Way Support		
Piatetsky-Shapiro	G_5	Low confidence High recall
Klosgen		
Specificity		
Recall	G_6	Lowest confidence Highest recall, Favors rare targets
Collective Strength		

4.1.3 Empirical evaluation

We present an empirical evaluation of the 34 measures for association rules introduced in Section 4.1.2. Recall that our goal is to assist the analyst in selecting measures. Our evaluation consists in comparing rankings produced by these measures on retail data to discover which measures differ significantly in practice. We then use that similarity to classify ranking measures into *groups*. We annotate these groups based on the properties common to the group. We discuss key insights obtained from experimentation on each group. The goal of this evaluation is to automatically detect similarities between interestingness measures and reduce the number of candidate measures to present to analysts in the user study (Section 4.1.4).

We first present methods used to compare ranked list. Then, we compare the resulting rankings and select representative measures.

Ranking similarity measures

In this section, we discuss some methods for comparison of ranked lists. The first three methods are taken from the literature. We then introduce *NDCC*, a new parameter-free ranking similarity designed to emphasize differences at the top of the ranking.

We are given of a set of association rules $\mathcal{R}$ to rank. We interpret each measure, m, as a function that receives a rule and generates a score, $m : \mathcal{R} \to \mathbb{R}$. We use $L^m_{\mathcal{R}}$ to denote an ordered list composed of rules in $\mathcal{R}$, sorted by decreasing score. Thus, $L^m_{\mathcal{R}} =< r_1, r_2, \ldots >$ s.t. $\forall i > i'\ m(r_i) < m(r_{i'})$. We generate multiple lists, one for each measure m, from the same set $\mathcal{R}$. $L^m_{\mathcal{R}}$ denotes a ranked list of association rules according to measure m where the rank of rule r is given as $rank(r, L^m_{\mathcal{R}}) = |\{r'|r' \in \mathcal{R},\ m(r') \geq m(r)\}|$. To assess the dissimilarity between two measures, m and m', we compute the dissimilarity between their ranked lists, $L^m_{\mathcal{R}}$ and $L^{m'}_{\mathcal{R}}$. We use r^m as a shorthand notation for $rank(r, L^m_{\mathcal{R}})$.

Spearman's rank correlation coefficient Given two ranked lists $L^m_{\mathcal{R}}$ and $L^{m'}_{\mathcal{R}}$, *Spearman's rank correlation* [34] computes a linear correlation coefficient that varies between 1 (identical lists) and -1 (opposite rankings) as shown below.

$$Spearman(L^m_{\mathcal{R}}, L^{m'}_{\mathcal{R}}) = 1 - \frac{6 \sum\limits_{r \in \mathcal{R}} (r^m - r^{m'})^2}{|\mathcal{R}|(|\mathcal{R}|^2 - 1)}$$

This coefficient depends only on the difference in ranks of the element (rule) in the two lists, and not on the ranks themselves. Hence, the penalization is the same for differences occurring at the beginning or at the end of the lists.

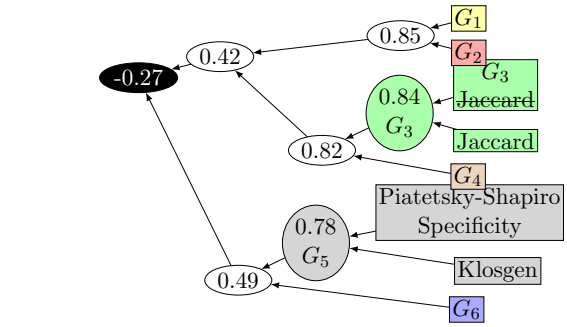
Kendall's τ rank correlation coefficient *Kendall's τ rank correlation coefficient* [60] is based on the idea of agreement among element (rule) pairs. A rule pair is said to be *concordant* if their order is the same in $L^m_{\mathcal{R}}$ and $L^{m'}_{\mathcal{R}}$, and *discordant* otherwise. τ computes the difference between the number of concordant and discordant pairs and divides by the total number of pairs as shown below.

$$\tau(L^m_{\mathcal{R}}, L^{m'}_{\mathcal{R}}) = \frac{|C| - |D|}{\frac{1}{2}|\mathcal{R}|(|\mathcal{R}| - 1)}$$

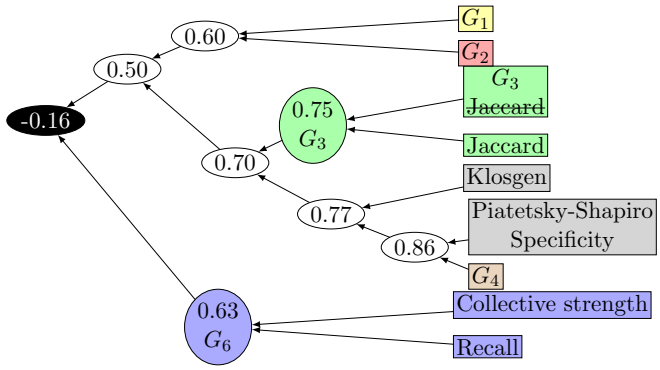
$$C = \{(r_i, r_j)|r_i, r_j \in \mathcal{R} \wedge i < j \wedge \\ \text{sign}(r^m_i - r^m_j) = \text{sign}(r^{m'}_i - r^{m'}_j)\}$$

$$D = \{(r_i, r_j)|r_i, r_j \in \mathcal{R} \wedge i < j \wedge \\ \text{sign}(r^m_i - r^m_j) \neq \text{sign}(r^{m'}_i - r^{m'}_j)\}$$

Similar to *Spearman's*, τ varies between 1 and -1, and penalizes uniformly across all positions.



(a) *NDCC*



(b) τ

Figure 4.1: Hierarchical clustering of interestingness measures for a single target

Overlap@k Overlap@k is another method for ranked lists comparison widely used in Information Retrieval. It is based on the premise that in long ranked lists, the analyst is only expected to look at the top few results that are highly ranked. While *Spearman* and τ account for all elements uniformly, Overlap@k compares two rankings by computing the overlap between their top-k elements only.

$$Overlap@k(L_{\mathcal{R}}^{m}, L_{\mathcal{R}}^{m'}) = \frac{|\{r \in \mathcal{R} \mid r^m \leq k \wedge r^{m'} \leq k\}|}{k}$$

Normalized Discounted Correlation Coefficient Overlap@k, *Spearman*'s and τ sit at two different extremes. The former is conservative in that it takes into consideration only the top k elements of the list whereas the latter two take too liberal an approach by penalizing all parts of the lists uniformly. In practice, we aim for a good tradeoff between these extremes.

To bridge this gap, we propose a new ranking correlation measure coined *Normalized Discounted Correlation Coefficient* or *NDCC*. *NDCC* draws inspiration from *NDCG, Normalized Discounted Cumulative Gain* [57], a ranking measure commonly used in Information Retrieval. The core idea in *NDCG* is to reward a ranked list $L_{\mathcal{R}}^{m}$ for placing an element r of relevance rel_r by $\frac{rel_r}{\log r^m}$.

The logarithmic part acts as a smoothing discount rate representing the fact that as the rank increases, the analyst is less likely to observe r. In our setting, there is no ground truth to properly assess rel_r. Instead, we use the ranking assigned by m' as a relevance measure for r, with an identical logarithmic discount. When summing over all of $\mathcal{R}$, we obtain *DCC*,

which presents the advantage of being a symmetric correlation measure between two rankings $L^m_{\mathcal{R}}$ and $L^{m'}_{\mathcal{R}}$.

$$DCC(L^m_{\mathcal{R}}, L^{m'}_{\mathcal{R}}) = \sum_{r \in \mathcal{R}} \frac{1}{\log(1 + r^{m'}) \log(1 + r^m)}$$

We compute *NDCC* by normalizing *DCC* between 1 (identical rankings) and -1 (reversed rankings).

$$NDCC(L^m_{\mathcal{R}}, L^{m'}_{\mathcal{R}}) = \frac{dcc - avg}{max - avg}$$

$$\text{where } dcc = DCC(L^m_{\mathcal{R}}, L^{m'}_{\mathcal{R}}),\ max = DCC(L^{m'}_{\mathcal{R}}, L^{m'}_{\mathcal{R}})$$
$$min = DCC(L*, L^{m'}_{\mathcal{R}}),\ L* = rev(L^{m'}_{\mathcal{R}})$$
$$avg = (max + min)/2$$

Ranking comparison by example We illustrate the difference between all ranking correlation measures with an example in Table 4.4. This shows correlation of a ranking L^1 with 3 others, according to each measure. *NDCC* does indeed penalize differences at higher ranks, and is more tolerant at lower ranks.

Rankings comparison

We perform a comparative analysis of ranking measures, on our 3 mining scenarios summarized in Table 4.2. We generate association rules $A \rightarrow B$ where B is a single product among a set of 64 previously studied by analysts. Overall we obtain 1,651,024 association rules, and we compute one rule ranking per product and per interestingness measure. Our first observation is that the results we obtain for all scenarios lead to the same conclusions. Therefore, we only report numbers for `prod_assoc_c`.

While all measures are computed differently, we notice that some of them always return the same ranking for association rules of a given target. We identify them in Table 4.3 using symbols. Other notable similarities include *Sebag-Schoenauer* [47] and *lift* (89% of rankings are equal), as well as *Loevinger* and *lift* (87%). This difference between the number of interestingness measures considered (34) and the number of different rankings obtained (25) can easily be explained analytically in the case of a fixed target. Indeed, for a given ranking, $P(B)$ is constant, which eliminates some of the differences between interestingness measures. In addition, some measures only have subtle differences which only appear when selecting extreme values for $P(A)$, $P(B)$ and $P(AB)$, which do not occur in practice in our retail dataset.

Comparative analysis We now evaluate similarity between interestingness measures that do not return the same rankings. We compute a 34×34 correlation matrix of all rankings according to each correlation measure described in Section 4.1.3, and average them over the 64 target products. This gives us a ranking similarity between all pairs of measures. We then rely on hierarchical clustering with average linkage [86] to obtain a dendrogram of interestingness measures and analyze their similarities. The dendrograms for *NDCC* and τ are presented in Figure 4.1. For better readability, we merge sub-trees when correlation is above 0.9. To describe the results more easily, we partition interestingness measures into 6 groups, as indicated in the third column in Table 4.3.

Table 4.4: Example rankings and correlations

Ranking	Content
L^1	r_1, r_2, r_3, r_4
L^2	r_2, r_1, r_3, r_4
L^3	r_1, r_2, r_4, r_3
L^4	r_2, r_3, r_1, r_4

	Spearman	τ	*Overlap*@2	*NDCC*
L^2	0.80	0.67	1	0.20
L^3	0.80	0.67	1	0.97
L^4	0.40	0.33	0.5	−0.18

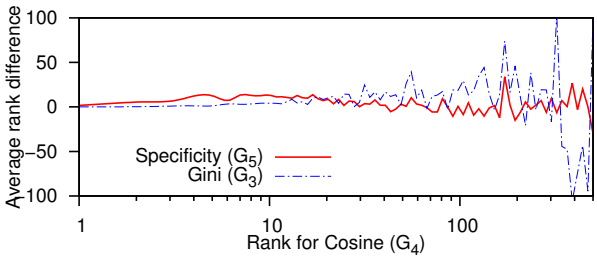


Figure 4.2: Rank correlations

G_1 is by far the largest group: in addition to 4 measures that always generate the same rankings, 14 other measures output similar results. A second group, G_2, comprising 2 measures, is quite similar to G_1 according to *NDCC*. τ also discovers this similarity, but considers it lower, which shows that it is mostly caused by high ranks. *Jaccard* is as a slight outlier in G_3 according to *NDCC*. Indeed, when focusing on the first 20 elements (*Overlap*@20), only an average of 71% are shared between *Jaccard* and the rest of G_3. This situation also occurs between *Klosgen* and the rest of G_5. Interestingly, we observe that, according to *NDCC*, G_5 is closest to G_6 and is negatively correlated with the other groups. However, according to τ, G_5 is very similar to G_4 and is negatively correlated with G_6. This difference between ranking measures illustrates the importance of accounting for rank positions. When the top of the ranking is considered more important, some similarities emerge.

We illustrate this behavior in Figure 4.2 by displaying correlation between rankings obtained with different interestingness measures. This experiment clearly shows that overall, *cosine* (G_4) is closer to *specificity* (G_5) than *Gini* (G_3), as the rank difference observed in the results is overall smaller. However, when focusing on the top-10 results of *cosine*, *Gini* assigns closer ranks than *specificity*. This explains the difference in clustering between *NDCC*/*overlap* and τ/*Spearman*.

Annotating groups While using hierarchical clustering on interestingness measures allows the discovery of families of measures, and their relative similarity, it does not fully explain which types of results are favored by each of them. We propose to compare their outputs according to the two most basic and intuitive interestingness measures employed in data mining: *recall* and *confidence*. *recall* represents the proportion of target items that can be retrieved by a rule, that is, $P(A|B)$. Its counterpart, *confidence*, represents how often the

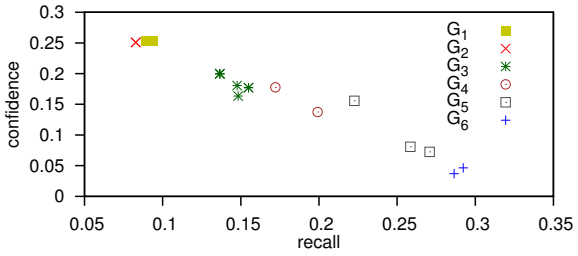


Figure 4.3: Avg. recall/confidence of the top-20 results of interestingness measures

consequent is present when the antecedent is, that is, $P(B|A)$. We present, in Figure 4.3, the average *recall* and *confidence* of the top-20 rules ranked according to each interestingness measure. G_1 contains *confidence*, so it is expected to score the highest on this dimension. G_2 is extremely close to G_1, but obtains slightly lower *confidence* and *recall*. We then have, in order of increasing *recall* and decreasing *confidence* G_3, G_4 and G_5. Finally, G_6, which contains *recall*, obtains the highest *recall* but the lowest *confidence*. Figure 4.3 also shows that executing a Euclidean distance-based clustering, such as k-means, with recall/confidence coordinates would lead to groups similar to the ones obtained with hierarchical clustering. Hence, this analysis is consistent with the hierarchical grouping and the correlation with *NDCC*.

While we believe that *NDCC* better reflects the interpretation of analysts browsing rules, it is important to note that the grouping of interestingness measures created through this evaluation is stable across all 4 correlation measures and for all 3 scenarios. Correlation between different families of measures may vary, but measures within a single family always have a high similarity. Thus, we conjecture that the obtained results are true in the general case of food retailers and we can rely on these groups to reduce the number of options presented to analysts.

Selecting representative measures We summarize the findings of the comparative evaluation in the last column of Table 4.3. We identify 6 families of measures that behave similarly. Each family offers a different trade-off in terms of confidence and recall, and thus ranks association rules differently. We select the quality measure that most represents each family of measures (i.e. with highest average similarity) in order to confront the results of this analysis with the opinion of domain experts in our user study. Taking a general data mining perspective leads us to considering G_3 and G_4 as the most promising families for finding interesting association rules. Indeed, it is important to achieve a good trade-off between *recall* and *confidence* in order to find reliable association rules that can be applied in a significant number of cases. Hence, *F1* score, that combines *recall* and *confidence*, would prefer G_3 and G_4 to others.

4.1.4 User study

We now report the results of a user study with domain experts from Intermarché. The goal of this study is to assess the ability of interestingness measures to rank association rules according to the needs of an analyst. As explained in Section 4.1.3, we identified 6 families of measures, and selected a representative of each group for the user study (their names are in bold in Table 4.3). We rely on the expertise of our industrial partner to determine, for

each analysis scenario, which family produces the most interesting results. This experiment involved 2 experienced analysts from the marketing department of Intermarché. We setup CAPA and let analysts select targets multiple times in order to populate a web application's database with association rules. We let our analysts interact with CAPA without any time restriction, and collect their feedback in a free text form.

Each analyst firstly has to pick a mining scenario among `demo_assoc`, `prod_assoc_t`, or `prod_assoc_c`. Then she picks a target category or a target product in the taxonomy. In `prod_assoc_t` and `prod_assoc_c`, she also has the option to filter out rules whose antecedent products are not from the same category as the target. Finally, she chooses one of our 6 ranking measures to sort association rules. Neither the name of the measure nor its computed values for association rules are revealed, because we wanted analysts to evaluate rankings without knowing how they were produced.

Resulting association rules are ranked according to a selected measure. Each rule is displayed with its support, confidence and recall, such that analysts can evaluate it at a glance. For each scenario, our analysts are asked which representative measure highlights the most interesting results (as detailed below, in all cases a few of them were chosen).

Scrolling behavior

Once the analyst selects a target, *all* matching rules are returned. The initial motivation of this choice was to determine how many results are worth displaying and are actually examined by the analysts. According to the follow-up interview with the analysts, they carefully considered the first ten results, and screened up to a hundred more. Interestingly, analysts mentioned that they also scrolled down to the bottom of the list in order to see which customer segments are not akin to buying the selected category. For example, when browsing demographic association rules, they expected to find $\{$*50-64*$\} \rightarrow$ *pet food* among top results, but also expected $\{< 35,$ *Paris*$\} \rightarrow$ *pet food* among bottom results. This confirms that all rules should remain accessible. This also indicates that while interestingness measures favor strong associations, it could also be useful to highlight *anti*-rules.

Feedback on ranking measures

We let marketing experts explore all 3 scenarios and express their preference towards groups of measures.

In the `demo_assoc` case, G_1 and G_3 were both highly appreciated. G_1 favors rules such as $\{< 35, M,$ Oise$\} \rightarrow$ *Flat and Carbonated drinks*. These rules are very specific and thus have a very high confidence (31,58 % in this particular case). However, this comes at the cost of recall (0,08 %). Experts involved in this study value *confidence* much more than *recall*, as their priority is finding rules that they consider reliable. A low support is not necessarily an issue, and can lead to the discovery of surprising niche rules that can be exploited nonetheless. As discussed in Section 4.1.3, G_3 offers a more balanced trade-off between confidence and recall, and prioritizes rules such as $\{< 35,$ *, *$\} \rightarrow$ *Baby food* (confidence 8,57 %, recall 37,61%). These rules are interesting because they capture a large fraction of the sales of a given category, but are less reliable and generally less surprising. G_2 and G_4 were considered as less interesting than G_1 and G_3 respectively. Their results offer similar trade-offs, but with lower confidence each time. G_5 and G_6 were considered unusable because of their very low confidence.

When experimenting with `prod_assoc`, we observed a slightly different behavior. By default, the analysts favored G_1 and G_2 because of the confidence of their results. Then, we offered the analysts the possibility of filtering the rules to only keep the ones in which the antecedent contains products from the same category as the target. This led to analysts favoring G_3 and G_5. This difference is caused by an important criterion: the ability of a measure to filter out very popular products. For example, the rule {*van. cream, emmental*}→ *choc. cream* usually appears just above its shorter version {*van. cream*}→ *choc. cream*, because the first one has a confidence of 32% and the second 31%. However, experts prefer the second one, because *emmental* (cheese) is among the heavy hitters in stores. Its addition to the rule is hence considered insignificant. This "noise" generally increases with *recall*. Hence, when no filtering is available, G_1 is selected, but analysts prefer the *recall* and *confidence* trade-off provided by G_3 and G_5. Again, G_4 suffered from its proximity to G_3 with lower confidence, while G_6's confidence was too low.

In all cases, analysts mentioned G_6 as uninteresting because it selects rules of low *confidence*. In general, sorting by decreasing *lift* (which is close to sorting by decreasing *confidence*) is the preferred choice. Combined with the minimum support threshold used in the mining phase, this ranking promotes rules that are considered reliable. However, in the case where analysts are given the ability to filter out noisy products (very frequent ones), they prefer the ranking produced by *Piatetsky-Shapiro*'s measure [77]. That could be explained by the fact this measure provides a good compromise between *confidence* and *support*. The noisy products that this measure may introduce can be filtered out by analysts.

4.1.5 Conclusion

Mining algorithms often generate thousands of results. Identifying the most interesting ones and showing them to the analysts first is important, as otherwise they may be ignored. When searching for an appropriate ranking function in the literature, we face too many choices: over 34 interestingness measures have been proposed, with no guidelines on which should be used.

We proposed CAPA, a framework for automatically analyzing interestingness functions by comparing the rankings they produce in real use-cases. CAPA clusters the measures into groups and determines the characteristics of each group. This scales down the problem to a much smaller number of options, so it becomes manageable for analysts. A user study can then find out which is the best option among the initial 34.

In the context of retail data and the dataset provided by Intermarché, we concluded that lift and Piatetsky-Shapiro best fit the needs of analysts, as they ensure high confidence. Our user study also led us to thinking about future research directions, including the extraction of negative results (anti-rules).

4.2 Integrated approach for building Composite Items

4.2.1 Context

Some queries are inherently multi-faceted and cannot be answered by a single item. For instance, planning a city tour is not simply asking either for an hotel suggestion or one location to visit. Instead, the user requires a bundle that contains an hotel, a few locations to visit and suggestions of restaurants. Building a bundle is more than aggregating the results of independent queries on different types of objects. Indeed, this one-type-at-a-time approach does not necessarily guarantee that items in the bundle (e.g. an hotel, a restaurant and a museum) will be close to each other, i.e., *cohesive*. Instead, we consider the problem of selecting this bundle in one step: this approach is known as building Composite Items.

CIs have been shown to be very effective in solving complex information needs such as organizing vacations, selecting books for a reading club, or organizing a movie rating contest [8, 10, 19, 23, 31, 49, 56, 83, 97]. In those applications, a CI is a set of close items (e.g., geographically close points of interest - POIs, movies rated by the same users) that satisfy *a budget* (e.g., at least two schools and one theater, at least one movie per genre). When summarizing POIs, a CI may correspond to geographically close places that have different types (e.g., theater and museum) and whose total visit time does not exceed 3 hours. When selecting books, a CI may be formed by similar books, i.e., on similar topics, written by different authors and whose total price is less than a maximum amount. When organizing a movie contest, a CI is a set of comparable movies, i.e., having common reviewers, and with different genres or release years. The budget constraint of a CI can therefore be used to *glue together* heterogeneous items, i.e., items with different types. In that case, we say that a CI is *valid*. The problem of summarizing heterogeneous item collections can therefore be formulated as finding k valid, cohesive and representative CIs, i.e., each CI satisfies budget constraints, is formed of "close" items, and the set of CIs "covers" all input items.

Forming valid, cohesive and representative CIs can be naturally expressed as a constrained optimization problem [8]. Existing solutions to solve this problem usually rely on two phases: in one solution, many valid CIs (i.e. satisfying the budget constraint) are built, and then the k farthest are chosen thereby resulting in representative CIs. In the other, a k-clustering is performed in the first stage to address representativity, and then one valid CI, i.e. satisfying constraints, is picked from each cluster in order to produce k CIs overall. This process decouples budget constraint satisfaction (e.g., a CI must contain one museum and 2 restaurants) from the optimization goal (e.g., each CI is a set of closely located POIs). As a result, we can argue that while clustering is a natural solution to finding CIs, existing formulations are not well-adapted to achieve validity, cohesiveness and representativity simultaneously. We hence advocate the seamless integration of validity, cohesiveness and representativity when building CIs.

The work presented in this section is the result of a collaboration between the SLIDE and AMA groups from LIG, and the KEIO University. It is detailed in the following publications:

- **Building Representative Composite Items**
  Vincent Leroy, Sihem Amer-Yahia, Eric Gaussier and Hamid Mirisaee
  In Proceedings of the ACM Conference on Information and Knowledge Management (CIKM), 2015, pages 1421–1430.

- **Task Composition in Crowdsourcing**

Sihem Amer-Yahia, Ria Borromeo, Eric Gaussier, Vincent Leroy, Julien Pilourdault and Motomichi Toyama
In Proceedings of the International Conference on Data Science and Advanced Analytics (DSAA), 2016, pages 194–203.

Section 4.2.2 contains our formalization and general problem statement. Section 4.2.3 describes our integrated algorithm, KFC. We evaluate our approach in Section 4.2.4. We conclude in Section 4.2.5.

4.2.2 Model and Problem

In this section, we first define our formal model and discuss the link between clustering, validity, cohesiveness and representativity. We then formalize the problem of *finding a set of k, possibly overlapping, valid, cohesive and representative CIs*.

Data Model

We are given a set $\mathcal{X}$ of items where $x \in \mathcal{X}$ is uniquely identified. $\mathcal{X}$ is a heterogeneous set of items each of which may have one or several types in $\mathcal{T} = \{t_1, \ldots, t_n\}$. For example, the movie Titanic has two types: romance and drama. A book type could be novel or adventure and the type of a point of interest could be museum, park, etc. We use $x.type$ to refer to the type(s) of x. We furthermore assume that an item x may have a cost, that will be denoted as $x.cost$. For a book, this would typically be its selling price. For a museum, it could either be the cost of an entry ticket or the average time required to visit it.

We define a budget vector $b = \langle \#t_1, \ldots, \#t_n, \#_\$ \rangle$ where each $\#t_i$ specifies a cardinality for an item type $t_i \in \mathcal{T}$ and $\#_\$$ is a total cost (e.g., maximum price a user is willing to pay for a movie or maximum time a user is willing to spend visiting a place). For example, the vector $\langle 1, 2, 1, 90 \rangle$ applied on books would represent 1 novel, 2 art books, and 1 self-help book, assuming those are the only available book types, whose total price does not exceed 90$. The same vector applied on points of interest in a city would be interpreted very differently and represent 1 gym, 2 subway stops and 1 bakery and a total time not exceeding 90 minutes.

We will make use of a distance function, noted $d(,)$ to compare a pair of items $(x, x') \in \mathcal{X} \times \mathcal{X}$. For instance, if x and x' are points of interest in a city, it is natural to use their geographic distance. If items are books, it is more appropriate to compare them according to their content, e.g. based on the cosine between their vectors; similarly, if items are movies, their distance can be inversely proportional to the fraction of reviewers who like both x and x'.

Representativity through fuzzy clustering

We are interested in identifying valid, cohesive and representative sets of items where each item has one or several types. The *validity* of a set of items is expressed in terms of the budget vector $b = \langle \#t_1, \ldots, \#t_n, \#_\$ \rangle$ introduced above. The *cohesiveness* is the ability to identify sets of items relatively close to each other, whereas the *representativity* is the ability to cover the input dataset. The clustering literature contains many proposals for finding representative points of a dataset. Indeed, representative points are typically obtained, in any given dataset, as the centroids of the clusters present in that dataset: The set of clusters "covers" the whole dataset and their centroids represent a summary of the content of each

cluster. In *hard* clustering, items are divided into distinct clusters and each item belongs to exactly one cluster, a framework well adapted to homogeneous items [56]. However, in the case of a heterogeneous set of items, an item may have different types and hence belong to more than one cluster. Therefore, we propose to study the applicability of *fuzzy* clustering [16] to the problem of finding valid, cohesive and representative items.

The most popular fuzzy clustering algorithm is Fuzzy C-Means (FCM) [16]. FCM assigns a set of items $\mathcal{X}$ to a collection of k fuzzy clusters represented through their centroids $v_j,\ 1 \leq j \leq k$ (the set of centroids will be denoted V). More precisely, given a set of N items, $\mathcal{X}$, the algorithm returns both the k centroids and a partition matrix $W = w_{i,j} \in [0,1],\ i \in [1,N],\ j \in [1,k]$ where each w_{ij} represents the degree to which item x_i belongs to cluster j. Given a distance function $d(,)$, the standard objective function of FCM is as follows:

$$\underset{V,W}{\operatorname{argmin}} \sum_{i=1}^{N} \sum_{j=1}^{k} w_{ij}^{m} d(x_i, v_j)$$

$$s.t.\ \forall i \in [1,N],\ \sum_{j=1}^{k} w_{ij} = 1$$

where m is a weighting exponent, greater than one. A large value of m results in smaller memberships w_{ij} and hence, fuzzier clusters, whereas setting m to 1 leads to hard clustering [16]. The problem above is typically solved through an alternate optimization process in which one fixes v (respectively w) and solves for w (respectively v). The proof that such an approach converges is given in [15]; furthermore, initialization of k-means++ [12] can also be used for the centroids.

Fuzzy clustering thus represents a direct way to identify clusters in a dataset and their representative points defined by their centroids. Furthermore, its fuzzy nature enables each point to be assigned to different clusters (and centroids) through membership values.

Problem Statement

We seek to find a set of k valid, cohesive and representative items. Intuitively, validity finds sets of items that satisfy a budget constraint (i.e., cardinality and/or cost) b which glues together items of different types into *composite items*, CIs. Cohesion and representativity intuitively try to identify those CIs formed of close items that cover the input dataset (i.e., that are close to cluster centroids). We first define a valid CIs as follows:

Definition 1. *Given a set of items $\mathcal{X}$ and a budget b, a valid CI, denoted $\{x_1, \cdots, x_{le}; x_i \in X,\ 1 \leq i \leq le\}$, is a set of items such that :*

$$\begin{cases} (i)\ \forall \#t_j \in b,\ \sum_{i=1}^{le} \mathbb{1}(t_j, x_i.type) \geq \#t_j \\ (ii)\ \sum_{i=1}^{le} x_i.cost \leq \#_{\$} \end{cases}$$

where $\mathbb{1}$ is an indicator function which is 1 if both arguments are equal and 0 otherwise. le is the number of items in the CI and is such that $le \geq n$, where n is the number of type values considered. The set of all valid CIs will be denoted as $\mathcal{V}_{CI}$.

We can now formulate our problem as a joint optimization problem where one part aims at identifying good summaries (i.e. cluster centroids that are representative) of the set of items whereas the other part ensures that the representatives chosen are "close" to valid CIs, which are in turn cohesive, i.e., formed of closeby items. The closest cohesive CIs to the obtained centroids are thus valid and representative of the set of items. We in fact face a minimization problem involving the distance function $d(,)$. Note that the weighting exponent m of the fuzzy clustering problem takes values in $[1, \infty]$. This leads to:

$$\begin{cases} \textbf{General formulation} \\ \underset{V,W}{\text{argmin}}(1-\lambda) \underbrace{\sum_{j=1}^{k}\sum_{i=1}^{N} w_{ij}^{m} d(x_i, v_j)}_{FC} + \\ \qquad\qquad \lambda \underbrace{\sum_{j=1}^{k} \min_{C \in \mathcal{V}_{CI}} \left(\sum_{x \in C} d(x, v_j) \right)}_{\texttt{CRCI}} \\ s.t.\ \forall i \in [1, N],\ \sum_{j=1}^{k} w_{ij} = 1 \end{cases} \tag{4.1}$$

where V denotes a set of k points (centroids) and W a partition matrix of size $N \times k$. λ is a parameter that controls the influence of the two aspects of the problem: identifying cluster centroids that are representative of the complete dataset (FC - *Fuzzy Clustering*) while ensuring that the centroids obtained are close to some valid CI (`CRCI` - *Close Representative CI*). Minimizing the sum of the distances of all the items of the CI to the centroid in `CRCI` additionally ensures the cohesion of the valid CI considered. It is the compromise between these different aspects that allows one to identify valid, cohesive and representative CIs. It is important to note that the above formulation corresponds to an integrated approach that directly yields valid, cohesive and representative CIs. This contrasts with most previous solutions that rely on a two-step approach in which candidate CIs are first generated and then filtered [8].

4.2.3 Algorithmic solution

We present here an algorithmic solution for the optimization problem above, focusing on the Euclidean distance for $d(,)$. Prior to that, we first introduce a slight generalization that partly circumvents the minimization problem in `CRCI`.

Given a set of items $\mathcal{X}$, $\mathcal{X} \subseteq \mathbb{R}^p$, a budget constraint b and the set of valid CIs $\mathcal{V}_{CI}$, let f be a function that associates to a point $v \in \mathbb{R}^p$ a valid CI from $\mathcal{V}_{CI}$: $f : \mathbb{R}^p \rightarrow \mathcal{V}_{CI}$. As before, we will denote by V a set of k points (centroids) and by W a partition (weight) matrix of size

$N \times k$. We consider the following general minimization problem using the Euclidean distance:

$$\begin{cases} \textbf{Euclidean distance-based formulation} \\ \underset{V,W}{\operatorname{argmin}}(1-\lambda)\sum_{j=1}^{k}\sum_{i=1}^{N} w_{ij}^{m}||x_i - v_j||_2^2 + \\ \qquad\qquad \lambda\sum_{j=1}^{k}\sum_{x\in C_j}||x - v_j||_2^2 \\ with:\ C_j = f(v_j) \\ and\ s.t.\ \forall i \in [1,N],\ \sum_{j=1}^{k} w_{ij} = 1 \end{cases} \tag{4.2}$$

In the remainder, we will use $\mathcal{G}_{eucl}(V,W,f)$ to denote $(1-\lambda)\sum_{j=1}^{k}\sum_{i=1}^{N} w_{ij}^{m}||x_i - v_j||_2^2 + \lambda\sum_{j=1}^{k}\sum_{x\in C_j}||x - v_j||_2^2$, with C_j obtained from v_j through f.

If the set V is fixed and f is given, so that C_j is known for $1 \leq j \leq k$, then $\mathcal{G}_{eucl}(V,W,f)$ is a convex function of W and the W that minimizes it can be obtained by setting the derivative of the Lagrangian of $\mathcal{G}_{eucl}$ (that integrates the constraints on W) with respect to W to 0 and solving for W. This leads to the following update rule for W (equivalent to the standard FCM update rule [16]):

$$w_{ij}^{(l+1)} = \left(\sum_{k=1}^{k}\left(\frac{||x_i - v_j^{(l)}||_2^2}{||x_i - v_k^{(l)}||_2^2}\right)^{\frac{1}{(m-1)}}\right)^{-1} \tag{4.3}$$

where l serves to indicate that new values are computed from known (old) ones. Similarly, for fixed W and given C_j, the function $\mathcal{G}_{eucl}(V,W,f)$ is convex in V. The values of V minimizing $\mathcal{G}_{eucl}$ are obtained by setting the derivatives of $\mathcal{G}_{eucl}$ with respect to V to 0 and solving for V, leading to:

$$v_j^{(l+1)} = \frac{(1-\lambda)\sum_{i=1}^{N}(w_{ij}^{(l)})^m x_i + \lambda\sum_{x\in C_j^{(l)}} x}{(1-\lambda)\sum_{i=1}^{N}(w_{ij}^{(l)})^m + \lambda|C_j^{(l)}|} \tag{4.4}$$

where $|C_j^{(l)}|$ represents the number of items in $C_j^{(l)}$.

For the valid composite item C_j associated to the centroid v_j, two cases may arise depending on the function f considered. Either the valid composite item provided by f for the new centroid $v_j^{(l+1)}$ leads to a better solution than the one associated to $v_j^{(l)}$, and it is kept, or it does not lead to a better solution, in which case the previous valid composite item is used. This can be formalized as:

$$C_j^{(l+1)} = \begin{cases} f(v_j^{(l+1)}) \text{ if } \sum_{x\in f(v_j^{(l+1)})}||x - v_j^{(l+1)}||_2^2 \\ \qquad\qquad \leq \sum_{x\in C_j^{(l)}}||x - v_j^{(l+1)}||_2^2 \\ C_j^{(l)} \text{ otherwise} \end{cases} \tag{4.5}$$

Algorithm 1

Require: $\mathcal{X}$, budget constraint b, k, λ, step η, procedure f
Ensure: Set S of k CIs

$S \leftarrow \emptyset$; $\lambda' = \lambda$; $\lambda = 0$
Initialize (e.g. through random assignment) V and $W \rightarrow V^{(0)}, W^{(0)}, f^{(0)}(V^{(0)}) = f(V^{(0)})$
repeat
    repeat
        Update W through Eq. 4.3
        Update V through Eq. 4.4
        Update $f(V)$ through Eq. 4.5
    until $\mathcal{G}_{eucl}$ (resp. $\mathcal{G}_{cos}$) does not change
    $\lambda = \lambda + \eta$
until $\lambda \geq \lambda'$
$S \leftarrow f(V)$ (with the final f and V obtained)

The above update rules guarantee that, starting with $W^{(l)}$, $V^{(l)}$ and f, one has:

$$\mathcal{G}_{eucl}(V^{(l+1)}, W^{(l+1)}, f) \leq \mathcal{G}_{eucl}(V^{(l)}, W^{(l)}, f)$$

as, for each update of W and V, the function $\mathcal{G}_{eucl}$ is minimized and does not decrease when updating the CIs provided by f. Thus, the algorithm iterating over the update rules defined by Eq. 4.3, 4.4 and 4.5 convergences (as $\mathcal{G}_{eucl}$ is lower bounded by 0) and provides a local minimum for the problem with the Euclidean distance.

Algorithm 1 summarizes the steps followed. As one can note, we first set λ to 0 and gradually increase its value. By doing so, one first identifies fuzzy centroids that are then moved towards valid, cohesive CIs.

Choice of f

Because the budget constraints b considered here have two parts, related respectively to type cardinality and cost (see Definition 1), we rely on two scenarios associated to two different choices for f. In the first scenario, we restrict ourselves to budget constraints b that only contain type cardinality constraints: $b = \langle \#t_1, \ldots, \#t_n \rangle$. In that particular case, it is possible to efficiently compute, for any v_j, $\min_{C \in \mathcal{V}_{CI}} \sum_{x \in C} ||x - v_j||_2^2$ through the following process:

1. Set $C \leftarrow \emptyset$
2. For $i = 1$ to n, add to C the $\#t_i$ items of type t_i closest to v_j
3. Return C

The function f defined by the above algorithm, the complexity of which is $\mathcal{O}(kN)$ in the worst case, directly yields the minimizer of `CRCI` in Problem 4.1 as there is no other valid CI closer to the given point v_j.

In the second scenario, we consider cost constraints in addition to type cardinality constraints, leading to the general budget constraint: $b = \langle \#t_1, \ldots, \#t_n, \#_\$ \rangle$. In that case one cannot directly use the above approach and we resort in this study to backtracking: we first select the closest item to a given v_j with a type in b, and iteratively add the next closest item

to v_j compatible with the constraint in b. If the cost constraint is violated, the process backtracks until all the constraints are satisfied. Lastly, the backtracking process may not lead to an optimal solution in the sense of the minimization problem defined in `CRCI` (Problem 4.1); it will nevertheless yield a valid CI (close to the centroid considered), which is required to solve Problem 4.2.

4.2.4 Evaluation

We demonstrate the benefits of using an integrated approach for building k CIs through the following example. Consider the case of Mary whose job is to train future users of products developed by a large software company. Mary often travels to different places where she spends extended periods of time, i.e., at least 2 weeks, during which she rents an apartment. In her free time, Mary enjoys going to the theater and dining out wherever she stays. She also practices yoga and likes swimming. Mary would be interested in exploring a map with representative CIs in different areas in the city she is planning to visit. The validity constraints for each CI is that they should contain at least one theater, a pharmacy, a gym, two restaurants and a subway station, and cohesiveness is measured through geographic distance. Figures 4.4a, 4.4b and 4.4c show three sets of CIs for Paris produced using a dataset from Tourpedia[1] with three different methods: the one-at-a-time approach that summarizes each homogeneous item collection separately, the two-stage approach that decouples validity, cohesiveness and representativity, and the integrated approach, detailed in this chapter, that optimizes validity, cohesiveness and representativity together. The CIs generated using the integrated approach (KFC, Figure 4.4c) offer the best trade-off between validity, cohesiveness and representativity. Indeed, the CIs in Figure 4.4a tend to favor representativity (coverage of the city) to the expense of cohesiveness (items in each CI are not close to each other). Those in Figure 4.4b are located on the edges of the city because this two-stage approach first produces the most cohesive valid CIs, which limits their representativity in the second stage.

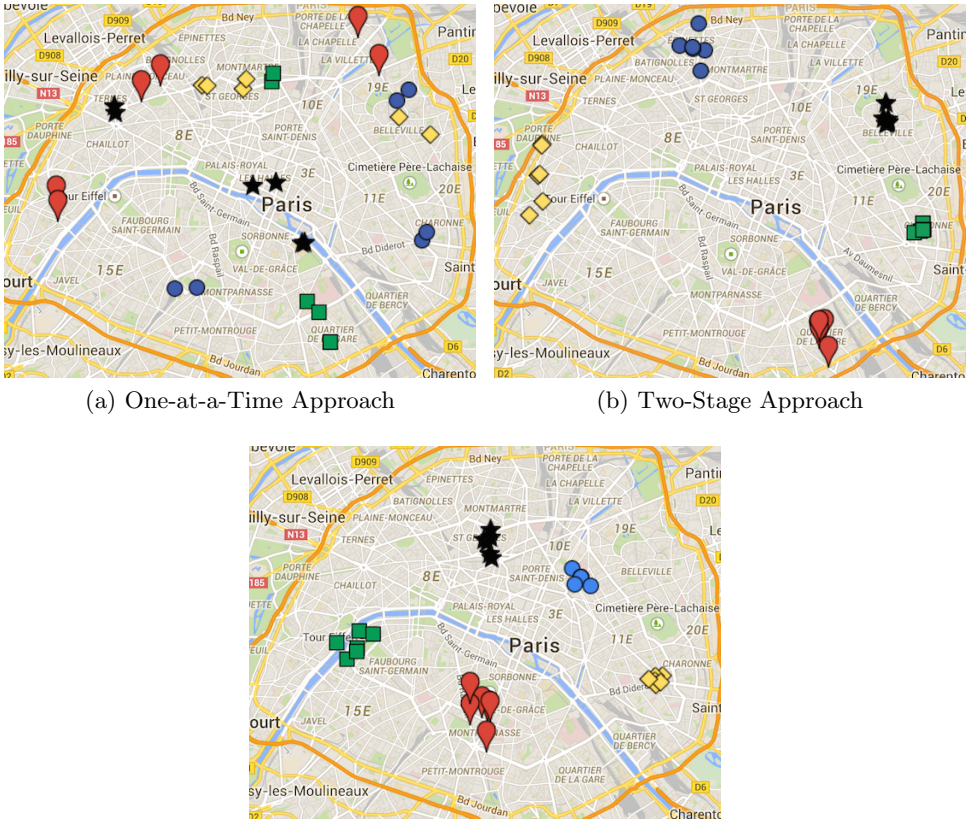
4.2.5 Conclusion

On the Web, simple navigation queries can be easily answered by a list of individual links to Websites. However, as the needs of the user become more complex, Composite Items become more appropriate, as they cover multiple facets of the needs of the user simultaneously. In this chapter, we propose an integrated approach for building valid and cohesive CIs, while also ensuring that our results are representative of the dataset. This is a departure from previous approaches that use disjoint processes phases to achieve this goal.

We designed a new integrated algorithm that builds the k most valid, cohesive and representative CIs of an input dataset. Our algorithm integrates constraint satisfaction into fuzzy clustering in order to simultaneously optimize for validity, cohesion and representativity. Experiments on real datasets show that the integrated approach outperforms two-stage ones resulting in CIs that achieve very good representativity of existing items.

While the evaluation presented in this chapter only builds CIs of POIs, we also experimented with CIs in a very different context: crowdsourcing. We show in [7] that CIs can be used to efficiently aggregate micro-tasks into a composite task, which helps worker navigate crowdsourcing platforms such as Amazon Mechanical Turk [6] more efficiently.

[1] http://datahub.io/dataset/tourpedia



(a) One-at-a-Time Approach (b) Two-Stage Approach

(c) Integrated Approach (KFC)

Figure 4.4: Alternative approaches to build composite items

4.3 Summary of contributions

Analyzing increasingly large amounts of data stresses the importance of having a qualitative approach. Analysts are unable to browse thousands of results. It is our role, as data miners, to synthesize the most relevant information of a dataset before returning it.

One of the directions to achieve this goal is to focus on raking functions, and the ability to assign a score to each result. This is the traditional approach of information retrieval, and the goal is to extend this to different types of data mining algorithms. Surprisingly, the challenge often does not come from devising new interestingness functions, as there are in practice plenty in the literature. Instead, it is of selecting the right relevance function for the right dataset. In our work on pattern mining, presented in Section 4.1, we propose CAPA, a general approach for automatically comparing a large number of interestingness functions. This is followed by a user study with marketing experts, in which we identify the most appropriate ranking function for their use-case.

A different direction consists in re-defining what constitutes a result. We are specifically interested in the composite-items approach, that, instead of individual results, returns a bundle of items that complement each other. Section 4.2 describes KFC, an algorithm that aims at increasing the quality of composite items by using an integrated approach that optimizes validity, cohesiveness and representativity simultaneously. We demonstrate the benefits of composite-items in a variety of applications, including recommending POIs for visiting new cities, recommending movies [65], and crowd-sourcing.

Chapter 5

Perspectives

While data analysis has been a very active area of research, there are still many avenues of research to explore. As stated in the introduction, I enjoy working on diverse topics, and this is something I would like to preserve in the future. In the remainder of this section, I list research topics I plan to pursue in the next few years.

Systems design

Distributed exploration of a result-space

Popular distributed data processing platform (namely Hadoop and Spark) expose an execution model in which large tasks are partitioned into independent stages. During the execution of a stage, there are no communications between executors. This model is problematic for any algorithm doing a *dynamic exploration* of a results space for two main reasons: (i) it is very difficult to initially produce a balanced partition of the workload, as the result candidates are generated dynamically and their number can vary a lot from one partition to the next, and (ii) results discovered in one partition cannot be shared with other partitions to eliminate portions of the search-space.

In Chapter 3, I describe two algorithms for processing top-k queries on these platforms. The algorithms presented remain within the bounds of Map-Reduce, but compensate for its limitations by using strategies specific to the type of queries considered. In other cases [51], researchers have deployed a Publish-Subscribe platform (namely Kafka) in addition to the Map-Reduce one in order to obtain message-passing functionalities. This requires a lot of effort from the developer, and the solution implemented is specific to the problem considered. A different approach would be to extend the programming model and offer more possibilities to the developer.

Coming up with a proper abstraction for large-scale data processing is difficult. The popularity of Map-Reduce comes from its simplicity: developing a Map-Reduce application does not require extensive knowledge about distributed systems, parallelism, or fault tolerance. Hence, Map-Reduce is accessible to most developers and is taught to many students. There is clearly a huge gap between Map-Reduce platforms, and low level primitives such as MPI. The challenge is to significantly extend the possibilities offered by the model, without transferring the complexity to the developer. I believe what is required is an integrated abstraction for distributed shared objects that can be accessed by multiple executors simultaneously to share information. These shared objects have to integrate seamlessly with the fault tolerance policy

of the platforms, which is non trivial as the executions are no longer isolated. Furthermore, they need to have a simple general purpose API accessible to most developers.

Algorithms development

Hardware-accelerated algorithms for machine learning

Innovation in algorithms and software for machine learning leads to new applications and better results. However, hardware can also be a driving force behind discoveries in data analysis. Specialized hardware support significantly improves the efficiency of data processing, both in terms of speed and power usage. Companies like Microsoft added reconfigurable hardware (FPGA) to their servers to accelerate some modules of the Bing search engine. In addition, GPUs have seen a lot of use in the domain of neural networks.

For the past two years, I have worked with STMicroelectronics and the TIMA laboratory in Grenoble to develop data analysis algorithms with hardware support. Our first results on pattern mining [80] and deep neural networks [4] are very promising, and outperformed existing industrial products such as IBM TrueNorth. This work demonstrates that the co-development of the algorithm and the hardware solution to support it leads to more progress than the development of algorithms for existing hardware or of hardware for existing algorithms. Indeed, the cost of algorithmic operation varies significantly depending on the hardware support available, so they influence each other. I will continue this collaboration and extend this work to different types of data mining algorithms, with a specific focus on data streams.

Mining massive graph datasets

Many interesting datasets are represented as graphs showing relations between entities. The YAGO dataset for instance contains 120M facts about 10M entities. The structure of molecules and proteins can also be represented as a graph, and their properties are related to specific layouts. Analyzing these graphs is important to detect structural characteristics, and predict new facts. Processing large graphs is challenging because of the connected nature of the data: a graph cannot be easily split into independent partitions.

Several general-purpose platforms were built to process these graphs at scale in parallel and distributed platforms such as MapReduce and Spark (e.g. Arabesque [89]). To the best of my knowledge, the platforms able to perform graph mining all require the graph to be replicated on each executor, which limits scalability in terms of graph size. More specialized algorithms backed by a standard databases approach (joins) were developed to mine graphs (e.g. AMIE++ [43] and Ontological Pathfinding [29]). These algorithms are unable to scale beyond patterns of 3 edges, because they search for instances from scratch each time instead of building on previous results. In addition, I suspect these existing approaches to be unable to deal efficiently with some specific graph structures such as cliques and stars, that generate a number of instances that grows exponentially with the pattern size.

Over the past years I have developed an expertise both in distributed systems and mining algorithms. My goal is to design graph mining algorithms that will benefit from large-scale distributed platforms while being optimized to solve specific graph mining problems. Preliminary experiments on Spark show promising results.

Information discovery

A data mining approach to debugging applications

Debugging application can be a difficult problem. Many tools exist to solve functional bugs, in which a function does not output the correct result. However, there are many more subtle bugs that emerge in complex systems for which these tools (debuggers...) are unable to help developers. Consider for instance a video playback software. A frame decoded correctly but displayed late is the symptom of a temporal bug. The code was executed correctly and does not contain errors, but some system activity occurring in the background delayed the execution of the function decoding the frame, resulting in screen tearing.

Data mining can help developers analyze the behavior of their program to better identify the origin of these subtle bugs. I have supervised in collaboration with STMicroelectronics a Ph.D student working on this problem [53]. In addition, I am currently supervising a Ph.D student applying data mining to model checking [14].

For my future research activities, I am starting a collaboration with François Trahay and Gaël Thomas from Telecom SudParis. Our goal is to develop data mining algorithms to better understand the performance of large-scale parallel and distributed systems. This requires pattern mining algorithms to identify structure in application traces, and an integration of quantitative measures such as CPU usage, proportion of cache misses, and frequency context switches, to correlated patterns with performance issues.

Contextual recommendation

The goal of a recommendation strategy and is to estimate a user's interest for items she has not expressed interest for before, and return the items she is most likely to appreciate. Context-aware recommendations refer to the need to take into account additional information in serving recommendations in serving content to users. Context refers to many different dimensions, temporal (time of day or time of year), geographical (at home or at work), presence of absence of others (in the company of friends or in the company of kids).... Context can be utilized at various stages of the recommendation process, including at the pre-filtering and the post-filtering stages and also as an integral part of the contextual modeling.

In this work, I will investigate how various techniques of using the contextual information can be combined into a single recommendation approach to improve recommendation accuracy. In collaboration with Sihem Amer-Yahia, I will take part in a project funded by Total and starting in 2018 to explore this issue. Total will provide a dataset of user-data obtained using the beacon technology that makes use of customers' smartphones to better understand their needs.

Bibliography

[1] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 99–110, 2010.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 207–216, 1993.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[4] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and FRédéric Pétrot. Ternary neural networks for resource-efficient ai applications. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2017.

[5] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[6] Amazon Mechanical Turk. https://www.mturk.com/mturk/welcome.

[7] S. Amer-Yahia, E. Gaussier, V. Leroy, J. Pilourdault, R. M. Borromeo, and M. Toyama. Task composition in crowdsourcing. In *Proceedings of the IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 194–203, 2016.

[8] Sihem Amer-Yahia, Francesco Bonchi, Carlos Castillo, Esteban Feuerstein, Isabel Méndez-Díaz, and Paula Zabala. Composite retrieval of diverse and complementary bundles. *IEEE Transactions on Knowledge and Data Engineering*, 26(11):2662–2675, 2014.

[9] Chris Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More.* Hyperion, 2006.

[10] Albert Angel, Surajit Chaudhuri, Gautam Das, and Nick Koudas. Ranking objects based on relationships and fixed associations. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 910–921, 2009.

[11] Apache Samza. http://samza.apache.org/.

[12] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1027–1035, 2007.

[13] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vassilis Plachouras, and Luca Telloli. On the feasibility of multi-site web search engines. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, pages 425–434, 2009.

[14] Gianluca Barbon, Vincent Leroy, and Gwen Salaüun. Ternary neural networks for resource-efficient ai applications. In *Proceedings of the IPM International Conference on Fundamentals of Software Engineering (FSEN)*, pages 1–15, 2017.

[15] James C. Bezdek. A convergence theorem for the fuzzy ISODATA clustering algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):1–8, 1980.

[16] James C. Bezdek, Robert Ehrlich, and William Full. FCM: The Fuzzy c-Means Clustering Algorithm. *Computers & Geosciences*, 10(2-3):191–203, 1984.

[17] Roi Blanco, Edward Bortnikov, Flavio Junqueira, Ronny Lempel, Luca Telloli, and Hugo Zaragoza. Caching search engine results over incremental indices. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 82–89, 2010.

[18] Angela Bonifati, Martin Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of xml views. *ACM Transactions on Database Systems*, 38(3):14:1–14:45, September 2013.

[19] Horatiu Bota, Ke Zhou, Joemon M. Jose, and Mounia Lalmas. Composite retrieval of heterogeneous web search. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 119–130, 2014.

[20] Ulf Brefeld, B. Barla Cambazoglu, and Flavio P. Junqueira. Document assignment in multi-site search engines. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, pages 575–584, 2011.

[21] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[22] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[23] Alexander Brodsky, Sylvia Morgan Henshaw, and Jon Whittle. Card: a decision-guidance framework and application for recommending composite alternatives. In *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, pages 171–178, 2008.

[24] Jean-Paul Calbimonte, Jose Mora, and Oscar Corcho. *Query Rewriting in RDF Stream Processing*, pages 486–502. Springer International Publishing, 2016.

[25] B. Barla Cambazoglu, Emre Varol, Enver Kayaaslan, Cevdet Aykanat, and Ricardo Baeza-Yates. Query forwarding in geographically distributed search engines. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 90–97, 2010.

[26] B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, pages 411–420, 2010.

[27] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Technical Committee on Data Engineering*, 38(4):28–38, 2015.

[28] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing interval joins on map-reduce. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 463–474, 2014.

[29] Yang Chen, Sean Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. Ontological pathfinding. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 835–846, 2016.

[30] Eden Chlamtác, Michael Dinitz, Christian Konrad, Guy Kortsarz, and George Rabanca. The densest k-subhypergraph problem. In *Proceedings of APPROX-RANDOM*, volume 60, pages 6:1–6:19, 2016.

[31] Munmun De Choudhury, Moran Feldman, Sihem Amer-Yahia, Nadav Golbandi, Ronny Lempel, and Cong Yu. Automatic construction of travel itineraries using social breadcrumbs. In *Proceedings of the ACM conference on Hypertext and hypermedia*, pages 35–44, 2010.

[32] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[33] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 3(1-2):48–57, 2010.

[34] W.W. Daniel. *Applied Nonparametric Statistics*. Houghton Mifflin, 1978.

[35] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[36] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap interval partition join. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1459–1470, 2014.

[37] Maciej Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009.

[38] Didier Dubois, Allel HadjAli, and Henri Prade. Fuzziness and uncertainty in temporal reasoning. *J. UCS*, 9(9):1168, 2003.

[39] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 683–694, 2004.

[40] Ronald Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 216–226, 1996.

[41] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 102–113, 2001.

[42] Jonathan Finger and Neoklis Polyzotis. Robust and efficient algorithms for rank join evaluation. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 415–428, 2009.

[43] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Fast rule mining in ontological knowledge bases with amie++. *The VLDB Journal*, 24(6):707–730, 2015.

[44] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, 2005.

[45] Weizheng Gao, Hyun Chul Lee, and Yingbo Miao. Geographically focused collaborative crawling. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 287–296, 2006.

[46] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern numa systems. *Communications of the ACM*, 58(12):59–66, 2015.

[47] Liqiang Geng and Howard J. Hamilton. Interestingness Measures for Data Mining: A Survey. *ACM Computing Surveys*, 38(3), 2006.

[48] Sharad Goel, Andrei Broder, Evgeniy Gabrilovich, and Bo Pang. Anatomy of the long tail: ordinary people with extraordinary tastes. In *Proceedings of the Third International Conference on Web Search and Data Mining (WSDM)*, pages 201–210, 2010.

[49] Adrian Graham, Hector Garcia-Molina, Andreas Paepcke, and Terry Winograd. Time as essence for photo browsing through personal digital libraries. In *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 326–335, 2002.

[50] Jiawei Han, Jianyong Wang, Ying Lu, and Petre Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 211–218. IEEE, 2002.

[51] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combinations. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 7(4):301–312, 2013.

[52] Bernardo A. Huberman, Daniel M. Romero, and Fang Wu. Social networks that matter: Twitter under the microscope. *First Monday*, 14(1), 2009.

[53] Oleg Iegorov. *Data Mining Approach to Temporal Debugging of Embedded Streaming Applications*. PhD thesis, Universit Grenoble Alpes, 2016.

[54] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 754–765, 2003.

[55] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11:1–11:58, 2008.

[56] Alexander Jaffe, Mor Naaman, Tamir Tassa, and Marc Davis. Generating summaries and visualization for large collections of geo-referenced photographs. In *Proceedings of the ACM SIGMM International Workshop on Multimedia Information Retrieval (MIR)*, pages 89–98, 2006.

[57] Kalervo Järvelin and Jaana Kekäläinen. Cumulated Gain-based Evaluation of IR Techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.

[58] Flavio P. Junqueira, Vincent Leroy, and Matthieu Morel. Reactive index replication for distributed search engines. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 831–840, 2012.

[59] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing (SISC)*, 20(1):359–392, 1998.

[60] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, 1938.

[61] M. Kirchgessner, V. Leroy, S. Amer-Yahia, and S. Mishra. Testing interestingness measures in practice: A large-scale analysis of buying patterns. In *Proceedings of the IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 547–556, 2016.

[62] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 239–250, 2015.

[63] Victor Lavrenko and W. Bruce Croft. *Relevance Models in Information Retrieval*, pages 11–56. Springer Netherlands, 2003.

[64] V. Leroy, M. Kirchgessner, A. Termier, and S. Amer-Yahia. Toppi: An efficient algorithm for item-centric mining. *Information Systems*, 64:104 – 118, 2017.

[65] VIncent Leroy, Sihem Amer-Yahia, Eric Gaussier, and Hamid Mirisaee. Building representative composite items. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, pages 1421–1430, 2015.

[66] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the second conference on Recommender systems (RecSys)*, pages 107–114, 2008.

[67] Guimei Liu, Mengling Feng, Yue Wang, Limsoon Wong, See-Kiong Ng, Tzia Liang Mah, and Edmund Jon Deoon Lee. Towards exploratory hypothesis testing and analysis. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 745–756, 2011.

[68] Hongjun Lu, Beng Chin Ooi, and Kian-Lee Tan. On spatially partitioned temporal join. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 546–557, 1994.

[69] Luís Marujo, Ricardo Ribeiro, Anatole Gershman, David Martins de Matos, João P. Neto, and Jaime Carbonell. Event-based summarization using a centrality-as-relevance model. *Knowledge and Information Systems*, 50(3):945–968, 2017.

[70] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 398–412, 2005.

[71] Shin-Ichi Minato, Takeaki Uno, Koji Tsuda, Aika Terada, and Jun Sese. A fast method of statistical assessment for combinatorial hypotheses based on frequent itemset enumeration. In *Machine Learning and Knowledge Discovery in Databases*, volume 8725, pages 422–436. Springer, 2014.

[72] Benjamin Négrevergne, Alexandre Termier, Jean-Francois Mhaut, and Takeaki Uno. Discovering closed frequent itemsets on multicore: Parallelizing computations and optimizing memory accesses. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, pages 521–528, 2010.

[73] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *IEEE International Conference on Data Mining Workshops*, pages 170–177, 2010.

[74] Nikos Ntarmos, Ioannis Patlakas, and Peter Triantafillou. Rank join queries in nosql databases. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 7(7):493–504, 2014.

[75] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 949–960, 2011.

[76] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 398–416, 1999.

[77] Gregory Piatetsky-Shapiro. *Knowledge Discovery in Databases*. Menlo Park, CA: AAI/MIT, 1991.

[78] Julien Pilourdault. *Scalable Algorithms for Monitoring Activity Traces*. PhD thesis, Universit Grenoble Alpes, 2017.

[79] Julien Pilourdault, Vincent Leroy, and Sihem Amer-Yahia. Distributed evaluation of top-k temporal joins. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1027–1039, 2016.

[80] Adrien Prost-Boucle, FRédéric Pétrot, Vincent Leroy, and Hande Alemdar. Efficient and versatile fpga acceleration of support counting for stream mining of sequences and frequent itemsets. *ACM Transactions on Reconfigurable Technology and Systems*, 10(3):21:1–21:25, 2017.

[81] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikolaos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: Scaling online social networks. *IEEE/ACM Transactions on Networking*, 20(4):1162–1175, 2012.

[82] Marie-Christine Rousset and Federico Ulliana. Extracting bounded-level modules from deductive rdf triplestores. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, pages 268–274, 2015.

[83] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 843–854, 2010.

[84] Rodrygo L. T. Santos, Craig Macdonald, and Iadh Ounis. Search result diversification. *Found. Trends Inf. Retr.*, 9(1):1–90, March 2015.

[85] Karl Schnaitter and Neoklis Polyzotis. Evaluating rank joins with optimal cost. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 43–52, 2008.

[86] R. R. Sokal and C. D. Michener. A Statistical Method for Evaluating Systematic Relationships. *The University of Kansas science bulletin*, 38:1409–1438, 1958.

[87] Igor Tatarinov. An efficient LFU-like policy for web caches. Technical report, 1998.

[88] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 963–972, 2011.

[89] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 425–440, 2015.

[90] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014.

[91] Twitter Timeline Scalability. http://www.infoq.com/presentations/Twitter-Timeline-Scalability.

[92] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. An efficient algorithm for enumerating closed patterns in transaction databases. In *Discovery Science*, pages 16–31, 2004.

[93] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Proceedings of the Workshop on Frequent Itemset Mining Implementations (FIMI)*, 2004.

[94] Sheng Wang, David Maier, and Beng Chin Ooi. Fast and adaptive indexing of multi-dimensional observational data. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 9(14):1683–1694, 2016.

[95] Xuanhui Wang, Michael Bendersky, Donald Metzler, and Marc Najork. Learning to rank with selection bias in personal search. In *Proceedings of the ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 115–124, 2016.

[96] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 205–218, New York, NY, USA, 2009.

[97] Min Xie, Laks V.S. Lakshmanan, and Peter T. Wood. Breaking out of the box of recommendations: From items to packages. In *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2010.

[98] Chengxiang Zhai and John Lafferty. Model-based feedback in the language modeling approach to information retrieval. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, pages 403–410, 2001.

[99] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2007.

[100] Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 5(11):1184–1195, 2012.